Credits

The MPS user guide has evolved thanks to numerous big and small contributions of many contributors. Although most of the work has been done by the individual members of the MPS core team, external MPS users have done their share, as well. On this page we list the prominent external contributors, who have helped evolve the user guide:

- Markus Voelter
- Marco Lombardo
## Contents

Front cover i
Credits i
Contents ii

### I Cookbooks

1 Getting the dependencies right 2
   1.1 Motivation ........................................... 2
   1.2 Useful keyboard shortcuts ................................ 2
   1.3 Solution ........................................... 2
      1.3.1 Common ........................................... 3
      1.3.2 Dependencies ........................................... 3
      1.3.3 Used Languages ........................................... 4
      1.3.4 Java ........................................... 4
      1.3.5 Facets ........................................... 4
   1.4 Solution models ........................................... 4
      1.4.1 Dependencies ........................................... 5
      1.4.2 Used languages ........................................... 5
      1.4.3 Advanced ........................................... 5
      1.4.4 Virtual packages ........................................... 5
   1.5 Adding external Java classes and jars to a project - runtime solutions ........................................... 5
   1.6 Language ........................................... 5
      1.6.1 Common ........................................... 6
      1.6.2 Dependencies ........................................... 6
      1.6.3 Used Languages ........................................... 6
      1.6.4 Runtime ........................................... 6
      1.6.5 Java ........................................... 6
      1.6.6 Facets ........................................... 7
   1.7 Language models/aspects ........................................... 7
      1.7.1 Dependencies / Used Languages / Advanced ........................................... 7
   1.8 Generator ........................................... 7
      1.8.1 Common ........................................... 7
      1.8.2 Dependencies ........................................... 7
      1.8.3 Used Languages ........................................... 7
      1.8.4 Generators priorities ........................................... 8
      1.8.5 Java ........................................... 8
      1.8.6 Facets ........................................... 8
      1.8.7 Generator models ........................................... 8

2 Building an interpreter cookbook 9
   2.1 Instant preview ........................................... 12
   2.2 Defining an Interpreter ........................................... 15
      2.2.1 A whole-scene Interpreter ........................................... 15
      2.2.2 Shape preview ........................................... 17
      2.2.3 Scene preview ........................................... 18
## 3 Editor cookbook

3.0.4 How to properly indent a block of code ........................................... 20  
3.0.5 Alias ......................................................................................... 20  
3.0.6 Unique constraint ........................................................................... 21  
3.0.7 An indented vertical collection ....................................................... 22  
3.0.8 Optional visibility ........................................................................... 22  
3.0.9 Defining and reusing styles ............................................................ 22  
3.0.10 Reusing the BaseLanguage styles ............................................... 23  
3.0.11 Making proper keywords .............................................................. 23  
3.0.12 Adjust abstract syntax for easy editing ........................................ 24  
3.0.13 Specify, which concept should be the default for lists ................... 24  
3.0.14 Make empty lines helpful to further editing ................................... 25  
3.0.15 Hiding concepts from the completion menu .................................. 26  
3.0.16 Easy node replacement ............................................................... 26  
3.0.17 Vertical space separators ............................................................. 26  
3.0.18 Handling empty values in constants ............................................ 27  
3.0.19 Horizontal list separator ............................................................. 28  
3.0.20 Matching braces and parentheses ................................................. 28  
3.0.21 Empty blocks should look empty ................................................... 30  
3.0.22 Make empty constants editable .................................................... 31  

### 3.1 Common editor patterns

3.1.1 Prepending flag keywords ........................................................... 31  
3.1.2 Appending parametrized keywords .............................................. 33  
3.1.3 Node substitution actions ............................................................ 34  
3.1.4 Substitution for custom string values .......................................... 34  
3.1.5 Including parents transform actions ............................................ 36  

### 3.2 Conceptual questions

3.2.1 Extending an existing editor .......................................................... 37  
3.2.2 Design an editor for extension ....................................................... 37  
3.2.3 How to add a refactoring to the menu .......................................... 37  

### 3.3 How to define multiple editors for the same concept

3.3.1 A sample first .............................................................................. 37  
3.3.2 Hints ......................................................................................... 39  
3.3.3 Editor hints for Editor Components ............................................. 40  
3.3.4 Pushing hints from the IDE .......................................................... 40  

### 3.4 How to manipulate and apply editor hints programmatically

### 4 Editor language generation API

4.0.1 API: EditorCell contract .................................................................. 43  
4.0.2 API: EditorCellFactory is now available only within UpdateSession .. 43  
4.0.3 Language Runtime: AbstractEditorBuilder .................................. 43  
4.0.4 Generator: EditorBuilder classes ................................................. 43  
4.0.5 Contextual parameters available for cell builders .......................... 44  
4.0.6 CellFactoryContextClass ........................................................... 44  
4.0.7 GenericCellCreationContext ....................................................... 44  
4.0.8 New signature for createCell() methods ...................................... 44  
4.0.9 Automatic migration script for new createCell() methods .......... 44  
4.0.10 Mapping labels .......................................................................... 44  
4.0.11 CellLayoutConstructor switch introduced ..................................... 45  
4.0.12 New generator for RefCellServiceProvider sub-classes ................ 45  
4.0.13 InlineServiceProvider replaced with EditorBuilder sub-class ....... 45  
4.0.14 Editor Styles generator ............................................................. 45  
4.0.15 StyleClassItem constraints modification ...................................... 45  

## 5 Description comments

5.1 The Calculator language ................................................................... 46  
5.2 Changes to the fields ....................................................................... 46  
5.3 The DescriptionAnnotation concept ............................................... 47  
5.4 Further editor enhancements ............................................................ 49  
5.5 How are we doing so far? ................................................................. 52  
5.6 Updating the generator ................................................................... 53  
5.7 Summary ....................................................................................... 54
**CONTENTS**

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Generator cookbook</td>
<td>55</td>
</tr>
<tr>
<td>6.0.1</td>
<td>How does the generator process the rules?</td>
<td>55</td>
</tr>
<tr>
<td>6.0.2</td>
<td>Can I have more generators for a single language?</td>
<td>55</td>
</tr>
<tr>
<td>6.0.3</td>
<td>Can I have multiple mapping configurations per generator?</td>
<td>55</td>
</tr>
<tr>
<td>6.0.4</td>
<td>How to generate multiple targets from single source?</td>
<td>55</td>
</tr>
<tr>
<td>6.0.5</td>
<td>What macros are available in the templates?</td>
<td>56</td>
</tr>
<tr>
<td>6.0.6</td>
<td>Where shall I put utility classes?</td>
<td>56</td>
</tr>
<tr>
<td>6.0.7</td>
<td>Where shall I put runtime classes?</td>
<td>56</td>
</tr>
<tr>
<td>6.0.8</td>
<td>How do I generate unique names?</td>
<td>56</td>
</tr>
<tr>
<td>6.0.9</td>
<td>How to generate enumeration datatypes?</td>
<td>56</td>
</tr>
<tr>
<td>6.0.10</td>
<td>What is the difference between root mapping rules and reduction rules?</td>
<td>57</td>
</tr>
<tr>
<td>6.0.11</td>
<td>What happens to untransformed roots?</td>
<td>58</td>
</tr>
<tr>
<td>6.0.12</td>
<td>How to generate multiple nodes for a single node?</td>
<td>58</td>
</tr>
<tr>
<td>6.0.13</td>
<td>Why is editing the macros so weird?</td>
<td>58</td>
</tr>
<tr>
<td>6.0.14</td>
<td>How can I generate root nodes for non-root nodes?</td>
<td>58</td>
</tr>
<tr>
<td>6.0.15</td>
<td>How to remove no-longer needed roots from the model?</td>
<td>58</td>
</tr>
<tr>
<td>6.0.16</td>
<td>How do I generate nodes for nodes in accessory models?</td>
<td>58</td>
</tr>
<tr>
<td>6.0.17</td>
<td>How to use mapping labels?</td>
<td>58</td>
</tr>
<tr>
<td>6.0.18</td>
<td>How to handle references during generation?</td>
<td>59</td>
</tr>
<tr>
<td>6.0.19</td>
<td>Can I debug the generation process?</td>
<td>59</td>
</tr>
<tr>
<td>6.0.20</td>
<td>How do I figure out the actual generation steps?</td>
<td>61</td>
</tr>
<tr>
<td>6.0.21</td>
<td>Can I specify the order in which generators should be run?</td>
<td>61</td>
</tr>
<tr>
<td>6.0.22</td>
<td>How to check that the model has no errors before starting the generator?</td>
<td>62</td>
</tr>
<tr>
<td>6.0.23</td>
<td>How to extend an existing generator?</td>
<td>62</td>
</tr>
<tr>
<td>6.0.24</td>
<td>How to build an extensible generator?</td>
<td>62</td>
</tr>
<tr>
<td>6.0.25</td>
<td>What shall I use TextGen for?</td>
<td>63</td>
</tr>
<tr>
<td>6.0.26</td>
<td>How to generate multiple files per root node?</td>
<td>63</td>
</tr>
<tr>
<td>7</td>
<td>Cross-model generation cookbook</td>
<td>64</td>
</tr>
<tr>
<td>8</td>
<td>Cookbook - Type System</td>
<td>65</td>
</tr>
<tr>
<td>8.1</td>
<td>Inference rules</td>
<td>65</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Equality</td>
<td>65</td>
</tr>
<tr>
<td>8.1.2</td>
<td>Inequality</td>
<td>65</td>
</tr>
<tr>
<td>8.2</td>
<td>Replacement rules</td>
<td>66</td>
</tr>
<tr>
<td>8.3</td>
<td>Subtyping rules</td>
<td>67</td>
</tr>
<tr>
<td>8.4</td>
<td>Comparison rules</td>
<td>67</td>
</tr>
<tr>
<td>8.5</td>
<td>Substitute Type rules</td>
<td>68</td>
</tr>
<tr>
<td>8.6</td>
<td>Checking and Quick-fixes</td>
<td>68</td>
</tr>
<tr>
<td>8.7</td>
<td>When-concrete, overloaded operations</td>
<td>69</td>
</tr>
<tr>
<td>9</td>
<td>HowTo – Integration with the Data Flow Engine</td>
<td>71</td>
</tr>
<tr>
<td>9.1</td>
<td>Dataflow</td>
<td>71</td>
</tr>
<tr>
<td>9.2</td>
<td>Building a Data Flow Graph</td>
<td>71</td>
</tr>
<tr>
<td>9.2.1</td>
<td>Simple, Linear Dataflow</td>
<td>71</td>
</tr>
<tr>
<td>9.2.2</td>
<td>Branching</td>
<td>73</td>
</tr>
<tr>
<td>9.3</td>
<td>Integrating Data Flow Checks into your Language</td>
<td>75</td>
</tr>
<tr>
<td>9.4</td>
<td>Building your own Analyzers</td>
<td>76</td>
</tr>
<tr>
<td>10</td>
<td>Dataflow</td>
<td>77</td>
</tr>
<tr>
<td>10.0.1</td>
<td>Reading a value</td>
<td>77</td>
</tr>
<tr>
<td>10.0.2</td>
<td>Writing a value</td>
<td>77</td>
</tr>
<tr>
<td>10.0.3</td>
<td>Code for</td>
<td>77</td>
</tr>
<tr>
<td>10.0.4</td>
<td>Jump</td>
<td>77</td>
</tr>
<tr>
<td>10.0.5</td>
<td>Ifjump</td>
<td>78</td>
</tr>
<tr>
<td>10.0.6</td>
<td>Inserting instructions</td>
<td>78</td>
</tr>
<tr>
<td>10.1</td>
<td>The data flow API</td>
<td>79</td>
</tr>
<tr>
<td>11</td>
<td>Custom language aspect cookbook</td>
<td>80</td>
</tr>
<tr>
<td>11.1</td>
<td>Development cycle of custom aspects</td>
<td>80</td>
</tr>
<tr>
<td>11.2</td>
<td>Look around the sample project</td>
<td>80</td>
</tr>
<tr>
<td>11.3</td>
<td>Language runtime</td>
<td>81</td>
</tr>
<tr>
<td>11.3.1</td>
<td>Using the language aspects</td>
<td>82</td>
</tr>
<tr>
<td>11.4</td>
<td>Implementing custom aspect</td>
<td>82</td>
</tr>
</tbody>
</table>
## 12 Icon description
- **12.1 First impression**
- **12.2 Creating icon prototypes**
- **12.3 The language explained**
  - **12.3.1 Extending the language**
  - **12.3.2 icon{} vs iconResource{}**

## 13 Progress indicators
- **13.1 Asynchronous tasks**
- **13.2 Monitoring progress**
- **13.3 Running in the background**
- **13.4 Proper locking when accessing resources**
- **13.5 Undoable actions**

## 14 Removing bootstrapping dependency problems
- **14.1 Definition:**
- **14.2 Why is this a problem**
- **14.3 The detection and elimination toolchain in MPS**
- **14.4 Fixing the problem**
  - **14.4.1 Analyzing the bootstrapping dependencies**
  - **14.5 Typical cases of bootstrapping dependencies**
  - **14.6 What else needs to be done**
  - **14.7 A quick way to suppress the problem**

## 15 HowTo – Adding additional Tools (aka Views)
- **15.1 Adding additional Tools (aka Views)**
- **15.2 The Outline Tool itself**
- **15.3 An Action to Open the Tool**
- **15.4 An Action Group**
- **15.5 Managing the Tool Lifecycle**
  - **15.5.1 Editor Activation and Focus**
  - **15.5.2 Tracking the Model Lifecycle**
- **15.6 Synchronizing Node Selections**
  - **15.6.1 Tracking Editor Selection**
  - **15.6.2 Tracking Changes in the Model Structure**
  - **15.6.3 The way back: Tracking Tree Selection**
- **15.7 Swing’s Artifacts: Tree Model and Renderer**
  - **15.7.1 Tree Cell Renderer**
  - **15.7.2 Tree Model**

## 16 Debugger API
- **16.1 Debugger**
  - **16.1.1 Where to start**
  - **16.1.2 Key classes**
  - **16.1.3 Final remarks**

## II Command line tools

## 17 HowTo – MPS and Git

## 18 HTTP support plugin
- **18.0.4 Node URLs**
- **18.0.5 YouTrack and TeamCity Integration**
- **18.0.6 Built-in server extensions**

## 19 HowTo – MPS and ant
- **19.1 Working with MPS and ant**
- **19.2 Building the Languages in a Project**
- **19.3 Generating/Building Solutions**
- **19.4 Running Tests**
# Table of Contents

## III Obsolete documents

### 20 Build languages (obsolete)

- 20.1 Build Languages .................................................. 123
- 20.2 Table of contents .................................................. 123

### 21 Packaging Language (obsolete)

- 21.1 Packaging Language ............................................... 124
  - 21.1.1 Introduction .................................................... 124
  - 21.1.2 Language Structure .......................................... 124
  - 21.1.3 Using Packaging Language .................................... 128

### 22 Build Language (obsolete)

### 23 Generating MPS models from Ant (obsolete)

- 23.1 Generating MPS models from Ant ............................... 135
  - 23.1.1 Introduction .................................................... 135
  - 23.1.2 Parameters ..................................................... 135
  - 23.1.3 Nested .......................................................... 135
  - 23.1.4 Configuration and caches directories ....................... 136
  - 23.1.5 JVM options ................................................... 136
  - 23.1.6 Examples ....................................................... 136
- 23.2 Testing your models on TeamCity ............................... 137
  - 23.2.1 Generation Tests ............................................. 137
  - 23.2.2 Difference between mps.test.generation and mps.generate ............................................. 137
  - 23.2.3 Broken References Tests .................................... 137

### 24 Custom MPS Language (obsolete)

- 24.1 Custom MPS Language ............................................. 138
  - 24.1.1 Introduction .................................................... 138
  - 24.1.2 Building custom MPS for your project ....................... 138
Part I

Cookbooks
Chapter 1

Getting the dependencies right

1.1 Motivation

Modules and models are typically interconnected by a network of dependencies of various types. Assuming you have understood the basic principles and categorisations of modules and models, as described at the MPS project structure page, we can now dive deeper as learn all the details.

Getting dependencies right in MPS is a frequent cause of frustration among inexperienced users as well as seasoned veterans. This page aims to solve the problem once and for all. You should be able to find all the relevant information categorised into sections by the various module and dependency types.

All programming languages have a notion of "imports". In Java you get packages and the "import" statement. In Ruby or Python you have "modules" and "require" or "import" statements. In MPS we provide a similar mechanism for packaging code and expressing dependencies in a way that works universally across languages - code is packages into models and these models can express mutual dependencies.

In addition, since MPS is a multi-language development environment, models can specify the languages (aka syntaxes) enabled in them. This is different from when writing code in Java, Ruby or other languages, where the language to be used is given and fixed.

1.2 Useful keyboard shortcuts

Whenever positioned on a model or a node in the left-hand-side Project Tool Window or when editing in the editor, you can invoke quick actions with the keyboard that will add dependencies or used languages into the current model as well as its containing solution.

- Control + L - Add a used language
- Control + M - Add a dependency
- Control/Cmd + R - Add a dependency that contains a root concept of a given name
- Control/Cmd + Shift + A - brings up a generic action-selection dialog, in which you can select the desired action applicable in the current context

1.3 Solution

Solutions represent programs written in one or more languages. They typically serve two purposes:

1. Sandbox solutions - these solutions hold an end user code. The IDE does not treat the code in any special way.
2. Runtime solutions - these solutions contain code that other modules (Solutions, Languages or Generators) depend on. The code can consist of MPS models as well as of Java classes, sources or jar files. The IDE will reload the classes, whenever they get compiled or changed externally.
3. Plugin solutions - these solutions extend the IDE functionality in some way. They can contribute new menu entries, add side tool panel windows, define custom preference screens fro the Project settings dialog, etc. Again, MPS will keep reloading the classes, whenever they change. Additionally the IDE functionality will be updated accordingly.

We'll start with the properties valid for all solutions and then cover the specifics of runtime and plugin solutions.
1.3.1 Common

Properties

- **Name** - name of the solution
- **File path** - path to the module file
- **Generator output path** - points to the folder, where generated sources should be placed
- **Left-side panel** - contains model roots, each of which may hold one or more models.
- **Right-side panel** - displays the directory structure under the model root currently selected in the left-side panel. Folders and jar files can be selected and marked/unmarked as being models of the current model root.

Model root types

Solutions contain model roots, which in turn contain models. Each model root typically points to a folder and the contained models lie in one or more sub-folders of that folder. Depending on the type of contained models, the model roots are of different kinds:

- **default** - the standard MPS model root type holding MPS models
- **java_classes** - a set of directories or jar files containing Java class files
- **javasource_stubs** - a set of directories or jar files containing Java sources

When included in the project as models, Java classes in directories or jar files will become first-class citizens of the MPS model pool and will become available for direct references from other models, which import these stub models. A second option to include classes and jars in MPS is to use the Java tab and define them as libraries. In that case the classes will be loaded, but not directly referenceable from MPS code. This is useful for libraries that are needed by the stub models.

1.3.2 Dependencies

The dependencies of a solution are other solutions and languages, the models of which will be visible from within this solution.

The **Export** flag then specifies whether the dependency should be transitively added as a dependency to all modules that depend on the current solution. For example, if module A depends on B with export on and C depends on A, then C depends on B.
1.3.3 Used Languages
The languages as well as devkits that the solution’s models may use are listed among used languages. Used languages are specified on the model level and the Used Languages tab on modules only shows a collection of used languages of its all models.

1.3.4 Java
This is where the different kinds of Solutions differ mostly.

The Java tab contains several options:

- **Solution kind** - different kinds of solutions are treated slightly differently by MPS and have access to different MPS internals
  - **None** - default, used for user code, which does not need any special class-loading strategy - use for Sandbox solutions
  - **Other** - used by typical libraries of reusable code that are being leveraged by other languages and solutions - use for Runtime solutions
  - **Core plugin** - used by code that ties into the MPS IDE core and needs to have its class-loading managed accordingly - use for Plugin solutions
  - **Editor plugin** used by code that ties into the MPS editor and needs to have its class-loading managed in sync with the rest of the editor - use for Plugin solutions that only enhance the editor

- **Compile in MPS** - indicates, whether the generated artifacts should be compiled with the Java compiler directly in MPS and part of the generation process

- **Source Paths** - Java sources that should be made available to other Java code in the project

- **Libraries** - Java classes and jars that are required at run-time by the Java code in one or more models of the solution

1.3.5 Facets

- **Idea Plugin** - checked, if the solution hooks into the IDE functionality

- **Java** - checked, if the solution relies on Java on some way. Keep this checked in most cases.

- **tests** - checked, if the solution contains test models

1.4 Solution models

Solutions contain one or more models. Models can be mutually nested and form hierarchies, just like, for example, Java packages can. The properties dialog hides a few configuration options that can be tweaked:
1.4.1 Dependencies
Models from the current or imported modules can be listed here, so that their elements become accessible in code of this model.

