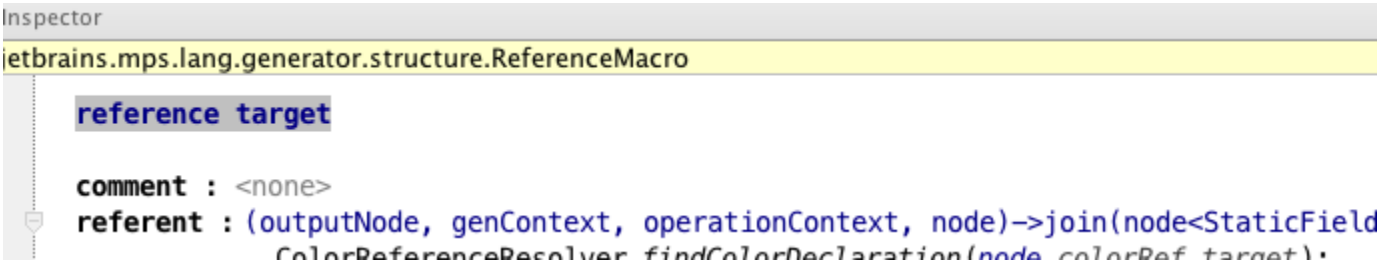
What's new in MPS 2017.2

EAP1

Generator: pointer to a node from reference macro

Reference macro in templates (`->$`) now supports `SNodeReference` as specification of a new target. With that, templates don't need access to a target's node model the moment they are applied.

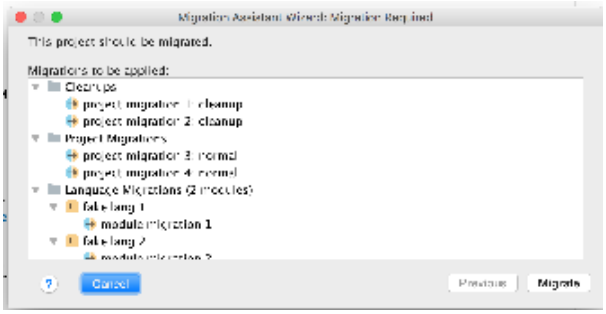


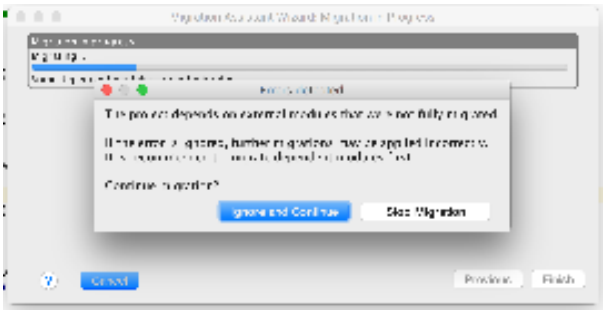
Editor: Default changed - R/O model access cell not editable

Read-Only model access cells are generated as with editable == false by-default. In most of known cases instances of this cell should not be editable, so we changed default behaviour of MPS editor generator in order to reflect it. If user would like to make this cell editable, it's still possible by overriding default value with editor cell style: "editable == true".

Migration assistant redesigned

The migration assistant has been rewritten from scratch. It now offers a better view of which migrations should be applied and has an ability to ignore and skip non-critical errors in the project. We will continue to evolve it.





EAP2

Cross-model generation story

In EAP 2, we enabled generation plans to generate descriptor models for languages (known as `<language.name>@descriptor`). If you got a custom aspect, you may need to make sure its generator extends generator of `jetbrains.mps.lang.descriptor` l anguage, as it's the way to get custom extensions activated for the plan.

Public API for Intentions aspect

New set of public API interfaces was extracted from the existing (closed) interfaces used for Intentions language aspect. New interfaces are located in jetbrains.mps.openapi.intentions.* package. All previously existing interfaces (jetbrains.mps.intentions.*) were deprecated and will be removed in the next MPS release. As a side-effect, the type of "intentionExecutable" parameter, available inside transformation menu parts for intentions was changed to return open API interface. Most probably this change will not affect any user code. In case the code was affected by this change, it should be revisited and rewritten in order to use API interface.

