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 Introduction to MPS 1

1 MPS User's Guide 2
  1.1 MPS User Guide for Language Designers .................................................. 2
  1.1.1 Beginner's Fast Track to MPS ................................................................. 2
  1.1.2 Structure of the User Guide ................................................................. 2

2 Basic notions 9
  2.1 Basic notions ............................................................................................... 9
    2.1.1 Abstract Syntax Tree (AST) ................................................................. 9
    2.1.2 Node ...................................................................................................... 9
    2.1.3 Concept ............................................................................................... 10
    2.1.4 Language ............................................................................................. 11
    2.1.5 Generator ........................................................................................... 11

3 MPS project structure 13
  3.1 Introduction ............................................................................................... 13
  3.2 Project ....................................................................................................... 13
  3.3 Models ....................................................................................................... 13
  3.4 Modules ..................................................................................................... 13
    3.4.1 Module meta information ..................................................................... 13
    3.4.2 Solutions .............................................................................................. 14
    3.4.3 Languages ........................................................................................... 14
    3.4.4 Generators .......................................................................................... 15
    3.4.5 DevKits ............................................................................................... 15
  3.5 Projects ...................................................................................................... 15
    3.5.1 Java compilation .................................................................................. 15

4 MPS user guide for DSL users 16
  4.1 Before you start .......................................................................................... 16
  4.2 Essentials ................................................................................................... 16
  4.3 Setting things up ........................................................................................ 16
    4.3.1 Configuration ....................................................................................... 16
  4.4 Editor specialities ....................................................................................... 17
  4.5 Resolving difficulties, understanding reported errors ............................... 17
  4.6 VCS integration ......................................................................................... 17

5 MPS user guide for Java developers (IntelliJ IDEA) 18
  5.1 Before you start .......................................................................................... 18
  5.2 Essentials ................................................................................................... 18
  5.3 Setting things up ........................................................................................ 18
  5.4 Editor specialities ....................................................................................... 19
  5.5 Resolving difficulties, understanding reported errors ............................... 19
  5.6 Where to find language plugins ................................................................. 19
  5.7 Platform Languages .................................................................................. 19
# CONTENTS

5.8 IDE integration - Java interoperability, cross-navigation, joint compilation ........................................ 19  
5.9 VCS integration ........................................................................................................................................ 19  
5.10 Debugger ................................................................................................................................................ 20  

6 MPS screen-casts ........................................................................................................................................... 21  
6.0.1 Customer stories .................................................................................................................................. 21  
6.0.2 First steps .............................................................................................................................................. 21  
6.0.3 Introduction to MPS .............................................................................................................................. 21  
6.0.4 Commanding the MPS IDE .................................................................................................................. 22  
6.0.5 MPS basics .......................................................................................................................................... 22  
6.0.6 Interoperability .................................................................................................................................. 23  
6.0.7 Language Extension .............................................................................................................................. 23  
6.0.8 Advanced aspects of language definition ............................................................................................ 24  
6.0.9 Tutorials .............................................................................................................................................. 24  

II Using MPS .................................................................................................................................................... 25  
7 MPS Java compatibility .................................................................................................................................. 26  
8 Commanding the editor ................................................................................................................................ 27  
8.0.10 Most useful key shortcuts .................................................................................................................. 28  
8.0.11 A complete listing ................................................................................................................................ 28  
9 IDE configuration .......................................................................................................................................... 29  
9.0.12 Plugins ............................................................................................................................................... 30  
10 Finding your way out ..................................................................................................................................... 31  
10.0.13 Reflective editor .................................................................................................................................. 31  
10.0.14 Node Explorer .................................................................................................................................... 31  
10.0.15 Check out the type of the node .......................................................................................................... 32  
10.0.16 Check the concept of the node under carret ...................................................................................... 33  
10.0.17 Check the editor of the node under carret ......................................................................................... 33  
10.0.18 Type-system Trace .............................................................................................................................. 33  
10.0.19 Investigate the structure .................................................................................................................... 33  
10.0.20 Importing elements ............................................................................................................................. 35  
10.0.21 Save transient models ......................................................................................................................... 35  
10.0.22 Why the heck do I get this error/warning? .......................................................................................... 36  
11 Where to find language plugins ................................................................................................................... 38  
12 Version Control ............................................................................................................................................. 40  
12.1 Compare two nodes .................................................................................................................................... 40  
12.2 VCS menu ................................................................................................................................................ 40  
12.3 VCS configuration .................................................................................................................................... 41  
12.4 Changes View .......................................................................................................................................... 42  
12.5 VCS Add-ons .......................................................................................................................................... 42  
12.5.1 What are VCS Add-ons ...................................................................................................................... 43  
12.5.2 Different VCS Add-ons ....................................................................................................................... 43  
13 Using MPS Debugger .................................................................................................................................... 44  
13.1 Using MPS Debugger ................................................................................................................................. 44  
13.1.1 Execution ........................................................................................................................................... 44  
13.1.2 Breakpoints ........................................................................................................................................ 46  
13.1.3 Runtime ............................................................................................................................................ 48  
13.1.4 Expressions ....................................................................................................................................... 48  
14 Migrations ..................................................................................................................................................... 51  
14.1 Language version ....................................................................................................................................... 51  
14.2 Migration assistant ..................................................................................................................................... 52  
14.3 Defining language migrations ................................................................................................................... 53  
14.3.1 Numbering of languages and migrations ............................................................................................ 53  
14.3.2 Structure of a migration ....................................................................................................................... 55  
14.3.3 Ordering of migration scripts .............................................................................................................. 55  
14.3.4 Languages for defining migrations ...................................................................................................... 55
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.4 Concept replacement</td>
<td>56</td>
</tr>
<tr>
<td>14.5 Defining project migrations</td>
<td>56</td>
</tr>
<tr>
<td>14.5.1 Adding a new project migration</td>
<td>57</td>
</tr>
<tr>
<td>14.5.2 Saving data from project migrations</td>
<td>57</td>
</tr>
<tr>
<td>14.6 Multiple branches</td>
<td>57</td>
</tr>
<tr>
<td>14.7 Migration Ant Task</td>
<td>57</td>
</tr>
<tr>
<td>14.8 Examples</td>
<td>57</td>
</tr>
<tr>
<td>14.9 Changes made by migrations in Local History view</td>
<td>57</td>
</tr>
<tr>
<td>14.10 Migration assistant in IntelliJ IDEA</td>
<td>59</td>
</tr>
<tr>
<td>15 Using Migrations with branching</td>
<td>61</td>
</tr>
<tr>
<td>15.1 Using migrations with branching</td>
<td>61</td>
</tr>
<tr>
<td>15.1.1 The main rule of painless merge</td>
<td>61</td>
</tr>
<tr>
<td>16 Console</td>
<td>63</td>
</tr>
<tr>
<td>16.0.2 Console commands</td>
<td>64</td>
</tr>
<tr>
<td>16.0.3 Scopes of Console queries</td>
<td>65</td>
</tr>
<tr>
<td>16.0.4 Copying nodes into Console</td>
<td>66</td>
</tr>
<tr>
<td>III Defining Languages</td>
<td>67</td>
</tr>
<tr>
<td>17 Structure</td>
<td>68</td>
</tr>
<tr>
<td>17.1 Concepts and Concept Interfaces</td>
<td>68</td>
</tr>
<tr>
<td>17.1.1 Concepts inheritance</td>
<td>69</td>
</tr>
<tr>
<td>17.2 Concept members</td>
<td>69</td>
</tr>
<tr>
<td>17.2.1 Properties</td>
<td>69</td>
</tr>
<tr>
<td>17.2.2 References</td>
<td>69</td>
</tr>
<tr>
<td>17.2.3 Children</td>
<td>70</td>
</tr>
<tr>
<td>17.2.4 Specialized references and children</td>
<td>70</td>
</tr>
<tr>
<td>17.2.5 Alias</td>
<td>70</td>
</tr>
<tr>
<td>17.3 Constrained Data Types</td>
<td>70</td>
</tr>
<tr>
<td>17.4 Enumeration Data Types</td>
<td>70</td>
</tr>
<tr>
<td>17.4.1 Presentation vs. Value vs. Identifier</td>
<td>71</td>
</tr>
<tr>
<td>17.4.2 Deriving identifiers automatically</td>
<td>71</td>
</tr>
<tr>
<td>17.4.3 Programmatic access</td>
<td>71</td>
</tr>
<tr>
<td>17.5 Attributes</td>
<td>72</td>
</tr>
<tr>
<td>18 SModel language</td>
<td>73</td>
</tr>
<tr>
<td>18.1 Treatment of null values</td>
<td>73</td>
</tr>
<tr>
<td>18.2 Types</td>
<td>73</td>
</tr>
<tr>
<td>18.3 Operation parameters</td>
<td>73</td>
</tr>
<tr>
<td>18.4 Comparison</td>
<td>74</td>
</tr>
<tr>
<td>18.5 Queries</td>
<td>74</td>
</tr>
<tr>
<td>18.5.1 Getting nodes by name</td>
<td>74</td>
</tr>
<tr>
<td>18.5.2 Getting concepts by name</td>
<td>74</td>
</tr>
<tr>
<td>18.5.3 Features access</td>
<td>74</td>
</tr>
<tr>
<td>18.5.4 Null checks</td>
<td>75</td>
</tr>
<tr>
<td>18.5.5 IsInstanceOf check and type casts</td>
<td>75</td>
</tr>
<tr>
<td>18.5.6 Node collection cast</td>
<td>75</td>
</tr>
<tr>
<td>18.5.7 Parent</td>
<td>75</td>
</tr>
<tr>
<td>18.5.8 Children</td>
<td>76</td>
</tr>
<tr>
<td>18.5.9 Sibling queries</td>
<td>76</td>
</tr>
<tr>
<td>18.5.10 Ancestors</td>
<td>76</td>
</tr>
<tr>
<td>18.5.11 Descendants</td>
<td>76</td>
</tr>
<tr>
<td>18.5.12 Containing root and model</td>
<td>76</td>
</tr>
<tr>
<td>18.5.13 Model queries</td>
<td>76</td>
</tr>
<tr>
<td>18.5.14 Search scope queries</td>
<td>77</td>
</tr>
<tr>
<td>18.5.15 The Concept literal</td>
<td>77</td>
</tr>
<tr>
<td>18.5.16 Concept operation</td>
<td>77</td>
</tr>
<tr>
<td>18.5.17 Migrating away from deprecated types</td>
<td>77</td>
</tr>
<tr>
<td>18.5.18 Concept hierarchy queries</td>
<td>77</td>
</tr>
<tr>
<td>18.5.19 The hasRole operation</td>
<td>77</td>
</tr>
<tr>
<td>18.5.20 Link queries</td>
<td>78</td>
</tr>
</tbody>
</table>
CONTENTS

18.5.21 Containing link queries .................................................. 78
18.5.22 Reference operations ..................................................... 78
18.5.23 ........................................................................... 79
18.5.24 Downcast to lower semantic level ..................................... 79
18.6 Modification operations ....................................................... 79
18.6.1 Feature changes ............................................................. 79
18.6.2 New node creation ......................................................... 79
18.6.3 Copy ......................................................................... 80
18.6.4 Replace with ............................................................... 80
18.6.5 Delete and detach operations ........................................... 80
18.7 smodel.query language ....................................................... 80
18.7.1 Operation parameters ..................................................... 80
18.7.2 Scope specification ........................................................ 81
18.7.3 Queries defined in the smodel.query language ................. 81
19 Open API - accessing models from code .................................. 82
19.1 Using Open API ............................................................... 82
19.2 Some API details ............................................................... 83
19.2.1 Meta-model level .......................................................... 83
19.2.2 Model level ................................................................. 83
19.2.3 Open API usage patterns .............................................. 83
19.3 Using commands .............................................................. 83
19.4 Concurrent access ............................................................ 84
19.4.1 Model-locking example ................................................ 84
19.5 Custom persistence .......................................................... 84
19.5.1 Alternative file format ................................................... 84
19.5.2 Alternative storage ....................................................... 85
19.5.3 Custom Find Usage and Navigation participants .............. 85
20 Quotations .................................................................. 86
20.1 Quotations ................................................................. 86
20.2 Antiquotations ............................................................... 87
20.3 Light quotations (quotation builders) .................................... 87
21 Pattern .................................................................... 89
21.1 The Pattern language ....................................................... 89
21.2 PatternExpression .......................................................... 89
21.3 Examples .................................................................. 89
21.4 Using patterns .............................................................. 90
21.4.1 Match statement ........................................................ 90
21.4.2 Other usages ............................................................ 90
22 Using model & module dependencies FAQ ............................... 91
23 Constraints ................................................................ 93
23.1 Can be child/parent/ancestor/root ..................................... 93
23.2 Property constraints ....................................................... 94
23.2.1 Example - customizing the description in the completion menu .................. 94
23.3 Referent constraints ....................................................... 95
23.4 Default scope .............................................................. 96
24 Scopes .................................................................. 97
24.1 Two ways ................................................................. 97
24.2 Inherited scopes ........................................................... 97
24.2.1 Scope helper implementations .................................... 102
24.2.2 VariableReference ..................................................... 102
24.3 Using reference scope .................................................... 104
25 Behavior ................................................................ 106
25.1 Concept instance methods .............................................. 106
25.1.1 Overriding behavior methods ..................................... 106
25.2 Concept constructors .................................................... 106
25.3 Concept static methods .................................................. 106
## 26 Editor

**26.1 Editor Overview** .......................................................... 108  
26.2 Types Of Cell Models .................................................. 109  
26.2.1 Editor Components and editor component cells ................. 111  
26.3 Cell layouts ........................................................................ 111  
26.4 Styles ................................................................................. 112  
26.5 Style properties ................................................................. 113  
26.5.1 Boolean style properties .................................................. 113  
26.5.2 Padding properties .......................................................... 113  
26.5.3 Punctuation properties ..................................................... 114  
26.5.4 Color style properties ...................................................... 114  
26.5.5 Indent layout properties ................................................... 114  
26.5.6 Other style properties ..................................................... 115  
26.5.7 Style properties propagation .......................................... 116  
26.5.8 Custom styles .................................................................... 116  
26.5.9 Style inheritance ............................................................. 116  
26.5.10 Style priorities ............................................................... 118  
26.6 Cell actions ........................................................................... 118  
26.6.1 Key maps ........................................................................ 118  
26.6.2 Action maps ..................................................................... 119  
26.7 Cell menus ........................................................................... 119  
26.7.1 Property values menu part ................................................ 119  
26.7.2 Property postfix hints menu part ....................................... 119  
26.7.3 Primary replace child menu ............................................. 120  
26.7.4 Primary choose referent menu ........................................... 120  
26.7.5 Replace node menu (custom node’s concept) .................... 120  
26.7.6 Replace child menu (custom child’s concept) .................... 120  
26.7.7 Replace child menu (custom action) ................................. 120  
26.7.8 Generic menu item .......................................................... 120  
26.7.9 Action groups .................................................................. 121  
26.7.10 Cell menu components .................................................. 121  
26.8 Customizing reference presentation ...................................... 121  
26.9 Migration of presentation query in reference constraints ....... 122  
26.10 Two-step deletion ............................................................... 123  
26.10.1 Invoking two-step deletion from code ............................ 123  
26.10.2 More complex cases ....................................................... 124

## 27 Editor Actions

27.1 Paste wrappers ..................................................................... 125  
27.2 Copy-paste handlers ........................................................... 126

## 28 Diagramming Editor

28.0.1 Adding elements ............................................................. 127  
28.0.2 Samples ........................................................................... 128  
28.1 Dependencies ....................................................................... 130  
28.2 Diagram definition .............................................................. 131  
28.2.1 Palette ........................................................................... 131  
28.2.2 Blocks ............................................................................ 131  
28.2.3 Custom figures ............................................................... 133  
28.2.4 Connectors ...................................................................... 134  
28.2.5 Rendering ports ............................................................. 135  
28.2.6 Using implicit ports ........................................................ 135

## 29 Context Assistant

29.1 Overview .............................................................................. 136  
29.2 Using Context Assistant UI ................................................ 136  
29.3 Using Context Assistant Framework ................................... 136  
29.3.1 Placeholder Cells ............................................................ 137  
29.3.2 Menu Lookup ............................................................... 137  
29.3.3 Placeholder Lookup ........................................................ 137