1.4.2 Used languages
The languages used by this model must be listed here.

1.4.3 Advanced
A few extra options are listed on the Advanced tab:

- **Do not generate** - exclude this model from code generation, perhaps because it cannot be meaningfully generated
- **File path** - location of the model file
- **Languages engaged on generation** - lists languages needed for proper generation of the model, if the languages are not directly or indirectly associated with any of the used languages and thus the generator fails finding these languages automatically

1.4.4 Virtual packages
Nodes in models can be logically organised into hierarchies of virtual packages. Use the Set Virtual Package option from the node’s context pop-up menu and specify a name, possibly separating nested virtual folder names with the dot symbol.

1.5 Adding external Java classes and jars to a project - runtime solutions
Runtime solutions represent libraries of reusable code in MPS. They may contain models holding MPS code as well as models referring to external Java sources, classes or jar files. To properly include external Java code in a project, you need to follow a few steps:

1. Create a new Solution
2. In the Solution properties dialog (Alt + Enter) specify the Java code, such that:
   
   (a) **Common tab** - click on Add Model Root, select javaclasses for classes or jars, select javasource_stubs for Java sources and navigate to your lib folder.
   
   (b) Select the folder(s) or jar(s) listed in the right-side panel of the properties dialog and click on the blue "Models" button.
   
   (c) Also on the **Java tab** add all the jars or the classes root folders to the Libraries part of the window, otherwise the solution using the library classes would not be able to compile. When using java_sourcestubs, add the sources into the Source paths part of the Java tab window, instead.

3. A new folder named stubs should appear in your solution
4. Now after you import the solution into another module (solution, language, generator) the classes will become available in that module’s models
5. The languages that want to use the runtime solution will need to refer to it in the **Runtime Solutions** section of the Runtime tab of their module properties

1.6 Language
Languages represent a language definition and consist of several models, each of which represent a distinct aspect of the language. Languages also contain a single Generator module. The properties dialog for languages is in many ways similar
CHAPTER 1. GETTING THE DEPENDENCIES RIGHT

to the one of Solutions. Below we will only mention the differences:

1.6.1 Common
A language typically has a single model root that points to a directory, in which all the models for the distinct aspects are located.

1.6.2 Dependencies
The dependencies of a language are other solutions and languages, the models of which will be visible from within this solution. The Export flag then specifies whether the dependency should be transitively added as a dependency to all modules that depend on the current language.

A dependency on a language offers thee Scope options:

- Default - only makes the models of the other language/solution available for references
- Extends - allows the language to define concepts extending concepts from the there language
- Generation Target - specifies that the current language is generated into the other language, thus placing a generator ordering constraint that the other language must only be generated after the current one has finished generating

1.6.3 Used Languages
This is the same as for solutions.

1.6.4 Runtime

- Runtime Solutions - lists solutions of reusable code that the language requires. See the "Adding external Java classes and jars to a project - runtime solutions" section above for details on how to create such a solution.
- Accessory models - lists accessory models that the language needs. Nodes contained in these accessory models are implicitly available on the Java classpath and the Dependencies of any model using this language.

1.6.5 Java
This is the same as for solutions, except for the two missing options that are not applicable to languages.
1.6.6 Facets

This is the same as for solutions.

When using a runtime solution in a language, you need to set both the dependency in the Dependencies tab and the Runtime Solutions on the Runtime tab.

1.7 Language models/aspects

1.7.1 Dependencies / Used Languages / Advanced

These settings are the same and have the same meaning as the settings on any other models, as described in the Solution section.

1.8 Generator

The generator module settings are very similar to those of other module types:

1.8.1 Common

This is the same as for languages.

1.8.2 Dependencies

This is the same as for solutions. Additionally generator modules may depend on other generator modules and specify Scope:

- Default - only makes the models of the other language/solution available for references
- Extends - the current generator will be able to extend the generator elements of the extended generator
- Design - the target generator is only needed to be referred from a priority rule of this generator

1.8.3 Used Languages

This is the same as for languages.
1.8.4 Generators priorities

This tab allows to define priority rules for generators, in order to properly order the generators in the generation process. Additionally, three options are configurable through the check-boxes at the bottom of the dialog:

- **Generate Templates** - indicates, whether the generator templates should be generated and compiled into Java, or whether they should be instead interpreted by the generator during generation.

- **Reflective queries** - indicates, whether the generated queries will be invoked through Java reflection or not. (Check out the Generator documentation for details)

- **IOperationContext parameter** - indicates, whether the generator makes use of the operationContext parameter passed into the queries. The parameter will be removed in the future and generators should gradually stop using it.

1.8.5 Java

This is the same as for languages.

1.8.6 Facets

This is the same as for languages.

1.8.7 Generator models

This is the same as for solutions.
Chapter 2

Building an interpreter cookbook

Check out the Shapes sample project, which is bundled with MPS since version 3.1. It can do some fancy tricks in the editor.

The sample projects are automatically installed by MPS upon its first run and they should be located in $USER_HOME/MPSSamples.

By default, the editor shows plain code that consists of commands to draw visual shapes of given sizes and colors on a canvas.

The code can be generated into Java and run. This will start a new Java process running the application that just has been generated from the code above:
MPS can, however, also interpret the code without generating Java. Just hit Alt + Enter anywhere in the code to invoke the Intentions pop-up menu and choose "Preview Scene". You’ll get a new frame containing your scene, which is interpreting the code:
CHAPTER 2. BUILDING AN INTERPRETER COOKBOOK

2.1 Instant preview

With the ability to interpret code the editor can now serve the results of the interpreter back to the developer through the editor. It can either interpret individual shapes and draw them in the editor next to the code that defines each of them:

Or the whole scene can be drawn next to the code and react instantaneously to the changes made to the code.
To try these capabilities yourself, right-click the editor, pick "Push Editor Hints" and select the hints that will indicate to the editor, which of the preview capabilities you want to enable:
This was a quick teaser, now we can have a look at how to build such simple interpreting capabilities into your language.

### 2.2 Defining an Interpreter

MPS is primarily focused on code generation. Programs represented as ASTs get translated into code in a target language, which then may get compiled and run. There are, however, situations when interpreting your code directly may be a better option. You could give the developers some sort of Preview of what the code does and thus providing instant feedback to their code changes. You could detect and report errors that only show at run-time or you could just want to speed up code-run-fix cycle turnaround by bypassing the generation and compilation phases.

MPS does not come with any infrastructure for building interpreters at the moment, but the Behavior aspect of your language gives you some power to build interpreters on your own. Ideally, just like with code generators, you would create a runtime solution holding the interpreter runtime classes. The Behavior aspect of your language concepts would then cooperate with the runtime classed to navigate the AST properly and interpret all nodes in the mode.

We’ll have a look at how the Shapes language enabled interpreting its code.

#### 2.2.1 A whole-scene Interpreter

Let’s start with the scenario when the user invokes the interpreter explicitly by an Intention. First, the intention needs to be created:
The intention invokes the `interpret()` method on the `Canvas` node, which is defined in the `Behavior` aspect of `Canvas`.

```java
public void interpret() {
    JFrame frame = new JFrame("Preview: " + this.name);
    final JPanel panel = PreviewFactory.createPanel(this);
    frame.add(panel);
    frame.setVisible(true);
    frame.pack();
}
```

The method builds a new `Frame` that will hold a panel with a customized `paintComponent()` method. The panel is being reused by other preview functionality, so it has been extracted into a separate helper class `PreviewFactory`. For other than trivial cases the class should be placed into a runtime solution, but here we’ve just put it directly into the `Behavior` aspect of the language.

The current model is traversed with the code: `thisCanvas.shapes.forEach((it -> it.drawShape(graphics)))`. This is where the interpreting happens. The `drawShape()` method is implemented by each `Shape` subconcept so that they can get rendered on the screen. Notice that the traversal code is wrapped inside a `ReadAction` to be guaranteed a read permission to the model.

The `drawShape()` method has to be implemented by all `Shapes`:
This is enough to get our little interpreter off the ground.

### 2.2.2 Shape preview

To preview individual shapes next to their definition in code, we need to change the editor or define a new one with a different `editor hint`, so that it holds a `swing component` containing the shape’s visualization. In the sample we chose to leverage multiple projections (see the Multiple Projections video to see how multiple projections work in MPS) and thus we created a new editor with the `ShapePreview` hint specified in the header. Only when the user enables the `ShapePreview` hint, this editor will be used instead of the default text-only one.
The **swing component** defines a **JPanel**, which in the `paintComponent()` method obtains a read lock on the model and then draws the current **Shape** node at a specific location. Notice that a new `drawShapeAt()` method has been added to **Shapes** to draw the **Shapes** ignoring their defined position within the canvas.

### 2.2.3 Scene preview

To preview a whole scene, we’ll need to create a new editor for the **Canvas** concept and hook it to the **ScenePreview** editor hint.

```java
concept behavior Circle {

    constructor {
        <no statements>
    }

    public void drawShape(Graphics graphics) {
        overrides Shape.drawShape {
            super.drawShape(graphics);
            graphics.drawOval(this.x, this.y, this.radius, this.radius);
        }
    }

    public void drawShapeAt(Graphics graphics, int x, int y) {
        overrides Shape.drawShapeAt {
            super.drawShapeAt(graphics, x, y);
            graphics.drawOval(x, y, this.radius, this.radius);
        }
    }
}
```
The swing component cell contains a customized JPanel, which is again created by the PreviewFactory class just like when we were interpreting on explicit user request earlier on.
Now, this is it. I hope you liked it.
Chapter 3

Editor cookbook

This document, intended for advanced language designers, should give you the answers to the most common questions related to the MPS editor. You may also like to read the Editor, which contains exhaustive information on the subject.

The RobotKaja sample project, which comes bundled with MPS, can serve you as a good example of how to build a fluent text-like projectional editor.

You may also like to check out a couple of screen-casts that focus on defining smooth editors:

- Smoothing the Editor Experience
- Optional visibility in the editor
- Using Editor Components
- Editor Transform Actions

3.0.4 How to properly indent a block of code

Nested blocks of code are typically represented using indented collections of vertically organized statements. In MPS you almost exclusively use indented layout for such collections, if you aim at making the experience as text-like as possible. Vertical and horizontal layouts should be only considered for positional or more graphical-like layouts.

So for properly indented code blocks you need to create an Indented collection ([- ... -]) and set:

- indent-layout-new-line-children so that each collection element (child) is placed on a new line
- indent-layout-indent to make the whole collection indented
- indent-layout-new-line to place a new line mark behind the collection, so that next nodes get placed on a new line underneath the collection
- indent-layout-on-new-line Optionally you may need this flag to place the whole collection on a new line instead of just appending it to the previous cell

3.0.5 Alias

Concepts can have an alias defined, which will represent them in code-completion pop-up menu and which will allow users to insert instances of these concept just by typing the alias. The editor can then refer to the text of the alias using the
AliasEditorComponent editor component. This is in particular useful, when multiple concrete concepts inherit an editor from an abstract parent and the alias is used to visually differentiate between the multiple concepts.

```java
concept And extends LogicalOperator
    implements <none>
    instance can be root: false

    alias: and
    short description: <no short description>

concept Or extends LogicalOperator
    implements <none>
    instance can be root: false

    alias: or
    short description: <no short description>

editor for concept LogicalOperator

node cell layout:
[- { % left % "# AliasEditorComponent | # % right % } -]

inspected cell layout:
<choose cell model>
```

You then refer to the alias value through the `conceptAlias` property:

```java
matching text
{ parameterObject, currentNodeNode, parentNode, parameterObject.conceptAlias; }
```

### 3.0.6 Unique constraint

In general, constraints allow you to restrict allowed nodes in roles and values for properties and report violations to the user.

Frequently you’d like to apply constraints to limit allowed values in your code. E.g. the names of method definitions should be unique, each library should be imported only once, etc. MPS gives you two options:

1. **Use constraints** - these define hard validity rules for abstract syntax (structure) of your language. Use these if you want to disallow certain nodes or properties to be ever made part of the AST.

2. **Use NonTypesystemRules** - these provide additional level of model verification. You are still allowed to enter incorrect values in the editor, but you will get an error message and a red underline that notifies you about the rule violation. Unlike with constraints you can also specify a custom error message with additional details to give the developer helpful hints.

```java
concepts constraints RoutineDefinition {
    can be child
    { node, parentNode, link, childConcept, scope, operationContext } -> boolean {
        { parentNode.isInstanceOf(CommandList) & parentNode.parent.isInstanceOf(Script) } ||
        parentNode.isInstanceOf(Library);
    };

    can be parent <none>

    can be ancestor <none>

    routine jump means
    end

    routine jump means
    end

    routine jump means
    end
```

```java
routine jump means
end
```

```java
routine jump means
end
```

```java
routine jump means
end
```

```java
routine jump means
end
```
### 3.0.7 An indented vertical collection

Indent layout is the preferred choice instead of vertical/horizontal ones where applicable. To create an indented vertical collection you can create a wrapping indent collection (marked with [- -]) and set

1. `indent-layout-indent: true`
2. `selectable: false`. (false means the collection will be transparent when expanding code selection).

Then, on the child collection use `indent-layout-new-line-children: true`.

### 3.0.8 Optional visibility

Elements, that should only be visible under certain condition, should have its `show if` property set:

### 3.0.9 Defining and reusing styles

Each cell can have its visual style defined in the Inspector. In addition to defining style properties individually, styles can be pre-defined and then reused by multiple languages.

You can define your own styles and make them part of your editor module so that they can be used in your editors as well as in all the editors that import your editor module.
3.0.10 Reusing the BaseLanguage styles

BaseLanguage comes with a rich collection of pre-defined styles. All you need to do in order to be able to use the styles defined in another language is to import the language into the editor of your language.

3.0.11 Making proper keywords

Keywords should have the `keyword` style applied so that they stand out in text. Also, making a keyword `editable` will make sure users can freely type inside or next to the keyword and have transforms applied. With `editable` set to `false` MPS will interfere with user’s input and ignore characters, which don’t match any applicable transformation.
3.0.12 Adjust abstract syntax for easy editing

Making all concepts descent from the same (abstract) concept allows you to mix and match instances of these concepts in any order inside their container and so give the users a very text-like experience. Additionally, if you include concepts for empty lines and (line) comments, you will give your users the freedom to place comments and empty lines anywhere they feel fit.

In our example, the SStructureContainer does not impose any order, in which the members should be placed. The allowed elements all descend from SStructurePart and so are all allowed children.

3.0.13 Specify, which concept should be the default for lists

When you hit Enter in a text editor, you get a new line. Identically, you may want to an empty line concept or some other reasonable default concept of your language to be added to the list at the current cursor position, when the user presses Enter.

The behavior in the second image below should be preferred to the one visible on the first one:

The red placeholder is caused by either a missing node in the position or an instance of an abstract concept. Since abstract concepts should not be instantiated, the editor always shows them in red.

When the user hits Enter on a collection cell a new node is created and inserted into the collection. The concept of the newly created and inserted node is the same as the declared concept of the collection. If the concept is abstract, the newly created and inserted node will be in instance of that abstract concept and thus rendered in red.

There are three ways to prevent such situations:

1. Use a non-abstract concept to declare the concept of the child collection. BaseLanguage uses this approach for Statement, which is a common super-concept to all statements and is not abstract in order to represent empty lines.

2. Specify a default concrete concept in the Constraints of the abstract concept. This will use the specified concrete concept whenever the abstract one is to be instantiated. Such a concrete concept will then be used an a concept for empty lines.

3. Use element factory on all collections, in which a different than the abstract concept should be used for creating newly inserted nodes (described below).

Specifying default concept is as easy as setting the element factory property of the collection editor:
3.0.14 Make empty lines helpful to further editing

Making empty lines to appear as default after hitting Enter is the first important step to mimic the behavior of text-editors. The next step that will get you even closer is to make empty lines react reasonably to user input and provide handy code completion.

You should make your empty lines similar to the one above - add an empty constant cell, make it editable and specify the super-type of all items to populate the code-completion menu with. If you followed the earlier advice and have all your concepts, which could take the position of the empty line, descend from a common ancestor, this is the type to benefit. Specify that ancestor as the concept to potentially replace empty lines with.
3.0.15 Hiding concepts from the completion menu

The completion menu can be customized using the Transformation Menu Language. By default it shows all non-abstract concepts that can be substituted in the current location. Some concepts, such as empty lines or comments, you may want to exclude from the completion menu explicitly. To do so, you need to define a Default Substitute Menu for the concerned concept or its super-concept in the editor aspect of your language and leave it empty. MPS uses that menu to populate the completion menu with items whenever your concept is considered for substitution. Leaving the menu empty will result in your concept never getting inserted into the completion menu.

3.0.16 Easy node replacement

Similarly, you will frequently want to allow developers to replace one node with another in-place, just by typing the name of the new element:

```java
concept <no name> extends BaseConcept {
}

inconcept <no name> extends BaseConcept {
}

interface concept <no name> {
  @interface concept (AbstractConcept in j.m.core.structure)

  interface concept <no name> {
    ...
  }
}
```

Just like with empty lines, for this to work, you set the replacement concept to the common ancestor of all the applicable candidate concepts. These will then appear in the code-completion menu.

3.0.17 Vertical space separators

To create some vertical space in order to separate elements, constant cells are very handy to utilize. Give them empty contents, set noAttraction for focus to make it transparent and put it on a separate line with indent-layout-new-line.
3.0.18 Handling empty values in constants

A value of a property can may either hold a value or be empty. MPS gives you three knobs to tune the editor to react properly to empty values. Depending on the values of the `allow-empty`, `text*` and `empty text*` flags, the editor cell may or may not turn red or display a custom message when the property is empty.

- **Empty values not allowed.**
  - The cell is displayed in red to indicate an error.

- **Empty values not allowed.**
  - A custom message has been provided to empty cells.

- **Empty values are allowed.**
  - An empty cell displays a default message in gray color.

- **Empty values are allowed.**
  - A custom message has been provided to empty cells.

- **Empty values are allowed.**
  - The empty cell is visually transparent.
Chapter 3. Editor Cookbook

3.0.19 Horizontal list separator

The `separator` property on collection cells allows you to pick a character that will

1. Visually separate elements of the list
2. Allow the user to append or insert new elements into the list

Although separators can be any string values, it is more convenient to keep them at one character length.

3.0.20 Matching braces and parentheses

Use the `matching-label` property to pair braces and parentheses. This gives the users the ability to quickly visualize the wrapped block of code and its boundaries:

Since MPS 3.4 you can also use `show-boundaries-in` style on a collection cell to specify how its boundaries are to be shown. The style has two values, `gutter` and `gutter-and-editor`. If you set the style of a collection to `gutter-and-editor`, the collection’s first and last cell will be highlighted in the editor when one of the cells is selected and a bracket will be shown in the left-hand editor gutter. Setting the style to `gutter` will only show the bracket and may be useful if you want to show where a particular collection begins and ends but there isn’t a well-defined last cell in the collection.

Example:
The result:
3.0.21 Empty blocks should look empty

By default an empty block always takes one line of the vertical real-estate. It also contains a default empty cell, which gives the developer a hint that there’s a list she can add elements to.

```plaintext
concept InterfaceClassifier extends BaseConcept {
    << ... >>
}
```

You may hide the « ... » characters by setting the empty cell value to be an empty constant cell, which gives you a slightly more text-like look:

```plaintext
concept InterfaceClassifier extends BaseConcept {
}
```

One additional trick will hide the empty line altogether:

```plaintext
concept InterfaceClassifier extends BaseConcept {
    }
```

What you need to do is to conditionally alter the indent-layout-new-line and the punctuation-right properties of the opening brace to add/remove a new line after the brace and assign control of the caret position right after the brace to the following cell. Since the empty constant cell for the members collection follows and is editable, it will receive all keyboard input at the position right after the brace. This will allow the developer to type without starting a new empty line first.
3.0.22 Make empty constants editable

It is advisable to represent empty cells for collections with empty constant values that have the editable property set to true. This way users will be able to start typing without first creating a new collection element (via Enter or the separator key).

3.1 Common editor patterns

3.1.1 Prepending flag keywords

Marker keywords prepending the actual concept, such as final, abstract or public in Java, are quite common in programming languages.

```java
private abstract class MyCalculator {
    ...
}
```

Developers have certain expectations of how these can be added, modified or deleted from code and projectional editors should follow some rules to achieve pleasant intuitiveness and convenience levels.

- Typing part of the keyword anywhere to the left of the main concept name should insert the keyword
- Hitting delete while positioned on the keyword should remove it
- The keyword should only be visible when the associated flag is true - for example, the final keyword is only shown for final classes in Java

Notice that the keywords are optional, with the show if condition querying the underlying abstract model.
The action map property refers to an action map, which specifies that when the DELETE action is invoked (perhaps by pressing the delete key), the underlying abstract model should be updated, which will in turn make the flag keyword disappear from the screen.

Notice that the approveDelete call gives the user a chance to revoke her decision to delete the flag and also how positioning the cursor after deleting the flag is handled.