EAP3

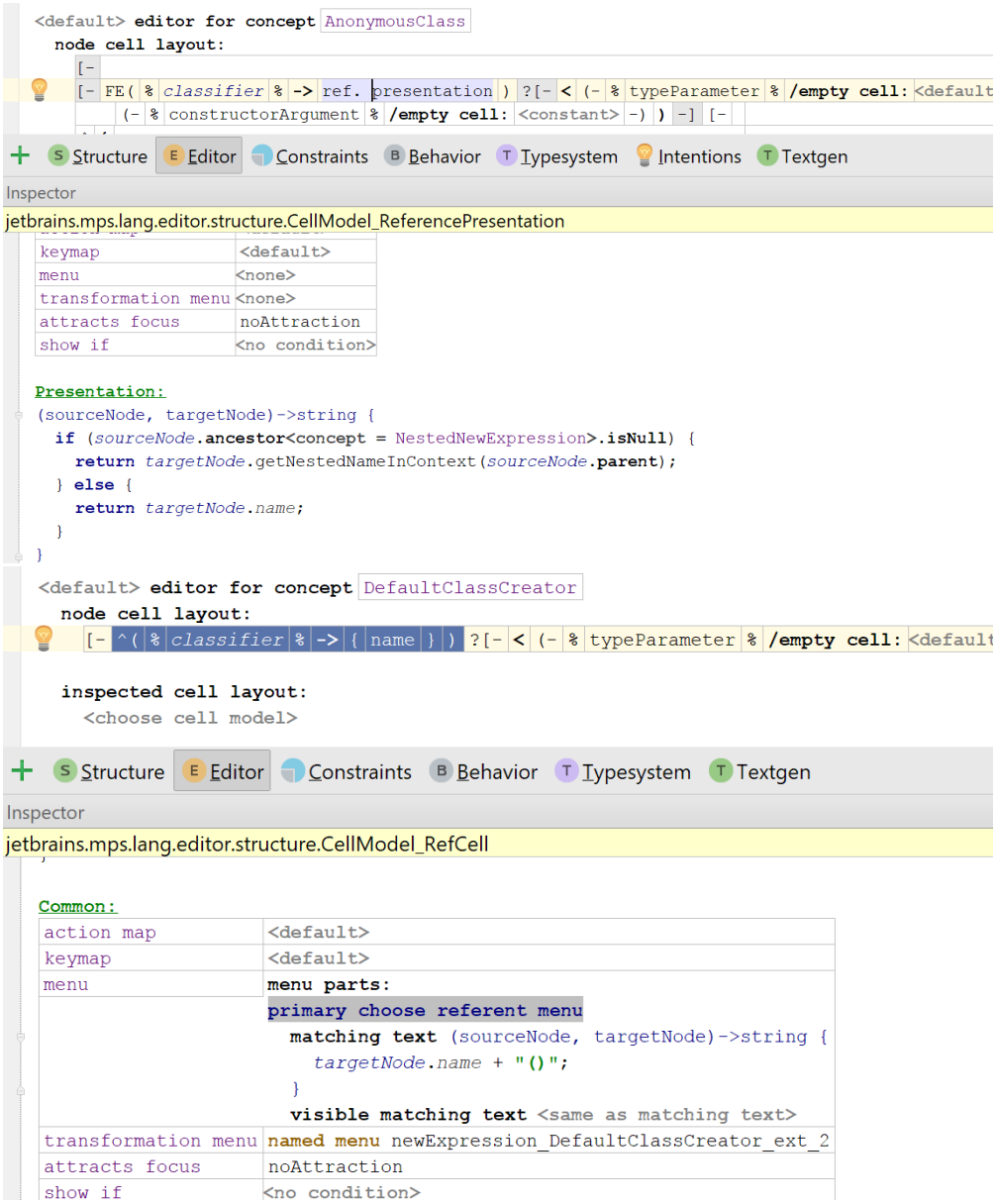
'reference actions' substitute menu parts

Suppose you have a concept which has a reference to another concept and on creating the instances of this concept in the editor you want to show instances of the referent concept that are in scope. Now it's possible with using 'reference actions' sub stitute menu parts functionality that was added in Substitute Menu Language.

```
substitute menu ClassifierClassExpression_SmartReference for concept ClassifierClassExpression
  reference actions classifier(output concept: default)
    matching text (parentNode, currentTargetNode, link, editorContext, model, pattern, referencedNode)->string {
      referencedNode.getNestedNameInContext(parentNode) + ".class";
    }
    visible matching text <same as matching text>
    description text <default>
```

reference presentation text customization options

Now you can specify matching text and in-editor textual presentation for references directly in the editor aspect.