## 30 Context actions tool

.......................................................... 138
### CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Transformation Menu Language</td>
<td>139</td>
</tr>
<tr>
<td>31.1</td>
<td>Overview</td>
<td>139</td>
</tr>
<tr>
<td>31.2</td>
<td>Defining a Menu</td>
<td>139</td>
</tr>
<tr>
<td>31.2.1</td>
<td>Default Menu</td>
<td>139</td>
</tr>
<tr>
<td>31.2.2</td>
<td>Named Menu</td>
<td>140</td>
</tr>
<tr>
<td>31.2.3</td>
<td>Menu Contributions</td>
<td>141</td>
</tr>
<tr>
<td>31.2.4</td>
<td>Section locations</td>
<td>141</td>
</tr>
<tr>
<td>31.2.5</td>
<td>Menu Parts</td>
<td>142</td>
</tr>
<tr>
<td>31.3</td>
<td>Side transformations</td>
<td>143</td>
</tr>
<tr>
<td>31.4</td>
<td>Substitute menus</td>
<td>143</td>
</tr>
<tr>
<td>31.4.1</td>
<td>Default behavior for code-completion</td>
<td>144</td>
</tr>
<tr>
<td>31.4.2</td>
<td>Substitute menu (default)</td>
<td>144</td>
</tr>
<tr>
<td>31.4.3</td>
<td>Substitute menu (named)</td>
<td>145</td>
</tr>
<tr>
<td>31.4.4</td>
<td>Substitute menu contribution</td>
<td>145</td>
</tr>
<tr>
<td>31.4.5</td>
<td>Options</td>
<td>145</td>
</tr>
<tr>
<td>31.5</td>
<td>Interaction of Cell Menu and Cell-specific Transformation Menu</td>
<td>145</td>
</tr>
<tr>
<td>31.6</td>
<td>Include transformation menu for the property/reference</td>
<td>145</td>
</tr>
<tr>
<td>31.6.1</td>
<td>Example on references</td>
<td>145</td>
</tr>
<tr>
<td>31.6.2</td>
<td>Example on properties</td>
<td>146</td>
</tr>
<tr>
<td>31.7</td>
<td>Show Item Trace</td>
<td>147</td>
</tr>
<tr>
<td>32</td>
<td>Extending the Transformation Menu Language</td>
<td>149</td>
</tr>
<tr>
<td>32.1</td>
<td>Overview</td>
<td>149</td>
</tr>
<tr>
<td>32.2</td>
<td>Adding Locations and Features</td>
<td>149</td>
</tr>
<tr>
<td>32.3</td>
<td>Adding Menu Part Types</td>
<td>149</td>
</tr>
<tr>
<td>33</td>
<td>Intentions</td>
<td>150</td>
</tr>
<tr>
<td>33.0.1</td>
<td>Using intentions</td>
<td>150</td>
</tr>
<tr>
<td>33.0.2</td>
<td>Intention types</td>
<td>150</td>
</tr>
<tr>
<td>33.0.3</td>
<td>Common Intention Structure</td>
<td>150</td>
</tr>
<tr>
<td>33.0.4</td>
<td>Where to store my intentions?</td>
<td>151</td>
</tr>
<tr>
<td>34</td>
<td>Generator</td>
<td>152</td>
</tr>
<tr>
<td>34.1</td>
<td>Introduction</td>
<td>152</td>
</tr>
<tr>
<td>34.2</td>
<td>Overview</td>
<td>152</td>
</tr>
<tr>
<td>34.2.1</td>
<td>Generator Module</td>
<td>152</td>
</tr>
<tr>
<td>34.2.2</td>
<td>Transformation</td>
<td>154</td>
</tr>
<tr>
<td>34.3</td>
<td>Generator Language Reference</td>
<td>158</td>
</tr>
<tr>
<td>34.3.1</td>
<td>Mapping Configuration</td>
<td>158</td>
</tr>
<tr>
<td>34.3.2</td>
<td>Generator Rule</td>
<td>158</td>
</tr>
<tr>
<td>34.3.3</td>
<td>Root Template</td>
<td>160</td>
</tr>
<tr>
<td>34.3.4</td>
<td>External Template</td>
<td>160</td>
</tr>
<tr>
<td>34.3.5</td>
<td>Mapping Label</td>
<td>160</td>
</tr>
<tr>
<td>34.3.6</td>
<td>Export Label</td>
<td>161</td>
</tr>
<tr>
<td>34.3.7</td>
<td>Macro</td>
<td>161</td>
</tr>
<tr>
<td>34.3.8</td>
<td>Template Switch</td>
<td>163</td>
</tr>
<tr>
<td>34.3.9</td>
<td>Generation Context (operations)</td>
<td>164</td>
</tr>
<tr>
<td>34.3.10</td>
<td>Mapping Script</td>
<td>166</td>
</tr>
<tr>
<td>34.4</td>
<td>The Generator Algorithm</td>
<td>166</td>
</tr>
<tr>
<td>34.4.1</td>
<td>Defining the Generators Involved</td>
<td>166</td>
</tr>
<tr>
<td>34.4.2</td>
<td>Dependency scope/kind - 'Generation Target' and 'Design'</td>
<td>167</td>
</tr>
<tr>
<td>34.4.3</td>
<td>Defining the Order of Priorities</td>
<td>167</td>
</tr>
<tr>
<td>34.4.4</td>
<td>Optimized Generation Plan</td>
<td>167</td>
</tr>
<tr>
<td>34.4.5</td>
<td>Ignored priority rules</td>
<td>167</td>
</tr>
<tr>
<td>34.4.6</td>
<td>Implicit priorities</td>
<td>167</td>
</tr>
<tr>
<td>34.4.7</td>
<td>Model Transformation</td>
<td>168</td>
</tr>
<tr>
<td>34.4.8</td>
<td>Handling of node attributes during generation</td>
<td>168</td>
</tr>
<tr>
<td>34.4.9</td>
<td>In-place transformation</td>
<td>169</td>
</tr>
<tr>
<td>34.4.10</td>
<td>Generation trace</td>
<td>169</td>
</tr>
<tr>
<td>34.4.11</td>
<td>Support for non-reflective queries</td>
<td>169</td>
</tr>
<tr>
<td>34.5</td>
<td>Mapping Priorities</td>
<td>169</td>
</tr>
<tr>
<td>34.6</td>
<td>Generation plan</td>
<td>170</td>
</tr>
<tr>
<td>34.6.1</td>
<td>Motivation</td>
<td>170</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>35 TextGen</td>
<td>TextGen language aspect</td>
<td>180</td>
</tr>
<tr>
<td>35.1</td>
<td>Introduction</td>
<td>180</td>
</tr>
<tr>
<td>35.2</td>
<td>Operations</td>
<td>180</td>
</tr>
<tr>
<td>35.3</td>
<td>Indentation</td>
<td>180</td>
</tr>
<tr>
<td>35.4</td>
<td>Root concepts</td>
<td>181</td>
</tr>
<tr>
<td>35.5</td>
<td>TextGen in extended concepts</td>
<td>181</td>
</tr>
<tr>
<td>35.6</td>
<td>Layout</td>
<td>182</td>
</tr>
<tr>
<td>35.7</td>
<td>Context objects</td>
<td>182</td>
</tr>
<tr>
<td>35.8</td>
<td>Handling attributes in TextGen</td>
<td>183</td>
</tr>
<tr>
<td>35.9</td>
<td>Examples</td>
<td>183</td>
</tr>
<tr>
<td>36 Data flow</td>
<td>Data Flow</td>
<td>185</td>
</tr>
<tr>
<td>36.1</td>
<td>Commands of intermediate language</td>
<td>185</td>
</tr>
<tr>
<td>36.2</td>
<td>May be unreachable</td>
<td>185</td>
</tr>
<tr>
<td>36.3</td>
<td>Links:</td>
<td>186</td>
</tr>
<tr>
<td>37 Typesystem</td>
<td>Defining A Typesystem For Your Language</td>
<td>187</td>
</tr>
<tr>
<td>37.1</td>
<td>What is a typesystem?</td>
<td>187</td>
</tr>
<tr>
<td>37.2</td>
<td>Types</td>
<td>187</td>
</tr>
<tr>
<td>37.3</td>
<td>Inference Rules</td>
<td>187</td>
</tr>
<tr>
<td>37.4</td>
<td>Inference Methods</td>
<td>188</td>
</tr>
<tr>
<td>37.5</td>
<td>Overriding</td>
<td>188</td>
</tr>
<tr>
<td>37.6</td>
<td>Equations And Inequations</td>
<td>188</td>
</tr>
<tr>
<td>37.7</td>
<td>Weak And Strong Subtyping</td>
<td>188</td>
</tr>
<tr>
<td>37.8</td>
<td>Subtyping Rules</td>
<td>189</td>
</tr>
<tr>
<td>37.9</td>
<td>Comparison Inequations And Comparison Rules</td>
<td>189</td>
</tr>
<tr>
<td>37.10</td>
<td>Examples Of Inference Rules</td>
<td>189</td>
</tr>
<tr>
<td>37.11</td>
<td>Type Variables</td>
<td>190</td>
</tr>
<tr>
<td>37.12</td>
<td>Meet and Join types</td>
<td>191</td>
</tr>
<tr>
<td>37.13</td>
<td>&quot;When Concrete&quot; Blocks</td>
<td>192</td>
</tr>
<tr>
<td>37.14</td>
<td>Overloaded Operators</td>
<td>192</td>
</tr>
<tr>
<td>37.15</td>
<td>Replacement Rules</td>
<td>193</td>
</tr>
<tr>
<td>37.16</td>
<td>Motivation</td>
<td>193</td>
</tr>
<tr>
<td>37.17</td>
<td>Different semantics</td>
<td>194</td>
</tr>
<tr>
<td>37.18</td>
<td>Type-system, trace</td>
<td>195</td>
</tr>
<tr>
<td>37.19</td>
<td>Advanced features of typesystem language</td>
<td>198</td>
</tr>
<tr>
<td>37.20</td>
<td>Overriding default type node</td>
<td>198</td>
</tr>
<tr>
<td>37.21</td>
<td>Check-only inequations</td>
<td>198</td>
</tr>
<tr>
<td>37.22</td>
<td>Dependencies</td>
<td>199</td>
</tr>
<tr>
<td>37.23</td>
<td>Overriding type of literal or expression</td>
<td>199</td>
</tr>
<tr>
<td>37.24</td>
<td>Inference rules</td>
<td>199</td>
</tr>
<tr>
<td>37.25</td>
<td>Inference rules for node attributes</td>
<td>199</td>
</tr>
<tr>
<td>37.26</td>
<td>Conditionally overriding type inference</td>
<td>200</td>
</tr>
<tr>
<td>37.27</td>
<td>Checking rules</td>
<td>201</td>
</tr>
<tr>
<td>37.28</td>
<td>Severity</td>
<td>201</td>
</tr>
</tbody>
</table>
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>37.20.2 Quick-fixes</td>
<td>202</td>
</tr>
<tr>
<td>37.20.3 Invoking quick-fixes</td>
<td>202</td>
</tr>
<tr>
<td><strong>38 Using typesystem</strong></td>
<td>203</td>
</tr>
<tr>
<td>38.1 Using a typesystem</td>
<td>203</td>
</tr>
<tr>
<td>38.2 Type Operation</td>
<td>203</td>
</tr>
<tr>
<td>38.3 Is Subtype expression</td>
<td>203</td>
</tr>
<tr>
<td>38.4 Coerce expression</td>
<td>203</td>
</tr>
<tr>
<td>38.5 Coerce Statement</td>
<td>203</td>
</tr>
<tr>
<td><strong>39 Typesystem Debugging</strong></td>
<td>204</td>
</tr>
<tr>
<td>39.1 Try it out for yourself</td>
<td>204</td>
</tr>
<tr>
<td>39.2 Inspecting the types</td>
<td>205</td>
</tr>
<tr>
<td>39.3 Type errors</td>
<td>206</td>
</tr>
<tr>
<td>39.4 Type-system Trace</td>
<td>206</td>
</tr>
<tr>
<td>39.5 Step-by-step debugging</td>
<td>207</td>
</tr>
<tr>
<td><strong>40 Refactoring</strong></td>
<td>209</td>
</tr>
<tr>
<td>40.1 Changes in the Refactoring language</td>
<td>209</td>
</tr>
<tr>
<td><strong>41 Testing languages</strong></td>
<td>210</td>
</tr>
<tr>
<td>41.1 Testing languages</td>
<td>210</td>
</tr>
<tr>
<td>41.2 Introduction</td>
<td>210</td>
</tr>
<tr>
<td>41.3 Quick navigation table</td>
<td>210</td>
</tr>
<tr>
<td>41.4 Tests creation</td>
<td>211</td>
</tr>
<tr>
<td>41.4.1 1. Create a Test aspect in your language</td>
<td>211</td>
</tr>
<tr>
<td>41.4.2 2. Create a test model</td>
<td>211</td>
</tr>
<tr>
<td>41.4.3 Unit testing with BTestCase</td>
<td>213</td>
</tr>
<tr>
<td>41.4.4 TestInfo</td>
<td>213</td>
</tr>
<tr>
<td>41.5 Testing aspects of language definitions</td>
<td>214</td>
</tr>
<tr>
<td>41.5.1 Node tests</td>
<td>215</td>
</tr>
<tr>
<td>41.5.2 Editor tests</td>
<td>220</td>
</tr>
<tr>
<td>41.5.3 Migration tests</td>
<td>222</td>
</tr>
<tr>
<td>41.6 Running the tests</td>
<td>224</td>
</tr>
<tr>
<td>41.6.1 Inside MPS</td>
<td>224</td>
</tr>
<tr>
<td>41.6.2 From a build script</td>
<td>227</td>
</tr>
<tr>
<td>41.7 Running Editor tests in IDEA Plugin</td>
<td>228</td>
</tr>
<tr>
<td><strong>42 Common language patterns</strong></td>
<td>229</td>
</tr>
<tr>
<td>42.1 Sample project with language patterns</td>
<td>229</td>
</tr>
<tr>
<td>42.2 Initial tips</td>
<td>230</td>
</tr>
<tr>
<td>42.2.1 Naming concepts</td>
<td>230</td>
</tr>
<tr>
<td>42.2.2 Valid identifiers</td>
<td>230</td>
</tr>
<tr>
<td>42.2.3 Unique names</td>
<td>231</td>
</tr>
<tr>
<td>42.2.4 Not seeing type-system errors</td>
<td>231</td>
</tr>
<tr>
<td>42.2.5 Regular expressions</td>
<td>231</td>
</tr>
<tr>
<td>42.2.6 Safe operators and operations</td>
<td>232</td>
</tr>
<tr>
<td>42.3 Basic patterns</td>
<td>232</td>
</tr>
<tr>
<td>42.3.1 Container - component</td>
<td>232</td>
</tr>
<tr>
<td>42.3.2 Customized presentation</td>
<td>233</td>
</tr>
<tr>
<td>42.3.3 Declaration - references</td>
<td>234</td>
</tr>
<tr>
<td>42.3.4 Fluent editing</td>
<td>234</td>
</tr>
<tr>
<td>42.3.5 Override editor component</td>
<td>235</td>
</tr>
<tr>
<td>42.3.6 Seamless substitution</td>
<td>235</td>
</tr>
<tr>
<td>42.3.7 Hierarchical scopes</td>
<td>237</td>
</tr>
<tr>
<td>42.3.8 Seamless substitution for your own references</td>
<td>237</td>
</tr>
<tr>
<td><strong>43 Accessories</strong></td>
<td>240</td>
</tr>
<tr>
<td>43.1 Example</td>
<td>240</td>
</tr>
<tr>
<td>43.1.1 Define the concept to represent colors</td>
<td>240</td>
</tr>
<tr>
<td>43.1.2 Update ColorReference</td>
<td>240</td>
</tr>
<tr>
<td>43.1.3 Define a method to obtain the real Color</td>
<td>241</td>
</tr>
<tr>
<td>43.1.4 Change the generator templates for circle and square</td>
<td>241</td>
</tr>
<tr>
<td>43.1.5 Define colors in the Accessories model</td>
<td>241</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>53 HowTo – Integrating into the MPS Make Framework</td>
<td>282</td>
</tr>
<tr>
<td>53.1 Build Facets</td>
<td>282</td>
</tr>
<tr>
<td>53.2 Overview</td>
<td>282</td>
</tr>
<tr>
<td>53.3 The targets within a facet can exchange configuration parameters.</td>
<td>283</td>
</tr>
<tr>
<td>For example, a target that is declared to run early in the overall</td>
<td></td>
</tr>
<tr>
<td>make process may collect configuration parameters and pass them to</td>
<td></td>
</tr>
<tr>
<td>the second facet, which then uses the parameters. The mechanism to</td>
<td></td>
</tr>
<tr>
<td>achieve this intra-facet parameter exchange is called <strong>properties</strong>.</td>
<td></td>
</tr>
<tr>
<td>In addition, targets can use queries to obtain information from the</td>
<td></td>
</tr>
<tr>
<td>user during the make process.</td>
<td>283</td>
</tr>
<tr>
<td>53.4 Building an Example Facet</td>
<td>284</td>
</tr>
<tr>
<td>53.4.1 The first target: Collecting Directories</td>
<td>284</td>
</tr>
<tr>
<td>53.4.2 The second Target: Running Make</td>
<td>285</td>
</tr>
<tr>
<td>54 Extension support</td>
<td>287</td>
</tr>
<tr>
<td>54.1 Quick howto</td>
<td>287</td>
</tr>
<tr>
<td>54.2 Extension language</td>
<td>287</td>
</tr>
<tr>
<td>54.3 Java API</td>
<td>289</td>
</tr>
<tr>
<td>55 Custom Persistence Cookbook</td>
<td>290</td>
</tr>
<tr>
<td>55.1 What is custom persistence?</td>
<td>290</td>
</tr>
<tr>
<td>55.2 Look around the sample project</td>
<td>290</td>
</tr>
<tr>
<td>55.3 Define the persistence format</td>
<td>291</td>
</tr>
<tr>
<td>55.3.1 Plugin ID</td>
<td>291</td>
</tr>
<tr>
<td>55.4 Build the MPS plugin</td>
<td>292</td>
</tr>
<tr>
<td>55.5 Build the IntelliJ IDEA plugin</td>
<td>294</td>
</tr>
<tr>
<td>55.6 Using the custom persistence plugin in MPS</td>
<td>295</td>
</tr>
<tr>
<td>55.7 Using the custom persistence plugin in IntelliJ IDEA</td>
<td>296</td>
</tr>
<tr>
<td>55.8 Debugging the plugin</td>
<td>296</td>
</tr>
<tr>
<td>V IDE tools</td>
<td>298</td>
</tr>
<tr>
<td>56 Dependencies Analyzer - analyze model dependencies</td>
<td>299</td>
</tr>
<tr>
<td>57 Module Dependencies Tool</td>
<td>301</td>
</tr>
<tr>
<td>58 Run Configurations</td>
<td>304</td>
</tr>
<tr>
<td>58.0.1 Introduction</td>
<td>304</td>
</tr>
<tr>
<td>58.0.2 Settings</td>
<td>305</td>
</tr>
<tr>
<td>58.0.3 Commands</td>
<td>306</td>
</tr>
<tr>
<td>58.0.4 Configurations</td>
<td>307</td>
</tr>
<tr>
<td>58.0.5 Useful examples</td>
<td>307</td>
</tr>
<tr>
<td>59 Changes highlighting</td>
<td>312</td>
</tr>
<tr>
<td>59.0.6 Project tree view</td>
<td>312</td>
</tr>
<tr>
<td>59.0.7 Editor tabs</td>
<td>312</td>
</tr>
<tr>
<td>59.0.8 Editor</td>
<td>313</td>
</tr>
<tr>
<td>60 Default Keymap Reference</td>
<td>315</td>
</tr>
<tr>
<td>61 Module Cloning</td>
<td>319</td>
</tr>
<tr>
<td>VI Platform Languages</td>
<td>322</td>
</tr>
<tr>
<td>62 Base Language</td>
<td>323</td>
</tr>
<tr>
<td>63 Base Language Extensions Style Guide</td>
<td>324</td>
</tr>
<tr>
<td>63.1 Quick Reference</td>
<td>324</td>
</tr>
<tr>
<td>63.2 Keywords</td>
<td>327</td>
</tr>
<tr>
<td>63.3 Curly braces</td>
<td>327</td>
</tr>
<tr>
<td>63.4 Parentheses</td>
<td>327</td>
</tr>
<tr>
<td>63.5 Identifiers</td>
<td>327</td>
</tr>
<tr>
<td>63.6 Punctuation</td>
<td>327</td>
</tr>
</tbody>
</table>
## 64 MPS Java compatibility
- 64.1 Configuration ....................................................... 328
- 64.2 Build scripts .................................................... 328
- 64.3 Using java classes compiled with JDK 1.8  
  - 64.3.1 Static interface method call ............................. 329
  - 64.3.2 Default interface methods ............................... 329
- 64.4 Using Java platform API ...................................... 330