To allow the developers to add the keyword just by typing it, we need to define a left transform action, which, in our example, when applied to a non-final element will make the element final after typing "final" or any unique prefix of it. The Transformation Menu language gives us Transformation Menus, which we can utilize here:
A Named transformation menu can then be attached to the editor cells that precede or succeed our flag cell:

```plaintext
<default> editor for concept ClassConcept
node cell layout:

[-] # DeprecatedPart #
[-] ^ static ? ^ abstract ? ^ final ^ class F( name )
/- GenericDeclaration_TypeVariables Component #
| |- | extends | ^ % superclass % -]
| |[ | implements | ^ % implementedInterface % /empty cell: <constant> -] [-]
|
```

### 3.1.2 Appending parametrized keywords

Supporting keywords similar to Java’s `implements` is also very straightforward in MPS.

First, the whole `implements A, B C` part must be optionally visible.
Only when the list of implemented interfaces is non-empty, the collection including the `implements` keyword is displayed. Second, we need a right transformation to add a new child into the `implements` collection, when `implements` or a part of it is typed right after the class name or after the reference to the `extended` class. Similarly, a left transformation is needed to add a new child when `implements` is typed right before the code block’s left brace. Again, the Transformation Menu language will let us create named transformation menus and attach them to the cells.

### 3.1.3 Node substitution actions

Node substitution actions specify how certain nodes can be replaced with others. For example, you may want to change `logical and` to `or` and vice versa, yet preserve the boolean conditions specified in the child nodes:

```plaintext
if (wall ahead and heading south) do
  and
  Or
end
```

Let’s use Default substitute menu from Transformation Menu language:

```plaintext
Default substitute menu for concept LogicalOperator
parameterized
class LogicalOperator

parameter type: concept<LogicalOperator>
parameter values:
  (parentNode, currentTargetNode, link, editorContext, model)->sequence<concept<LogicalOperator>> {
    concept<LogicalOperator>/sub-concepts(model).where({it !it.isAbstract(); }); }

substitute action(output concept: default) =:
  create node
  (parameterObject, parentNode, currentTargetNode, link, editorContext, model, pattern)->node<LogicalOperator>
  newInitializedInstance =
  parameterObject.new initialized instance(currentTargetNode);
  newInitializedInstance;

  matching text
  (parameterObject, parentNode, currentTargetNode, link, editorContext, model, pattern)->string {
    parameterObject.getConceptAlias();
  }

  description text
  (parameterObject, parentNode, currentTargetNode, link, editorContext, model, pattern)->string {
    parameterObject.getName();
  }
```

Since both `And` and `Or` concepts inherit from `LogicalOperator`, we can refer to `LogicalOperator` in the action. In essence, the action above allows replacing any `LogicalOperator` with any non-abstract subconcept of `LogicalOperator`. The replacing concept is instantiated and its left and right children populated from the children of the node that is being replaced.

### 3.1.4 Substitution for custom string values

Substitution rules can also be used to convert plain strings into nodes of a desired concept. Imagine, for example, a kind of variable declaration that starts with the name of the variable, like in Python:

```plaintext
myVariable = 10;
```

To enter such variable declaration, you can always pick the variable concept (by its alias) from the completion menu and then fill in the name:
You can, however, add a simple substitution action into the substitution menu that will enable users to simply type the desired name of the variable on the empty line and have the variable created for you automatically:

The menu may look something like this:

1. It can be substituted whenever a non-empty pattern has been typed and it does not match an alias of any of the known sub-concepts applicable in the position
2. A new node is created when the action is run and the pattern is copied into the name of the variable
3. The "selection handler" ensures that the cursor stays within the "name" cell after creating the variable so that the user can continue typing the name even after the variable has already been created (it is created as soon as there is no other option in the completion menu (the "selection handler" returns null, since when a non-null node is returned, MPS would set the cursor itself on the returned node and would ignore the "select" command)
You may also experiment with checking the "strictly" flag to further customize the behavior:

```kotlin
Script Maze runs as

step
  Step forward Turn left Define a routine Control flow Other actions Scene builder
  variable called s
  (AbstractCommand in jetbrains.mps.samples.Kaja)
  (AbstractBuilderCommand in jetbrains.mps.samples.KajaSceneConstruction)

destroy wall

Here's the menu:

default substitute menu for concept AbstractCommand

subconcepts menu <no filter>
substitute action(output concept: default)

create node
  (parentNode, currentTargetNode, link, editorContext, model, pattern)->node<AbstractCommand> {
    node<Variable> var = currentTargetNode.replace with new initialized(Variable);
    var.name = pattern;
    return var;
  }

description text (parentNode, currentTargetNode, link, editorContext, model, pattern)->string {
  "variable called " + pattern;
}

matching text (parentNode, currentTargetNode, link, editorContext, model, pattern)->string {
  pattern;
}

selection (parentNode, link, editorContext, model, pattern, createdNode)->void {
  createdNode.select(f: editorContext, cell: FIRST_EDITABLE, selectionStart: -1);
}

can substitute
  (parentNode, currentTargetNode, link, editorContext, model, pattern, strictly)->boolean {
    sequence<concept<AbstractCommand>
      subConcepts = concept/AbstractCommand/.sub-concepts{/model}.where{[-it => it.isAbstract(); !]};
    return pattern.isNotEmpty &
      (subConcepts.all([-it => it.conceptAlias.startsWith(pattern); ])
        || !strictly);
  }

Changes:

1. The **description** provides a customized description message to display in the completion menu alongside the "variable" alias.

2. The **pattern** ensures the so far typed text is suggested as the name of the variable to be created (when the code-completion menu is visible).

3. The **can substitute** handler reacts to the value of the strictly parameter, so that as long as there are other subconcepts available in the menu this action is also available, but only for non-strict matching.

The "strictly" parameter is set to "true" when MPS is detecting whether your action perfectly matches the text typed so far. This gives actions the ability to distinguish between two situations:

1. The "pattern" is a prefix of what the action reacts to and so it wants to stay in the menu.

2. The "pattern" is an exact match of what the action reacts to and so if there are no other options in the completion menu, the action should be triggered instantly.

If your action is the only item in the completion menu and the "can substitute" function of your action, when called with "strictly==true", returns true, the action is performed. By always returning "false" for "strictly==true" you ensure that the action never matches strictly (perfectly) the pattern and so the variable is only created when the user explicitly triggers the action (Control+Space, Enter). In that case the "selection handler" can let the cursor move to the initializer, since the user has already finished typing the name of the variable, anyway.

### 3.1.5 Including parents transform actions

Your nodes can optionally include transform actions applicable to different nodes, e.g. parents. For example, if we allow for appending **logical and** and **or** to logical expressions, we may still get into problems when the logical expression is more complex and, for example, the last cell of its editor belongs to a child node.

```
if heading south
  do
    and
      or
    and
  end
```

In our example, **heading south** is a logical expression, however, **south** itself is a child of logical expression with a concept
3.2 Conceptual questions

3.2.1 Extending an existing editor

The MPS editor assigns visual cells to nodes from the model and so delegates to the node’s concept the responsibility for rendering the corresponding values and accepting user input. This mechanism will work irrespective of the language the concepts has been defined in. So an embedded language such as, e.g. a math formula, will render itself correctly, no matter whether it is part of a Java program or an electrical circuit simulation model, for example.

New sub-concepts will by default re-use the editor of their parent concept, unless a specific editor is available. On the other hand extending languages may supply their own editors for inherited concepts and thus override the concrete syntax derived from the inherited editor.

3.2.2 Design an editor for extension

A good strategy is to use Editor components to modularize the editor. This will allow language extensions to override the components without having to redefine the editor itself.

References to properties, such as conceptAlias, from within the editor should be preferred to hardcoded literals, since the reference will allow the editor to adapt to the subconcepts without having to override the editor.

When specifying actions’ and intentions’ applicability rules, bear in mind that some subconcepts may need to opt out from these actions of their parent. Making these actions check a behavior method in their applicability rules is advisable in such scenarios.

3.2.3 How to add a refactoring to the menu

Use InlineField or IntroduceVariable as good examples. In general, you need to define an Action from the jetbrains.mps.lang.plugin language, which specifies its applicability, collects contextual information and the user input, initializes the actual refactoring procedure and invokes it. The refactoring functionality is typically extracted into a BaseLanguage class and potentially reused by multiple actions.

3.3 How to define multiple editors for the same concept

3.3.1 A sample first

The MultipleProjections sample project bundled with MPS provides good introductory guidelines to learn how to define multiple editors per concepts and how to allow switching between them.
The sample languages allow you to define workflows, which consist of one or more state machines. State machines can be expressed either structurally or as tables. The programmer can switch between notations used for each state machine simply by typing either *structural* or *tabular* at the beginning of the corresponding state machine definition:

```plaintext
workflow container org.jetbrains.workflow
    tabular workflow simpleIssueTracking
    state Open;
    state WaitingVerification;
    state Closed;
    event Close;
    event Verify;
    event Reopen;

    on event Close
        Open -> WaitingVerification
    on event Verify
        WaitingVerification -> Closed
    on event Reopen
        WaitingVerification -> Open
    on event Reopen
        Closed -> Open
```

The sample consists of three languages and a sandbox project.
The requestTracking language provides the concepts and root concepts to wrap state machines and use them to define simple workflows. This could serve as an example of language that needs to embed a state machine language and allow for alternative notations for that embedded language.

The stateMachine language defines the basic concepts of the state machine language plus default editors. It has no artifacts specific to multiple projections at all. To illustrate the power of language extension in MPS the alternative editor projections for some of the concepts have been defined in stateMachine.tabular, which extends stateMachine.

While the default editors specified in stateMachine indicate the fact of being default with the default value in the upper-left corner, the editors in stateMachine.tabular specify the tabular hint. Specifying multiple hints for a single editor is also possible:

### 3.3.2 Hints

The key element in choosing the right projection are Hints. Editors specify, which hints will trigger them to show up on the screen. Hints are defined using the new ConceptEditorContextHints concept.
This concept lets you define the ID and a short description for each hint recognized in the particular language or a language extension. So in our sample project, `stateMachine` and `stateMachine.tabular` both define their own set of hints.

Notice also the Can be used as a default hint flag that can be set in the inspector. When set to true, the hint will be available to be pushed to editors from the IDE. See the details below, in the "Pushing hints from the IDE" section.

With hints defined, languages can offer the user to switch between notations by adding/removing hints into/from the context. The sample `requestTracking` language does it by exposing a presentation property of an enumeration type to the user. The property influences the collection of hints passed down into the state machine editor, as specified in the inspector window:

3.3.3 Editor hints for Editor Components

Not only editors can have hints specified. Editor Components that override other editor components have the same capability.

3.3.4 Pushing hints from the IDE

The hints that have the "Can be used as a default hint" flag enabled, can be pushed by the IDE to the editors as the new defaults. This allows the developers to customize the default projection used for different languages in their IDEs.
One way to customize the projection is to use the **Push Editor Hints** action in the editor context menu and select the hints that you want to be pushed as defaults to the active editor frame:

The active editors will then use projections that match the selected hints. The second option is to make some hints pushed by the IDE through the corresponding **Settings** panel. These choices are then applied to all editor windows as default preferences.

You may consider combining the ability to switch between notations with splitting the editor frame. This allows you to lay several different projections of the same piece of code displayed next to one-another. Your changes in one will be immediately reflected in the other:

The right panel has the *tabular* notation pushed as the default, which the left panel does not. Both projections visualize the same code.
3.4 How to manipulate and apply editor hints programmatically

If you want to create your own UI elements that will adjust editor hints, you’ll need to write code that will manipulate the editor hints applied to editor cells. EditorContext is the entry point for manipulating the hints. You’ll need to obtain a `jetbrains.mps.openapi.editor.update.Updater` interface implementation, most likely using the `EditorComponent.getUpdater()` method. The Updater interface provides several methods for manipulating hints:

- `setInitialEditorHints()` - sets the hints that the editor will start applying from the top of the editor component hierarchy down to all of the components, unless they explicitly remove these hints from the context, returns true if the hints actually changed as a result of this operation
- `getInitialEditorHints()` - gets the hints the the editor applies to all cells, may return null
- `addExplicitEditorHintsForNode()` - adds hints to apply to a particular node’s editor and the editors it embeds
- `removeExplicitEditorHintsForNode` - removes the specified hints from the list of hints explicitly applied to the node’s editor and the editors it embeds
- `getExplicitEditorHintsForNode` - gets the list hints of hints explicitly applied to the node’s editor and the editors it embeds, may return null
- `clearExplicitHints` - clears all explicitly specified hints for all editors

Also, after changing the hints, the editor needs to be rebuilt in order to reflect the changes. This can be done through the `Updater.update()` method.

The `multipleProjections` sample project contains three sample intentions that programmatically manipulate editor hints applied to the editor. Check out the sample for more details.
Chapter 4

Editor language generation API

The editor language is supposed to be extended by numerous MPS users, so we designed the generator for the Editor language for ease of use - straightforward templates, human-readable generated code and use of meta-information more at the generation-time than at run-time. If any of your languages extend the editor language in order to provide new cell types, this document is for you.

4.0.1 API: EditorCell contract

The contract of EditorCell.setBig().getBig() methods was slightly changed. Please check the javadoc for details.

4.0.2 API: EditorCellFactory is now available only within UpdateSession

We made the EditorCellFactory instance controlled by the current UpdateSession. In the same time we put some caches inside the EditorCellFactory implementation, making the editor building process faster in some situations. The EditorContext.getCellFactory() method was deprecated and will be removed in the next release.

4.0.3 Language Runtime: AbstractEditorBuilder

AbstractEditorBuilder runtime class was introduced and should be used as a common super-class of any classes containing cell factory methods. This class implements common utility methods and provides access to generic contextual parameters of editor cell creation process like:

- editorContext
- node
- CellFactory
- UpdateSession

AbstractEditorBuilder is used to capture some context of cell creation process and execute consequent cell factory methods within this context.

4.0.4 Generator: EditorBuilder classes

A separate sub-class of AbstractEditorBuilder class will be generated now as a root class for each of available editor declaration hierarchies:

- ConceptEditorDeclaration.cellModel
- ConceptEditorDeclaration.inspectedCellModel
- EditorComponentDeclaration
- InlineEditorComponent

The MPS editor generator will continue creating classes, implementing ConceptEditor & ConceptEditorComponent. These classes were used earlier as containers for cell factory methods. In the new version of MPS these classes are used as descriptors providing access to the contextual hints information & instantiating actual EditorBuilders. Descriptor classes may be cached by the EditorCellFactory implementation.
4.0.5 Contextual parameters available for cell builders

The code, generated as a part of AbstractEditorBuilder sub-classes may access contextual parameters by using existing methods of AbstractEditorBuilder class. In addition to that, all available meta-information is used to generate private fields with more specific types than those available in the method signatures of AbstractEditorBuilder. For now each sub-class of AbstractEditorBuilder will hold private node<TheConcept> myNode field, where TheConcept is the actual concept associated with this AbstractEditorBuilder. This means that any cell factory method may use such a private field in order to get typed access to the contextual node and directly access available properties, links and other information from the contextual node using s-model language.

4.0.6 CellFactoryContextClass

CellFactoryContextClass is a handy utility class providing necessary context for templates, generating the code included into one of the generated AbstractEditorBuilder sub-classes. By using this class as a contextual class template authors will automatically obtain all available methods and fields, the code generation environment will be supported by MPS platform, and so it’s not necessary to reconstruct it for each and every editor template anymore. At the same time, CellFactoryContextClass can be used as a marker interface highlighting templates, which will generate code for one of the EditorBuilders, simplifying the process of locating such code & supporting it in the future.

4.0.7 GenericCellCreationContext

The GenericCellCreationContext interface provides limited subset of contextual information, which is always available for the code called either as a part of EditorBuilders or from a separate class, executed as a part of cell creation (editor update) process. This interface should be used as a template context instead of CellFactoryContextClass in those cases, when template authors are going to reuse the same template across the EditorBuilders generation process and some other places. For example, query methods which may be generated either inside EditorBuilders or within some style class.

4.0.8 New signature for createCell() methods

In the previous version of MPS, the cell factory methods were always generated with two additional parameters specifying the context of cell creation: EditorContext & node<>. From now on it’s not necessary to specify these parameters any more - the generated code can always access this information (as well as any other contextual info) by calling methods from the containing EditorBuilder class. The new editor generator will generate cell factory methods without any parameters.

4.0.9 Automatic migration script for new createCell() methods

For compatibility with the existing generators, we provide a migration script patching available templates and introducing the new createCell() methods, which delegate to the old ones (with the two additional parameters) as a fallback. We recommend to execute this script first and then check all modifications and verify, if the modified generator still works correctly. The provided automatic migration supports only most frequent situations, so in some specific cases you may need to manually modify your generator in order to make it work again. The template, which generates the compatibility methods is called template_cellFactoryCompatibility. If you later modify your generator to generate directly the new createCellMethods, you should remove any calls to template_cellFactoryCompatibility. We do recommend to review all existing generators & patch obsolete templates generating the legacy createCell(...) methods in the scope of the current MPS release - we are going to drop the compatibility template in the next version.

4.0.10 Mapping labels

Several mapping labels were introduced into the Editor generator (MAPPING_main) and may be used to simplify code generation:

cellFactory.class.concept

cellFactory.class.concept : ConceptEditorDeclaration -> ClassConcept
This label expose a java class, generated for the EditorBuilder of ConceptEditorDeclaration.cellModel

cellFactory.class.inspector

cellFactory.class.inspector : ConceptEditorDeclaration -> ClassConcept
This label exposes a java class, generated for the EditorBuilder of ConceptEditorDeclaration.inspectedCellModel

cellFactory.class.component

cellFactory.class.component : EditorComponentDeclaration -> ClassConcept
This label exposes a java class, generated for the EditorBuilder of EditorComponentDeclaration
cellFactory.constructor

`cellFactory.constructor : EditorCellModel -> ConstructorDeclaration`

Used to mark the constructor of the generated `EditorBuilder` class.

cellFactory.factoryMethod

`cellFactory.factoryMethod : EditorCellModel -> InstanceMethodDeclaration`

The replacement for obsolete `cellFactoryMethod` label, containing the new `cellFactory` methods. This label should be used instead of `cellFactoryMethod` at the moment of modification of existing templates making them generating new `cellFactory` methods.

generated.constructor

`generated.constructor : <no input concept> -> ConstructorDeclaration`

This label may be used together with existing `generatedClass` one to mark generated constructor instances. This label may be used to avoid the ugly code for locating first constructor instance inside `node<ClassConcept>`, returned from the `generatedClass` mapping.

4.0.11 CellLayoutConstructor switch introduced

This template switch is used to instantiate proper cell layout while creating a collection cell. The previously used static `createxxx()` methods inside `EditorCell_Collection` class have been deprecated and will be removed.

4.0.12 New generator for RefCellCellProvider sub-classes

The generator for `CellModel_RefCell` has been modified. The newly generated anonymous inner classes for `RefCellCellProvider` do not use the logic located inside `RefCellCellProvider.createRefCell()` runtime method. The meta-information, available at generation-time, are used in order to create complete content of this method. If you do generate sub-classes of `RefCellCellProvider` within your generators, you should consider reviewing such places and aligning your templates with the templates from MPS.

4.0.13 InlineCellProvider replaced with EditorBuilder sub-class

`InlineCellProvider` is not being used by the MPS generator anymore. MPS uses the generated sub-classes of `AbstractEditorBuilder` instead. Nevertheless, we modified some constraints inside `InlineCellProvider` in order to make the lifecycle more transparent. We recommend to check the javadoc for `InlineCellProvider`, if you are still using it.

4.0.14 Editor Styles generator

A separate static inner class will be generated for each entry inside `StyleSheet & StyleKeyPack` instances. The provided `applyStyleClass` template may be used to properly instantiate & call the new `Style` classes. Legacy static `applyxxx()` methods should be removed in the next release.

4.0.15_STYLEClassItem constraints modification

We removed the `canBeChild` constraints from the `StyleClassItem` concept. These constraints were replaced with `canBeParent` constraints of the node, containing the `StyleClassItem`. In addition to that `isApplicableToCell(node<EditorCellModel> cellModel)` behaviour method has been deprecated and is not used anymore. Instead we have introduced the following methods:

- `isApplicableToCellConcept()`
- `isApplicableForLayout()`
- `isApplicableInLayout()`

We recommend you to check the javadoc of `StyleClassItem` behavior methods, if you are implementing any custom `StyleClassItem` in your language.
Chapter 5

Description comments

We are going to learn how to leverage the capability of adding attributes to nodes. This feature is fully described in the Attributes section of the Structure chapter. In this cookbook we'll create a simple addition to the Calculator tutorial language that will allow the input and output fields of the calculator definitions to have descriptive comments attached to them. These descriptive comments will be propagated into the generated Java code in the form of "pop-up tooltips" attached to the Swing text fields corresponding to the input and output fields.

5.1 The Calculator language

We chose the Calculator tutorial language as a testbed for our experiments. You can find the calculator-tutorial project included in the set of sample projects that comes with the MPS distribution. I recommend you skinned through the tutorial to familiarize yourself with the language before we continue.

5.2 Changes to the fields

Let’s start building the machinery for adding descriptive comments to input and output fields. Our approach will be based on annotations that when added to nodes will have impact on their visual appearance as well as on the generated code. There are several vital components that will enable our desired functionality:

- A marker interface that indicates, which nodes can hold the description comments
- The annotation itself to indicate whether a node holds the description comment and what the description is
- An action providing a keyboard shortcut to quickly add and remove the description comment to/from a node under caret
- Updating the code generation process in order to generate the visual "tooltips" for the described components

We start with a new interface to mark all commentable nodes:

```
interface concept ICanHaveDescription extends <none>
```

Now both InputField and OutputField concepts have to implement the new ICanHaveDescription interface.
5.3 The DescriptionAnnotation concept

We are still missing the core piece of our puzzle - the actual annotation that will be attributed to input and output fields to specify the description comment. Let’s create it now:

Once you make the concept extend `NodeAttribute`, MPS will hint you to provide further information detailing the attribution. Invoking a quick-fix through Alt + Enter will add the necessary information for you to customize:
You need to specify additional qualities of the new annotation by setting values for role as well as attributed concepts. We set the role for the annotation as descriptionComment. This is the name to use in queries in order to find out, whether a node has been annotated with DescriptionAnnotation or not. We also indicate that the annotation can only be attributed to nodes implementing ICanHaveDescription, which in our case is InputField and OutputField.