smart reference attributes

MPS already has a functionality which is called 'smart reference' that give more suitable textual presentation and code completion. Now you can enable it for a concept by annotating it declaration with 'smart reference' attribute.

```
@smart reference
  reference: variableDeclaration
  presentation: <default>
concept VariableReference extends    Expression
                          implements TypeAnnotable
                                     IVariableReference
                                     ILocalReference

  instance can be root: false
  alias: <no alias>
  short description: reference to variable

  properties:
  << ... >>

  children:
  << ... >>

  references:
  variableDeclaration : VariableDeclaration[1]
```

Note that other way (heuristics based on concept alias) now is deprecated and will no longer support in future releases. All existent smart reference will be migrated automatically to use the attribute.

migration of presentation query in reference constraints

Reference presentation part in the constraints aspect has been designed purely and now can be replaced with new functionality. So we decided to deprecate it and migrate it to new options. Most of the code will be migrated automatically. Some code that produced with migration can be simplified so consider to review it.

There is a case when a presentation query can not be migrated: suppose you have an editor for a concept with reference link and then have a reference constraint with defined presentation part for its reference in one of its subconcepts. If editor component doesn't overriden in subconcept, MPS doesn't know where this presentation part should be inlined. In this case, you should manually migrate the presentation part usage to prevent uncorrected reference presentation in user code. There are several alternatives to do it:

- Simply override the editor in subconcept. Move the code from presentation part to the proper reference cell.
- Extract the reference cell into a separate component and override the component for subconcept.
- Create new behavior method that provides a presentation for the reference. Make reference cell delegates to created method. Override this method in subconcept.

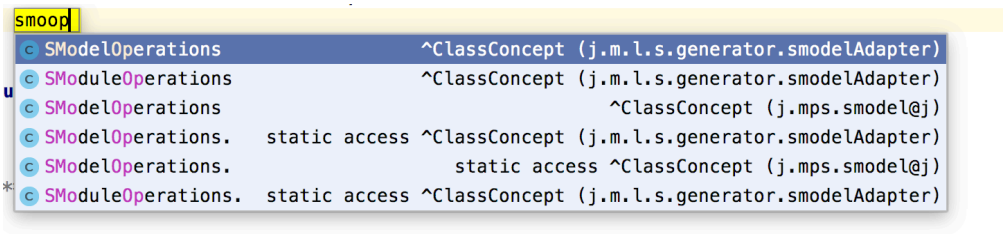
If you are expecting that your language may be extended in another project by someone else, do not remove deprecated presentation parts. Otherwise, extending languages may be migrated improperly.

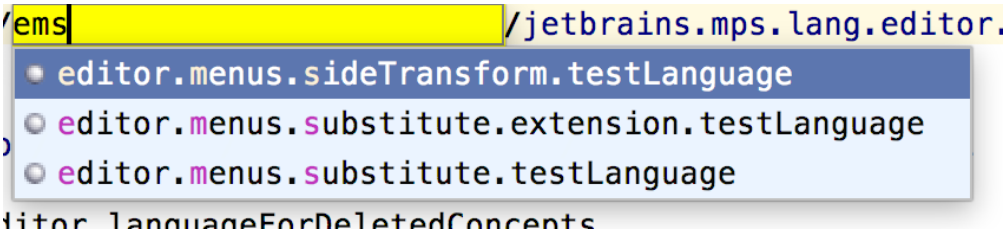
Shiftless code completion

You should not press shift anymore in the completion.

It supports the shiftless pattern matching now (I works same as in idea:https://blog.jetbrains.com/idea/2011/04/shiftless-code-completion-and-navigation-in-intellij-idea-105/) and substring search as before.