## 65 Closures
- 65.1 Introduction .................................................... 331
- 65.2 Function type .................................................. 331
- 65.3 Closure literal ................................................ 331
- 65.4 Closure invocation ......................................... 332
  - 65.4.1 Recursion ................................................. 332
- 65.5 Closure conversion ......................................... 332
- 65.6 Yield statement ............................................. 333
- 65.7 Functions that return functions ......................... 333
- 65.8 Runtime ....................................................... 333
- 65.9 Differences from the BGGA proposal ................... 333

## 66 Collections language
- 66.1 Introduction ................................................... 335
- 66.2 Null handling ................................................ 335
- 66.3 Skip and stop statements .................................. 335
- 66.4 Collections Runtime ....................................... 335
- 66.5 Sequence ....................................................... 336
- 66.6 List ............................................................. 344
- 66.7 Set ............................................................. 347
- 66.8 Sorted Set ..................................................... 348
- 66.9 Map ............................................................. 349
- 66.10 Sorted Map ................................................... 351
- 66.11 Stack .......................................................... 353
- 66.12 Queue ........................................................ 355
- 66.13 Deque ........................................................ 356
- 66.14 Iterator ....................................................... 357
- 66.15 Modifying Iterator ....................................... 358
- 66.16 Enumerator .................................................. 358
- 66.17 Mapping ...................................................... 359
- 66.18 Custom Containers ....................................... 359
- 66.19 Primitive Containers .................................. 360

## 67 Tuples
- 67.1 Indexed tuples ............................................... 363
- 67.2 Named tuples ................................................ 363
  - 67.2.1 Named tuple declaration .................................. 363

## 68 Lightweight DSL
- 68.1 Enhancing plain classes .................................... 364
- 68.2 Defining a DSLDescriptor  
  - 68.2.1 Properties .................................................. 366
  - 68.2.2 Methods ..................................................... 366
  - 68.2.3 Custom members .......................................... 367
  - 68.2.4 Initializer .................................................. 369
- 68.3 Enhancing custom ClassConcept sub-concepts .......... 370
- 68.4 Samples ....................................................... 371

## 69 Dates language
- 69.1 Introduction ................................................... 372
- 69.2 Types .......................................................... 372
- 69.3 Properties ..................................................... 373
- 69.4 Operations .................................................... 374
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>70 Regular expressions</td>
<td>376</td>
</tr>
<tr>
<td>70.1 Regular expressions language overview</td>
<td>376</td>
</tr>
<tr>
<td>70.1.1 Introduction</td>
<td>376</td>
</tr>
<tr>
<td>70.1.2 Language overview</td>
<td>376</td>
</tr>
<tr>
<td>70.1.3 Language structure</td>
<td>377</td>
</tr>
<tr>
<td>70.1.4 Editor</td>
<td>384</td>
</tr>
<tr>
<td>70.1.5 Scopes</td>
<td>386</td>
</tr>
<tr>
<td>70.1.6 Actions</td>
<td>387</td>
</tr>
<tr>
<td>70.1.7 Type System</td>
<td>387</td>
</tr>
<tr>
<td>70.1.8 Generator</td>
<td>388</td>
</tr>
<tr>
<td>70.1.9 Further reading</td>
<td>389</td>
</tr>
<tr>
<td>71 Regexp language</td>
<td>391</td>
</tr>
<tr>
<td>71.1 Defining regular expression</td>
<td>391</td>
</tr>
<tr>
<td>71.1.1 Re-using Definitions</td>
<td>391</td>
</tr>
<tr>
<td>71.1.2 Pattern Match Operator</td>
<td>391</td>
</tr>
<tr>
<td>71.1.3 Capturing Text</td>
<td>391</td>
</tr>
<tr>
<td>71.1.4 Examples</td>
<td>392</td>
</tr>
<tr>
<td>72 Type Extension Methods</td>
<td>393</td>
</tr>
<tr>
<td>72.0.5 Root Nodes</td>
<td>394</td>
</tr>
<tr>
<td>73 Builders</td>
<td>396</td>
</tr>
<tr>
<td>73.1 Building Builders</td>
<td>397</td>
</tr>
<tr>
<td>74 Logging</td>
<td>400</td>
</tr>
<tr>
<td>75 XML language</td>
<td>402</td>
</tr>
<tr>
<td>75.0.1 Structure</td>
<td>402</td>
</tr>
<tr>
<td>75.0.2 Editing</td>
<td>402</td>
</tr>
<tr>
<td>75.0.3 Generation</td>
<td>403</td>
</tr>
<tr>
<td>76 Other languages</td>
<td>404</td>
</tr>
<tr>
<td>76.1 Checked dots</td>
<td>404</td>
</tr>
<tr>
<td>76.2 Overloaded operators</td>
<td>404</td>
</tr>
<tr>
<td>76.3 Custom constructors</td>
<td>405</td>
</tr>
<tr>
<td>VII Delivering languages to the users</td>
<td>406</td>
</tr>
<tr>
<td>77 Build Language</td>
<td>407</td>
</tr>
<tr>
<td>77.1 What is MPS build language?</td>
<td>407</td>
</tr>
<tr>
<td>77.2 Build script structure</td>
<td>407</td>
</tr>
<tr>
<td>77.3 Macros</td>
<td>409</td>
</tr>
<tr>
<td>77.4 Built-in plugins</td>
<td>409</td>
</tr>
<tr>
<td>77.4.1 Java plugin</td>
<td>409</td>
</tr>
<tr>
<td>77.4.2 MPS plugin</td>
<td>410</td>
</tr>
<tr>
<td>77.4.3 Module resources</td>
<td>411</td>
</tr>
<tr>
<td>77.4.4 Module Testing plugin</td>
<td>414</td>
</tr>
<tr>
<td>77.4.5 MPS-runner plugin</td>
<td>414</td>
</tr>
<tr>
<td>77.4.6 Control over the repository</td>
<td>415</td>
</tr>
<tr>
<td>77.5 How-to's</td>
<td>415</td>
</tr>
<tr>
<td>78 Building IntelliJ IDEA language plugins</td>
<td>417</td>
</tr>
<tr>
<td>78.0.1 Starting point</td>
<td>417</td>
</tr>
<tr>
<td>78.0.2 Generate an Ant build xml file</td>
<td>422</td>
</tr>
<tr>
<td>78.0.3 Run Ant</td>
<td>423</td>
</tr>
<tr>
<td>78.0.4 Pick the generated files and share them</td>
<td>424</td>
</tr>
</tbody>
</table>
## 79 Using MPS inside IntelliJ IDEA

- 79.0.5 Get the MPS core IntelliJ IDEA plugin .............................................. 425
- 79.0.6 Turn MPS on and off ........................................................................... 426
- 79.0.7 Try BaseLanguage inside IntelliJ IDEA .............................................. 426
- 79.0.8 Playing with MPS models ..................................................................... 428
- 79.0.9 Code generation .................................................................................. 430
- 79.0.10 Install the language plugin ................................................................. 430
- 79.0.11 Adding runtime libraries ..................................................................... 431
- 79.0.12 Interoperability between Java and BaseLanguage .............................. 432
- 79.0.13 Converting Java code to BaseLanguage ............................................. 432
- 79.0.14 MPS configuration options ................................................................. 432
- 79.0.15 Configure the generator ..................................................................... 433

## 80 Building MPS language plugins

### 81 Building standalone IDEs for your languages

- 81.1 Introduction ................................................................................................. 437
- 81.2 Process Overview ...................................................................................... 437
- 81.3 Building an example RCP build ............................................................... 437
  - 81.3.1 Creating the Solution and the Build Script ........................................ 437
  - 81.3.2 Investigating the build scripts ............................................................ 440
  - 81.3.3 Editing the build scripts .................................................................... 441
  - 81.3.4 Creating the Platform-Specific Distributions .................................... 446
  - 81.3.5 Creating a Windows distribution ...................................................... 446
- 81.4 The final Robot Kaja IDE build scripts ..................................................... 449
  - 81.4.1 Kajak ................................................................................................. 449
  - 81.4.2 KajakDistribution ............................................................................ 451
- 81.5 Generating and running the Build Script ................................................ 452
- 81.6 Inspecting the Generated Directory Structure ......................................... 454

## 82 Extending the user interface

- 82.1 Creating wizards ....................................................................................... 455

## 83 Removing sources from generated code

- 83.1 Remove sources from BaseLanguage code .............................................. 456
  - 83.1.1 A few handy notes ............................................................................. 458
- 83.2 Customizing implementation stripping for your own language ............ 458
  - 83.2.1 Robot Kaja sample .......................................................................... 458
  - 83.2.2 Initial situation .................................................................................. 458
  - 83.2.3 Goal .................................................................................................. 460
  - 83.2.4 Changes to the Robot Kaja language ................................................. 460
  - 83.2.5 Stub creation ..................................................................................... 462
  - 83.2.6 Build script for the library authors ................................................... 463
  - 83.2.7 General guidelines and additional notes ........................................... 464

## 84 FAQ

- 84.1 DSLs, LOP and Programming ................................................................ 468
  - 84.1.1 What are DSLs? How are they different from "real" programming languages? .............................................. 468
  - 84.1.2 What are the benefits of DSLs? Why should I care? ......................... 468
  - 84.1.3 How do DSLs and regular Code fit together? .................................... 468
  - 84.1.4 What is Language Oriented Programming? ..................................... 469
  - 84.1.5 Why do I want to extend a language? Aren’t libraries good enough? ................................................ 469
- 84.2 Projectional Editing .................................................................................. 469
  - 84.2.1 What is Projectional Editing? ............................................................. 469
  - 84.2.2 Why use Projectional Editing? Aren’t text editors good enough? .... 469
  - 84.2.3 What are the key benefits of using projectional editors instead of parsed languages ......................... 470
  - 84.2.4 If it isn’t text, how does the integration with version control work? Diff, Merge? ...................................... 470
  - 84.2.5 How long does it take to get used to the editor? ................................. 471
- 84.3 MPS Specifics ............................................................................................ 471
  - 84.3.1 How is MPS licensed? ....................................................................... 471
  - 84.3.2 Which languages are already in MPS so I can extend them? .............. 471
Part I

Introduction to MPS
Chapter 1

MPS User’s Guide

1.1 MPS User Guide for Language Designers

Credits

Welcome to MPS. This User Guide is a complete reference documentation to MPS and it will navigate you through the many concepts and usage patterns that MPS offers and will give you a hand whenever you need to know more details about any particular aspect of the system.

1.1.1 Beginner’s Fast Track to MPS

Here’s our offer for new comers: Try our Fast Track to MPS tutorial, which was designed specifically for professionals, who are completely new to MPS and prefer a guided tour through the MPS landscape. You will walk the beaten path one step at a time, following clear marks that show you where to go next. The information is structured so that you progress from simpler concepts to the more involved ones and at the end of the journey you’ll understand MPS and will be able to use it effectively on your projects.

1.1.2 Structure of the User Guide

1. First, the Introduction section will offer a high-level overview of the basic notions and their roles.

2. In the second section, named Using MPS, you’ll get familiar with the interface through which you’ll communicate with MPS. Although very small, there still are some differences between how you interact with MPS and how you typically use other common programming tools.

3. In the third section, called Defining Languages, we’ll get to the meat of MPS. We’ll show details on how to define the many aspects of your custom languages - their structure, editors, generators and type systems rules.

4. The IDE integration section will then provide some additional context necessary to help you improve the IDE aspect of your languages and integrate them nicely into MPS.

5. The Platform languages section gives you details on all languages bundled with MPS including the corner stone language of MPS - the BaseLanguage.

6. The Delivering languages to the users and Java IDE integration section covers the build language and the process of building standalone IDEs as well as language plugins.

7. Don’t forget to check out our tutorials and focused cookbooks listed in the Tutorials and Cookbooks sections, to learn more about individual aspects of MPS.

8. Whatever didn’t fit the mentioned scheme was placed into the last Miscellaneous section.

For your convenience, you can also view the MPS User’s Guide (created by Marco Lombardo) or MPS User’s Guide (one page) version.
Alternative User Guides

Not a language designer yet? Try out one of our simplified MPS user guides

You do not need to design your own languages and DSLs to benefit from MPS. You may well enjoy using languages designed and developed by somebody else. These languages are typically distributed as language plugins for Java IDEs or they may come bundled with their own standalone IDEs. Since using existing languages is a much simpler tasks than creating them, we prepared dedicated user guides covering the topics relevant to the DSL users.

- Check out the MPS user guide for Java developers (IntelliJ IDEA) if you want to learn quickly how to use MPS for Java development.
- The MPS user guide for DSL users collects the essential knowledge required for developing in MPS-based DSLs using an MPS-based standalone IDE.

User guide for language designers

1. Introduction to MPS
   - Basic notions
   - Glossary
   - FAQ
   - MPS project structure
   - Default Keymap Reference

2. Using MPS
   - MPS Java compatibility
   - Commanding the editor
   - IDE configuration
   - Getting the dependencies right
   - Finding your way out
   - Where to find language plugins
   - Version Control
   - Using MPS Debugger
   - Migrations
   - Using Migrations with branching
   - Console

3. Defining Languages - defining a language involves specifying several aspects, each of which codifies some part of the language - the allowed AST structure, the appearance on the screen, type-system rules, etc.

<table>
<thead>
<tr>
<th>Aspect name</th>
<th>Description</th>
<th>Core documentation</th>
<th>Other links</th>
</tr>
</thead>
</table>
| Structure   | Defines the kinds of nodes (called Concepts) that may be used in user models. Each node in the program (model) refers to its concept. Concepts specify, which properties, children and references nodes may have. Concepts can extend other Concepts and implement ConceptInterfaces. | Structure | SModel language  
- programatic access to the model  
Open API - accessing models from code  
- API reference for accessing models  
Quotations  
- building AST snippets  
Pattern  
- language for pattern matching nodes  
Using model & module dependencies FAQ  
Video - Introduction to JetBrains MPS, part 1: Projects  
Video - Introduction to JetBrains MPS, part 2: Structure |
### Constraints

Restricts the relationships between nodes as well as the allowed values for properties beyond the rules defined in Structure.

Constraints typically define:
- the target scope for references (a collection of allowed nodes a reference can point to)
- situations, in which a node can be a child/parent/ancestor of another node
- allowed values for properties
- property accessor methods (getters and setters)

### Behavior

Just like classes in OOP hold methods, Concepts may define methods and static methods that can be invoked on nodes in a polymorphic way. Nodes thus carry behaviour alongside their properties and relationships.

### Editor

Instead of defining a parser that would translate code from an editable form (i.e. text) into the tree-like structure that a computer could manipulate, MPS offers the concept of **projectional editor**, which let’s the user edit the AST directly. The Editor aspect enables language designers to create a UI for editing their concept concepts.

### Actions

The Actions aspect provides means to specify advanced editor behavior, such as copy/paste or node initialization.