```
@attribute info
  multiple: <inherited>
  role: descriptionComment

attributed concepts: ICanHaveDescription

concept DescriptionAnnotation extends NodeAttribute
  implements <none>

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

  properties:
    << ... >>

  children:
    << ... >>

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

The DescriptionAnnotation also has to store the actual text of the description - we'll use a string property descriptionText for this:
The editor of DescriptionAnnotation is the key element in defining the visual behavior of nodes with description comments attached to them. We prepend the annotated node with a "described" constant and append the actual description text wrapped in parentheses. The attributed node cell represents the editor of the actual node - the node that the attribute is attached to.

5.4 Further editor enhancements

To toggle the description comment of a node on and off, we will create new KeyMap.
To edit the code we will use concepts from *BaseLanguage* and *smodel*, which should be available by default. If not, import them through \textit{Control/Cmd} + L. We will also need the *actions* language in order to create initialized nodes easily. This one will most likely need to be imported explicitly (\textit{Control} + L).

Now we can complete the action:
Whenever Control + Alt + Shift + D is pressed on a node implementing ICanHaveDescription, the action will add or remove the DescriptionAnnotation to/from the node. The KeyMap now needs to be added to all fields that should react to Control + Alt + Shift + D. This is to InputField and OutputField in our case. The editor cells that should react to the Control + Alt + Shift + D key combination must have the key map specified in the keycap property.
5.5 How are we doing so far?

After compilation you are able to use `Control + Alt + Shift + D` both on input fields and output fields to toggle the description comment:
5.6 Updating the generator

The `DescriptionAnnotation` is currently lost during the generation process. Instead, we would like to generate the Swing "tooltips" for the text fields that represent the commented input and output fields.

When generating code we currently simply take all input or output fields in the calculator and generate JTextFields for them. In the constructor we also generate initialization code for each generated JTextField - this is the place, where we can attach a Swing "tooltip" the the JTextFields that represent an input or output fields with description comments.

The `$IFS$` node macro will ensure that a "tooltip" is only generated for nodes with the `DescriptionComment` annotation attached.
The *property macro* of the text of the "tooltip" will set the *descriptionText* of the *DescriptionAnnotation* attached to the current node:

After rebuilding the language and the sandbox solution, the generated Calculator form will have "tooltips" attached to the fields that have been marked with the *DescriptionAnnotation*:

### 5.7 Summary

We can finish our tour here. Both input fields and output fields can be described with *DescriptionAnnotation* and the generated code contains a "tooltip" for each such description.
Chapter 6

Generator cookbook

This document is intended to give answers to the most common questions related to the MPS generator. You may alternatively consult the Generator and check out the Generator Demos.

6.0.1 How does the generator process the rules?

Generation gradually converts an input model into an output model, which may or may not be then turned into text with TextGen. The generation process itself consists of steps. Each step has three phases:

1. Executing pre-mapping scripts
2. Template-based model transformation
3. Executing post-mapping scripts

The template-based model transformation phase consists of one or more micro-steps. A micro-step is a single-pass model transformation of and input model into a transient (output) model. The generator during a generation step will iterate over the model and search in the generators of all languages involved in this step for rules applicable to the nodes in the input model. When no applicable rules are found, the generation step stops. The next generation step (if any) will receive the output model of previous generation step as its input.

You can check out Generator in the generator documentation.

6.0.2 Can I have more generators for a single language?

No, MPS only allows one generator per language at the moment. If you need several generators for a language, it is advisable to create empty languages that extend your language and in each of these extending languages to define the desired alternative generators.

6.0.3 Can I have multiple mapping configurations per generator?

Yes, they will all be treated as equal during the generation process. Additionally, each mapping configuration can have its generation process priority specified separately, which gives you more flexibility to tune the generation.

6.0.4 How to generate multiple targets from single source?

You may want to generate code for several target platforms from the same code-base. There are at least two ways you can achieve that:

- Put all the generation rules for different platforms into a single generator, perhaps logically separated into virtual packages and multiple mapping configurations, and use the condition of each rule to activate the rule based on the currently selected target. The developer will indicate the intended target platform by setting a flag in her code.

- Have no generator in you language directly and provide generators for each target platform in a separate empty language, which extends the original language. The developer will select the desired target platform by importing the appropriate extension language.
6.0.5 What macros are available in the templates?

- **property macro** - computes a value for a property
- **reference macro** - computes the target (node) of a reference
- **node macro** - is used to control template filling at generation time
  - `$IF$` - conditional generation
  - `$CALL$` - insert another template at the current position and process it passing in the current node
  - `$LOOP$` - iterate over a collection of nodes and set the current node for each iteration
  - `$COPY_SRC$` - copies the specified node and replaces the wrapped node. Reduction rules are applied to the copied node and its children during the process
  - `$COPY_SRCL$` - copies the specified collection of nodes and replaces the wrapped code. Reduction rules are applied to the copied nodes and their children during the process
  - `$MAP_SRC$` - sets a node to be used as the current node inside the wrapped code
  - `$MAP_SRCL$` - sets a collection of nodes, each of which should to be used in turn as the current nodes inside the wrapped code
  - `$SWITCH$` - chooses the template to use for generation from multiple choices
  - `$LABEL$` - stores the wrapped node in a mapping label for easy discovery by other macros
  - `$VAR$` - sets a value into a variable, which will then be accessible in the wrapped nodes through `genContext.varName`
  - `$TRACE$` - stores information required to trace back the original node for the wrapped node and stores the mapping between a source node and the resulting generated text into the `trace.info` file - when Save Transient Models is on, this enables the Reveal Origin Node option in the Debug pop-up menu item in transient models
  - `$WEAVE$` - invokes a specific weaving rule
  - `$INSERT$` - inserts a node into the output model at the current position
  - `$EXPORT$` - saves a node for cross-model reference, so it can be retrieved when generating other models

6.0.6 Where shall I put utility classes?

Create a new model inside the generator. Make sure it does not have the `generator` stereotype attached to it. The model should typically depend on `BaseLanguage` so that you can create classes in it. The original `generator` model should then import the utility model.

6.0.7 Where shall I put runtime classes?

Classes and libraries that the generated code relies on should be put into separate runtime solutions, as described in the "Getting the dependencies right" section of Getting the dependencies right.

6.0.8 How do I generate unique names?

Use the `genContext` parameter, which gives you access to the generator context object and call:

```
unique name from <base name> in context <node>
```

- **base name** - is an arbitrary string that must be part of the generated name
- **context node** - If specified, then MPS tries its best to generated names ‘contained’ in a scope (usually a root node). Then when names are re-calculated (due to changes in input model or in generator model), this won’t affect other names outside the scope. The context node parameter is optional, though we recommend to specify it to guarantee generation stability. The uniqueness of generated names is secured throughout the whole generation session.

⚠️ Clashing with names that weren’t generated using this service is still possible.

6.0.9 How to generate enumeration datatypes?

Having to reduce an enumeration datatype into a Java enum, using the `$SWITCH$` macro is usually the best option.
Notice the way the actual enumeration datatype is tested for equality using the `is()` method:

```plaintext
template switch ReduceDirectionEnum extends <none>

null-input message: <none>

cases:

- concept Looking<nbsp>inherited false
  condition (genContext, node, operationContext)->boolean {
    node.direction.is(< north >);
  }
  --> <T Direction.north T>

- concept Looking<nbsp>inherited false
  condition (genContext, node, operationContext)->boolean {
    node.direction.is(< east >);
  }
  --> <T Direction.east T>

- concept Looking<nbsp>inherited false
  condition (genContext, node, operationContext)->boolean {
    node.direction.is(< south >);
  }
  --> <T Direction.south T>

- concept Looking<nbsp>inherited false
  condition (genContext, node, operationContext)->boolean {
    node.direction.is(< west >);
  }
  --> <T Direction.west T>
```

### 6.0.10 What is the difference between root mapping rules and reduction rules?

During generation models, which have the form of a tree (Abstract Syntax Tree, see Basic notions), are translated into the generated models, which are also trees. Typically root nodes are generated into root nodes (e.g. a robot Script into a Java class) and non-root nodes into non-root nodes (e.g. a robot Step command into a Java BlockStatement). Thus templates for root mapping rules represent a starting point of the generated model and cannot specify template context, unlike reduction templates, which may hold context and indicate the actual generation contents with the `template fragment` marks.

Nodes of rootable concepts, when positioned in a non-root location, will have reduction rules applied to them. Root mapping rules are only applied to nodes in root positions.
6.0.11 What happens to untransformed roots?

Roots, for which there are no applicable transformation rules, are carried unchanged over to the next generation step.

6.0.12 How to generate multiple nodes for a single node?

A template fragment in a template indicates, which node will be replacing the current input node. A template fragment can only be attached to a single node inside the template, but a generator template can contain multiple template fragments, provided they are all attached to nodes in the same role under the same parent. This way multiple nodes can be generated for a single input node.

```java
protected void perform() {
   <TF {moveKaja(); } <TF
   <TF {pause(); } <TF
}
```

6.0.13 Why is editing the macros so weird?

Indeed, the generator macros and template fragment marks are a bit rough to edit at first. But once you understand the underlying principles of how they work, you’ll be able to use them efficiently.

The macros have been implemented as `Node/property/reference attributes` (see the Attributes section of the Structure chapter). As such, they can be used in any language and do not require any up-front support to be available in that language. This is important, because any language can be used as a generation target in MPS. Macros are thus attributes attached to the node that they wrap. Deleting or replacing the node wrapped by a macro will delete the macro as well and you have to re-enter the macro to attach it to a new node. A recommended approach is to enter the code for the template first and then start adding macros. If you need to change the node wrapped by a macro, you can at least preserve the Inspector window contents on the macro on the old node by copy-pasting it into the macro on the new node.

6.0.14 How can I generate root nodes for non-root nodes?

Simply create a root mapping rule for the concept, perhaps further restrict it with a condition. The rule is called root, because it generates a root node, not necessarily that it takes a root as input. Alternatively you can either use conditional root rules, which insert root nodes based on evaluating a boolean predicate, or with pre-processing scripts, which can inspect the input model and create root nodes as needed.

6.0.15 How to remove no-longer needed roots from the model?

Root nodes are removed automatically once they have been transformed using root mapping rules. For roots that are not directly transformed use the abandon root rule to prevent them from being propagated into the next generation step.

6.0.16 How do I generate nodes for nodes in accessory models?

Nodes in the accessory models are not handled by the generator in any way. In order to generate nodes based on nodes in accessory models, you can leverage either the conditional root rules, or pre-processing scripts.

```java
script script IncludeColorsFromAccessoryModels

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

(genContext, model, operationContext)->void {
   model.nodesIncludingImported(Color).forEach({-it ->
      node<ColorRepresentation> c = new node<ColorRepresentation>();
      c.name = it.name;
      model.add root(c);
   });
}
```

Reduction rules can then be used to further generate the "inserted nodes" into the desired target language.

6.0.17 How to use mapping labels?

Mapping labels are defined in mapping configurations (e.g. main). They must then be populated by wrapping the node that you want to be stored with macros, such as $LOOP$, $LABEL$, $COPY_SRC$ and others.
6.0.18 How to handle references during generation?

Reference macros should be used to set proper target nodes on generated references. You typically use reference macros together with mapping labels.

The fundamental task here is that for a reference $R$ that points to a node $N$, we need to ensure that a reference $G(R)$ generated from $R$ must point to a node $G(N)$ that was generated from $N$.

If $R \rightarrow N$, $N$ generates to $G(N)$, $R$ generates into $G(R)$

then $G(R) \rightarrow G(N)$

Mapping labels serve as dictionaries, which store nodes as keys and whatever they have been generated into as values - $N \rightarrow G(N)$ in our example here. The generator has to be instructed explicitly to store the mapping between a node and its generated node into a mapping label. Template fragments as well as node macros allow the user to specify a mapping label into which the generated node should be saved.

When reducing $R$ into $G(R)$, a reference macro on $G(R)$ can retrieve $G(N)$ from the mapping label using $N$ as a key when calling the generator context in genContext.get output by label and input.

Generator User Guide Demo7 of the Generator Demos covers mapping labels in more detail.

6.0.19 Can I debug the generation process?

The first option is to enable saving transient models. The intermediate models that MPS creates during generation will be preserved and you can access them in the Project View panel on the left-hand side of your screen:

The saved transient models allow you to investigate how the models looked like at various stages of the generation process. The second option is using the Generator Tracer Tool. For this tool the "Save transient models" option also has to be set to on. The Generator Tracer Tool gives you the ability to investigate in detail the generation process for a selected node. When you select a node in your original model or in any of the transient ones, the context menu gives you the options to either trace the generation process forward or backward. As a result you get a tree-shaped report giving you access to all the stages, through which the particular node went during generation and the rules that influence its generation.
Additionally, the `$TRACE$` generator macro gives you the option to mark parts of your templates for tracing, so that you will be able to navigate back from a generated node in a transient model to its original node using the Reveal origin node option in the Context menu -> Language Debug menu:

```java
public class $[map_Script]$ extends KajaFrame {
    public map_Script() {
        <no statements>
    }

    protected void perform() {
        $LOOP$ [$COPY_SRC$][System.out.println("" ); ]
    }

    $LOOP$ [$COPY_SRCL$] public void routine() {
        <no statements>
    }

    public void libraryRoutine() {
        <no statements>
    }

    public static void main(String[] args) {
        record input node as origin of the output

        comment : <none>
        mapping label : <no label>
        input node : (genContext, node, operationContext)->node<>
    }
6.0.20 How do I figure out the actual generation steps?

Use the *Show Generation Plan* action from the generator context menu. This will give a detailed report showing all the planned generation phases and listing the mapping configurations run in each phase.

6.0.21 Can I specify the order in which generators should be run?

The *Generator priorities* tab in the generator’s *Module properties dialog* enables you to specify ordering rules between two distinct *mappings configurations*. 
For more complex scenarios, the new functionality of Generation plan gives you full control over the generator process.

6.0.22 How to check that the model has no errors before starting the generator?

Run the Check model (language/solution) context-menu action on the model/language/solution to check.

The Preferences -> Build, Execution, Deployment -> Generator -> Check models for errors before generation configuration flag specifies whether the model checker should be run automatically each time the generator is triggered.

6.0.23 How to extend an existing generator?

There is one rule to follow - make sure your extending generator is run before the extended one - give it higher priority in the generator order priority dialog.

6.0.24 How to build an extensible generator?

By extending the generator, extensions can alter the semantics of the original language. The MPS generator is extensible by design – it resolves all generator rules and mapping configurations from all involved languages and builds a global generation plan. Any language that is attached to the project will have its rules included. The plan specifies the execution order for the generator rules based on their mutual relative priorities expressed in mapping configurations. This enables language extensions to inject their own desired generation rules into the most suitable generation phase. Since priorities are expressed as a collection of relative ordering between mapping configurations, a language extension does not need to know about all other generators involved in generation in order to work. Potential (and rare) clashes are detected and left up to the developer to resolve. Once created, the generation plan is used to iteratively invoke the generators, potentially leveraging parallelism of the underlying hardware for mutually independent rules.

Providing additional reduction rules is one way to extend a language. Using a Generator Switch is another option. If the parent language uses a generator switch to choose the right reduction rules, the language extension may extend that generator switch with its additional logic for picking the reduction rules – typically to include new rules contributed by extension languages.
6.0.25 What shall I use TextGen for?

While the generator performs model-to-model transformations, TextGen does mapping to text. This is typically the last step in generation and is fully managed by the generator. Roots are picked by the generator and converted to files. The TextGen intentionally offers very little flexibility to the language designer. It is the Generator, where the generation process should primarily be configured and handled. If you feel a need to workaround the limitations of TextGen, you’re most likely trying things the wrong way. Generator is the place where you have the most flexibility.

6.0.26 How to generate multiple files per root node?

TextGen only allows one file per root node. Also, you can only have one TextGen component per concept. The flexibility should be encoded into the generator, instead. You get the generator create multiple copies of the node, maybe wrapping each of them in a new node of different concepts. These new nodes then have their TextDef defined to perform the desired model-to-file conversions.
Chapter 7

Cross-model generation cookbook
Chapter 8

Cookbook - Type System

8.1 Inference rules

This cookbook should give you quick answers and guidelines when designing types for your languages. For in-depth description of the typesystem please refer to the Typesystem section of the user guide.

8.1.1 Equality

Use type equation when the type of a node should always be a particular concrete type. Use the typeof command to declare that the type of the desired node should equal to a particular type.

```plaintext
rule typeof_StringLiteral {
  applicable for concept = StringLiteral as nodeToCheck
  applicable always
  overrides false

  do {
    typeof(nodeToCheck) :==: <string>;
  }
}
```

Note quotation is used to refer to a type. `<string>` is equivalent to typing `new node<StringType>()`. The type of an element is equal to the type of some other element. For example, the to express that parentheses preserve the type of the wrapped element, the ParenthesizedExpression concept declares:

```plaintext
rule typeOf_ParenthesizedExpression {
  applicable for concept = ParenthesizedExpression as parExpr
  applicable always
  overrides false

  do {
    typeof(parExpr) :==: typeof(parExpr.expression);
  }
}
```

8.1.2 Inequality

When the types should be sub-types or super-types of other types, use the infer typeof command. See the ternary operator as an example:
The `ForEachStatement` concept illustrates how to solve quite an involved scenario. The type of the loop variable must be equal to the type of elements in the iterated collection, while the type of the collection must be a sub-type of either a sequence or an array of elements of the `elementType` type.

```
rule typeof_ForEachStatement {
  applicable for concept = ForEachStatement as forEachStatement
  applicable always
  overrides false
  do {
    node<ForEachVariable> variable = forEachStatement.variable;
    node<Expression> inputSequence = forEachStatement.inputSequence;
    if (inputSequence.isNotNull && variable.isNotNull) {
      var elementType;
      infer <join(sequence<%( elementType)%>| %( elementType)%[])> :>=: typeof(inputSequence);
      typeof(variable) :==: elementType;
    }
  }
}
```

Notice, we use `var elementType` to declare a variable, which we then use to tie together the type of the collection elements and the type of the loop variable. Also, `%(...)%` demarcates so called anti-quotation, which allows you to provide values from your local context into the AST you are manipulating or retrieve them back.

### 8.2 Replacement rules

Replacement rules indicate to the type system the possibility to replace one type with another. For example, `NullType` is a subtype of all types (except for primitive types) and so the type system can simply remove the inequation between `NullType` and `BaseConcept`.

```
replacement rule any_type_supertypeof_nulltype
  applicable for concept = NullType as nullType <: concept = BaseConcept as baseConcept
  custom condition: ()->boolean {
    ! (baseConcept.isInstanceOf(RuntimeTypeVariable));
  }
  rule {
    if (baseConcept.isInstanceOf(PrimitiveType) || baseConcept.isInstanceOf(PrimitiveTypeDescriptor)) {
      error "null type is not a subtype of primitive type" -> equationInfo.getNodeWithError();
    }
  }
```

Replacement rules are also handy to declare covariance and contravariance. For example, covariance for sequences is declared in MPS as follows:
replacement rule sequence_subtypeOf_sequence

applicable for concept = SequenceType as left <: concept = SequenceType as right

custom condition: true

rule {
    if (right.elementType.isNotNull) {
        infer left.elementType :<=: right.elementType;
    }
}

The original rule claiming that the left collection is a subtype of the right collection gets replaced with a rule ensuring that the type of elements in the left collection is a subtype of the type of elements in the right collection.

### 8.3 Subtyping rules

Subtyping rules allow you to specify where the particular type belongs in the type hierarchy. The rule returns a collection of types, which it identifies as its direct super-types. The following rule, for example, declares that `Long` variables can be cast to `Float`.