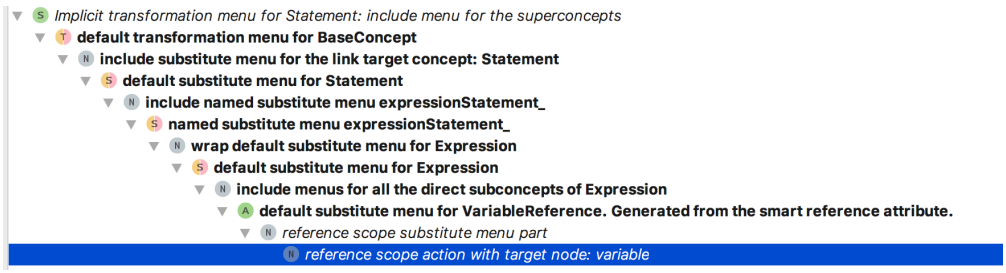
Here are some examples:





Show Item Trace

Sometimes it is hard to track how the action appeared in the completion or context assistant because of there are many substitute and transformation menus including each other.
Now you may select some action in completion (by arrows) or in the context assistant (by pressing cmd/ctrl+alt+Enter) and then press cmd/ctrl+alt+B.
You will see the trace in the project tool. This is the trace of the menu and menu part declaration which include each other starting from the top-level menu and ending with the action declaration. If the menu or the menu part declaration is explicit and is in the project, then it has bold style in tool and you can click on it and go to the declaration.
Here how it looks like when we put the caret on the statement, show completion for the variable reference and then invoke "Show Item Trace":



We see that what we see in completion is the default transformation menu for the Statement which includes the menu for superconcept which is the BaseConcept. It in its turn includes the substitute menu for the Statement, which in it's turn wraps the menu for the Expression. Then it comes to subconcepts of Expression, one of which is the VariableReference. VariableReference is the "smart reference" concept, so it tries to find all visible target of the Variable concept. So that's how the variable reference appears in the menu for the statement.

Cross-model generation story

In this EAP, we transform few language aspect with a help of generation plans, and restore cross-model references between language and aspect descriptors with regular mapping labels, no dirty magic ("make up a class qualified name and hope it works") involved. Affected aspects are: structure, textgen, typesystem, dataflow and constraints.

Public Preview

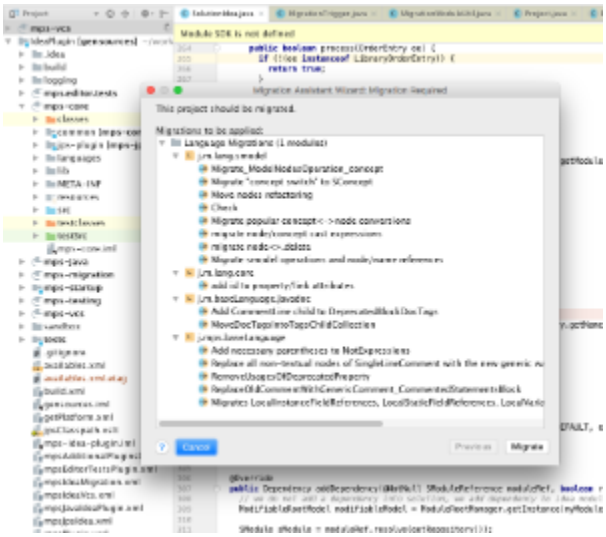
Running Editor tests in IDEA Plugin

With the new JUnit test suite (jetbrains.mps.idea.core.tests.PluginsTestSuite) it is possible to execute editor tests for your language within MPS plugin for IntelliJ IDEA. To make use of this functionality you have to create simple ANT script installing all necessary plugins into IDEA platform & executing tests by specifying test module name(s). Demo will be provider.