### Intentions

All modern IDEs assist developers with instant code manipulating action available under a handy key-shortcut (Alt + Enter in MPS). Language authors can define such little code transformations for their languages in the **Intentions** aspect.
| **Generator** | Models written in one or more languages get ultimately translated into runnable code in some target general-purpose language and platform, such as Java. Along the way models get gradually transformed so that repeatedly concepts get replaced with concepts from a lower level of abstraction until the bottom-line level is reached. The rules for translating concepts and their proper ordering is defined in the **Generator** aspect. | **Generator** | Generation plan | Generator cookbook  
Building an interpreter cookbook  
Video - Introduction to JetBrains MPS, part 8: Generator |
| **TextGen** | During code generation after the **Generator** has reached the bottom-line AST representation, the **TextGen** phase kicks in and translates all nodes in the model into their textual representation and saves the resulting textual source files on disk. | **TextGen** |  
Video - Introduction to JetBrains MPS, part 9: TextGen |
| **Dataflow** | The ability to understand the flow of values and the flow of control through language constructs helps languages report issues such as **unreachable code** or **potential null-pointer error**. Language designer can leverage the **Dataflow** aspect to define the flow for each concept, which MPS will then use to calculate the dataflow for the whole program. | **Data flow** |  
Video - Introduction to JetBrains MPS, part 10: Dataflow |
| **Typesystem** | Language that need to type-check their code need to provide type-system rules. The MPS type-system engine will evaluate the rules on-the-fly, calculate types for nodes and report errors, wherever the calculated type differs from the expectations. So called **checking rules** may additionally be defined to verify non-typesystem assertions about the model. | **Typesystem** |  
Cookbook - Type System  
Using typesystem  
Typesystem Debugging  
Video - Introduction to JetBrains MPS, part 11: Typesystem |
| **Refactoring** | Modern IDEs allow the developer to seamlessly and flawlessly change the structure of their code through refactoring. MPS allows the language designers to prepare such refactorings and make them part of their languages. | **Refactoring** |  
 |
### Migrations

When a new version of a language is released to the public, projects that use the previous version of the language must be migrated so that they use the new language constructs. **Migration scripts**, prepared by the language authors, will manipulate user code and automatically update it to the most recent version of the language.

<table>
<thead>
<tr>
<th>Migrations</th>
<th>Migrations</th>
<th>Using Migrations with branching</th>
</tr>
</thead>
</table>

### Testing

Various aspects of language definition can be automatically tested. Language authors may create tests that will verify that the editor, actions, type-system, data flow or constraints of their languages behave according to the specifications.

<table>
<thead>
<tr>
<th>Testing</th>
<th>Testing languages</th>
</tr>
</thead>
</table>

### Scripts

<table>
<thead>
<tr>
<th>Scripts</th>
</tr>
</thead>
<tbody>
<tr>
<td>TODO</td>
</tr>
</tbody>
</table>

### Accessories

The **Accessories Models** can be stored at two places - either as an aspect of a language (recommended), or as a regular model under a solution. In both cases, the model needs to be added to the Language Runtime Language Settings so as it could be used. A typical use case would be a default library of Concept instances to be available at any place the language is used.

<table>
<thead>
<tr>
<th>Accessories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessories</td>
</tr>
</tbody>
</table>

4. **Languages for IDE Integration** - how to customise MPS, add language-specific visual extensions, use different persistence format, etc.

- Generic placeholders and generic comments
- Commenting out nodes
- Custom Aspect
- Icon description - describing icons by text
- Plugin - extending the UI (menus, tool windows, tabs, preferences, etc.)
- Find usages - customising the way users discover nodes
- Suppressing Errors
- Debugger
- HowTo – Integrating into the MPS Make Framework
- Extension support
- Custom Persistence Cookbook
- HowTo – MPS and ant
- HowTo – MPS and Git
- HTTP support plugin

5. **IDE tools** - tools that MPS offers you to manipulate the languages

- Dependencies Analyzer - analyze model dependencies (Analyze model dependencies)
- Module Dependencies Tool (Analyze module dependencies)
CHAPTER 1. MPS USER’S GUIDE

- Run Configurations
- Changes highlighting
- Default Keymap Reference
- Module Cloning

6. Platform Languages - out-of-the-box languages ready for use

- Base Language
  - Base Language Extensions Style Guide
  - MPS Java compatibility
  - Concept Functions
- Closures
- Collections language
- Tuples
- Lightweight DSL
- Dates language
- Regexp language
- Type Extension Methods
- Builders
- Logging
- XML language
- Other languages

7. Delivering languages to the users and Java IDE integration - building languages from the command line, Ant integration, continuous integration, creating and using plugins, obfuscating code

- Build Language
- Building IntelliJ IDEA language plugins
- Using MPS inside IntelliJ IDEA
- Building MPS language plugins
- Building standalone IDEs for your languages
- Extending the user interface
- Removing sources from generated code (Implementation stripping)

8. Tutorials (MPS User’s Guide)

- Fast Track to MPS
- Shapes - an introductory MPS tutorial
- JetBrains MPS Calculator tutorial
- Generator Demos
- How to Add JARs to a Jetbrains MPS Project by Federico Tomasseti
- MindMaps by Antoine Gagnon
- Markus Voelter’s tutorial
- The beginners screen-cast series on JetBrands TV
- The MPS screen-casts page
- The MPS channel on JetBrands TV


- Common language patterns - a how-to guide covering recurring language design patterns
- Getting the dependencies right
- Editor cookbook
- Generator cookbook
- Building an interpreter cookbook
• Description comments - a cookbook showing how to leverage attributes, scopes and error suppression to add support for description comments on arbitrary code elements of your languages
• Cookbook - Type System
• Dataflow
• Regular expressions
• Open API - accessing models from code
• Custom Persistence Cookbook
• Custom language aspect cookbook
• Finding your way out - a brief collection of guidelines that should help you move forward when you get stuck somewhere.

10. Miscellaneous

• Glossary
• FAQ
• Dependencies and Classpath in MPS
• MPS public roadmap
• Contributing to JetBrains MPS Project - guidelines for MPS contributors and developers on how to get around the MPS source code
• Copyrights

On-line help

MPS comes with bundled help, which shows context-sensitive help information when you hit the F1 key. As an alternative you can view the help pages on-line:

• An on-line variant of the MPS help
Chapter 2

Basic notions

2.1 Basic notions

This chapter describes the basic MPS notions: nodes, concepts, and languages. These are key to proper understanding of how MPS works. They all only make sense when combined with the others and so we must talk about them all together. This section aims to give you the essence of each of the elements. For further details, you may consider checking out the sections devoted to nodes, concept (structure language), and languages (project structure).

2.1.1 Abstract Syntax Tree (AST)

MPS differentiates itself from many other language workbenches by avoiding the text form. Your programs are always represented by an AST. You edit the code as an AST, you save it as an AST you compile it as, well, as an AST. This allows you to avoid defining grammar and building a parser for your languages. You instead define your language in terms of types of AST nodes and rules for their mutual relationships. Almost everything you work with in the MPS editor is an AST-node, belonging to an Abstract Syntax Tree (AST). In this documentation we use a shorter name, node, for AST-node.

2.1.2 Node

Nodes form a tree. Each node has a parent node (except for root nodes), child nodes, properties, and references to other nodes.
CHAPTER 2. BASIC NOTIONS

The AST-nodes are organized into **models**. The nodes that don’t have a parent, called **root nodes**. These are the top-most elements of a language. For example, in *BaseLanguage* (MPS’ counterpart of Java), the root nodes are classes, interfaces, and enums.

### 2.1.3 Concept

Nodes can be very different from one another. Each node stores a reference to its declaration, its **concept**. A concept sets a “type” of connected nodes. It defines the class of nodes and coins the structure of nodes in that class. It specifies which children, properties, and references an instance of a node can have. Concept declarations form an inheritance hierarchy. If one concept extends another, it inherits all children, properties, and references from its parent.

Since everything in MPS revolves around AST, concept declarations are AST-nodes themselves. In fact, they are instances of a particular concept, *ConceptDeclaration*. 
2.1.4 Language

A language in MPS is a set of concepts with some additional information. The additional information includes details on editors, completion menus, intentions, typesystem, dataflow, etc. associated with the language. This information forms several language aspects.

Obviously, a language can extend another language. An extending language can use any concepts defined in the extended language as types for its children or references, and its concepts can inherit from any concept of the extended language. You see, languages in MPS form fully reusable components.

2.1.5 Generator

While languages allow their users to create code, which is stored in models, generators can transform these source models into target models. Generators perform model-to-model conversion on AST models. The target models use different languages than the source models and serve one or more purposes:

- can be converted to text source files and then compiled with standard compilers (Java, C, etc.)
- can be converted to text documents and used as such (configuration, documentation - property files, xml, pdf, html, latex)
- can be directly interpreted
- can be used for code analysis or formal verification by a third party tool (CBMS, state-machine reachability analysis, etc.)
- can be used for simulation of the real system

Generators typically rely on a Domain framework - a set of libraries that the generated code calls or inherits from. The framework encodes the stable part of the desired solution, while the variable part is contained in the actual generated code.
Generation in MPS is done in phases - the output of one generator can become the input for another generator in a pipeline. An optional model-to-text conversion phase (TextGen) may follow to generate code in a textual format. This staged approach helps bridge potentially big semantics gaps between the original problem domain and the technical implementation domain. It also encourages re-use of generators.
Chapter 3

MPS project structure

3.1 Introduction

When designing languages and writing code, good structure helps you navigate around and combine the pieces together. MPS is similar to other IDEs in this regard.

3.2 Project

Project is the main organizational unit in MPS. Projects consist of one or more modules, which themselves consist of models. Model is the smallest unit for generation/compilation. We describe these concepts in detail right below.

3.3 Models

Here’s a major difference that MPS brings along - programs are not in text form. Ever. You might be used to the fact that any programming is done in text. You edit text. The text is then parsed by a parser to build an AST. Grammars are typically used to define parsers. AST is then used as the core data structure to work with your program further, either by the compiler to generate runnable code or by an IDE to give you clever code assistance, refactorings and static code analysis.

Now, seeing that AST is such a useful, flexible and powerful data structure, how would it help if we could work with AST from the very beginning, avoiding text, grammar and parsers altogether? Well, this is exactly what MPS does. To give your code some structure, programs in MPS are organized into models. Think of models as somewhat similar to compilation units in text based languages. To give you an example, BaseLanguage, the bottom-line language in MPS, which builds on Java and extends it in many ways, uses models so that each model represents a Java package. Models typically consist of root nodes, which represent top level declarations, and non-root nodes. For example, in BaseLanguage classes, interfaces, and enums are root nodes. (You can read more about nodes Basic notions).

Models need to hold their meta information:

- models they use (imported models)
- languages (and also devkits) they are written in (in used languages section)
- a few extra params, such as the model file and special generator parameters

This meta information can be altered in Model Properties of the model’s pop-up menu or using Alt + Enter when positioned on the model.

3.4 Modules

Models themselves are the most fine-grained grouping elements. Modules organize models into higher level entities. A module typically consists of several models accompanied with meta information describing module’s properties and dependencies. MPS distinguishes several types of modules: solutions, languages, devkits, and generators. We’ll now talk about the meta-information structure as well as the individual module types in detail.

3.4.1 Module meta information

Now when we have stuff organized into modules, we need a way to combine the modules together for better good. Relationships between modules are described through meta information they hold. The possible relationships among modules can be categorized into several groups:
• **Dependency** - if one module depends on another, and so models inside the former can import models from the latter. The `reexport` property of the dependency relationship indicates whether the dependency is transitive or not. If module A depends on module B with the `reexport` property set to true, every other module that declares dependency on A automatically depends on B as well.

• **Extended** language dependency - if language L extends language M, then every concept from M can be used inside L as a target of a role or an extended concept. Also, all the aspects from language M are available for use and extension in the corresponding aspects of language L.

• **Generation Target** dependency - a relation between two languages (L2 and L1), when one needs to specify that Generator of L2 generates into L1 and thus needs L1’s runtime dependencies.

• **Used language** - if module A uses language L, then models inside A can use language L.

• **Used devkit** - if module A uses devkit D, then models inside A can use devkit D.

• **Generator output path** - generator output path is a folder where all newly generated files will be placed. This is the place you can look for the stuff MPS generates for you.

Now we’ll look at the different types of modules you can find in MPS.

### 3.4.2 Solutions

Solution is the simplest possible kind of module in MPS. It is just a set of models holding code and unified under a common name. There are several types of solutions:

• **Sandbox solutions** - these solutions hold an end user code. The IDE does not treat the code in any special way.

• **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.

• **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 for the Project settings dialog, etc. Again, MPS will keep reloading the classes, whenever they change. Additionally the IDE functionality will be updated accordingly.

### 3.4.3 Languages

Language is a module that is more complex than a solution and represents a reusable language. It consists of several models, each defining a certain aspect of the language: `structure`, `editor`, `actions`, `typesystem`, etc.

Languages can extend other languages. An extending language can then use all concepts from the extended language - derive its own concepts, use inherited concepts as targets for references and also place inherited concepts directly as children inside its own concepts.

Languages frequently have runtime dependencies on third-party libraries or solutions. You may, for example, create a language wrapping any Java library, such as Hibernate or SWT. Your language will then give the users a better and smoother alternative to the standard Java API that these libraries come with.

Now, for your language to work, you need to include the wrapped library with your language. You do it either through a runtime classpath or through a runtime solution. Runtime classpath is suitable for typical scenarios, such as Java-written libraries, while runtime solutions should be preferred for more complex scenarios.

• **Runtime classpath** - makes library classes available as stubs language generators

• **Runtime solutions** - these models are visible to all models inside the generator

#### Language aspects

Language aspects describe different facets of a language:

• **structure** - describes the nodes and structure of the language AST. This is the only mandatory aspect of any language.

• **editor** - describes how a language will be presented and edited in the editor

• **actions** - describes the completion menu customizations specific to a language, i.e. what happens when you type `Control + Space`

• **constraints** - describes the constraints on AST: where a node is applicable, which property and reference are allowed, etc.

• **behavior** - describes the behavioral aspect of AST, i.e. AST methods
CHAPTER 3. MPS PROJECT STRUCTURE

- typesystem - describes the rules for calculating types in a language
- intentions - describes intentions (context dependent actions available when light bulb pops up or when the user types Alt + Enter)
- plugin - allows a language to integrate into MPS IDE
- data flow - describes the intended flow of data in code. It allows you to find unreachable statements, uninitialized reads etc.

You can read more about each aspect in the corresponding section of this guide.

To learn all about setting dependencies between modules and models, check out the Getting the dependencies right page.

3.4.4 Generators

Generators define possible transformations of a language into something else, typically into another languages. Generators may depend on other generators. Since the order in which generators are applied to code is important, ordering constraints can be set on generators. You can read more about generation in the Generator.

3.4.5 DevKits

DevKits have been created to make your life easier. If you have a large group of interconnected languages, you certainly appreciate a way to treat them as a single unit. For example, you may want to import them without listing all of the individual languages. DevKits make this possible. When building a DevKit, you simply list languages to include. As expected, DevKits can extend other DevKits. The extending DevKit will then carry along all the inherited languages as if they were its own ones.

3.5 Projects

This one is easy. A project simply wraps modules that you need to group together and work with them as a unit. You can open the Properties of a project (Alt + Enter on the Project node in the Project View panel) and add or remove modules that should be included in the project. You can also create new modules from the project nodes' context pop-up menu.

3.5.1 Java compilation

MPS was born from Java and is frequently used in Java environment. Since MPS models are often generated into java files, a way to compile java is needed before we can run our programs. There are generally two options:

- Compiling in MPS (recommended)
- Compiling in IntelliJ IDEA (requires IntelliJ IDEA)

When you compile your classes in MPS, you have to set the module’s source path. The source files will be compiled each time the module gets generated, or whenever you invoke compilation manually by the make or rebuild actions.
Chapter 4

MPS user guide for DSL users

Languages created with MPS are typically packaged together with their tooling into dedicated IDEs (Integrated Development Environment). These IDEs are then used by the end users - domain experts, who create models using the supplied languages. Since all the language design functionality and unrelated languages have been removed, such IDEs offer less clutter, greater focus and better performance in comparison with using full-blown MPS. Administrator consoles, business rule editors or other similar types of applications are examples of the domains that these dedicated IDEs typically aim at. This page provides the fundamental knowledge that the users of such single-purpose IDEs need in order to operate the tools efficiently.

4.1 Before you start

Other user guides

- MPS User’s Guide - if you want to learn how to create languages of your own in MPS
- MPS user guide for Java developers (IntelliJ IDEA) - if you are a Java developer and look for ways to seamlessly combine MPS-based languages with Java
- MPS user guide for DSL users - covers the topic of using MPS-based standalone IDEs

4.2 Essentials

- Basic notions of MPS: nodes, concepts, and languages
- Glossary
- FAQ
- MPS project structure
- Default keymap reference

4.3 Setting things up

In order to program in MPS-based languages you need to start up MPS or a standalone IDE that was built specifically for these languages. These dedicated IDEs are essentially cut-down variants of MPS itself, which do not distract a DSL user with unrelated languages and language-design capabilities. MPS-based IDEs are typically standalone Java applications with their own installation procedure. Please follow the instructions provided by the IDE vendor to properly setup the tool and run it.

4.3.1 Configuration

Just like MPS itself, standalone MPS-based IDEs can be configured and customized using the Settings dialog, which you can invoke through the menu, a toolbar or using a keyboard shortcut (Control + Alt + S / Cmd + ,). Although the actual contents of the dialog may differ among different applications, they typically offer some basic settings for the editor, generator and perhaps VCS. Some may also allow to manage plugins, enable and disable individual languages or integrate third-party tools.
Please refer to the documentation of your particular tool for more details on its configuration.

4.4 Editor specialities

The clever projectional editor in MPS is slightly different from what you may be used to. Since the editor is the most visible element, which you’ll get to know as soon as you start interacting with your code, we prepared a short introductory document - Commanding the editor.

4.5 Resolving difficulties, understanding reported errors

The Finding your way out page should give you a hand whenever you run into a problem.

4.6 VCS integration

Being projectional in the world that has been built on text brings about some challenges. Integration with VCS systems is one of them. In order to be able to standard VCS systems, such as Subversion or Git, MPS provides an add-on that hooks into the important VCS events and gives you a projectional way to merge code and resolve conflicts. With this add-on installed your MPS-based IDE will give you the experience you expect from a VCS for your projectional code. Check out the details at the Version Control page.
Chapter 5

MPS user guide for Java developers (IntelliJ IDEA)

One of the primary goals for MPS is to serve Java developers so that they could seamlessly combine DSLs with Java code as an integrated part of their Java projects. Various database queries, business rules, hardware-specific code, system configuration or any other code in languages developed by language vendors as well as those developed yourself can participate in Java projects and be used from Java IDEs directly. This user guide describes MPS from the Java developer’s perspective and provides the essential information needed to jump-start coding with DSLs in IntelliJ IDEA.