```markdown
subtyping rule long_extends_float {
    weak = false
    applicable for concept = LongType as longType

    rule {
        return <float>;
    }
}
```

Here MPS declares, that `LinkedList` is a subtype of either a `List`, a `Deque` or a `Stack`:

```markdown
subtyping rule supertypesOf_linkedlist {
    weak = false
    applicable for concept = LinkedListType as llt

    rule {
        nlist<> res = new nlist<>;
        res.add(<list<%( llt.elementType)%>>);
        res.add(<deque<%( llt.elementType)%>>);
        res.add(<stack<%( llt.elementType)%>>);
        return res;
    }
}
```

### 8.4 Comparison rules

When two types should be interchangeable, use comparison rules to define that. For example, the following rule makes `NullType` comparable with any type, except for primitive ones:

```markdown
comparison rule any_type_comparable_with_nulltype

applicable for concept = BaseConcept as baseConcept , concept = NullType as nullType

rule {
    if (baseConcept.isInstanceOf(PrimitiveType) || baseConcept.isInstanceOf(PrimitiveTypeDescriptor)) { return false; }
    return true;
} weak = false
```

Similarly, the `MapType` from `BaseLanguage` and the `Map` interface from Java (here referred to through the `ClassifierType` concept inside a pattern) should be comparable:
8.5 Substitute Type rules

These instruct the type-system to replace nodes representing a type with defined substitutes. For example, one might decide to use different types for different program configurations, such as using int or long depending on whether the task requires using one type or another. This is different from simply using the generator to produce the correct "implementation" type, as the substitution is done at the time the typechecking is performed, so possible errors can be caught early.

In its simplest form the type substitution can be used by creating an instance of Substitute Type Rule in the typesystem model.

```
substitute type rule substituteType_MyType {
    applicable for concept = MyType as mt
    substitute {
        if (mt.isConditionSatisfied()) {
            return new node<IntegerType>;
        }
        null;
    }
}
```

The Substitute Type Rule is applicable to nodes that represent types. Whenever a new type is introduced by the typechecker, it searches for applicable substitution rules and executes them. The rule must either return an instance of 'node<>' as the substitution, or null value, in which case the original node is used to represent the type (the default behaviour).

One other possibility to override types used by the typechecker comes with the use of node attributes. If there is a node attribute contained by the original type node, the typechecker tries to find a Substitute Type Rule applicable to the attribute first. This way one can override the type nodes even for languages, which implementation is sealed.

```
substitute type rule substituteType_SubstituteAnnotation {
    applicable for concept = SubstituteAnnotation as substituteAnnotation
    substitute {
        if (substituteAnnotation.condition.isSatisfied(attributedNode)) {
            return substituteAnnotation.substitute;
        }
        null;
    }
}
```

The rule above is defined for the attribute node, and it’s the attribute node that is passed to the rule as the explicit parameter. The rule can check whether the condition for substituting the type node is satisfied, and it can also access the attributed node representing original type via attributedNode expression.

8.6 Checking and Quick-fixes

Checking rules become part of the MPS code analysis process and will report found issues to the user interactively in the editor. For example, this is a check for superfluous type casts:

```
comparison rule map_type_comparableWith_Map
applicable for concept = MapType as mapType, > Map<# KEY, # VALUE> < as classifierMapType
rule {
    return true;
}
weak = true
```
Checking rule `CheckExcessTypeCasts` {
    applicable for concept = CastExpression as expr
    overrides false

do {
    if (isStrongSubtype(expr.expression.type :<< expr.type)) {
        info "Typecast expression is superflous" -> expr ;
    }
}
}

Now you can define a quick-fix that will pop-up to the user whenever the problem above is reported. The user can then quickly invoke the quick-fix to correct the reported issue.

**Quick fix RemoveExcessTypeCast**

**Arguments:**
node<CastExpression> castExpr

**Fields:**
<< ... >>

description(node)->string {
    "Remove Excess Typecast";
}

execute(node)->void {
    castExpr.replace with(castExpr.expression);
}

The hook the quick-fix to the reported error, you need to specify the quick-fix as **intention linked with info message(optional)**:

Additionally, you can pass parameters to the quick-fix and mark it with **apply immediately**, in which case the quick-fix will be applied automatically as soon as the error is discovered in the editor.

### 8.7 When-concrete, overloaded operations

When-concrete blocks allow you to perform type checks once the type a node has been calculated. In the example below we are checking, that the calculated type of an operation matches the type suggested by the operation type command based on the operator overriding rules:
rule typeof_BinaryOperation {
    applicable for concept = BinaryOperation as operation
    overrides false

do {
    when concrete (typeof(operation.leftExpression) as leftType) {
        when concrete (typeof(operation.rightExpression) as rightType) {
            node<> opType = operation type( operation , leftType , rightType );
            if (opType.isNotNull) {
                typeof(operation) := opType;
            } else {
                error "operation is not applicable to these operands" -> operation;
            }
        }
    }
}
}
Chapter 9

HowTo – Integration with the Data Flow Engine

9.1 Dataflow

Data flow analysis supports detecting errors that cannot be found easily by "looking at the model" with static constraints or type checks. Examples include dead code detection, missing returns in some branches of a method’s body or statically finding null values and preventing null pointer exceptions.

The foundation for data flow analysis is the so-called data flow graph. This is a data structure that describes the flow of data through a program's code. For example, in `int i = 42; j = i + 1;` the 42 is "flowing" from the init expression in the local variable declaration into the variable i and then, after adding 1, into j. Data flow analysis consists of two tasks: building a data flow graph for a program, and then performing analysis on this data flow graph to detect problems in the program.

MPS comes with predefined data structures for data flow graphs, a DSL for defining how the graph can be derived from language concepts (and hence, programs), a framework for defining your own analyses on the graph as well as a set of default analyses that can be integrated into your language. We will look at all these ingredients in this section.

To play with the data flow graph, you can select a method in a Java program and then use the context menu on the method; select `{Language Debug -> Show Data Flow Graph}`. This will render the data flow graph graphically and constitutes a good debugging tool when building your own data flow graphs and analyses.

9.2 Building a Data Flow Graph

9.2.1 Simple, Linear Dataflow

In this section we look at the code that has to be written to create a data flow graph for a language similar to C and Java (in fact, it is for the mbeddr.com C base language).

Data flow is specified in the Dataflow aspect of language definitions. Inside that aspect, you can add data flow builders (DFBs) for your language concepts. These are programs that build the data flow graph for instances of those concepts in programs. To get started, here is the DFB for LocalVariableDeclaration.
Let's inspect this in detail. The framework passes in the node variable as a way of referring to the current instance of the concept for which this DFB is defined (LocalVariableDeclaration here). If the LocalVariableDeclaration has an init expression (it is optional!), then the DFB for the init expression has to be executed. The code for statement does this: it "calls" the DFB for the node that is passed as its argument. Then we perform an actual data flow definition: the write node = node.init specifies that write access is performed on the current node (there is also a read statement; this supports detection of read-before-write errors). The statement also expresses that whatever value was in the init expression is now in the node itself.

If there is no init expression, we still want to mark the LocalVariableDeclaration node as visited — the program flow has come across this node. A subsequent analysis reports all program nodes that have not been visited by a DFB as dead code. So even if a node has no further effect on a program's data flow, it has to be marked as visited using nop.

To illustrate a read statement, one can take a look at the LocalVariableRef expression which read-accesses the variable it references. Its data flow is defined as read node.var}, where {{var} is the name of the reference that points to the referenced variable. Here is the code:

For an AssignmentStatement, the data flow is as follows:

Note how we first execute the DFB for the rvalue and then "flow" the rvalue into the lvalue — the purpose of an assignment.

For a StatementList, we simply mark the list as visited and then execute the DFBs for each statement:

Finally, for a C function, at least for now, ignoring arguments, the DFB simply calls the DFB for the body, a StatementList.

We are now ready to inspect the data flow graph for a simple function. Below is the graph for the function.
Data flow analysis is typically limited to one function, method or similar concept. To signal the end of a one of those, we should use the `ret` statement. To illustrate this, here is the DFB for the `ReturnStatement`:

```java
if (node.expression != null) {
    code for node.expression
} ret
```

### 9.2.2 Branching

Linear dataflow, as described above, is relatively straightforward (no pun intended 😊). However, most interesting data flow analysis has to do with loops and branching. So specifying the correct DFBs for things like `if`, `switch` and `for` is important. It is also not as simple...

Let us take a step-by-step look at the DFB for the `IfStatement`. We start with the obligatory `nop` to make the node as visited. Then we run the DFB for the condition, because that is evaluated in any case. Then it becomes interesting: depending on whether the condition is true or false, we either run the `thenPart` or we jump to where the `{else if parts begin. Here is the code so far:

```java
nop
code for node.condition
ifJump after elseIfBlock  // elseIfBlock is a label defined later
code for node.thenPart
{ jump after node }
```

The `ifJump` statement means that we may jump to the specified label (i.e. we then execute the `{else if}s). If not (we just "run over" the `ifJump`), then we execute the `{thenPart}. If we execute the `thenPart`, we are finished with the whole `{IfStatement — no else if}s or `{else parts are relevant, so we jump after the current node (the `IfStatement`) and we’re done. However, there is an additional...
catch: in the thenPart}, there may be a {{return statement}. So we may never actually arrive at the jump after node statement. This is why it is enclosed in curly braces: this says that the code in the braces is optional. If the data flow does not visit it, that's fine (typically because we return from the method before we get a chance to execute this code).

Let's continue with the else if}s. We arrive at the {{elseIfBlock} label if the condition was false, i.e. the above if jump actually happened. We then iterate over the {{elseIf}s and execute their DFB. After that, we run the code for the elsePart, if there is one. The following code can only be understood if we know that, if we execute one of the {{else if}s, then we jump after the whole IfStatement. This is specified in the DFB for the ElseIfPart, which we'll illustrate below. Here is the rest of the code for the IfStatement's DFB:

```java
label elseifBlock
foreach elseif in node.elseIfs {
    code for elseif
}
if (node.elsePart != null) {
    code for node.elsePart
}
```

We can now inspect the DFB for the ElseIfPart. We first run the DFB for the condition. Then we may jump to after that else if, because the condition may be false and we want to try the next else if, if there is one. Alternatively, if the condition is true, we then run the DFB for the body of the ElseIfPart. Then two things can happen: either we jump to after the whole IfStatement (after all, we have found an {{else if} that is true}, or we don't do anything at all anymore because the current else if contains a return statement. So we have to use the curly braces again for the jump to after the whole if. Here is the code:

```java
code for node.condition
ifjump after node
  code for node.body
{ jump after node.ancestor<concept = IfStatement> }
```

The resulting data flow graph is shown below.
Loops
To wrap up data flow graph construction, we can take a
look at the for loop. This is related to branching again, because after
all, a loop can be refactored to branching and jumps. Here is the DFB for the
for loop:

```
0: nop
1: read i
2: ifjump 6
3: nop
4: write j NumberLiteral
5: jump 8
6: nop
7: write j NumberLiteral
8: end
```

We first execute the DFB for the iterator (which is a kind of
LocalVariableDeclaration, so the DFB for it above works). Then we define a
label start so we can jump to this place from further down. We then
execute the condition. Then we have an {ifjmp to after the whole loop
(which covers the case where the condition is false and the loop ends). In the
other case (the condition is still true) we execute the code for the body
and the incr part of the {{for loop We then jump to after the
start label we defined above.

9.3 Integrating Data Flow Checks into your Language

Data flow checks are triggered from NonTypesystemRules. There is a bit of
procedural code that needs to be written, so we create a class
DataflowUtil in the typesystem aspect model.
public class DataflowUtil extends <none> implements <none> {

    private Program prog;

    public DataflowUtil(node<> root) {
        // build a program object and store it
        prog = DataFlow.buildProgram(root);
    }

    @CheckingMethod
    public void checkForUnreachableNodes() {
        // grab all instructions that are unreachable (predefined functionality)
        sequence<Instruction> allUnreachableInstructions = ((sequence<Instruction>) prog.getUnreachableInstructions());
        // remove those that may legally be unreachable
        sequence<Instruction> allWithoutMayBeUnreachable = allUnreachableInstructions.where({~instruction => !(Boolean.TRUE.equals(instruction.getUserObject("mayBeUnreachable"))); });

        // get the program nodes that correspond to the unreachable instructions
        sequence<node<> unreachableNodes = allWithoutMayBeUnreachable.
            select({~instruction => ((node<>) instruction.getSource()); });

        // output errors for each of those unreachable nodes
        foreach unreachableNode in unreachableNodes {
            error "unreachable code" -> unreachableNode;
        }
    }
}

The class constructs a {{Program} object in the constructor. {{Program ]]s are wrappers around the data flow graph and provide access to the graph, as well as to a set of predefined analyses on the graph. We will make use of one of them here in the checkForUnreachableNodes method. This method extracts all unreachable nodes from the graph (see comments in the code above) and reports errors for them. To be able to use the error statement, we have to annotate the method with the @CheckingMethod annotation.

To actually run the check, we call this method from a NonTypesystemRule for C functions:

```java
checking rule check_DataFlow {
    applicable for concept = Function as fct
    overrides false
    do {
        new DataflowUtil(fct.body).checkForUnreachableNodes();
    }
}
```

Inspecting the Program class, you can see the set of existing other data flow analyses: uninitialized reads (read before write), unreachable instructions (dead code) and unused assignments. We’ll look at what to do if those aren’t enough in the next section.

### 9.4 Building your own Analyzers

Data flow analysis is a non trivial topic. To build meaningful analyses you will probably need a background in this topic, or read up on the respective literature.

For the actual integration of custom data flow analyses, we will provide an additional article later.
Chapter 10

Dataflow

This cookbook should give you quick answers and guidelines when designing dataflow for your languages. For in-depth description of the typesystem please refer to the Dataflow section of the user guide.

10.0.1 Reading a value

The read operation instructs the dataflow engine that a particular value is read:

```java
data flow builder for LocalVariableReference {
    (node)->void {
        if (node.isVariableDefinedInThisMethod()) {
            read node.localVariableDeclaration
        }
    }
}
```

10.0.2 Writing a value

Similarly the write operation indicates that a value gets written to. In the example, a variable declaration with an initializer first executes the initializer through the `code for` command and then marks the node as being written the result of the initializer:

```java
data flow builder for LocalVariableDeclaration {
    (node)->void {
        nop
        if (node.initializer.isNotNull) {
            code for node.initializer
            write node = node.initializer
        }
    }
}
```

10.0.3 Code for

As seen above in the `LocalVariableDeclaration` dataflow or below in the `DotExpression` dataflow, the `code for` command indicates nodes that get executed and when. In the `DotExpression`, for example, `code for` the operand runs before the actual dot operation:

```java
data flow builder for DotExpression {
    (node)->void {
        code for node.operand
        code for node.operation
    }
}
```

10.0.4 Jump

Dataflow for the `TernaryOperatorExpression` is a very straightforward example of using both conditional and unconditional jumps. Once the condition gets evaluated we can optionally jump to the `ifFalse` branch. Similarly, once the `ifTrue` branch
is completed we unconditionally jump out of the scope of the node:

```java
data flow builder for TernaryOperatorExpression {
    (node)->void {
        code for node.condition
        ifjump before node.ifFalse
        code for node.ifTrue
        jump after node
        code for node.ifFalse
    }
}
```

10.0.5 Ifjump

The `WhileStatement` shows a more involved usage of the dataflow language. Not also the built-in detection of boolean constants. Trying to use `while(false)` will thus be correctly reported by MPS as a while-loop with unreachable body. This is thanks to the unconditional jump to `after node` if the constant is `false`.

```java
data flow builder for WhileStatement {
    (node)->void {
        code for node.condition
        if (node.condition.isInstanceOf(BooleanConstant)) {
            node<BooleanConstant> constant = node.condition : BooleanConstant;
            if (!(constant.value)) {
                jump after node
            }
        } else {
            ifjump after node
        }
        code for node.body
        { jump before node }
    }
}
```

10.0.6 Inserting instructions

The `TryStatement` has even more needs from the dataflow language. It must insert extra `ifjump` instructions to jump to a catch clause wherever the particular exception can be thrown in the code:
Notice, we’re using a few other helper methods and commands here - `get code for` to retrieve the dataflow instruction set for a node, `isRet`, `isJump` and `isNop` to exclude certain types of instructions (returns, jumps and no-operations respectively), `label` to create named places in the dataflow instruction set that we can jump to from elsewhere, and finally the `insert` command to insert a new command into an existing dataflow instruction set.

### 10.1 The data flow API

Check out the `jetbrains.mps.dataflow.framework` package for the classes that compose the API for accessing data flow information about code.
Chapter 11

Custom language aspect cookbook

Alongside the usual language aspects, such as Structure, Editor, Type-system, etc., it is possible for language authors to create custom language aspects (e.g. interpreter, alternative type-system, etc.), have them generated into a language runtime and then use these generated aspects from code.

This document will use the customAspect sample bundled with MPS to teach you how to define custom language aspects. The custom aspect feature is still under development, so some java-stuff that can be seen in this doc will gradually be replaced with a specific DSL constructions.

What is a custom aspect?
Language definitions in MPS can be thought of as a collection of aspects: structure, editor, typesystem, generator. Each of the aspects consists of declarations used by the corresponding aspect subsystem. For example, the type-system aspect consists of type-system rules and is used by the type-system engine.

Each aspect of a language is now defined in a separate aspect model. For example, the editor aspect of language L is defined in the L.editor model.

Each aspect is described using a set of aspect’s main languages. E.g. there’s the j.m.lang.editor language to describe the editor aspect.

Declarations in an aspect model may or may not be bound to some concept (like an editor of the concept is bound to a concept, mapping configuration in the generator aspect is not bound).

The aspect can be generated into a language’s aspect runtime, which represents this aspect at runtime, in other words, when the language is used.

Since version 3.3, MPS allows language authors to define new aspects for its languages.

11.1 Development cycle of custom aspects

1. Create a language to describe the aspect - you may reuse existing languages or create ones specific to the needs of the aspect. For example, each of the core MPS aspects uses its own set of languages, plus a few common ones, such as BaseLanguage or smodel.

2. Declare that this language (and maybe some others) describes some aspect of other languages - create an aspect descriptor

3. Develop a generator in the created language to generate aspect’s runtime classes (if needed)

4. Develop the aspect subsystem that uses the aspect’s runtime

We’ll further go through each step in detail.

11.2 Look around the sample project

If you open the customAspect sample project, you will get five modules.
Chapter 11. Custom Language Aspect Cookbook

81

The **documentation** language and its **runtime** solution are used to define new **documentation aspects** for any other language. **sampleLanguage** utilizes this new aspect - it uses it to document its concepts. The **sandbox** solution shows the **sampleLanguage**’s usage to view its documentation ability. The aspect subsystem is represented by the **pluginSolution**, which defines an action that shows the documentation for the concept of the currently focused node.

We won’t cover creation of the documentation’s language concepts as well as creation of sampleLanguage and sandbox solution in this cookbook, since these are core topics covered by all entry-level MPS tutorials. We will only focus on the specifics of creating a new language aspect.

### 11.3 Language runtime

Before we move on, let’s consider for a second how language aspects work at runtime.

- For each language in MPS, a language descriptor is generated (the class is called **Language.java**).

- Given an **aspect interface**, the language descriptor returns a **concrete implementation** of this aspect in this language (let’s call it **AspectDescriptor**).

- The **AspectDescriptor** can be any class with any methods, the only restriction is that it should implement a marker interface **ILanguageAspect**. We suggest that an **AspectDescriptor** contains no code except getters for entities described by this aspect.

This is how a typical language runtime looks like:

```java
public class Language extends LanguageRuntime {
    ...
    @Override
    protected <T extends ILanguageAspect> T createAspect(Class<T> aspectClass) {
        if (aspectClass == EditorAspectDescriptor.class) {
            return (T) new EditorAspectDescriptorImpl();
        }
        if (aspectClass == StructureAspectDescriptor.class) {
            return (T) new decl.structure.StructureAspectDescriptor();
        }
        return super.createAspect(aspectClass);
    }
}
```

The **createAspect()** method checks the type of the parameter expecting one of interfaces declared in aspects and returns a corresponding newly instantiated implementation.

This is how the interfaces defined in aspects may look like (this example is defined in the **Intentions** aspect):
11.3.1 Using the language aspects

Now, let’s suppose we would like to use some of the aspects. E.g. while working with the editor, we’d like to acquire a list of intentions, which could be applied to the currently selected node.

1. We first find all the language runtimes corresponding to the languages imported
2. then get the intentions descriptors for each of them
3. and finally get all the intentions from the descriptors and check their for applicability to the current node

The overall scheme is: Languages->LanguageRuntimes->Required aspect->Get what you want from this aspect
So your custom aspect need to hook into this discovery mechanism so that the callers can get hold of it.

11.4 Implementing custom aspect

Let’s look in detail into the steps necessary to implement your custom aspect using the customAspect sample project:

1. To make MPS treat some special model as a documentation aspect (that is our new custom aspect), an aspect declaration should be created in the documentation language. To do so, we create a plugin aspect in the language and import the customAspect language.

2. Create an aspect declaration in the plugin model of the language and fill in its fields. This tells MPS that this language can be used to implement a new custom aspect for other languages.
3. After making/rebuilding the documentation language, it’s already possible to create a **documentation aspect** in the sample language and create a doc concept in it.
4. Now, we should move to the language runtime in order to specify the functionality of the new aspect as it should work inside MPS. In our example, let’s create an interface that would be able to retrieve and return the documentation for a chosen concept. To do so, we create a runtime solution, add it as a runtime module of our documentation language and create an interface in it. Note that the runtime class must implement the ILanguageAspect interface. To satisfy our needs, the method must take a concept as a parameter and return a string with the corresponding documentation text.

6. In the generator for the documentation language we now need to have an implementation of the interface from above generated. A conditional root rule and the following template will do the trick and generate the documentation descriptor class:

```java
public interface DocumentationAspectDescriptor extends ILanguageAspect {
    String getConceptDocumentation(concept concept);
}
```

The condition ensures that the rule only triggers for the models of your custom aspect, i.e. in our case models that hold the documentation definitions (jetbrains.mps.samples.customAspect.sampleLanguage.documentation).

The useful feature here is the concept switch construction, which allows you to ignore the concept implementation details. It simply loops through all documented concepts (the LOOP macro) and for each such concept creates...
a matching case (exactly -> $[ConceptDocumentation]$) that returns a string value obtained from the associated ConceptDocumentation.

7. So we have an interface and an implementation class. Now, we need to tie them together - we have to generate the part of the LanguageRuntime class, which will instantiate our concept, i.e. whenever the documentation aspect is required, it will return the DocumentationDescriptor class. To understand how the following works, look at how the class Language.java is generated (see Language class in model j.m.lang.descriptor.generator.template.main). The descriptor instantiation is done by a template switch called InstantiateAspectDescriptor, which we have to extend in our new aspect language so that it works with one more aspect model:

Essentially, we’re adding a check for the DocumentationAspectDescriptor interface to the generated Language class and return a fresh instance of the DocumentationDescriptor, if the requested aspectClass is our custom aspect interface.

8. The only thing left is using our new aspect. For that purpose, an action needs to be created that will show documentation for a concept of a node under cursor on demand:

The jetbrains.mps.ide.actions@java_stub model must be imported in order to be able to specify the context parameters. The action must be created as part of a (newly created) plugin solution (more on plugin solutions at Plugin) with a StandalonePluginDescriptor and hooked into the menu through an ActionGroupDeclaration:

**standalone descriptor**

Generate initializer config | **false**

When true, generates a configuration file
9. This way the IDE **Code** menu will be enhanced.

10. Let’s now try it out! Rebuild the project, create or open a node of the `DocumentedConcept` concept in the `sandbox` solution and invoke the **Show Documentation** action from the **Code** menu:
Chapter 12

Icon description

The icon description language helps describing and instantiating icons for various MPS elements: concepts, actions etc. The language has two aims:
1. Provide a tool for quick icon prototyping (e.g. making new icons for concepts)
2. Make icons an extensible language construct

12.1 First impression

Wherever an icon is expected in the MPS language definition languages, you can enter a textual description of the desired icon instead of pointing to an existing .png file.

The \texttt{jetbrains.mps.lang.resources} contains two constructs:

- \texttt{icon\{}\texttt{\}} represents the image as an instance of \texttt{javax.swing.Icon} class.
- \texttt{iconResource\{}\texttt{\}} returns an instance of \texttt{jetbrains.mps.sm model.runtime.IconResource} class.

12.2 Creating icon prototypes

When describing an icon, you can get assistance from the \texttt{Create Icon} intention, which offers an automatic way to create a textual description of an icon and thus to prototype it quickly.
Invoking the intention will result in creating a straightforward icon definition.
This definition describes a circular icon with letter "X" inside of it. Upon regeneration the generated icon will take effect and shows up in the UI.
12.3 The language explained

An icon description consists of layers, each of which can be any of:

- a primitive graphical shape
- a custom image loaded from a file
- a character

These layers are then combined into a single image to represent the icon. These icon descriptions can be used:

- to specify icons in different places of the language definition languages - in concepts, actions, etc, where icons are expected
- in methods in the MPS UI that are supposed to return an Icon

12.3.1 Extending the language

The language is open for extension. To add new icon types, you need to create a new concept and make it implement the Icon interface. The Icon interface represents the desired icon and will get transformed to a .png file during the make process.

After generating a model, all Icons are collected from the output model and their generate() methods are called. These create .png files corresponding to the images described by the corresponding Icons. When an icon resource is requested (e.g. using the icon{} syntax), a resource referenced by Icon.getResourceId() method is loaded using the classloader of the corresponding module and converted into a Java Icon object.

12.3.2 icon{} vs iconResource{}

There are two constructs in the resources language to load resources. icon{} loads an image as an instance of javax.swing.Icon class, while iconResource{} returns an instance of jetbrains.mps.sm/model.runtime.IconResource class. The second one is used in core MPS aspects, which should not depend on the javax.swing package. All UI-related code uses icon{}. 
Chapter 13

Progress indicators

Actions by default block the UI for the duration of their execution. For actions that do not finish instantly it is advisable to indicate to the user that the action is active and perhaps also show a progress bar to give the user some feedback on the progress and time estimates. In this section we’ll see how to properly display and update progress indicators, how to allow the users to manually cancel actions as well as to send actions to the background.

The ProgressIndicators sample project will give you functional examples of actions that display progress bars, can be cancelled and run in the background.

13.1 Asynchronous tasks

You should aim at making all your actions quick to avoid freezing the UI. Long-lasting activities should be extracted from actions and placed into one or more asynchronously run tasks. Tasks extend the Task class from com.intellij.openapi.progress@java_stub and they can either display a modal dialog (Task.Modal class) or be sent to the background (Task.Backgroundable class). The task should be invoked on the EDT through the ApplicationManager.getApplication().invokeLater() method.
The task typically provides a `run()` method to perform the task and an `onCancel()` method to handle cancellation invoked by the user. The actual cancellation logic must be implemented by the task implementer - work in `run()` should be organised into chunks with checks for pending cancellation in between them. The `onCancel()` method will get fired only after `run()` finishes through cancellation and so serve mostly for cleaning up the partial work done by the task.

Non-cancelable modal tasks (the `canBeCancelled` parameter set to `false`) will force the user to wait until the action finishes completely and so should be used with care, perhaps for critical actions only, the cancellation of which would be difficult to handle properly.

### 13.2 Monitoring progress

The task’s `run()` method obtains an instance of IntelliJ’s progress indicator as a parameter. It is advisable to wrap it in `ProgressMonitorAdapter` coming from `jetbrains.mps.progress @java_stub`. `ProgressMonitorAdapter` represents the visual progress dialog and provides methods to set the actual progress bar value as well as and the labels shown to the user. It also holds the information regarding cancellation.
A few notes:

- The constructor of the Task includes a text that will be used as the title of the progress dialog
- The `start()` method provides a text to show above the progress bar and a number of steps/points to complete the task
- The `step()` method changes the text label displayed below the progress bar in the progress dialog
- The `advance()` method moves the progress bar to a new value by the specified number of steps/points
- Make the progress steps as small as possible to improve the user experience. Smaller steps provide smoother experience to the user

### 13.3 Running in the background

The `Task.Backgroundable` class should be used for tasks that can be processed in the background.
action BackgroundableProgressAction {
    mnemonic: <no mnemonic>
    execute outside command: false
    also available in: << ... >>

    caption: BackgroundableProgressAction
description: <no description>
icon: <no icon>

construction parameters

<< ... >>

action context parameters ( always visible = false )
    Project project key: PROJECT required
    MPSProject mpsProject key: MPS_PROJECT required

<update block>
execute(event)->void {
    boolean canBeCanceled = true;
    PerformInBackgroundOption showProgress = PerformInBackgroundOption.DEAF;

    Task.Backgroundable backgroundable = new Task.Backgroundable(BackgroundableProgressAction.this.project,
            public void run(@NotNull() final ProgressIndicator indicator) {

            @Override
            public void onCancel() {
                super.onCancel();
            }
            }
        );

    ApplicationManager.getApplication().invokeLater({ => ProgressManager.getInstance().run(backgroundable);

    
    
additional methods

private void doWork() {
    
}
}

A few notes:

- The backgroundable tasks may or may not allow cancellation
- The PerformInBackgroundOption interface allows you to create tasks that start in the foreground as well as in the background
- The user can move backgroundable tasks to the foreground as well as to the background
- The predefined constants for the PerformInBackgroundOption interface are DEAF (start in the foreground) and ALWAYS_BACKGROUND (start in the background, useful for non-critical actions that the user does not need to pay attention to, since no dialog would show up distracting the user, the UI remains fully usable all the time).

13.4 Proper locking when accessing resources

It is ok to obtain read and write locks to the MPS repository inside tasks as well as executing commands:
A few notes:

- When you need locking inside an action, prefer grouping of all modification into a single locking block
- Release the locks as soon as you do not need them to avoid blocking other potential user actions
- Do not use R/W actions or Commands in the EDT thread - this would lead to unpredictable updates of the progress and may even cause the UI to freeze

### 13.5 Undoable actions

Changes to the models require undoable actions, which can be executed through the `executeCommandInEDT()` method. However, you must not call the `ProgressIndicator`’s methods from within the command, since it itself is running in an EDT and all requests for the progress bar changes would be delayed until the command finishes. The recommended approach is to instruct the progress bar from the main action’s thread before invoking the command with `executeCommandInEDT()` and then block the main action’s thread until the command finishes, perhaps with a `CyclicBarrier` or other synchronisation primitive:
adapter.step("Do some work in command ...")
final CyclicBarrier barrier = new CyclicBarrier(2);

ModalProgressAction.this.mpsProject.getRepository().getModelAccess().executeCommandInEDT(new Runnable() {
    public void run() {
        try {
            model m = model/myModel/;
            m.new root node(MyRootConceptDeclaration);
        } finally {
            try {
                barrier.await();
            } catch (BrokenBarrierException e) {
                <no statements>
            } catch (InterruptedException e) {
                <no statements>
            }
        }
    }
});
try {
    barrier.await();
} catch (InterruptedException e) {
    <no statements>
} catch (BrokenBarrierException e) {
    <no statements>
}
adapter.advance(stepValue);
if (adapter.isCanceled()) { return; }

The Additional methods section can be leveraged to extract the locking implementation code out of the action’s body.

private void block(CyclicBarrier barrier) {
    try {
        barrier.await();
    } catch (BrokenBarrierException e) {
        <no statements>
    } catch (InterruptedException e) {
        <no statements>
    }
}
Chapter 14

Removing bootstrapping dependency problems

14.1 Definition:
Whenever you have

- a language that uses itself or
- a solution that uses a language and that language requires (i.e. indirectly or directly depends on) the same solution in order to work

you have bootstrapping dependency problem.

14.2 Why is this a problem

- you cannot rebuild the whole project from the ground up using build scripts
- you are forced to keep generated artifacts in the VCS repository

This bootstrapping dependency circle prevents your project from ever being rebuilt from the ground up completely. Instead, building your language requires the previous version of your solution build artifacts to have been generated. Similarly, building your solution requires the language to have been built first. Thus you cannot rebuild your whole project with a single command, you need to keep generated artifacts in VCS and the dependency structure of your project contains loops. The build scripts in MPS 2.5.4 and beyond can generate MPS code and so you no longer depend on the MPS workbench for code generation. Bootstrapping dependencies in the project structure, however, stand in the way.

14.3 The detection and elimination toolchain in MPS

We consider the inability to fully regenerate projects to be a serious issue. Serious enough so that we decided to provide tools that detect and help you eliminate all such dependencies in MPS starting with version 2.5.4. Nothing changes for rebuilding your projects in MPS. However, when rebuilding build scripts, MPS will detect and report circular dependencies between languages and modules that use them as errors.

When rebuilding a build script, you may get error reports like:

The output text contains a hint for solving the issue: right click on a module -> Analyze -> Analyze Module Dependencies
You will get the analyzer report panel displaying module dependencies. The gray (bootstrap dependency) indicator will highlight all problematic bootstrapping dependencies so you can quickly spot them. After selecting one in the left panel, the right panel will further detail the dependency circle. You see the dependency of the module on the language is listed, followed by the dependency of the language on the module. Now you can choose, which of the two dependency directions you want to remove in order to fix the problem.

14.4 Fixing the problem

Essentially, you need to break the dependency cycles. The first attempt you should make is to invoke Optimize Imports. If the dependencies between modules were only declared but not really used in code, this will remove them and potentially solve the bootstrapping problem. If the problem remains, you’ll have to play around with the analyzer a bit more.

14.4.1 Analyzing the bootstrapping dependencies

Imagine, for example, that we are fixing a frequent problem of a language uses itself for its own definition - a bootstrapping language.
We are only seeing a single problematic dependency direction in the right panel.

With the *Show Usages* pop-up menu command we get a third panel, which lists all the concrete occurrences in code, where the dependency is needed.

Unfortunately, you’ll have to decide yourself, which ones to remove and how. This will be a manual process.

### 14.5 Typical cases of bootstrapping dependencies

#### Case 1: Self-refering quotations

A very frequent example of a mistake that leads to a self-reference in a language is using the language in quotations, for example to define its own type-system rules. You declare your own type and then instantiate this type in the type-system aspect through quotation. Now the language needs itself to have been built in order to build.
Light quotations should be used in such situations instead of quotations. They provide lightweight quotation-like functionality without the need to use the editors of the nodes in quotation. For Light quotations to create appropriate nodes you need to supply the concept name and the required features. A handy intention for quick conversion from quotation to a Light quotation is also available.

Light quotations compared to quotations are slightly less convenient, but they avoids the bootstrapping problem. You may also favor them to quotations to better express your intent, when you need to combine quotations and anti-quotations heavily.

Case 2: A language uses itself in an accessory model

Languages can contain accessory models. If these accessory models use the containing language and they have not selected the "do not generate flag", we get a circular dependency between the accessory model and the owning language. Such a language cannot be rebuilt from scratch, since the accessory model needs to be generated together with the language. The solution to the problem is to move the accessory model to a separate solution. It is possible in MPS to have an accessory model reside outside of a language.

Case 3: A language uses itself inside patterns created by jetbrains.mps.lang.pattern

This problem is similar to the one with quotations. It does not prohibit rebuilding a language from scratch, but the cyclic dependency stays anyway. Currently, there is no straightforward solutions to that other than avoiding using the language inside patterns or Removing bootstrapping dependency problems.
Case 4: A runtime solution of a language uses the very same language

This one is not really a bootstrapping problem and will not prevent your build scripts from being build, however, it should still be considered to be a problematic and discouraged project dependency structure.

A runtime solution for a language typically contains the code executed by whatever gets generated from the models that use the language. So the runtime solution code logically belongs to the same abstraction level as the generated code, that is one level below the language level. It is a good practice to separate the lower-level code from the higher-level code and organize your project hierarchically.

The solution to the problem is to move the usages of language away from the runtime solution into another, separate module.

14.6 What else needs to be done

After attacking all problems individually, you need to optimize imports in your project and invoke the Reload modules from disk intention in the build script. This will clean up the dependency structure and allow your project to be fully rebuilt again.

14.7 A quick way to suppress the problem

A quick and dirty trick to suppress the error is to set the bootstrap property in the MPS settings section of the build script to true.
Consider this to be a last resort way to temporarily enable building your project. Proper solutions with eliminating all offending bootstrapping dependencies should be preferred.
Chapter 15

HowTo – Adding additional Tools (aka Views)

15.1 Adding additional Tools (aka Views)

This document describes how to build a new Tool (For Eclipse users, a Tool in MPS is like a View in Eclipse) for MPS. This can serve as an example to build arbitrary additional tools into MPS. This text emphasizes to aspects of Tool development: how to add a new tool to a language and the menu system, and how to synchronize the view with whatever is currently edited. In all other ways tools are just Swing UI programs. So for implementing sophisticated views, one needs Java Swing experience. In this tutorial we will not focus too much on the intricacies of building a tree view with Swing — an endeavour that is surprisingly non-trivial!

15.2 The Outline Tool itself

The MPS plugin language supports the definition of new Tools. Create a new instance of Tool and set the name. You can also give it a different caption and an icon. A default location (bottom, left, right, top) can also be defined.

We now add three fields to the Tool. The first one is used to remember the project, the second one connects the tool to MPS’s message bus, and the third one remember the ToolSynchronizer. The ToolSynchronizer updates the outline view in case the active editor changes. The message bus is IDEA platform’s infrastructure for event distribution. We use it for getting notified of selection changes in the currently active editor. More on these two later.

```java
private Project project;
private MessageBusConnection messageBusConn;
private ToolSynchronizer synchronizer;
```

We are then ready to implement the three main methods of a Tool: init, dispose and getComponent. Here is the code (with comments, please read them!):
15.3 An Action to Open the Tool

Actions are commands that live in the MPS UI. We have to add an action to open the outline view. Actions live in the plugins aspect of a language. Actions can define keyboard shortcuts, captions and icons, as expected. They also declare action parameters. These define which context needs to be available to be able to execute the action. This determines the presence of the action in the menu, and supports delivering that context into the action itself. In our case the context includes the currently opened project as well as the currently opened file editor.

```java
execute(event)->void {
    tool<Outline> outline = this.project.tool<Outline>;
    outline.setEditor(this.editor);
    outline.openTool(true);
}
```

15.4 An Action Group

Menu items are added via groups. To be able to add the Open Outline Action to the menu system, we have to define a new group. The group defines its contents (the action, plus a two separators) and it determines where in the menu it should go. Groups live in the plugin aspect as well.
15.5 Managing the Tool Lifecycle

The tool must play nicely with the rest of MPS. It has to listen for a number of events and react properly. There are two listeners dedicated to this task. The EditorActivationListener tracks which of the potentially many open editors is currently active, since the outline view has to show the outline for whatever editor is currently used by the user. It is also responsible to hooking up further listeners that deal with the selection status and model changes (see below). The ModelLifecycleListener tracks lifecycle events for the model that is edited by the currently active editor. The model may be replaced while it is edited, for example, by a revert changes VCS operation.

15.5.1 Editor Activation and Focus

The EditorActivationListener tracks which of the potentially many open editors is currently active. It is instantiated and hooked up to the MPS message bus by the tool as it is created by the following code (already shown above):

```java
this.messageBusConn = this.project.getMessageBus().connect();
this.messageBusConn.subscribe(FileEditorManagerListener.FILE_EDITOR_MANAGER, this.synchronizer);
```

In its constructor, it remember the outline tree. Then it sets up a new outline tree selection listener that listens to selection changes made by the user in the tree itself.

```java
this.outlineSelectionListener = new OutlineSelectionListener(this);
tree.addTreeSelectionListener(this.outlineSelectionListener);
```

It then sets up a listener to keep track of the model lifecycle (load, unload, replace) and it hooks up with the events collector (both explained below).