Migration Assistant in Idea Plugin

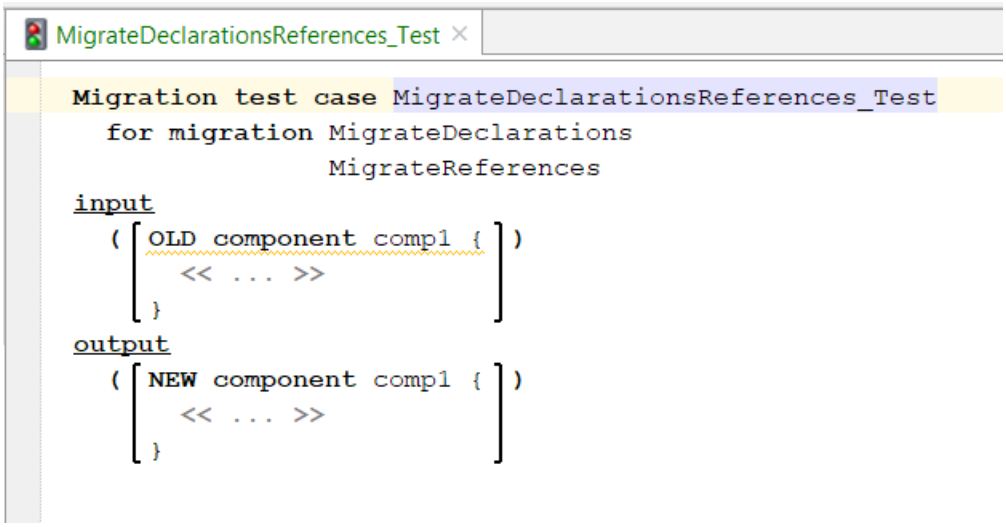
Language migrations are now supported in IntelliJ IDEA plugin.

Whether a migration was added to some MPS language used in IDEA project, the Migration Assistant will be shown and models in the projects will be updated to the newer languages.



Migration tests

Testing migration scripts becomes more convenient with Migration test cases. Migration test cases allow testing migration scripts by running them on specified nodes and checking produced result. See documentation page for details.



Two step deletion

Sometimes it is hard to predict what part of the code will be deleted when you press Delete or Backspace. For example, when the caret is on the semicolon of the baseLanguage statement and you press Backspace, the whole statement will be deleted. With the two step deletion feature, you now can see what part of the code which will be deleted.

Here how it works: you press Delete or Backspace and the part of the code which will be deleted becomes highlighted. If it suits you, you press Delete or Backspace again and the code will be deleted. If after highlighting you realize that you don't want to delete this piece of code you can press Escape or just move the caret and the highlighting will disappear.

Let's see the example:
Put the caret to the statement semicolon.

```
public static void main(string[] args) {
  System.out.println();
}
```

Press Backspace. The whole statement is highlighted. This means that if you press Backspace again, the statement will be deleted.

```
public static void main(string[] args) {
  System.out.println();
}
```

Press Backspace again. The statement is deleted.

```
public static void main(string[] args) {
  <no statements>
}
```

The same works by default for other nodes.

Note that if the node is selected, it will be removed immediately without highlighting. Also if the caret is on the editable text cell, the text parts will be also removed immediately.

To turn on the two step deletion, check the "two step deletion" checkbox in Preferences > Editor > General

The language designer may include the two step deletion scenario in her custom delete actions.

The ApproveDelete_Operation in jetbrains.mps.lang.editor is introduced for that purpose. This operation is applied to the node:

```
node.approveDelete [in: editorContext]
```

This operation returns true iff it succeed and the node was approved for deletion, or more formally all the following conditions were met:

1) The two-step deletion preferences option is checked.

2) The node was not fully selected.

3) The node was not approved for deletion already.

In this case, the node approved for deletion is highlighted and the custom delete action may stop.

If the same custom delete action is called immediately after approving the deletion, the operation will return false (because the node is approved already) and the action will proceed with the deletion.

Let's see the typical scenario from the baseLanguage:

```
if (node.operation.approveDelete [in: editorContext]) { return; }
node.operation = new node<AbstractOperation>();
```

This is the part of the delete action for the Dot_Expression's operation.

This action as described before firstly tries to approve the operation for the deletion and if succeed it stops.

If not, that means that either node's operation is already approved (= highlighted), or the operation is selected by the user or the "two step deletion" preferences option is turned off. In this case, we delete the operation and replace it with the node of the abstract concept.

Sometimes the custom delete action is more complicated than just deleting the node. This leads to more complicated changes of the editor. For those cases, we can specify the cell for deletion approving.

Let's see the scenario: we press delete on the "final" keyword on the IncompleteMemberDeclaration. There is the custom action which sets the final property to false. In the editor, there is the cell which is shown only if the final property of the node is true, so after the action, the cell wouldn't be shown.

If we want to highlight the final keyword, we approve it for the deletion:

```
if (node.approveDelete [in: editorContext, cell: finalKeyword]) { return; }
node.final.set(false);
```