5.1 Before you start

Other user guides

- MPS User’s Guide - if you want to learn how to create languages of your own in MPS
- MPS user guide for Java developers (IntelliJ IDEA) - if you are a Java developer and look for ways to seamlessly combine MPS-based languages with Java
- MPS user guide for DSL users - covers the topic of using MPS-based standalone IDEs

5.2 Essentials

- Basic notions of MPS: nodes, concepts, and languages
- Glossary
- FAQ
- MPS project structure
- Default keymap reference

5.3 Setting things up

To get started, you first need to install the essential MPS plugins into IntelliJ IDEA:

- MPS core
- MPS BaseLanguage support (in order to edit BaseLanguage in IDEA)
- MPS Version Control support (in order to use VCSs for MPS models in IDEA)

Both Community and Ultimate editions of IntelliJ IDEA will work. Once you have done that, pick a DSL that you want to use and install it into IntelliJ IDEA, as well. This will allow you to use MPS-based DSLs inside IntelliJ IDEA and interoperate between the MPS code and the rest of your project.

We’ve described all the steps required to get up and running at the Using MPS inside IntelliJ IDEA page. Check it out for details.
5.4 Editor specialities
The clever projectional editor in MPS is slightly different from what you may be used to. Since the editor is the most visible element, which you’ll get to know as soon as you start interacting with your code, we prepared a short introductory document - Commanding the editor.

5.5 Resolving difficulties, understanding reported errors
The Finding your way out page should give you a hand whenever you run into a problem.

5.6 Where to find language plugins
Languages come packaged as ordinary zip files, which you unzip into the IntelliJ IDEA plugin directory and which IDEA will load upon restart. Additionally, many of the plugins have been shared through the IntelliJ IDEA plugin repository and so can be comfortably downloaded through the IDEA’s plugin manager. The Using MPS inside IntelliJ IDEA page further details the necessary steps.

As examples, the BaseLanguage and Build language have been uploaded to the IntelliJ IDEA plugin repository for you to try.

- **BaseLanguage** is a projectional clone of Java, which allows you to write code in a language that you are familiar with, yet benefit from many advantages that MPS brings.
- **Build language** provides a comfortable and fully declarative build abstraction over Ant. This is the language we use internally to build MPS. You can use it, for example, to create build scripts for your Java projects or to compile and package your MPS-based languages. Check out the Build Language for details.

5.7 Platform Languages
On top of Base Language, which is merely a copy of Java 6 (defined using the MPS language-definition aspects), MPS provides several useful language additions that aim at making Java development much more efficient and enjoyable:

- Closures
- Collections language
- Tuples
- Dates language
- Regexp language
- Type Extension Methods
- Builders
- XML language
- Other languages

5.8 IDE integration - Java interoperability, cross-navigation, joint compilation
The MPS IDEA plugin integrates your MPS code tightly into the rest of the Java project. MPS code will seamlessly participate in IDEA’s build and make, you can cross-navigate to usages or definitions between Java and BaseLanguage as well as refactorings will correctly include all the sources.

5.9 VCS integration
Being projectional in the world that has been built on text brings about some challenges. Integration with VCS systems is one of them. In order to be able to standard VCS systems, such as Subversion or Git, MPS provides an add-on that hooks into the important VCS events and gives you a projectional way to merge code and resolve conflicts. With this add-on installed IntelliJ IDEA will give you the experience you expect from a VCS for your projectional code. When comparing versions or resolving conflicts in MPS models, you can rely on the MPS structural diff/merge tool, just like in MPS. The tool will render the model content in a domain specific rather then the persistence-specific way. Check out the details at the Version Control page.
5.10 Debugger

It is not yet possible to directly run a program represented by an MPS node - the generated code should be used to execute the program instead. However, we've already implemented the ability to set a breakpoint directly inside the DSL code and so the IntelliJ IDEA debugger will be stopped in the appropriate place allowing you to explore the stack trace and the variables.
Chapter 6

MPS screen-casts

This page lists and categorizes the MPS-related screen-casts published at the MPS channel of JetBrains TV.

6.0.1 Customer stories

Who and how uses MPS. Watch the MPS customers presenting their use of MPS.

- Why JetBrains MPS - an overview video about MPS, who should care and why
- The Voice Menu IDE - a more thorough example of what MPS can do. A voice menu definition language is used as a sample domain to illustrate the potential of MPS.
- Why the mbeddr.com project uses MPS - mbeddr.com explores the benefits of language extension and formal methods for embedded software development. We have implemented the C programming language in MPS, which allows us to extend it in a meaningful way, for example, with state machines, physical units or with support for product line variability. In this screencast we briefly explain why we chose MPS for this project.
- Modelwerkstatt application of MPS (part 1 and part 2) - modellwerkstatt.org in Innsbruck, Austria use MPS to develop powerful DSLs for building enterprise applications. The whole development process is realized with MPS. Watch this two-part screen-cast to get a taste of what MPS brings to the table in this particular domain.

6.0.2 First steps

Two videos giving you the initial insight into how MPS works

- The MPS projectional editor - an explanation of the principles of the MPS projectional editor, the differences from textual editors and the benefits of this approach
- Your first date with JetBrains MPS - an introductory screen-cast showing how to use the bundled sample Robot Kaja language
- Your second date with JetBrains MPS - an introductory screen-cast showing how the bundled sample Robot Kaja language has been defined
- Creating your first language in JetBrains MPS - a gentle introductory guide through creating a minimalistic language that exposes the basics of structure, editor and generator definition in MPS.

6.0.3 Introduction to MPS

Watch this series of short introductory videos to get a bird’s-eye-view of how languages get defined in MPS. After watching the whole series you’ll know the steps necessary to define a language and you’ll better understand the whole picture. (TODO)

1. Introduction to JetBrains MPS, part 1: Projects - This episode covers the MPS project setup and organisation of modules and models including their dependencies.
2. Introduction to JetBrains MPS, part 2: Structure - This episode provides a brief theoretical background into models, meta-models and abstract syntax trees and then applies the knowledge to the MPS structure aspect.
3. Introduction to JetBrains MPS, part 3: Constraints - This episode details the constraints aspect of language definition in MPS, mainly how to restrict properties, links and how to define scopes.
4. Introduction to JetBrains MPS, part 4: Behavior - This episodes adds a few useful tips on adding functionality to concepts and nodes.
5. Introduction to JetBrains MPS, part 5: Editor - Make’em see your code - defining editors that project the AST on the screen

6. Introduction to JetBrains MPS, part 6: Actions - Polish the editors - smoothing the editing experience by defining transformations and substitutions

7. Introduction to JetBrains MPS, part 7: Intentions - Assist the developer with context-sensitive hints and refactorings

8. Introduction to JetBrains MPS, part 8: Generator - Converting models - defining model-to-model transformations

9. Introduction to JetBrains MPS, part 9: Text-Gen - Here’s what I’ve written - converting models to text

10. Introduction to JetBrains MPS, part 10: Dataflow - Go with the flow - defining dataflow definitions so that MPS could automatically detect issues in code structure

11. Introduction to JetBrains MPS, part 11: Type-system - They are just my type - defining types and type-system rules to validate expressions early-on

You may also watch these as a single YouTube playlist.

6.0.4 Commanding the MPS IDE

These screen-casts should help you become fluent with the projectional MPS editor and various IDE aspects of MPS.

- **MPS Editor Tips** - Watch this introductory screen-cast and learn how to command the MPS editor. You’ll see how to navigate around your code, refactor it, invoke intentions, select regions and a lot more. If you’re new to MPS and want to master the MPS projectional editors, this screen-cast is for you.

- **Running MPS solutions** - This introductory screen-cast aims to show the ways to run MPS solutions. Using the MPS calculator tutorial project as a reference you’ll learn how to generate runnable Java code, create and manage run configurations and integrate MPS into your Java projects.

- **Context Assistant** - This video shows the use and definition of context assistant for the sample robot Kaja language.

- **Context Actions Tool** - This video shows the use and definition of context actions tool.

- **Context Assistant for language definition** - This video highlights the guidance that Context Assistant provides within the language definition languages - structure and editor.

- **Transformation Menu Language** - This video explains the new Transform Menu Language, which is used to define side-transformations, substitutions as well as other context-sensitive actions in the editor.

6.0.5 MPS basics

Videos covering the core principles of building DSLs with MPS

- **MPS basics - creating your first language** - This introductory screen-cast should help MPS newbies to get over the common difficulties that they are likely to face when creating their first DSLs. We’ll go through the three fundamental steps of creating a DSL - defining the language structure, the editor and the generator. Watch this short demo if you are new to MPS and want to dip your toe in the basics of the DSL definition process.

- **MPS basics - enhancing the language of constants** - This second in the series of introductory screen-casts aiming at MPS freshmen builds on the simple language for constants created in the previous episode, polishes several of its glitches and enhances the DSL with expressions and variable types. We’ll also touch on importing languages and defining simple constraints.

- **MPS basics - types and scopes of references** - In the third episode of the MPS introductory series of screen-casts we’re going to enhance our experimental constant language with references, simple type system rules and scoping constraints.

- **MPS basics - intentions and AST manipulation** - The fourth episode of the MPS basics series demonstrates Intentions and the ways you can manipulate the AST of your Domain Specific Languages. If you want to provide a smooth and pleasurable user experience to your DSLs, this screen-cast is for you.

- **MPS basics - generating text with TextGen** - In the fifth episode of the MPS beginners’ series you’ll familiarize yourself with the TextGen aspect. We’ll take our MPS-based DSL and generate Ruby code from it.

- **MPS basics - checking rules and quick fixes** - Static code analysis helps developers discover and eliminate bugs and problems in the code quickly. By highlighting suspicious pieces of code and offering automatic refactoring to fix the code, modern IDEs save development time and reduce the number of software defects. MPS expands this capabilities to the field of domain specific language.
CHAPTER 6. MPS SCREEN-CASTS

MPS basics - screen-casts covering the bundled sample projects

- The Java Extensions Sample
- The Simple State Machine (SecretCompartment Language)

6.0.6 Interoperability

MPS doesn’t live in vacuum. It can be used standalone as well as integrated into other development environments. Check out these screen-casts for the details.

- MPS languages inside IntelliJ IDEA - a parallel for loop example - The most notable new feature of MPS 2.5 is its integration with IntelliJ IDEA. You can use your DSLs and language extensions directly in your favorite Java IDE tightly integrated with your Java code. This screen-cast shows how this integration works and feels.

- Building an IntelliJ IDEA language plugin with MPS - This screen-cast will show you how to package your languages to make them available for IntelliJ IDEA. We’ll create a simple properties language, define a build script to generate and package the language and then use the properties language in IntelliJ IDEA directly.

- How to package your DSLs for IntelliJ IDEA - Watch this screen-cast to learn how to build and package your languages so that they can be shared with others and used inside both MPS and IntelliJ IDEA. We’ll create a build script for a sample language that will compile, build and package the language and create a proper IntelliJ plugin zip file.

- How to enable MPS in IntelliJ IDEA - Watch this screen-cast to learn how to add MPS core capabilities to IntelliJ IDEA and how to import third-party languages. We’ll go through the various options that enable the MPS plugin inside IntelliJ IDEA, configure the plugin, create our first demo and run it. We will then import a separate language plugin in order to use the contained language in our code.

6.0.7 Language Extension

Extending languages is one of the fundamental advantages of MPS. Watch these screen-casts and learn how to improve, customize and build upon existing languages.

- Language Extensions - Creating a new statement - Watch this screencast to learn about the MPS basics. We will discuss the internals of a simple language extension and explore the details of the language design process. This screen-cast is the first part of a series, which aims to gradually introduce the listeners into the fundamental principles of MPS.

- Language Extensions - Creating tabular expressions - In the second installment of the series we explore the implementation of a custom decision table expression. We will show how to define the structure, the editor and the type system rules for tables as well as the implementation of a generator, which transforms the table into a BaseLanguage expression. Watch this video to get a better command of tables or other non-trivial language constructs.

- Language Extensions - Creating new types and literals - In the third installment of the series we explore a custom money language. We will show how to customize the editor behavior through actions, how to override operators for your custom types, how to build a DSL on top of an existing Java library and most of all how to multiply money with MPS.

- Language Extensions - Describing the dataflow - In this episode of the series we investigate the dataflow aspect of language extensions. Watch the screen-cast and learn how to let MPS detect unreachable code, potential NullPointerExceptions or initialization problems for your custom language extensions.

- Language Extensions - Customizing suggested variable names - In the fifth episode of the series we briefly discuss the possibility to customize the list of suggested variable names. Each time the user creates a new variable either directly or through the "Introduce variable" refactoring, he or she is offered suggestions for names. While the default implementation would offer a name based on the actual type, designers of Base Language extensions may hook into the process and offer custom-tailored suggestions. This is a good example of how MPS lets you extend languages on a very fine-grained level.

- Language Extensions - Creating intentions - This time we will guide you through intentions. Intentions, (also called quick-fixes) offer users quick context-sensitive hints with suggestions for improving or altering the code at hands. Focusing on both plain and surround-with intentions, the screen-cast shows how to create them, hook them to the appropriate language elements and obviously also how to use them to your benefit.

- Memoization for java
- Tail Recursion Optimization for Java
- A parallel for loop for Java
CHAPTER 6  MPS SCREEN-CASTS

6.0.8  Advanced aspects of language definition

These screen-casts dive deeper into the waters of MPS language definition.

- Aspects of language definition - Typesystem - Watch this episode of the MPS screen-casts to learn more about the MPS typesystem. We discuss several ways to define types for custom languages and various aspects of the type resolution process in MPS.

- Smoothing the Editor Experience

- Optional Visibility in the Editor

- Using Editor Components

- Editor Transform Actions

- Node Substitution in the Editor

- Using the MPS Console to investigate and update the user models

- Aspects of language definition - Editor - In this episode we dive deeply into the editor aspect of language definition. On top of the fundamental principles of the editor specification we will also see ways to define optional elements, to control the cursor position and to customize the layout of the editor components.

- Aspects of language definition - Generator - Watching this episode you will learn about the details of code generation in MPS. The screen-casts will guide you through both text generators and model-to-model transformations. You will hear about the essential concepts of the MPS generator language, such as templates, macros or labels and see them all used in a concrete language definition. If you want to familiarize yourself with the code generation aspect of MPS, this screen-cast is for you.

- Extending existing languages through attributes - Watch this screen-cast to see how to leverage attributes in order to enhance capabilities of languages that you have no control over. Attributes allow you to add extra elements as children to existing concepts from other languages. They give you a powerful tool for language extension. Alongside the main topic you will get a brief tour round the new SampleJavaExtension sample project and find out what interesting aspects of MPS it covers.

- Checkpoints, cross-model generation and generation plans - This video shows how to define generation plans in order to enable cross-model generation. It also highlights th persisted checkpoint models, which help debug the persisted mapping labels.

6.0.9  Tutorials

These videos accompany the introductory Calculator Language tutorial.

- The introductory Calculator Language Tutorial - This sceencast provides an alternative way to learn MPS. Following the steps described in the on-line introductory MPS Tutorial the sceencast will guide you through the process of building a Calculator definition language and show you how to use the basic MPS concepts, such as the structure, the editor and the generator. If you find the on-line tutorial a bit dry and difficult to follow, this sceencast may help you get over it.

- The introductory Calculator Language Tutorial - constraints and type-system - This is a second part of the Calculator Language Tutorial covering constraints and type-system. This sceencast provides an alternative way to learn MPS. Following the steps described in the on-line introductory MPS Tutorial the sceencast will guide you through the process of creating scoping constraints and simple type-system rules for the Calculator Language we built in the first part.
Part II

Using MPS
Chapter 7

MPS Java compatibility
Chapter 8

Commanding the editor

When coding in MPS you will notice there are some differences between how you normally type code in text editors and how code is edited in MPS. In MPS you manipulate the AST directly as you type your code through the projectional editor. The editor gives you an illusion of editing text, which, however, has its limits. So you are slightly limited in where you can place your cursor and what you can type on that position. As we believe, projectional editor brings huge benefits in many areas. It requires some getting used to, but once you learn a few tricks you’ll leave your plain-text-editor colleagues far behind in productivity and code quality. In general only the items suggested by a completion menu can be entered. MPS can always decide, which elements are allowed and which are disallowed at a certain position. Once the code you type is in red color you know you’re off the track.

Code completion

Code completion (Control + Space) will be your good friend allowing you to quickly complete the statements you typed. Remember that CamelHumps are supported, so you only need to type the capital characters of long names and MPS will guess the rest for you.

Intentions

Frequently you can enhance or alter your code by means of predefined semi-automated procedures called Intentions. By pressing Alt + Enter MPS will show you a pop-up dialog with options applicable to your code at the current position. Some intentions are only applicable to a selected code region, e.g. to wrap code inside a try-catch block. These are called Surround With intentions and once you select the desired block of code, press Control + Alt + T to show the list of applicable intentions.

Navigation

Whenever you need to see the definition of an element you are looking at, press Control/Cmd + B or Control/Cmd + mouse click to open up the element definition in the editor. To quickly navigate around editable positions on the screen use the Tab/Shift + Tab key. Enter will typically insert a new element right after your current position and let you immediately edit it. The Insert key will do the same for a position right before your current position. When a piece of code is underlined in either red or yellow, indicating an error or a warning respectively, you can display a pop-up with the error message by pressing Control + F1.