```java
modelLifecycleListener = new ModelLifecycleListener(tree);
eventsCollector = new ModelChangeListener(tree);
```

The EditorActivationListener implements FileEditorManagerListener, so it has to implement the following three methods:

```java
void fileOpened(FileEditorManager p0, VirtualFile p1);
void fileClosed(FileEditorManager p0, VirtualFile p1);
void selectionChanged(FileEditorManagerEvent p0);
```

In our case we are interested in the selectionChanged event, since we’ll have to clean up and hook up all kinds of other listeners. Here is the implementation.
public void selectionChanged(FileEditorManagerEvent event) {
    // read action is required since we access the model
    read action {
        FileEditor oldEditor = event.getOldEditor();
        // grab the old editor and clean it up if there is one
        if (oldEditor != null) {
            this.cleanupOldEditor(oldEditor);
        }
        // call a helper method that sets up the new editor
        this.newEditorActivated(event.getNewEditor());
    }
}

The `cleanupOldEditor` method removes existing listeners from the old, now inactive editor:

```java
private void cleanupOldEditor(FileEditor oldEditor) {
    // Downcast from IDEA level to MPS specifics and
    // grab the NodeEditor component
    IEditor oldNodeEditor = ((MPSFileNodeEditor) oldEditor).getNodeEditor();
    if (oldNodeEditor != null && this.editorSelectionListener != null) {
        // remove the selection listener from the old editor
        oldNodeEditor.getCurrentEditorComponent().getSelectionManager().
            removeSelectionListener(this.editorSelectionListener);
        // grab the descriptor of the model edited by the old editor
        // and remove the model listener (cleanup!)
        SModelDescriptor descriptor = oldNodeEditor.getEditorContext().getModel().
            getModelDescriptor();
        descriptor.removeModelListener(modelLifecycleListener);
        // remove the model edited by the old editor from the events collector
        // ...we are not interested anymore.
        eventsCollector.remove(descriptor);
    }
}
```

The next method hooks up the infrastructure to the newly selected editor and its underlying model. Notice how we use SNodePointer whenever we keep references to nodes. This acts as a proxy and deals with resolving the node in case of a model is replaced and contains a "weak reference" to the actual node, so it can be garbage collected if the model is unloaded. This is important to avoid memory leaks!
public void newEditorActivated(FileEditor fileEditor) {
    if (fileEditor != null) {
        // remember the current editor
        this.currentEditor = ((MPSFileNodeEditor) fileEditor);
        // grab the root node of that new editor...
        SNode rootNode = this.currentEditor.getNodeEditor().
            getCurrentlyEditedNode().getNode();
        // ...wrap it in an SNodePointer....
        SNodePointer treeRoot = new SNodePointer(rootNode);
        // and create a new outline tree model
        OutlineModel model = new OutlineModel(treeRoot);
        tree.setModel(model);

        // create a new selection listener and hook it up
        // with the newly selected editor
        this.editorSelectionListener = new EditorSelectionListener(tree, 
            outlineSelectionListener);
        SelectionManager selectionManager = this.currentEditor.getNodeEditor().
            getCurrentEditorComponent().getSelectionManager();
        selectionManager.addSelectionListener(this.editorSelectionListener);

        // This is needed to detect reloading of a model
        ((MPSFileNodeEditor) fileEditor).getNodeEditor().
            getEditorContext().getModel().getModelDescriptor().
            addModelListener(modelLifecycleListener);
        eventsCollector.add(this.currentEditor.getNodeEditor().
            getEditorContext().getModel().getModelDescriptor());
    } else {
        tree.setModel(new OutlineModel(null));
    }
}

15.5.2 Tracking the Model Lifecycle

The ModelLifecycleListener extends the SModelAdapter class provided
by MPS. We are interested in the model replacement and hence overload the
modelReplaced method. It is called whenever the currently held model is
replaced by a new one, e.g. during a VCS rever operation.

In the implementation, we create a new new tree model for the same root. While
this code looks a bit nonsensical, note that we use an SNodePointer
internally which automatically re-resolves the proxied node in the new model.
We also add ourselves as a listener to the new model’s descriptor to be notified
of subsequent model replacement events.

@Override
public void modelReplaced(SModelDescriptor descriptor) {
    tree.setModel(new OutlineModel(((OutlineModel) tree.getModel()).getRealRoot()));
    descriptor.addModelListener(this);
}

15.6 Synchronizing Node Selections

15.6.1 Tracking Editor Selection

This one updates the selection (and expansion status) of the tree as the
selected node in the currently active editor changes. We have already made sure
(above) that the outline view’s tree is synchronized with the currently active
editor.

The class extends MPS’ SingularSelectionListenerAdapter because we are
only interested in single node selections. We overwrite the
selectionChangedTo method in the following way:
protected void selectionChangedTo(EditorComponent component, SingularSelection selection) {
    // do nothing is disabled -- prevents cyclic, never ending updates
    if (disabled) { return; }
    // read action, because we access the model
    read action {
        // gran the current selection
        SNode selectedNode = selection.getSelectedNodes().get(0);
        // ... only if it has changed...
        if (selectedNode != lastSelection) {
            lastSelection = selectedNode;
            // disable the tree selection listener, once again to prevent cyclic
            // never ending updates
            treeSelectionListener.disable();
            // select the actual node in the tree
            tree.setSelectionPath(((OutlineModel) tree.getModel()).
                getPathTo(selectedNode));
            treeSelectionListener.enable();
        }
    }
}

15.6.2 Tracking Changes in the Model Structure

The tree structure in the outline view has to be updated if nodes are added, changed (renamed), moved or deleted in the editor. Tree change events can be quite granular, and to avoid overhead, MPS collects them into batches related to more coarse-grained commands. By using MPS' EventsCollector base class, we can get notified when a significant batch of events has happened, and then inspect the event list for those that we are interested in using a visitor. The ModelChangeListener performs this task. To do so, we have to implement the eventsHappened method. We get a list of events, and use a an inner class that extends SModelEventVisitorAdapter to visit the events and react to those that we are interested in.

protected void eventsHappened(List<SModelEvent> list) {
    super.eventsHappened(list);
    foreach evt in list {
        evt.accept(new ModelChangeVisitor());
    }
}

The ModelChangeVisitor inner class, which acts as the visitor to notify the tree, overrides visitPropertyEvent to find out about nodes whose properties have changed in the current batch. It then notifies all the listeners of the tree model.

public void visitPropertyEvent(SModelPropertyEvent event) {
    OutlineModel outlineModel = ((OutlineModel) tree.getModel());
    foreach l in outlineModel.getListeners() {
        l.treeNodesChanged(new TreeModelEvent(this, outlineModel.getPathTo(event.getNode())));
    }
}

It also overwrites visitChildEvent to get notified of child additions/deletions of nodes. Except that the API in JTree is a bit annoying, the following commented code should be clear about what it does:
```java
@Override
public void visitChildEvent(SModelChildEvent event) {
    // grab the model
    OutlineModel outlineModel = ((OutlineModel) tree.getModel());
    // we need the following arrays later for the JTree API
    Object[] child = new Object[]{event.getChild()};
    int[] childIndex = new int[]{event.getChildIndex()};
    // we create a tree path to the parent notify all listeners
    // of adding or removing children
    TreePath path = outlineModel.getPathTo(event.getParent());
    if (path == null) { return; }
    // notify the tree model's listeners about what happened
    foreach l in outlineModel.getListeners() {
        if (event.isAdded()) {
            l.treeNodesInserted(new TreeModelEvent(this, path, childIndex, child));
        } else if (event.isRemoved()) {
            l.treeNodesRemoved(new TreeModelEvent(this, path, childIndex, child));
        }
    }
}
```

15.6.3 The way back: Tracking Tree Selection

Tracking a JTree's selection happens by implementing Swing's TreeSelectionListener and overwriting it's valueChanged method the following way:

```java
public void valueChanged(TreeSelectionEvent event) {
    // don't do anything if disabled --- preventing cyclic updates!
    if (!disableEditorUpdate) {
        JTree tree = ((JTree) event.getSource());
        if (editorActivationListener.currentEditor != null &&
            tree.getLastSelectedPathComponent() instanceof SNodePointer) {
            // grab the selected tree node
            SNodePointer pointer = ((SNodePointer) tree.get:\
                lastSelectedPathComponent());
            // disable the editor selection listener to prevent
            // cyclic, never ending updates
            editorActivationListener.editorSelectionListener.disable();
            // update the selection in the editor
            editorActivationListener.currentEditor.getNodeEditor().
                getCurrentEditorComponent().selectNode(pointer.getNode());
            editorActivationListener.editorSelectionListener.enable();
        }
    }
}
```

15.7 Swing's Artifacts: Tree Model and Renderer

In this section I want to describe a couple of interesting MPS-specific aspects of the implementation of the Swing artifacts.

15.7.1 Tree Cell Renderer

The tree cell renderer is responsible for rendering the cells in the tree. It uses the getPresentation method on the nodes, and the IconManager to get (a cached version of) the icon for the respective node.
public Component getTreeCellRendererComponent(JTree tree, Object object,
   boolean selected, boolean expanded,
   boolean leaf, int row, boolean hasFocus) {
   if (object instanceof SNodePointer) {
      string presentation;
      DefaultTreeCellRenderer component;
      read action {
         SNode node = ((SNodePointer) object).getNode();
         presentation = node.getPresentation();
         component = ((DefaultTreeCellRenderer) renderer.getTreeCellRendererComponent(
            tree, presentation, selected, expanded, leaf, row, hasFocus));
         component.setIcon(IconManager.getIconFor(node));
      }
   } else {
      return renderer.getTreeCellRendererComponent(tree, object, selected,
         expanded, leaf, row, hasFocus);
   }
}

15.7.2 Tree Model

The tree model is interesting since we include only those children in the tree
node whose concept has been annotated with the
ShowConceptInOutlineAttribute attribute. It is stored in the
storeInOutline role. In the tree model, we have to filter the children of
a node for the presence of this attribute. Here is the respective helper method:

private List findAllChildrenWithAttribute(SNodePointer pointer) {
   List result = new ArrayList();
   SNode node = pointer.getNode();
   if (node == null) { return new ArrayList(); }
   foreach child in node.getChildren() {
      SNode attribute = AttributeOperations.getNodeAttribute(
         child.getConceptDeclarationNode(), "showInOutline");
      if (attribute != null) {
         result.add(child);
      }
   }
   return result;
Chapter 16

Debugger API

16.1 Debugger

MPS provides a Java API for creating a debugger engine to work with MPS.

16.1.1 Where to start

A good way to start writing a custom debugger engine is to study the Java Debugger plugin implementation in MPS. The source code is located in solutions jetbrains.mps.debugger.java.api and jetbrains.mps.debugger.java.runtime. The Debugger API itself can be found in jetbrains.mps.debugger.api.api. Solution jetbrains.mps.debugger.api.runtime can be studied to understand how API classes are used in MPS, but classes from this solution should not be referenced from a custom debugger code. Both Debugger API and Java Debugger are packaged as IDEA plugins for MPS, and so should be every custom implementation of Debugger API. Implementing a debugger for MPS requires some knowledge of IntelliJ IDEA, so IntelliJ IDEA Plugin Development page should also be studied. And there is of course a Build Language to help with plugin packaging.

16.1.2 Key classes

Interface jetbrains.mps.debug.api.IDebugger represents a debugger as a whole. It should be registered in a jetbrains.mps.debug.api.Debuggers application component. There is a default implementation, AbstractDebugger. IDebugger implementation is required to provide 2 things: a way to start an application under debugger and a way to create breakpoints.

For that MPS Debugger API has 2 interfaces:

1. jetbrains.mps.debug.api.AbstractDebugSessionCreator—its main job is to start an application under debug and create an instance of AbstractDebugSession, which corresponds to a single application, executed under debug. A state of debugger (either running or paused on a breakpoint or other) is represented by jetbrains.mps.debug.api.AbstractUiState. When a debug session transfers from one state to another, a new instance of AbstractUiState should be created and set to AbstractDebugSession. AbstractUiState holds the information about current threads and stack frames, visible variables etc.

2. jetbrains.mps.debug.api.breakpoints.IBreakpointsProvider is responsible for creating breakpoints, providing editing UI and persisting breakpoints state to and from XML. A breakpoint class should implement IBreakpoint. AbstractBreakpoint is a base class to extend. For breakpoints that can be set on some location (which basically means "on a node", like a breakpoint on line of code) exists an interface ILocationBreakpoint. Each breakpoint is of some kind (IBreakpointKind), for example there can be kinds for line breakpoints, field breakpoints or exception breakpoints.

Values and watchables

jetbrains.mps.debug.api.programState package contains a number of interfaces to reflect a state of the program. When paused on a breakpoint AbstractUIState has a list of threads that are running in an application (interface IThread). Each thread has a list of stack frames (IStackFrame), and each stack frame has a list of watchables (IWatchable) visible there. A watchable can be a local variable, or "this" object or a "static context". Each watchable has one value (IValue). In turn, each value can have a number of watchables. For example, Java local variable is a watchable, an object assigned to this variable is a value, and fields that this object has are watchables etc. This tree-like structure is what is shown in "Variables" section of the "Debug" tool window.
CHAPTER 16. DEBUGGER API

SourcePosition and IPositionProvider

jetbrains.mps.debug.api.source.IPositionProvider is an interface to customize the way source code (text) is mapped into a location inside of MPS (a text file or a node). It calculates an instance of jetbrains.mps.debug.api.source.SourcePosition for a location either given by an instance of a ILocation interface or by name of a unit name, file name and line number. There are two predefined providers: jetbrains.mps.debug.api.source.NodePositionProvider and jetbrains.mps.debug.api.source.TextPositionProvider. They are responsible for finding a node or a text position for a location.

SourcePosition is a class that contains a position inside MPS. This class has two implementations: jetbrains.mps.debug.api.source.NodeSourcePosition (contains a pointer to a node) and jetbrains.mps.debug.api.source.TextSourcePosition (contains a text file). Default line highlighting in MPS supports only these two kinds of positions. Providers are registered in jetbrains.mps.debug.api.source.PositionProvider class. Each provider has a key, associated with it, which tells whether this provider returns a NodeSourcePosition or a TextSourcePosition. Also, a provider accepts a specific DebugSession, and only providers that are shipped with the MPS debugger by default can accept all sessions.

Here is an example of a position provider:

```java
/*
NodePositionProvider is extended here in order to provide NodeSourcePosition.
It is not necessary to extend one of the default providers.
*/
public class MyCustomNodePositionProvider extends NodePositionProvider {
    private final Project myProject;

    public MyCustomNodePositionProvider(Project project) {
        myProject = project;
    }

    @Nullable()
    @Override
    public node<> getNode(@NonNls string unitName, @NonNls string fileName, int position) {
        node<> node = ... (some code that calculates a node);
        return node;
    }

    public void init() {
        // call this method on your debugger initialization
        PositionProvider.getInstance(myProject).addProvider(this, NodeSourcePosition.class.getName());
    }

    public void dispose() {
        // call this method on your debugger disposal
        PositionProvider.getInstance(myProject).removeProvider(this);
    }

    @Override
    public boolean accepts(AbstractDebugSession session) {
        // it is necessary to override this method and to accept only AbstractDebugSession instances that are
        return session instanceof MyCustomDebugSession;
    }
}
```

TraceInfoResourceProvider

jetbrains.mps.generator.traceInfo.TraceInfoCache. TraceInfoResourceProvider is an interface that allows to customise how paths to the trace.info files are calculated for a module. As an example, modules that are written in BaseLanguage have their trace.info files in classpath. The instance of an interface is given a module and a name of resource to find (for example, "jetbrains/mps/core/util/trace.info" if a caller needs a location of a trace.info file for a model jetbrains.mps.core.util) and should return an instance of java.net.URL. TraceInfoResourceProviders are registered/unregistered in TraceInfoCache instance using addResourceProvider/removeResourceProvider methods.
16.1.3 Final remarks

1. Variables tree in debugger tool window now uses a background thread for its updates. Several methods in the API provide support for that:

- `IValue.initSubvalues()` method, which is called in the background thread and should perform all the calculations necessary for acquiring subvalues of this value. The `IValue.getSubvalues()` method is only supposed to return the data, which have been calculated in the `initSubvalues()` method before.

- The `AbstractUiState.invokeEvaluation()` method, which is used by the debugger api to invoke all evaluation code (i.e. the code, which does some calculations inside the debugged program; specifically this method is used to invoke `IValue.initSubvalues()`). By default this method just schedules the command to one of the threads in the thread pool, but one can override it for some custom behavior.
Part II

Command line tools
Chapter 17

HowTo – MPS and Git

Old wiki markup is not supported, page content has been ignored.
Chapter 18

HTTP support plugin

All nodes in MPS models can now be referenced with a url. This gives you the ability to share pointers to code with others and continuous integration together with bug-tracking services may use direct references in their reports for your easy navigation.

The HTTP support plugin provides:

- co-operating via node URLs
- integration with YouTrack and TeamCity services
- a DSL for defining custom extensions to the IDEA Platform built-in server

18.0.4 Node URLs

You can create URL references to your code via the context menu. The created URL will be copied to the clipboard and then can be pasted wherever you want. On clicking it, MPS will handle it and open the referenced code.

If you want to get a URL of a node programmatically, you should use the `getURL` operation defined in `jetbrains.mps.ide.httpsupport` language.

18.0.5 YouTrack and TeamCity Integration

MPS listens for requests that come from YouTrack and TeamCity. Upon clicking the ‘Open in IDE’ button in a browser, MPS will open the requested file/node. Moreover, if you are trying to open a generated file, MPS will open its sources in a proper location.

18.0.6 Built-in server extensions

The features above are implemented using the IDEA Platform built-in server. If you have any other necessities to handle HTTP requests in the IDE, you can define an extension to it server via `jetbrains.mps.ide.httpsupport` language. Note that the defined extensions should be placed in a plugin solution. See Plugin for more information.
CHAPTER 18. HTTP SUPPORT PLUGIN

```java
request handler NodeOpener {
    query prefix: /node

    query parameters:
    ref converter: default(SNodeReference) required
    project converter: default(Project)

    handle always
    handle(request) -> void throws Exception {
        if (project != null) {
            project.getModelAccess().runWriteInEDT({ => HandlerUtil.openNode(request, project, ref); });
        } else {
            error "No project is available."
            request.send response (HandlerUtil.FAILURE_STREAM : image/gif);
        }
    }
}
```
19.1 Working with MPS and ant

Editing code of course requires the MPS editor. But generating models and running tests can be done from the command line to integrate it with automatic builds. Ant is used as the basis. In this section we explain how to use MPS from the command line via ant.

For all of the examples we use a build.properties file that defines the following two properties:

```
mps.home = // installation directory of MPS itself
mbeddr.home = // the root directory relative to which all other directories // to projects etc. are specified
```

This build.properties file is included in all the build scripts we discuss in this section. In addition, we have to define a set of MPS-specific tasks using the taskdef element in ant. Also, a couple of JVM options are reused over and over. Consequently, the following is a skeleton of all the build files we will discuss:

```
<project name="com.mbeddr.core build and test" default="all">
  <property file="build.properties"/>
  <path id="mps.ant.path">
    <pathelement location="${mps.home}/lib/mps-backend.jar"/>
    <pathelement location="${mps.home}/lib/jdom.jar"/>
    <pathelement location="${mps.home}/lib/log4j.jar"/>
    <pathelement location="${mps.home}/lib/mps-core.jar"/>
  </path>
  
  <taskdef resource="jetbrains/mps/build/ant/antlib.xml"
           classpathref="mps.ant.path"/>
  
  <jvmargs id="myargs">
    <arg value="-ea"/>
    <arg value="-Xss1024k"/>
    <arg value="-Xmx1024m"/>
    <arg value="-XX:MaxPermSize=92m"/>
    <arg value="-XX:+HeapDumpOnOutOfMemoryError"/>
  </jvmargs>

  <!-- here is the place where all the following example code goes -->

</project>
```
19.2 Building the Languages in a Project

We start by building the contents of a project. Here is the necessary ant code that has to be surrounded by the skeleton ant file shown above:

```xml
<property name="project.mpr" value="relative/path/to/project/project.mpr"/>
<target name="build-languages-in-project">
    <mps.generate>
        <jvmargs refid="myargs"/>
        <project file="${mbeddr.home}/${project.mpr}"/>
    </mps.generate>
</target>
```

All modules within the project are generated. If only a subset of the modules in the project should be generated, a modules filesset can be used. The following code generates all the languages in a project; typically they reside in the languages directory below the project. Note how we define a different property that points to the project directory as opposed to the project (.mps) file.

```xml
<property name="project.mpr" value="relative/path/to/project/project.mpr"/>
<property name="project.dir" value="relative/path/to/project"/>
<mps.generate>
    <jvmargs refid="myargs"/>
    <project file="${mbeddr.home}/${project.mpr}"/>
    <modules dir="${mbeddr.home}/${project.dir}/languages"/>
</mps.generate>
```

Sometimes a project needs access to other languages in order to be compilable. These can be added with library elements, whose dir attribute has to point to a directory that (directly, or further below) contains the required languages.

```xml
<property name="some.other.project.dir" value="relative/p/to/other/project"/>
<mps.generate>
    <jvmargs refid="myargs"/>
    <project file="${mbeddr.home}/${project.mpr}"/>
    <library name="irrelevantName" dir="${mbeddr.home}/${some.other.project.dir}/languages"/>
</mps.generate>
```

19.3 Generating/Building Solutions

Building solutions that contain code written in a DSL is not fundamentally different from building languages. However, it is important to set up the libraries correctly so they point to the directories that contain the languages used in the solutions.
19.4 Running Tests

MPS supports a special testing language that can be used for testing constraints, type system rules and editor functionality. These tests can be run from the UI using the Run option from the solution or model context menu (see the figure below).

These tests can also be run from the command line. Here is the code you need:
The important ingredients here are the two system properties
mps.junit.project and mps.junit.pathmacro.mbeddr.home. The first one
specifies the project that contains the tests. The second one is a bit more
involved. The syntax mps.junit.pathmacro.XXX sets a value for a path
variable XXX in an MPS project. To make the tests run correctly, there has
to be a TestInfo node in the project that points to the project file. This
one uses a path variable (defined in MPS settings) to make it portable between
different machines and various locations in the file system. The
mps.junit.pathmacro.mbeddr.home thingy is used to supply a value for the
macro from the command line.
Part III

Obsolete documents
Chapter 20

Build languages (obsolete)

20.1 Build Languages

MPS has a group of languages responsible for build process. Those languages are:

- `jetbrains.mps.buildlanguage` (build language) – an MPS counterpart of Apache Ant;
- `jetbrains.mps.build.property` (property language) – an extension to build language designed for writing property files;
- `jetbrains.mps.build.packaging` (packaging language) – a high-level language for building software packages, including languages made in MPS.
- `jetbrains.mps.build.custommps` (custom mps language) – an extension to packaging language, allowing to build your own custom MPS distribution.

20.2 Table of contents

- Build Language
  - Features
    - Typesystem
    - Expressions
    - External properties
  - Using Build Language
    - Running build language project
- Packaging language
  - Language structure
    - Defining Build Structure
    - Build configurations
    - Variables
    - Macro
    - Blocks
    - Language elements table
  - Using packaging language
    - Generating build scripts for your project
    - Generating files from build script
    - Running build scripts
- Custom MPS language
  - Building custom MPS for your project
    - Creating custom MPS build script using a wizard
    - Running custom MPS build script from MPS
    - Running custom MPS build script manually
    - Build results
Chapter 21

Packaging Language (obsolete)

21.1 Packaging Language

21.1.1 Introduction

When we think about building a project, we often do not think about a process of building, we usually imaging the result we want to have, for example, an archive with a program, documentation and libraries. We want a tool, which could take a description of the desired artifacts and generated a build script, for example an Ant file. That is one of the reasons why packaging language was made. The second reason for creating this language was lack of ability of simple deployment procedure for languages created in MPS.

21.1.2 Language Structure

This section describes the packaging language structure.

Defining Build Structure

Each packaging language script consists of two sections. The first one is a place for build properties, like base directory, and some build stuff, like variables or configurations. The second part is used for defining a build structure itself.

Let’s talk more about the second part. The build structure is written exactly like it should be in the resulting program distribution. There are several language elements you can use to describe it, for example, Folder, Zip, Jar and others (see all language elements). Some element can contain others, like in your distribution an archive could contain files in it.

On the above screenshot you can see an example of packaging language script. In the build structure part a Zip element project.zip contains Folder element project, which contains Folder element copied from folder sources, and this element also contains File copied from build.number. This means that we want a build script to create a zip archive project.zip with a folder project, copy sources folder with a copy of build.number inside into project folder. As you can see, the definition is very literal.

Language elements, used in build structure description, are listed in language elements table. Next sections are considering some special features of packaging language.

Build Configurations

Sometimes build should be done in several slightly different ways. For example, we want to create two types of build: for other developers – with sources and some additional tools and for users. Packaging language allow to deal with this problem
using build configurations. Let’s see how it is done for already mentioned example.

Project layout

```java
basedir $project.home$ set false

generate compilation script false

configurations dev, external

variables
<< ... >>

folder project from . / << ... >>

folder <no title> from . / devtools include in dev
<entries>

folder <no title> from . / languages
<entries>

folder <no title> from . / src include in dev
<entries>
```

On the above screenshot a build script example is shown. Configurations are defined in configurations sections. As it can be seen, the script defines two configuration: dev and external for two build types. Some build elements marked with them. Elements, which are not marked at all, are included in all types of build. Those elements are Folder "project" and Folder "languages". Other elements marked with configuration dev. They are included in build for developers and not included in build for users. Those elements are Folder "devtools" and Folder "src".

Variables

Variables allow to use properties, passed to the script through command line, in elements names. Variables are declared in variables section of the build script. To declare a variable, you must define it’s name and a name of ant property, which would be passed to the script by your build environment. On the below screenshot you can see an example of how variables declarations look like.