Selection

The Control/Cmd + Up/Down key combination allows you to increase/decrease block selection. It ensures you always select valid subtrees of the AST. The usual Shift + Arrow keys way of text-like selection is also possible.

Investigation

To quickly find out the type of an element, press Control/Cmd + Shift + T. Alt + F12 will open the selected element in the Node Explorer allowing you to investigate the appropriate part of the AST. Alt + F7 will enable you to search for usages of a selected element. To quickly visualize the inheritance hierarchy of an element, use Control + H.

Inspector window

The Inspector window opens after you press Alt + 2. Some code and properties (e.g. editor styles, macros etc.) are shown and edited inside the Inspector window so it is advisable to keep the window ready.

We’ve prepared an introductory screen-cast showing you the basics of using the MPS editor.
8.0.10 Most useful key shortcuts

<table>
<thead>
<tr>
<th>Windows / Linux</th>
<th>MacOS</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control + Space</td>
<td>Cmd + Space</td>
<td>Code completion</td>
</tr>
<tr>
<td>Control + B</td>
<td>Cmd + B</td>
<td>Go To Definition</td>
</tr>
<tr>
<td>Alt + Enter</td>
<td>Alt + Enter</td>
<td>Intentions</td>
</tr>
<tr>
<td>Tab</td>
<td>Tab</td>
<td>Move to the next cell</td>
</tr>
<tr>
<td>Shift + Tab</td>
<td>Shift + Tab</td>
<td>Move to the previous cell</td>
</tr>
<tr>
<td>Control + Up/Down</td>
<td>Cmd + F9</td>
<td>Expand/Shrink the code selection</td>
</tr>
<tr>
<td>Shift + Arrow keys</td>
<td>Shift + Arrow keys</td>
<td>Select regions</td>
</tr>
<tr>
<td>Control + F9</td>
<td>Cmd + F9</td>
<td>Compile project</td>
</tr>
<tr>
<td>Shift + F10</td>
<td>Shift + F10</td>
<td>Run the current configuration</td>
</tr>
<tr>
<td>Control + Shift + T</td>
<td>Cmd + Shift + T</td>
<td>Show the type of the expression under carret</td>
</tr>
<tr>
<td>Alt + X</td>
<td>Control + X</td>
<td>Open the expression under carret in the Node Explorer to inspect the appropriate node and its AST surroundings</td>
</tr>
<tr>
<td>Control + H</td>
<td>Ctrl + H</td>
<td>Show the structure (inheritance hierarchy)</td>
</tr>
<tr>
<td>Ctrl + Alt + T</td>
<td>Cmd + Alt + T</td>
<td>Surround with...</td>
</tr>
<tr>
<td>Ctrl + O</td>
<td>Cmd + O</td>
<td>Override methods</td>
</tr>
<tr>
<td>Ctrl + I</td>
<td>Cmd + I</td>
<td>Implement methods</td>
</tr>
<tr>
<td>Ctrl + /</td>
<td>Cmd + /</td>
<td>Comment/uncomment the current node</td>
</tr>
<tr>
<td>Ctrl + Shift + /</td>
<td>Cmd + Shift + /</td>
<td>Comment/uncomment with block comment (available in Base-Language only)</td>
</tr>
<tr>
<td>Ctrl + X / Shift + Delete</td>
<td>Cmd + X</td>
<td>Cut current line or selected block to buffer</td>
</tr>
<tr>
<td>Ctrl + C / Ctrl + Insert</td>
<td>Cmd + C</td>
<td>Copy current line or selected block to buffer</td>
</tr>
<tr>
<td>Ctrl + V / Shift + Insert</td>
<td>Cmd + V</td>
<td>Paste from buffer</td>
</tr>
<tr>
<td>Ctrl + Shift + V</td>
<td>Cmd + Shift + V</td>
<td>Paste from history (displays a pop-up dialog that lists all previously copied code blocks)</td>
</tr>
<tr>
<td>Ctrl + Z</td>
<td>Cmd + Z</td>
<td>Undo</td>
</tr>
<tr>
<td>Ctrl + Shift + Z</td>
<td>Cmd + Shift + Z</td>
<td>Re-do</td>
</tr>
<tr>
<td>Ctrl + D</td>
<td>Cmd + D</td>
<td>Duplicate current line or selected block</td>
</tr>
</tbody>
</table>

8.0.11 A complete listing

Please refer to the Default Keymap Reference page for a complete listing of MPS keyboard shortcuts (Also available from the MPS Help menu).
Chapter 9

IDE configuration

Many aspects of MPS can be configured through the Settings dialog (Control + Alt + S / Cmd + ,).

To quickly navigate to a particular configuration items you may use the convenient text search box in the upper left corner. Since the focus is set to the text field by default, you can just start typing. Notice that the search dives deep into the individual screens:

![Settings dialog](image-url)
9.0.12 Plugins

MPS is modular and contains several plugins. If you open the MPS Plugin Manager you’ll see a list of plugins available in your installation.

Where to find language plugins are also listed here.

If some plugins are not necessary for your current work they can be simply switched off, which may have impact on the overall performance of the platform.
Chapter 10

Finding your way out

This document should give you instant step-by-step advice on what to do and where to look to get over a problem with MPS. It is an organized collection of patterns and how-tos fed with our own experience.

10.0.13 Reflective editor

Projectional editor by its nature presents the model to the user in a controlled way. Depending on the intent of the language designer the language may hide some information or some nodes from the user and prohibit some ways to manipulate the code. Also, if the editor definition is broken or incomplete in some sense, the editor may not allow the user to modify the code in a way she desires. Reflective editor provides the means to suppress the language editor and instead show the model in a default tree-like form. This way the developer has full and direct access to the model.

![Reflective editor](image)

F5 returns the editor to the normal.

10.0.14 Node Explorer

The Control + X keyboard shortcut gives the user a way to visualise the AST that represents the piece of code that has been selected in the editor.
10.0.15 Check out the type of the node

Knowing the type of the element you are looking at may give you very useful insight. All you need to do is pressing Control + Shift + T and MPS will pop-up a dialog window with the type of the element under carret.
10.0.16  Check the concept of the node under carret

The \textit{Control + Shift + S/Cmd + Shift + S} keyboard shortcut will get you to the definition of the concept of the node you are currently looking at or that you have selected.

10.0.17  Check the editor of the node under carret

The \textit{Control + Shift + E/Cmd + Shift + E} keyboard shortcut will get you to the definition of the editor for the concept you are currently looking at or that you have selected. This may be in particular useful if you want to familiarize yourself with the concrete syntax of a concept and all the options it gives you.

![Keyboard shortcuts](image)

10.0.18  Type-system Trace

When you run into problems with types, the \textit{Type-system Trace} tool will give you an insight into how the types are being calculated and so could help you discover the root of the issues. Check out the details in \textit{Typesystem} and in \textit{Typesystem Debugging}.

10.0.19  Investigate the structure

When you are learning a new language, the structure aspect of the language is most often the best place to start investigating. The shortcuts for easy navigation around concepts and searching for usages will certainly come in handy.
You should definitely familiarize yourself with Control + B / Cmd + B (Go To Definition), Control + N / Cmd + N (Go To Concept), Control + Shift + S / Cmd + Shift + S (Go To Concept Declaration) and Alt + F7 (Find usages) to make your investigation smooth and efficient.

Before you learn the shortcuts by heart, you can find most of them in the Navigate menu:
10.0.20 Importing elements

You are trying to use an element or a language feature, however, MPS doesn’t recognize the language construct or doesn’t offer that element in the code-completion dialog. So you cannot update your code the way you want. This is a symptom of a typical beginner’s problem - missing imports and used languages.

- In order to use language constructs from a language, the language has to be listed among Used Languages.
- To be able to enter elements from a model, the model must be imported first.
- Also, for your languages to enhance capabilities of another language, the language must be listed among the Extended Languages.

To quickly and conveniently add models or languages to the lists, you may use a couple of handy keyboard shortcuts in addition to the the Properties dialog:

Add Model Import Ctrl+M
Add Model Import by Root Ctrl+R
Add Language Import Ctrl+L

10.0.21 Save transient models

If you are getting errors from the generator, you may consider turning the Save Transient Models functionality on. This will preserve all intermediate stages of code generation for your inspection.
10.0.22 Why the heck do I get this error/warning?

You see that MPS is unhappy about some piece of code and you want to find out why. Use Control + Alt + Click / Cmd + Alt + Click to open up a dialog with the details.

```java
import java.util.Arrays;

public static void main(String[] args) {
    System.out.println("Foo");
    System.out.println("Bar");
    for (int i : Arrays.asList(1, 2, 3, 4, 5)) {
        System.out.println("\n" + i);
    }
}
```

Warning: Unused static field `StaticField`
The *Go To Rule* button will get you to the rule that triggers the error/warning.
Chapter 11

Where to find language plugins

MPS can be easily extended with additional languages. Languages come packaged as ordinary zip files, which you unzip into the MPS plugin directory and which MPS will load upon restart.

The most convenient way to install language plugins is through the Plugin Manager, which is available in the Settings dialog (Control + Alt + S / Cmd + .).

You can either install a zip file you’ve received previously (the Install plugin from disk... option) or you may click the Browse repositories button and pick the desired plugin from the list of plugins that have been uploaded to the MPS plugin reposi-
CHAPTER 11. WHERE TO FIND LANGUAGE PLUGINS

This component allows to editing properties like in a normal text editor. It supports:
- wrapping of long lines
- inserting line break by pressing enter
- pasting text that contains line breaks

Change Notes
Selection behavior fixed. It is now possible to select across word boundaries and COPY, CUT, PASTE multiple words.

Vendor
Sascha Lissan
http://github.com/lissan/mps-multiline

Version
0.1.3
Size: 126.4 K
Chapter 12

Version Control

12.1 Compare two nodes

Any two arbitrary nodes in the **Project View** tool window can be visually compared:

The standard VCS comparison dialog shows up that visualizes the mutual differences and allows easy modifications.

12.2 VCS menu

The VCS menu contains commands and configurations related to version control:
12.3 VCS configuration

In order to configure VCS for your project, open the Preferences (Control + Alt + S or Cmd + ,) and choose the Version Control item:

Among other things you must configure the project roots for the individual version control systems used.
12.4 Changes View

The Changes View tool window at the bottom (Alt + 9 or Cmd + 9) lists all files/models that have been modified. The view can be configured using the buttons on the side of the window. A context pop-up menu provides quick access to frequently used VCS-related actions applicable to the selected items:

The Log tab of the Changes View visualizes the commit history:

12.5 VCS Add-ons

When you first open MPS with version control or add VCS mapping for existing project, it offers you installing some global settings and install so called VCS Add-ons (they can also be installed from main menu: Version Control → Install MPS VCS Add-ons).
12.5.1 What are VCS Add-ons

VCS Add-ons are special hooks, or merge drivers for Subversion and Git, which override merging mechanism for special types of files. In case of MPS, these addons determine merging for model files (*.mps) and generated model caches (dependencies, generated and trace.info files, if you store them under version control). Every time you invoke some version control procedure (like merging branches or Git rebasing) which involves merging file modifications, these hooks are invoked. For models, it reads their XML content and tries to merge changes in high level, "model" terms (instead of merging lines of XML file which may lead to invalid XML markup). Sometimes models cannot be merged automatically. In that case, it stays in "conflicting" state, and it can be merged in UI of MPS.

In some cases during work of merge driver, there may happen id conflicts, situations when model has more than one node with the same id after applying all non-conflicting changes. In this situation, no automatic merging is performed, because it may lead to problems with references to nodes which are hard to find. In this case you should look through merge result by himself and decide if it okay.

For model caches merge driver works in a different way (if you store them under version control, of course). Generator dependencies (generated files) and debugger trace caches (trace.info files) are just cleared after merging, so you will need to regenerate corresponding models. Java dependencies (dependencies files) which are used on compilation are merged using simple union algorithm which makes compilation possible after merging.

12.5.2 Different VCS Add-ons

Look at the dialog:

There are several types of VCS Add-ons which can be installed. It is recommended to install them all.

- **Git global autocrlf setting.** Forces git to store text files in repository with standart Unix line endings (LF), while text files in working copy use local system-dependent line endings. Necessary when developers of your project use different operating systems with different line-endings (Windows and Unix).

- **Git global merge driver setting.** Registers merge driver for MPS models in global Git settings so they can be referred in .gitattributes files of Git repositories (see below). It only maps merge driver name (in this case, "mps") with path to actual merge driver command.

- **Git file attributes for repositories.** Enables MPS merge driver for concrete file types (*.mps, trace.info, etc) s in Git repositories used in opened MPS project. This creates or modifies .gitattributes file in root of Git repository. This file usually should be stored under version control so these settings will be shared among developers of the project.

- **Subversion custom diff3 cmd.** Registers MPS merger in config file of Subversion. MPS may use its own config folder for Subversion, so there are two different checkboxes. One updates global config used when you invoke Subversion procedures from command line or tools like TortoiseSVN. Another one modifies config only for MPS Subversion plugin. By the way, directory for Subversion config used in MPS can be defined in Subversion settings.
Chapter 13

Using MPS Debugger

13.1 Using MPS Debugger

MPS Debugger provides an API to create debuggers for custom languages. Java Debugger plugin, included into MPS distribution, allows user to debug programs which are written in languages which are finally generated into Base Language/Java. We use this plugin below to illustrate MPS Debugger features, which all are available to other languages via API.

13.1.1 Execution

We start with description of how to debug a java application. If a user has a class with main method, a Java Application run configuration should be used to run/debug such a program.

Creating an instance of run configuration

A Java Application or an MPS instance run configurations can be created for a class with a main method or an MPS project, respectively. Go to Run -> Edit Configurations menu and press a button with "+" as shown at the picture below:

A menu appears, choose Java Application from it and a new Java Application configuration will be created:
If you select **Java Application**, you will be able to specify the Java class to run, plus a few optional configuration parameters:

A name should be given to each run configuration, and a main node i.e. a class with a main method should be specified. Also VM and program parameters may be specified in a configuration. Look at Run Configuration chapter to learn more about run configurations.

**Debugging language definitions**

Select **MPS instance**, if you want to debug MPS language definition code. MPS will start a new instance of MPS with a project that uses your language (it could also be the current project) and you will set breakpoints and debug in your original MPS instance.

In the **Debug configuration** dialog you need to indicate, which MPS project to open in the new MPS instance - either the current one by checking the **Open current project** check-box, or any project you specify in the field below. You could also leave both empty and create/open a project from the menu once the new MPS instance starts.
Debugging a configuration

To debug a run configuration, select it from configurations menu and then press the Debug button. The debugger starts, and the Debugger tool window appears below.

There are two tabs in a tool: one is for the console view and other for the debugger view. In the console an application’s output is shown.

13.1.2 Breakpoints

Next section describes breakpoints usage.
CHAPTER 13. USING MPS DEBUGGER

Setting a breakpoint

A breakpoint can be set on a statement, field or exception. To set or remove a breakpoint, press Ctrl-F8 on a node in the editor or click on a left margin near a node. A breakpoint is marked with a red bubble on the left margin, a pink line inside the editor and a red frame around a node for the breakpoint. Exception breakpoints are created from the breakpoints dialog. When the program is stared, breakpoints on which debugger can not stop are specially highlighted.

When debugger stops at a breakpoint, the current breakpoint line is marked blue, and the actual node for the breakpoint is decorated with black frame around it.

Viewing breakpoints via breakpoints dialog.

All breakpoints set in the project could be viewed via Breakpoints dialog.

Java breakpoints features include:

- field watchpoints;
- exception breakpoints;
- suspend policy for java breakpoints;
- relevant breakpoint data (like thrown exception or changed field value) is displayed in variables tree.

Examining a state of a program at a breakpoint

When at a breakpoint, a Debugger tab can be used to examine a state of a program. There are three panels available:

- a "Frames" panel with a list of stack frames for a thread, selected using a combo box;
- a "Variable" tree which shows watchables (variables, parameters, fields and static fields) visible in the selected stack frame;
• a "Watches" panel with list of watches and their values.

In java debugger "Copy Value" action is available from the context menu of the variable tree.

13.1.3 Runtime

Controlling execution of a program

• To step over, use Run -> Step Over or F8.
• To step out from a method, use Run -> Step Out or Shift-F8.
• To step into a method call, use Run -> Step Into or F7.
• To resume program execution, use Resume button or Run -> Resume or F9.
• To pause a program manually, use Pause button or Run -> Pause. When paused manually i.e. not at a breakpoint, info about variables is unavailable.

There is a toolbar in Debugger window from where stepping actions are available.

13.1.4 Expressions

Expression evaluation

MPS Java debugger allows user to evaluate expressions during debug, using info from program stack. It is called low-level evaluation, because user is only allowed to use pure java variables/fields/etc from generated code, not entities from high-level source code.

To activate evaluation mode, a program should be stopped at a breakpoint. Press Alt-F8, and a dialog appears. In a dialog there’s a MPS Editor with a statement list inside it. Some code may be written there, which uses variables and fields from stack frame. To evaluate this code, press Evaluate button. The evaluated value will appear in a tree view below.
To evaluate a piece of code from the editor, select it and press Alt+F8, and the code will be copied to the evaluation window.

Watches

Watches API and low-level watches for java debugger are implemented. "Low-level" means that user can write expressions using variables, available on the stack. To edit a watch, a so-called "context" (used variables, static context type and this type) must be specified. If the stack frame is available at the moment, context is filled automatically.
Watches can be viewed in "Watches" tree in "Debug" tool window. Watches could be created, edited and removed via context menu or toolbar buttons.
Chapter 14

Migrations

After a language has been published and users have started using it, the language authors have to be careful with further changes to the language definition. In particular, removing concepts or adding and removing properties, children and references to concepts will introduce incompatibilities between the previous and the next language version. This impacts the users of the language if they update to the next language version, since they may discover that their model no longer matches the language definitions and get appropriate errors reported from their models.

MPS tracks versions of languages used in projects and provides automatic migrations to upgrade the usages of a language to the most recent versions. The language designers can create maintenance "migration" code to run automatically against the user code and thus change the user’s code so that it complies with the changes made to the language definition. This is called language migration.

The full language migration story has several aspects:

- Language designers can write scripts for migrating the user code and bundle them with the language
- MPS automatically tracks language versions used in the client code
- MPS controls that the user’s project is up-to-date with all language changes
- MPS runs the necessary migrations, when necessary

There are two types of migrations available in MPS:

- Language migrations - migrations that upgrade the project to comply with the next version of the language definition. Each language migration is attached to a version of the language definition.
- Project migrations - these are not triggered by language usages, but instead they themselves define the conditions, under which they should be run. These migrations are always applied to the whole project.

14.1 Language version

Languages store a version number in their module definition (.mpl) file. This number increases when a new migration is created in a language’s "migration" aspect

Modules that use languages contain a version number associated with each used language reference in the module (.msd, .mpl) file. These represent the language version used by the module. The number changes when the corresponding migration is run against this module to migrate it to a later language version.

The version number of a language can be viewed and modified manually in the Properties dialog for a language:

![Language Properties for nf](image)

Notice that there are two numbers available:

- Language version - updated each time the structure of the language changes
- Module version - updated each time the references to the nodes in the module were migrated. If you perform a migration on a module with sources, e.g. moving nodes, you need a migration, which will be run on references or on depending modules. Module version tracks that.
14.2 Migration assistant

When MPS detects that the modules within the currently open project refer to versions of languages older than the ones present, a Migration assistant is run. It prompts the user whether the migrations should be run in order to update the project to the most recent versions of the languages.

A detailed list of the migrations that will be run is presented to the user:

- **Project Migrations**
  - Remove .history files
  - Update generators of lang.editor extensions
  - Make empty language aspects 'generation required'
- **Language Migrations (7 modules)**
  - j.m.lang.smodel
    - Migrate popular concept<->node conversions
    - Migrate node/concept cast expressions
    - Migrate node<>.delete
    - Migrate smodel operations and node/name references
  - j.m.lang.dataFlow
    - Employ GP and devkit for dataflow aspect models
  - j.m.lang.behavior
    - Migrate 'this' and 'super' in static behavior methods
  - j.m.lang.structure
    - Employ dedicated devkit to specify used languages
    - Annotate 'smart reference' concepts
  - j.m.lang.typesystem
    - Employ GP and devkit for typesystem aspect models
  - j.m.lang.constraints
    - Migrate concept and link declaration nodes in 'canBe* function parameters
    - Migrate concept and link declaration nodes in scope factory parameters
    - Annotate ref. presentation queries as migrated
    - Employ GP and devkit for constraint aspect models

If the user triggers the migration, the project is fully migrated. In case of problems preventing the migration, a list of problems together with the list of not migrated code is presented to the user.
14.3 Defining language migrations

Migrations are defined as Migration Classes in the migrations aspect of your language definition. Migration Classes are nodes of the MigrationScript concept defined in the jetbrains.mps.lang.migration language.

14.3.1 Numbering of languages and migrations

- The name of each migration script holds a number
- Each migration script defines a from version property

When a new migration script is created, the language version is increased by 1 and the fromVersion field in the migration is set to old value of the language version. We can now say that the created migration script performs the migration from an old version to a new one.

Numbering of languages and migrations tips and tricks

- No migrations can be "missed". If a language contains a migration from version X and from version Y, it should also contain a migration for each versions between X and Y. If a migration is not found for some version, this means that no user is able to migrate from version X. Generation of such languages will end up with an error.
- It's not necessary to store all migrations for a language. If some language was "published" and it's necessary to remove some of the older migrations, they could be removed. The from-versions of migrations left should form a range A..B, where A is any older version and B = <current version> - 1
- If a migration is created by mistake and wasn’t published (meaning no user has run it on his project), it can be freely removed. After removing the migration, execute "Correct Language version" from the language's context menu - this action allows to synchronize the language's version with the last migration's version. BE VERY CAREFUL when doing this.
### 14.3.2 Structure of a migration

There are several optional elements that migrations may provide:

- **execute after** - to put an ordering constraint among migration scripts
- **required data** - specifies the `ConceptDeclaration` that will be used to represent the migration data produced by an earlier migration script. It also gives the data a logical name to represent it within this migration script.
- **produced data** - specifies the `ConceptDeclaration` that will be used to hold the migration data produced by this script and possibly consumed by a later migration script.
- **description** - a helpful textual description of the script
- **execute method** - each migration defines an `execute()` method, which performs the actual model conversion for user models. The method receives the user module as a parameter and may refer to the defined elements in the `required data` section.

#### Data production and consumption

The ability to pass data among migration scripts is useful in partitioning the migration process. One migration script may, for example, migrate nodes from an old concept to a new one, while a following migration script will migrate all references to the original nodes to point to the new nodes. For this to work, the first script has to store **ids** of the old and new nodes and publish the mapping as its produced data. The second migration script will consume the data as required data. Each time a reference to an old node has to be updated, the data will be used to find an id of the new node.

### 14.3.3 Ordering of migration scripts

The implicit dependencies between migration scripts expressed through the `required data` and `produced data` sections will take care of proper ordering of migration scripts. When script is migrating some module, it can use data stored for this module and all its dependencies, so consuming script will start migrating the module only after having run all the required producers on all dependencies of the module. There is no need to express those dependencies explicitly. However, in cases when it is necessary to execute some script only after some other scripts has been executed against the same module (without taking care about dependencies), such ordering constraint can be expressed through the `execute after` section. If, for example, some property was moved from one concept to its superconcept, which happens to be declared in another language, the migration can be expressed with two migration scripts. The first script, applicable to the subconcept, copies the property value from the old deprecated property to the new one. The second script is applicable to the superconcept, it initiates the new property for such instances of the superconcept, which are not instances of the subconcept, with some default value. And let us suppose that the second script does some other initialization which depends on value of the moved property. So, the second script should be executed only after the first one, and that on every module.

### 14.3.4 Languages for defining migrations

The `jetbrains.mps.lang.migration` language defines all concepts specific to migration scripts. When defining your migrations, you can either use `BaseLanguage` together with the `jetbrains.mps.lang.smodel` and `jetbrains.mps.lang.smodel.query` languages to manipulate the models. The `ofType<model>` construct may be of particular use to obtain models contained in the passed-in `SModule`:

```java
sequence<SModel> models = m.getModels();
models.ofType<model>.selectMany({~model => model.nodes(BaseDocComment); }).forEach({~node => ... });
```

A typical migration first excludes the migration aspect models from migration and then scans for nodes that need to be migrated. A new node is created and initialised with the values and children of the old node. The old node is then replaced with the new node. Setting the id of the new node to the value of the id of the old node will allow references to this node to be migrated without loosing their target.
void execute(SModule m) {
    sequence<model> models = ((sequence<model>) m.getModels()).where({-it => !it.isAspectModel(migration); });
    models.selectMany({-m => m.nodes(OldComponent); }).forEach({-oldNode =>
        node<NewComponent> newNode = <NEW component $( oldNode.name )$ {>
            *( oldNode/member )*
        ((SNode) newNode/).setId(((SNode) oldNode/).getNodeId());
        oldNode.replace with(newNode);
    });
}

Schematically:

1. The transformation is applied to some node. As a result, we have a reference to old node (call in No), and a new node (Nn).
2. IDs of No’s descendants are preserved automatically: if a was-descendant node is a descendant of the output node after the transformation, it already has the same id.
3. ID of No: MPS determines whether No is a descendant of an output node.
   (a) If yes, we already have the target for references that pointed to the No (this is for “wrap” cases - the node is “wrapped” in another node as a result of the transformation)
   (b) If no, the Nn gets the ID of No (that’s for the case when we changed the concept of a node, but the old node is semantically equivalent to the new one)
4. No is replaced with Nn in the containing model.

It’s sometimes much simpler to decompose a complex migration into a sequence of transform-migrations rather then writing the data-persisting logic and pass it and produced/required data.

14.4 Concept replacement

If a language designer decides to remove a language concept and perhaps replace it with a new one, she should not remove the concept definition from the language immediately. Instead, the concept should be deprecated first and a migration script should be provided to migrate the user code away from the deprecated concept.

It is generally a good strategy to move the concepts that are being deprecated into a separate virtual folder to keep them separate from the concepts that form the current way of using the language. Changing a virtual package of a concept has no impact on the using references in user models.

The deprecated concept can be completely removed (but does not need to) in the version following after the one, in which it was deprecated. The migration scripts that refer to the deprecated concept have then be removed, too.

14.5 Defining project migrations

Project migrations are not typically used by language developers, but rather by the MPS team to describe changes in the model file format, in the module dependencies system and other project-wide things. Project migration are run against the whole project, so it’s up to the MPS developer to think about how his migration will work when a part of a project changes. E.g. the user can update her project from the VCS, and in this case it may be not enough to know, that the project was migrated once; updated modules may still have to be migrated. MPS does not guarantee the order, in which project migrations will be run, so you basically can’t write mutually dependent project migrations.

Nevertheless, users can write their own project migrations. There’s no special language for project migrations, so they are basically written as Java/BaseLanguage classes and are contributed through plugin.xml. So’ further we’ll suppose that you already have an MPS plugin and write the project migration in it.

Note that if a project migration is written in a solution, this solution must have the IdeaPlugin enabled in the Facets tab of the Solution Properties dialog and the plugin id set in the Idea Plugin tab.
14.5.1 Adding a new project migration

- Create a class for the migration implementing the ProjectMigration interface. For most cases, it’s convenient to inherit from the BaseProjectMigration class.
- Create an ApplicationComponent that will contribute the new migrations. Do not forget to register it in plugin.xml.
- Contribute all your project migrations from created ApplicationComponent using the ProjectMigrationsRegistry.addProjectMigration() method.

14.5.2 Saving data from project migrations.

Project migrations can use the MigrationProperties project component to persist their data. The persisted data is stored in the .mps folder of the project and so it is shared between project’s developers through VCS.

14.6 Multiple branches

Migrating projects that use multiple branches has a few additional challenges. Check out the Using Migrations with branching documentation for details.

14.7 Migration Ant Task

There’s an ant task to run all migrations in a project from an ant script. This task can be used for automatic testing of migrations and/or for checking whether a project has been migrated. This task requires the MPS home path to be set by:

1. defining mpshome task attribute or
2. defining mps_home environment property or
3. defining mps.home environment property - this is the preferred way

Home path is the path to the folder that contains the build.txt file. E.g. under Mac OS this will end with "/Contents/"

Repository contents may be specified using the <repository> tag:

```xml
<migrate project="${projectPath}" mpshome="${mpsHome}">
  <repository>
    <allmpsmodules/>
  </repository>
</migrate>
```

14.8 Examples

For concrete examples on how to define migrations you can check out the migrations sample project that comes bundled with MPS. You will see migration scripts to migrate two simple mutually interconnected languages. One of them uses data to pass information about migrated nodes between two migration scripts, while the other relies of node id manipulation.

14.9 Changes made by migrations in Local History view

Migrations cooperate with the Local History functionality.
After running migrations, it’s possible to review all the changes made to the project by each of the migration. Open the Local History view for the project’s folder, a module or a model, select any two changes and press Ctrl + D to see the difference.
It’s also possible to revert a change or a group of changes from the Local History view as well as from the Diff dialogs.

14.10  Migration assistant in IntelliJ IDEA

The IntelliJ IDEA plugin can also run language migrations. Just like in MPS itself, the Migration assistant will update models in IDEA projects to match the currently installed versions of used languages.
Chapter 15

Using Migrations with branching

15.1 Using migrations with branching

To nicely support language evolution, migrations automatically upgrade client code to use the latest version of a language or multiple languages. The Migrations documentation describes the process of creating and running migrations in MPS. Briefly, in order to change models to comply with the language changes, language authors can provide migrations. MPS automatically tracks language versions used in projects and asks the end-user to apply needed migrations. While there are hardly any surprises when using a single branch of development, typical team setup involves multiple branches for features as well as bug fixes. With multiple branches teams will inevitably face situations like having different versions of languages in different branches, which makes the merge procedure much more challenging.

15.1.1 The main rule of painless merge

To merge two branches smoothly, all languages used in those branches should be of the same versions. The following describes a simple workflow that allows you to easily synchronise language versions in two branches.

No migrations were created in any of the merged branches

To sync language versions, just update languages in the outdated branch and run required migrations. Now both branches have the same versions of used languages and can be easily merged.

Some migrations were created

All considered cases have the same idea behind them: never merge branches that differ in versions of used languages. Thus we need to propagate all the migrations into both branches before merging them.

1. One migration was developed in one of the branches

The only special thing that should be done when you write a migration is that there should be a commit (name it C) containing the migration itself, but no code should be migrated in the same commit. The migrated files can be committed later, e.g. in the following commit. The main idea here is to give other branches a possibility to get the new migration without having to merge different versions of used languages.

Now, your history looks like this:

```
|  --- C  --- [branch 1]
|       |
|________ [branch 2]
```

To merge branches b1 and b2:

1. Merge b2 with commit C. Now you have the new migration in b2.
2. Run the migration on b2
3. Merge with b1’s HEAD

Quite simple, isn’t it?

It is a good habit to create a separate commit whenever you’re adding a new language migration. This allows other branches to get your migration safely without having to bother with the upgrades that the migration caused to code in your branch.
2. Two or more migrations in one of the branches. The principle remains the same, it is just repeated two or more times.

\[\text{\begin{verbatim}
\text{C1} - C2 - [branch 1]
\end{verbatim}\]}

1. merge b2 with C1
2. run C1's migration on b2
3. merge with C2
4. run the migration on b2
5. merge with b1

3. Migrations were created in both branches. In general, we recommend you to avoid this situation as it's harder to resolve. This scenario can best be avoided by having all migrations created in only one agreed-upon branch (e.g., master). But if you got into this situation, here's a way to resolve it gracefully.

\[\text{\begin{verbatim}
\text{C1} - [branch 1]
\end{verbatim}\]}

\[\text{\begin{verbatim}
\text{C2} - [branch 2]
\end{verbatim}\]}

1. Merge b2 and C1, b1 and C2
2. Run C1's migration on b2, C2's migration on b1
3. merge the branches
Chapter 16

Console

Console is a tool which allows developers to conveniently run DSL code directly in the MPS environment against the active models. It enables you to quickly query the model and change it. You can trigger actions against your models or study statistics about your project.

For example, you can quickly get all (or some) instances of a (deprecated) concept and migrate them to a new concept:

```plaintext
#instances(TryStatement).where({~it => it.catchClause.isNotEmpty; }).refactor({~node =>
    if (node.body.statement.size > 5) {
        node.replace with new(TryCatchStatement);
    }
});
```

The Console tool window allows line-by-line execution of any DSL construction in the realtime. After the command is written in the console, it is generated by the MPS generator and executed in the IDE’s context. This way the code in the console can access and modify the program’s AST, display project statistic, execute IDE actions, launch code generation or initiate classes reloading.

For discoverability reasons, most of the console-specific DSL constructs start with symbol ‘#’. Code-completion (Control + Space) assists developers to insert code in the console.

We shot a short informative screen-cast about using the MPS Console to investigate and update the user models. Check it out!

In general, there are 3 kinds of commands:

1. BaseLanguage statement lists. These commands can contain any BaseLanguage constructions. If some construction or class is not available in completion, it may not have been imported. Missing imports can easily be added as in the normal editor, using actions ‘Add model import’, ‘Add model import by root’, ‘Add language import’, or by the corresponding keyboard shortcuts.
2. BaseLanguage expressions. Expression is evaluated and, if its type is not void, printed in console as text, AST, or interactive response.

   > 2 + 2
   4

3. Non-BaseLanguage commands. These are simple non-customizable commands, such as `#reloadClasses`.

### 16.0.2 Console commands

There is also a set of languages containing the console commands and BaseLanguage constructions, which allow developers to easily make custom refactorings, complex usages search etc.

1. BaseLanguage constructions for iterating over IDE objects (`#nodes`, `#references`, `#models`, `#modules`). These expressions are lazy sequences, including all nodes/references/models/modules in project or in custom scope.
To inspect read-only modules and models, such as imported libraries and used languages, you need to include the \texttt{r/o+} parameter to the desired search scope.