```java
variables
var build = ${ build.number }
var revision = ${ build.vcs.number }
var configuration = ${ teamcity.buildConfName }
```

The shown section defines three variables: build, revision and configuration.

Variables can be used in titles of all elements in script. For example, having variables, defined on a previous screenshot, one can write a script like shown on a screenshot.

```java
zip MyProject-build.zip

    echo build.number-build\n    date\n    revision.number-revision\n    configuration.name-configuration > build.info
```

In this script a build number is used in a name of a zip archive with the project and build parameters are written into a file build.info.

You can use not only variables, defined in the script, but also a number of predefined variables which are listed in the following table.

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Ant Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>basedir</td>
<td>basedir</td>
<td>Base directory of the script.</td>
</tr>
<tr>
<td>date</td>
<td>DSTAMP</td>
<td>Current date.</td>
</tr>
<tr>
<td>/</td>
<td>line.separator</td>
<td>System line separator.</td>
</tr>
<tr>
<td>:</td>
<td>file.separator</td>
<td>System file separator.</td>
</tr>
<tr>
<td>:</td>
<td>path.separator</td>
<td>System path separator.</td>
</tr>
</tbody>
</table>

Macro

Some packaging language elements, such as folder or copy allow to enter a source path – a name of some file or directory. In many cases leaving those paths absolute would be a bad idea. That is why packaging language afford an opportunity to use macro in names of files and directories.

Two types of macro exists: path variables defined in "Settings" -> "IDE Setting" -> "Path Variables" and predefined macro: basedir – a base directory of build script (specified in the script beginning) and mps_home – an MPS installation directory. Build script base directory declaration also allow usage of macros but with exception of basedir.
Macro used in a build script, will be generated to checked external properties of build language. This means, they will be checked in a build start and a build would fail, if their values were not specified.

Of course, packaging language does not force users using macro. To enter an absolute path you can select no macro item from macro menu.

**Blocks**

Blocks were introduced for structuring large builds. Essentially, a block is a part of build, a group of build elements. A block could be referenced from the main script. During generation, a contents of a block is copied to places if is referenced from. On the following screenshot there is a block defining a part of MPS build script.

Apart from a title and contents, a block can define which build script it is referenced from. This way variables from main build script could be referenced from the block.

**Language Elements Table**

This table gives a brief description of packaging language elements.

<table>
<thead>
<tr>
<th>Element</th>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antcall</td>
<td>antcall [target declaration] from [project] &lt;excludes patterns&gt; &lt;includes patterns&gt;</td>
<td>A call of an ant target.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- target declaration – a link to declaration of target to call.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- project – an build language concept in which to look for the specified target.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- patterns – a coma- or space-separated patterns to exclude or include (can be omitted).</td>
</tr>
<tr>
<td>Block Reference</td>
<td>-&gt; [block name]</td>
<td>A reference to a block. One could use blocks to split big scripts in parts.</td>
</tr>
<tr>
<td>Copy</td>
<td>copy from [source] &lt;excludes patterns&gt; &lt;includes patterns&gt;</td>
<td>A copy of folder contents.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- source – a folder, which contents needs to be copied.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- patterns – a coma- or space-separated patterns to exclude or include (can be omitted).</td>
</tr>
<tr>
<td>Package</td>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>Echo</td>
<td><code>echo [message] &gt; &lt;title&gt;</code></td>
<td>Echoes some message to the output stream or to the file.</td>
</tr>
<tr>
<td>File</td>
<td><code>file &lt;title&gt; from &lt;source&gt;</code></td>
<td>A file.</td>
</tr>
<tr>
<td>Folder</td>
<td><code>folder &lt;title&gt; from &lt;source&gt; &lt;excludes patterns&gt; &lt;includes patterns&gt; &lt;entries&gt;</code></td>
<td>A folder.</td>
</tr>
<tr>
<td>Jar</td>
<td><code>jar [title] &lt;excludes patterns&gt; &lt;includes patterns&gt; &lt;entries&gt;</code></td>
<td>A jar archive.</td>
</tr>
<tr>
<td>Module</td>
<td><code>module [name]</code></td>
<td>A packaged language, solution or descriptor.</td>
</tr>
<tr>
<td>Replace</td>
<td><code>replace &lt;title&gt; from [source] &lt;pairs token-value&gt;</code></td>
<td>Make replacements in file.</td>
</tr>
</tbody>
</table>
21.1.3 Using Packaging Language

This section describes usage of packaging language.

Generating Build Scripts For Your Project

MPS allow to easily generate a build script on a packaging language for languages in your project. This functionality is very useful when you have a really big project and you do not want to enumerate every module name by hands.

Let’s see, how it is done using complex language as an example.

Let’s open a complex language project which is located in file %MPS_HOME%/samples/complexLanguage/Complex.mpr.

A first step in generating build process is calling a build generation wizard, by selecting "New" -> "Build Script" in project popup menu.

View as: Logical View

At first, the wizard asks to select, whether a new solution would be created for a build script, or an old one should be picked.

For this example we select to create a solution named Complex.build.
The next step is to choose a model for a build script. If we selected an existing solution on previous step, we could use one of the models in that solution. But as we create a new solution for our build script, we have only one option: to create a new model. Let's call the new model Complex.build.

The last step is to select modules to include into build. We choose only two modules: the language jetbrains.mps.samples.complex and it's runtime solution jetbrains.mps.samples.complex.runtime.
Generating Files From Build Script

For generating files from a build script a "Generate Build Files" action can be used.
It generates the model with build and place output files to build script's base directory, so the files a ready to use. For complex language build script, created in previous section, those files would be:

<table>
<thead>
<tr>
<th>File</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex-default.xml</td>
<td>The main script for building default configuration.</td>
</tr>
<tr>
<td>Complex-compile.xml</td>
<td>Compilation script.</td>
</tr>
<tr>
<td>Complex-languages.xml</td>
<td>Script for packaging modules into jars.</td>
</tr>
<tr>
<td>Complex.properties</td>
<td>Property file.</td>
</tr>
</tbody>
</table>

Running Build Scripts

To run build scripts from MPS it has a run configuration "Packaging Script". It could be created via "Run/Debug Configurations" dialog:
Build script could also be runned from context menu:
The build output is shown in "Run" tool window:
Chapter 22

Build Language (obsolete)

Old wiki markup is not supported, page content has been ignored.
Chapter 23

Generating MPS models from Ant (obsolete)

23.1 Generating MPS models from Ant

23.1.1 Introduction

Users can generate their projects not only from MPS gui, but from Ant-scripts. For that MPS has a special Ant-task, mps.generate. The task classes are located in file %MPS_HOME%/lib/mps-backend.jar.

To use the task you should add a taskdef declaration into Ant-project:

```xml
<taskdef resource="jetbrains/mps/build/ant/antlib.xml" classpath="${mps.home}/lib/mps-backend.jar"/>
```

In the above code mps.home property points to base directory of MPS installation.

23.1.2 Parameters

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>compile</td>
<td>Indicates whether generated code requires compilation.</td>
<td>no</td>
<td>true</td>
</tr>
<tr>
<td>failonerror</td>
<td>Indicates whether generation errors will fail the build.</td>
<td>no</td>
<td>true</td>
</tr>
<tr>
<td>fork</td>
<td>Indicates whether to run generation in separate process.</td>
<td>no</td>
<td>true</td>
</tr>
<tr>
<td>loglevel</td>
<td>Controls the level of output messages shown. Should be one of &quot;error&quot;, &quot;warning&quot;, &quot;info&quot;, &quot;debug&quot;.</td>
<td>no</td>
<td>&quot;info&quot;</td>
</tr>
<tr>
<td>mps.home</td>
<td>Sets MPS location</td>
<td>Yes, if property mps.home is not specified in project.</td>
<td></td>
</tr>
<tr>
<td>usepropertiesasmacro</td>
<td>Indicates that task should use properties defined in projects as MPS macro.</td>
<td>no</td>
<td>false</td>
</tr>
</tbody>
</table>

23.1.3 Nested elements

- modules

```
Nested elements modules and model are filesets, specifying models and modules files to generate.
```

- model

```
Nested element project is a fileset of project files to generate.
```

- jvmargs

```
Nested element jvmargs is used to pass command-line arguments to java virtual machine. This element can only be used unless fork attribute is set to false. Each command-line argument is specified via arg inner. jvmargs task can be used outside of any target, for example it can be specified in the beginning of a project and then be referenced in several tasks.
```

- library

```
Nested element library is used to define libraries, required for generation (in MPS ide they are located in "Settings">"Ide Settings">"MPS Library Manager").
```

135
## Attribute Description

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Library name.</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>dir</td>
<td>Library base directory.</td>
<td>yes</td>
<td></td>
</tr>
</tbody>
</table>

### 23.1.4 Configuration and caches directories

Places where MPS keeps its configuration and caches can be changed via `jvmargs` nested element. To do so, add the following children to `jvmargs`:

```xml
<jvmargs>
  <arg value="-Didea.config.path=%CONFIGURATION%"/>
  <arg value="-Didea.system.path=%SYSTEM%"/>
</jvmargs>
```

Replace `%CONFIGURATION%` and `%SYSTEM%` with path to the desired configuration and system directories. Note that ant task should be started in fork mode (`fork` property set to `true`) for using `jvmargs` element.

### 23.1.5 JVM options

By default, since 2.0 Milestone 6, MPS tasks are started in fork mode, to be able to use more memory for generation. Default options provided to java machine are `-Xss1024k -Xmx512m -XX:MaxPermSize=92m -XX:+HeapDumpOnOutOfMemoryError -client`. All of them, except for `-client`, you can change via `jvmargs` element. If you experience `java.lang.OutOfMemoryError`, increase heap size using `-Xmx` argument, like it is set to 1024m below:

```xml
<mps.generate>  
  <jvmargs id="myargs">  
    <arg value="-Xmx1024m"/>  
  </jvmargs>  
  ...
</mps.generate>
```

### 23.1.6 Examples

Generate project MyProject.

```xml
<mps.generate>  
  <project file="MyProject.mpr"/>  
</mps.generate>
```

Generate all models found in directory project1/languages/language1/languageModels. Fail if an error occurs during generation. Show debug messages.

```xml
<mps.generate failonerror="true" loglevel="debug">  
  <model dir="project1/languages/language1/languageModels"/>  
</mps.generate>
```

Generate model project1/languages/language1/languageModels/editor.mps. Note that base directory of language1, owner of model editor.mps, should be listed as library.

```xml
<mps.generate>  
  <library name="language1" dir="project1/languages/language1/"/>  
  <model file="project1/languages/language1/languageModels/editor.mps"/>  
</mps.generate>
```

Generate projects project1, all modules from project2/languages/ folder and all models from project3/languages/language3/languageModels/ in separate process with `-Xmx` parameter of jvm set to 512m. Before generation load library someLibrary from lib/library1.

```xml
<mps.generate>  
  <library name="language1" dir="project1/languages/language1/"/>  
  <library name="language3" dir="project3/languages/language3/"/>  
  <model file="project1/languages/language1/languageModels/editor.mps"/>  
  <model file="project3/languages/language3/languageModels/editor.mps"/>  
</mps.generate>
```
<jvmargs id="myargs">
    <arg value="-Xmx512m"/>
</jvmargs>

<mps.generate fork="true">
    <jvmargs refid="myargs"/>
    <library name="somelibrary" dir="lib/library1"/>
    <project file="project1.mpr"/>
    <modules dir="project2/languages/"/>
    <model dir="project3/languages/language3/languageModels/"/>
</mps.generate>

23.2 Testing your models on TeamCity

MPS offers Ant-tasks for testing MPS models on TeamCity. Executed, those tasks output specially formatted messages, signalling tests start, finish, failure.

There are two types of tests: generation tests and broken references tests.

23.2.1 Generation Tests

For generation tests mps.test.generation task is used. It has the same attributes and nesteds as mps.generate task, but also adds few more.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Required</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>invoketests</td>
<td>Automatically invokes unit tests in generated models.</td>
<td>no</td>
<td>false</td>
</tr>
<tr>
<td>showdiff</td>
<td>Calculate difference between files on disk and generated files and fails test if there is one.</td>
<td>no</td>
<td>false</td>
</tr>
</tbody>
</table>

When running mps.test.generation task, each model is considered as single thing to test. That means, if models model1 and model2 are loaded, task will run two tests named "generating model1" and "generating model2". When nested project is used, task will load a project and read its "test configuration" to find out which models to generate. If no test configuration can be found for project, task will print a warning message. To override this behaviour one need to set attribute wholeproject to true in project inner. Then all models from project will be loaded.

23.2.2 Difference between mps.test.generation and mps.generate

mps.generate does generation and compilation (if the corresponding option is set) of specified models, modules and projects. All used libraries have to be compiled before generation starts, so you have to set compile attribute to true for all libraries, distributed in sources. mps.generate always ensure that dependencies of a module are generated before module generation (if they are listed among targets to generate). mps.test.generation does not do so. You have to ensure that everything you need is generated and compiled before the task starts. It also does not save anything (like generated sources or compiled classes) on the disk and does not load compiled classes. It does only testing.

23.2.3 Broken References Tests

For testing broken references mps.test.broken.references task is used. This task’s parameters are the same as in mps.generate task.
Chapter 24

Custom MPS Language (obsolete)

24.1 Custom MPS Language

24.1.1 Introduction

The purpose of custom MPS language is to allow a programmer to create MPS distribution containing his own languages, so that users would be able to use those languages straight away.

The language adds two constructions into packaging language: mps-build and library. library is a special folder inside of MPS build, where packaged modules are located. While building MPS, a special configuration file is created and added to the build where all library folders are listed, so MPS will know, where to look for modules.

24.1.2 Building custom MPS for your project

In this section we will build a custom MPS distribution for a sample project. Our project will have three simple languages.
Creating custom MPS build script using a wizard

To create a custom MPS build script for the project, select "New" -> "Custom MPS Build Script" from project popup menu.

This action starts a wizard. The first step is to select a location of special zip file, MPS-buildTools.zip, containing files required to build MPS. This file can be downloaded from http://jetbrains.com/mps/download.

Next, the wizard asks to select, whether a new solution would be created for a build script, or an old one should be picked. For this example we select to create a solution named SimpleProject.build.
The next step is to choose a model for a build script. If we selected an existing solution on previous step, we could use one of the models in that solution. But as we create a new solution for our build script, we have only one option: to create a new model. Let’s call the new model `SimpleProject.build`.

The last step is to select modules, included in build. Let’s select all languages in project and exclude a solution `jetbrains.mps.simple1.sandbox`.
After pressing "OK" button the wizard creates a build script.

As you can see, resulting script has two libraries: simplenamespace library for modules in a namespace simpleNamespace and simpleproject library for modules with empty namespace.

Running custom MPS build script from MPS

To run custom MPS build script use "Custom MPS Script" run configuration.
You can also use a context menu item:
Running custom MPS build script manually

To run custom MPS script by hands you should generate build files by selecting "Generate Build Files" action from model's popup menu.
The files are generated into a folder inside of build script base directory. In our case it's folder "build" in project base directory.

```
~/MPSProjects/SimpleProject/build$ ls
help-build.xml installer.nsi mps.sh SimpleProject-compile.xml SimpleProject-default.xml SimpleProject.properties
Info.plist mps.bat mps.vmoptions SimpleProject-default-dist.xml SimpleProject-languages.xml
```

Here is the table with description of generated files.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help-build.xml</td>
<td>Script with different utility targets.</td>
</tr>
<tr>
<td>Info.plist</td>
<td>Configuration file for Mac OS X.</td>
</tr>
<tr>
<td>installer.nsi</td>
<td>Script for generating windows installer.</td>
</tr>
<tr>
<td>mps.bat</td>
<td>Windows startup file.</td>
</tr>
<tr>
<td>mps.sh</td>
<td>Unix startup file.</td>
</tr>
<tr>
<td>{YourProjectName}-compile.xml</td>
<td>File with java VM options.</td>
</tr>
<tr>
<td>{YourProjectName}-default-dist.xml</td>
<td>Compilation script.</td>
</tr>
<tr>
<td>{YourProjectName}-default.xml</td>
<td>Main script, packaging MPS for different platforms.</td>
</tr>
<tr>
<td>{YourProjectName}-default.xml</td>
<td>Script for creating folder with MPS.</td>
</tr>
<tr>
<td>{YourProjectName}-languages.xml</td>
<td>Script for packaging modules into jars.</td>
</tr>
<tr>
<td>{YourProjectName}.properties</td>
<td>Properties file.</td>
</tr>
</tbody>
</table>

To start custom MPS build you should use {YourProjectName}-default-dist.xml file (in our case it's SimpleProject-default-dist.xml). It requires mps_home property to be set to the directory, containing MPS. To run build from console you can use the following command.

```
ant -f {YourProjectName}-default-dist.xml -Dmps_home={PathToMPS}
```
Build results

Custom MPS build script creates folder "artifacts" in script base directory (in our case it's project home directory) with MPS build for different platforms. They all are marked with number of MPS build used when running build script.

<table>
<thead>
<tr>
<th>Artifact Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPS-{build.numer}.zip</td>
<td>Crossplatform build with bat and sh startup scripts.</td>
</tr>
<tr>
<td>MPS-{build.numer}-windows.exe</td>
<td>Windows installer (created only when running build script on windows).</td>
</tr>
<tr>
<td>MPS-{build.numer}-macos.zip</td>
<td>Zip for MacOsX.</td>
</tr>
<tr>
<td>MPS-{build.numer}-linux.tar.gz</td>
<td>Gzipped tar archive for Linux.</td>
</tr>
</tbody>
</table>

Let's unpack MPS-{build.numer}.zip and see, what is inside it.

~/MPSProjects/SimpleProject/artifacts/MPS$ ls
about.txt  build.number  entryPoints.xml  license  mps.sh  plugin  readme.mps.txt  samples.zip  Sim
bin        core          lib        mps.bat      platform     plugins     readme.txt     simpleNamespace  workbench

Here we see folders "simpleNamespace" and "SimpleProject".

~/MPSProjects/SimpleProject/artifacts/MPS/simpleNamespace$ ls
jetbrains.mps.simple2.mpsarch.jar  jetbrains.mps.simple3.mpsarch.jar

Folder "simpleNamespace" contains two packed languages jetbrains.mps.simple2 and jetbrains.mps.simple3.

~/MPSProjects/SimpleProject/artifacts/MPS/SimpleProject$ ls
jetbrains.mps.simple1.mpsarch.jar

Folder "SimpleProject" contains packed language jetbrains.mps.simple1.

Let's start MPS to make sure that those languages are available in it. Select "File" -> "Settings" -> "Ide Settings" -> "MPS Library Manager". You can see, that there are new simplenamespace and simpleproject libraries pointing to "simpleNamespace" and "SimpleProject" directories respectively.

Let's create a test project and try to write something on jetbrains.mps.simple1 language. Let's create a solution "Test.sandbox" with a model "Test.sandbox.sandbox" and import jetbrains.mps.simple1 into it. All three languages we wanted to pack are loaded by MPS and available for import.
Now we can create SimpleConcept1 instance in a model.