2. \textit{BaseLanguage} constructions for usages searching (\texttt{#usages}, \texttt{#instances}). These expressions are also sequences, which can be iterated over, but not lazy. When these expressions are evaluated, find usages mechanism is called, so it runs faster then iterating over all nodes or references and then filtering by concept/target.

\begin{verbatim}
> #usages{scope = project>(node/SwingUtilities/ -> invokeLater)
49 references
> #instances(DotExpression),where{it -> it.operation.isInstanceOf(AllOperation); }
50 nodes
\end{verbatim}

3. Commands for querying data from the IDE (\texttt{#stat}, \texttt{#showBrokenRefs}, \texttt{#showGenPlan})

4. Commands for interacting with the IDE (\texttt{#reloadClasses}, \texttt{#make}, \texttt{#clean}, \texttt{#removeGenSources})

\begin{itemize}
  \item To initiate a \textit{rebuild} of a model, first invoke \texttt{#clean} followed by \texttt{#make}.
\end{itemize}

5. \textit{BaseLanguage} constructions for showing results to user

\begin{itemize}
  \item \texttt{#show} expression opens usages view and shows there nodes, models or modules from sequence passed to the expression as a parameter.
  \item \texttt{#print} expression writes result to the console. There are also specialized versions of this construction:
    \begin{itemize}
      \item \texttt{#printText} converts result to string and add it to the response.
      \item \texttt{#printNode} is applicable only to nodes. This construction adds to response the the whole node and its subnodes. Since the response is also part of the AST, the node is displayed with its normal editor.
      \item \texttt{#printNodeRef} makes sense with only nodes locates in the project models. This construction prints to the console an interactive response, which can be clicked on in order to open the node in the editor.
      \item \texttt{#printSeq} is applicable to collections of nodes, models or modules. This command prints to the console an interactive response, which describes the size of the collection. When the response is clicked on, the usage view opens to show the nodes or the models.
      \item \texttt{#print} expression is a universal construction, which tries to choose the most appropriate way of displaying its argument, according to its type and value
    \end{itemize}
\end{itemize}

6. \textit{refactor} operation. This operation applies a function to sequence of nodes (like \texttt{forEach} operation), but before that it opens the found nodes in the usages view, where user can review the nodes before the refactoring is started and manually select the nodes to include/exclude in the refactoring and then apply or cancel the refactoring.

\begin{verbatim}
> #instances(ClassConcept),where{(~node =>
  node.name.isNotBlank || Character.isLowerCase(node.name.charAt(0));
).refactor{(~node =>
  node.name = Character.toUpperCase(node.name.charAt(0)) + node.name.substring(1, node.name.length());
})
\end{verbatim}

Additionally, the console languages can be extended by the user, if needed.

The model-querying commands used in the Console are defined in the \texttt{jetbrains.mps.lang.smodel.query} language. After importing the language you can use the commands in code to programmatically access and query the models, as well. Details can be found in the \texttt{SModel language Queries}.

\subsection*{16.0.3 Scopes of Console queries}

Queries can have scope specified to constraint the area of the repository to search. The scope is specified in pointy brackets:
• **project** - only the modules in the current project

• **editable** - only editable models in the current project (this is the default scope used when no scope is specified explicitly)

• **global** - the whole module repository

• **visible** - only visible modules from global repository

• **modules** - restrict to the listed modules

• **models** - restrict to the listed models

• **custom** - use the provided Java Scope class to filter the modules

### 16.0.4 Copying nodes into Console

In order to point to a concrete node in project from the console, this node can be copied from the editor and then pasted into the console. The node will be pasted as a special construction, called `nodeRef`, with a `BaseLanguage` expression of type `node<>`, with value of the pasted node. If there is a necessity to paste the piece of code as is, the 'Paste Original Node' action is available from the context menu.
Part III

Defining Languages
Chapter 17

Structure

Since MPS frees you from defining a grammar for your intended languages, you obviously need different ways to specify the structure of your languages. This is where the Structure Language comes in handy. It gives you all the means to define the language structure. As we discussed earlier, when coding in MPS you’re effectively building the AST directly, so the structure of your language needs to specify the elements, the bricks, you use to build the AST.

The bricks are called Concepts and the Structure Language exposes concepts and concept interfaces as well as their members: properties, references, children, concept(-wide) properties, and concept(-wide) links.

17.1 Concepts and Concept Interfaces

Now let’s look at those in more detail. A Concept defines the structure of a concept instance, a node of the future AST representing code written using your language. The Concept says which properties the nodes might contain, which nodes may be referred to, and what children nodes are allowed (for more information about nodes see the Basic notions section). Concepts also define concept-wide members - concept properties and concept links, which are shared among all nodes of the particular Concept. You may think of them as “static” members.

Apart from Concepts, there are also Concept Interfaces. Concept interfaces represent independent traits, which can be inherited and implemented by many different concepts. You typically use them to bring orthogonal concepts together in a single concept. For example, if your Concept instance has a name by which it can be identified, you can implement the INamedConcept interface in your Concept and you get the name property plus associated behavior and constraints added to your Concept.
17.1.1 Concepts inheritance

Just like in OO programming, a Concept can extend another Concept, and implement many Concept Interfaces. A Concept Interface can extend multiple other Concept Interfaces. This system is similar to Java classes, where a class can have only one super-class but many implemented interfaces, and where interfaces may extend many other interfaces.

If a concept extends another concept or implements a concept interface, it transitively inherits all members (i.e if A has member m, A is extended by B and B is extended by C, then C also has the member m)

Concept interfaces with special meaning

There are several concept interfaces in MPS that have a special meaning or behavior when implemented by your concepts. Here's a list of the most useful ones:

<table>
<thead>
<tr>
<th>Concept Interface</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDeprecatable</td>
<td>Used if instances of your concept can be deprecated. It’s <code>isDeprecated</code> behavior method indicates whether or not the node is deprecated. The editor sets a <em>strikeout</em> style for reference cells if <code>isDeprecated</code> of the target returns true.</td>
</tr>
<tr>
<td>INamedConcept</td>
<td>Used if instances of your concept have an identifying name. This name appears in the code completion list.</td>
</tr>
<tr>
<td>IType</td>
<td>Is used to mark all concepts representing types</td>
</tr>
<tr>
<td>IWrapper</td>
<td>Deleting a node whose immediate parent is an instance of IWrapper deletes the parent node as well.</td>
</tr>
</tbody>
</table>

17.2 Concept members

17.2.1 Properties

Property is a value stored inside a concept instance. Each property must have a type, which for properties is limited to: primitives, such as boolean, string and integer; enumerations, which can have a value from a predefined set; and constrained data types (strings constrained by a regular expression).

17.2.2 References

Holding scalar values would not get as far. To increase expressiveness of our languages nodes are allowed to store references to other nodes. Each reference has a name, a type, and a cardinality. The type restricts the allowed type of a reference target. Cardinality defines how many references of this kind a node can have. References can only have two types of cardinalities: 1:0..1 and 1:1.

Smart references

A node containing a single reference of 1:1 cardinality and with no alias defined is called a smart reference. These are somewhat special references. Provided the language author has not specified an alias for them, they do their best to hide from the language user and be as transparent as possible. MPS treats the node as if it was a the actual reference itself, which simplifies code editing and code-completion. For example, default completion items are created whenever the completion menu is required: for each possible reference target, a menu item is created with matching text equal to the presentation of a target node.

In order to make a reference smart when it does not meet the above mentioned criteria for being treated as smart automatically, the concept declaration has to be annotated with the `@smart reference` attribute: A typical use-case would be a concept that customizes the presentation of the reference or holds additional references.
17.2.3 Children

To compose nodes into trees, we need to allow children to be hooked up to them. Each child declaration holds a target concept, its role and cardinality. Target concept specifies the type of children. Role specifies the name for this group of children. Finally, cardinality specifies how many children from this group can be contained in a single node. There are 4 allowed types of cardinality: 1:1, 1:0..1, 1:0..n, and 1:1..n.

17.2.4 Specialized references and children

Sometimes, when one concept extends another, we not only want to inherit all of its members, but also want to override some of its traits. This is possible with children and references specialization. When you specialize a child or reference, you narrow its target type. For example, if you have concept A which extends B, and have a reference r in concept C with target type B, you might narrow the type of reference r in C’s subconcepts. It works the same way for concept’s children.

17.2.5 Alias

The alias, referred to from code as conceptAlias, optionally specifies a string that will be recognized by MPS as a representation of the Concept. The alias will appear in completion boxes and MPS will instantiate the Concept, whenever the alias or a part of it is typed by the user.

17.3 Constrained Data Types

Constrained Data Type allows you to define string-based types constrained with a regular expression. MPS will then make sure all property values with this constrained data type hold values that match the constraint.

17.4 Enumeration Data Types

Enumeration Data Types allow you to use properties that hold values from pre-defined sets.
Each enumeration data type member has a value and a presentation. Optionally an identifier can be specified explicitly.

17.4.1 Presentation vs. Value vs. Identifier

- **Presentation** - this string value will be used to represent the enum members in the UI (completion menu, editor)
- **Value** - this value, the type of which is set by the member type property, will represent the enum members in code
- **Identifier** - this optional value will be used as the name of the generated Java enum. This value is typically derived from either the presentation or the value, since it is meant to be transparent to the language users and has no meaning in the language. It only needs to be specified when the id deriving process fails to generate unique valid identifiers.
- **Name** - when accessing enum data type’s members from code, name refers to either presentation, value or identifier, depending on which option member identifier is active

17.4.2 Deriving identifiers automatically

When deriving identifiers from either presentation or values, MPS will make best efforts to eliminate characters that are not allowed in Java identifiers. If the derived identifiers for multiple enum data type members end up being identical, an error will be reported. Explicit identifiers should be specified in such cases.

17.4.3 Programmatic access

To access enumeration data types and their members programmatically, use the enum operations defined in the jetbrains.mps.lang.smodel language.

```
list<enummember<MonthType>> members = enum/MonthType/.members;
enummember<MonthType> january = enum/MonthType/.<Jan>;
enummember<MonthType> february = enum/MonthType/.memberForValue(2);
enummember<MonthType> march = enum/MonthType/.memberForName("id_3");

// Gets the value as string, returns "4"
string april = enum member value/MonthType : Apr/;

// returns 1
int janValue = january.value;
// returns "id_2"
string febName = february.name;
// returns "id_3"
string marchId = march.javaIdentifier;
// returns "March"
string mayPresentation = march/.toString();
```

Note that the name in memberForName and february.name above means the actual member identifier, whether it is set to be custom, derive from presentation or derive from internal value.
Checking a value of a property against an enum data type value can be done with the is operation. To print out the presentation of the property value, you need to obtain the corresponding enum member first:

```java
if (node.month.is(Feb)) {
    string currentMonthName = enum/MonthType/monthForValue(node.month).toString();
}
```

### 17.5 Attributes

Attributes, sometimes called Annotations, allow language designers to express orthogonal language constructs and apply them to existing languages without the need to modify them. For example, the generator templates allow for special generator marks, such as LOOP, ->$, and $[$, to be embedded within the target language:

```java
public CalculatorImpl() {
    setTitle("$Calculator");
    setLayout(new GridLayout(0, 2));
    $LOOPS$
        ->$inputField$.getDocument().addDocumentListener(this.listener);
        add(new JLabel("$Title");
        add(->$inputField$);
    }
    $LOOPS$
        add(new JLabel("Output"));
        add(this->$outputField$);
    }
    update();
    setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
    pack();
    setVisible(true);
}
```

The target language (BaseLanguage in our example here) does not need to know anything about the MPS generator, yet the generator macros can be added to the abstract model (AST) and edited in the editor. Similarly, anti-quotations and Patterns may get attributed to BaseLanguage concepts.

MPS provides three types of attributes:

- **LinkAttribute** - to annotate references
- **NodeAttribute** - to annotate individual nodes
- **PropertyAttribute** - to annotate properties

By extending these you can introduce your own additions to existing languages. For a good example of attributes in use, check out the Description comments.
Chapter 18

SModel language

The purpose of SModel language is to query and modify MPS models. It allows you to investigate nodes, attributes, properties, links and many other essential qualities of your models. The language is needed to encode several different aspects of your languages - actions, refactorings, generator, to name the most prominent ones. You typically use the jetbrains.mps.lang.smodel language in combination with BaseLanguage.

18.1 Treatment of null values

SModel language treats null values in a very safe manner. It is pretty common, in OO-languages, such as Java or C#, to have a lot of checks for null values in the form of `expr == null` and `expr != null` statements scattered across the code. These are necessary to prevent null pointer exceptions. However, they at the same time increase code clutter and often make the code more difficult to read. In order to alleviate this problem, MPS treats null values in a liberal way. For example, if you ask a null node for a property, you will get back a null value. If you ask a null node for its children list, you will get empty list, etc. This should make your life as a language designer easier.

18.2 Types

SModel language has the following types:

- `node<ConceptType>` - corresponds to an AST node (e.g. `node<IfStatement> myIf = ...`)
- `nlist<ConceptType>` - corresponds to a list of AST nodes (e.g. `nlist<Statement> body = ...`)
- `node-ptr<ConceptType>` - represents a pointer to a node, can be resolved into a node when provided a repository
- `model` - corresponds to an instance of the MPS model
- `model-ptr<ConceptType>` - represents a pointer to a model, can be resolved into a model when provided a repository
- `search scope` - corresponds to a search scope of a node’s reference, i.e. the set of allowed targets for the reference
- `reference` - corresponds to an AST node that represents reference instance
- `concept<Concept>` - corresponds to the org.jetbrains.mps.openapi.language.SConcept concept that represents a concept (e.g. `concept<IfStatement> = concept/IfStatement/`)
- `conceptNode<Concept>` - (deprecated) corresponds to an AST node that represents a concept (e.g. `conceptNode<IfStatement> = conceptNode/IfStatement/`)
- `enummember<Enum Data Type>` - corresponds to an AST node that represents an enumeration member (e.g. `enummember<FocusPolicy> focus = ...`)

Most of the SModel language operations are applicable to all of these types.

18.3 Operation parameters

A lot of the operations in the SModel language accept parameters. The parameters can be specified once you open the parameter list by entering `<` at the end of an operation. E.g. `myNode.ancestors<concept = IfStatement, concept = ForStatement>`.
The "+" symbol as a parameter to these operations means "include myself". That is, when the node, on which the operation is being invoked, matches the condition, "+" ensures that the node will also be included in the returned result - e.g. `myNode.ancestors<concept = IfStatement, +>` may return `myNode`, if it is an `IfStatement`.

MPS allows you to down-cast from the concepts of the smodel concepts to the underlying Java API (Open API), may you need more power when manipulating the model. Check out the Open API - accessing models from code for details.

18.4 Comparison

The :eq: and :ne: operators can be used to compare nodes for equality. The operators are null-safe and will compare the whole sub-trees represented by the two compared nodes.

18.5 Queries

18.5.1 Getting nodes by name

Use the `node-ptr/.../` construct to obtain a reference to a node using its name.

```java
def object = node-ptr/Object/;
def equals = node-ptr/Object->equals/;
def equalsNode = equals.resolve(editorContext.getRepository());
```

To check that a node is a specific one, there is the "is" operation available.

```java
def equals.is(Object->equals);
```

To get a pointer to a node, use the `pointer` construct:

```java
def pointer = node.parent.pointer;
```

18.5.2 Getting concepts by name

Use the `concept/.../` construct to obtain a concept declaration by specifying its name:

```java
concept<CommandList> myConcept = concept/CommandList/;
```

The `concept switch` construct can be used to branch off the logic depending on the concept at hands:

```java
class concept switch (node.concept) {
  exactly IfStatement :
    message info "IfStatement discovered", <no project>, <no throwable>;
  subconcept of Expression :
    message info "An expression discovered", <no project>, <no throwable>;
  default : <no defaultBlock>
}
```

18.5.3 Features access

The SModel language can be used to access the following features:

- properties
- children
- references
To access them, the following syntax is used:

```java
<node expression>.featureName.
```

If the feature is a property, then the type of whole expression is the property’s type. If the feature is a reference or a child of 0..1 or 1 cardinality, then the type of this expression is `node<LinkTarget>`, where `LinkTarget` is the target concept in the reference or child declaration. If the feature is a child of 0..n cardinality, then the type of this expression is `nlist<LinkTarget>`.

You can use so-called implicit select to access features of the child nodes. For example, the following query:

```java
thisNode.children.grandChildren
```

will be automatically transformed by MPS to something like:

```java
thisNode.children.selectMany({~it => it.grandChildren; })
```

resulting in a plain collection of all non-null model elements accessible through the specified chain of link declarations.

### 18.5.4 Null checks

Since nulls are treated liberally in MPS, we need a way to check for null values. The `isNull` and `isNotNull` operations are our friends here.

### 18.5.5 IsInstanceOf check and type casts

Often, we need to check whether a node is an instance of a particular concept. We can’t use Java’s `instanceof` operator since it only understands java objects, not our MPS nodes. To perform this type of check, the following syntax should be used:

```java
<node expression>.isInstanceOf(Concept)
```

Also, there’s the `isExactly` operation, which checks whether a node’s concept is exactly the one specified by a user. Once we’ve checked a node’s type against a concept, we usually want to cast an expression to a concept instance and access some of this concept’s features. To do so, the following syntax should be used:

```java
<node expression> : Concept
```

Another way to cast node to particular concept instance is by using `as` cast expression:

```java
<node expression> as Concept
```

The difference between the regular cast (using colon) and the `as` cast is in a way it handles the situation when the result of the left-side expression cannot be safely cast to the specified Concept instance: A NullPointerException will be thrown by the regular cast in this case, while null will be returned by the `as` cast.

Combine this with the **null-safe dot operator** in the `smodel` language and you get a very convenient way to navigate around the model:

```
node.parent as BinaryOperation.leftExpression as BinaryOperation.leftExpression.isLValue()
```

Intention are available to easily migrate from one type of cast expression to the other:

### 18.5.6 Node collection cast

A collection of nodes can be filtered and cast by the concept of the nodes using the `ofConcept` construct:

```java
node.statements.ofConcept<LocalVariableDeclaration>
```

### 18.5.7 Parent

In order to find a node’s parent, the `parent` operation is available on every node.
18.5.8 Children

The \texttt{children} operation can be used to access all direct child nodes of the current node. This operation has an optional parameter \texttt{linkQualifier}. With this parameter result of \texttt{children<\texttt{linkQualifier}>} operation is equivalent to \texttt{node.\texttt{linkQualifier}} operation call and so will recall only the children belonging to the \texttt{linkQualifier} group/role. E.g. \texttt{classDef.children<\texttt{annotation}, \texttt{member}>}

18.5.9 Sibling queries

When you manipulate the AST, you will often want to access a node’s siblings (that is, nodes with the same role and parent as the node under consideration). For this task we have the following operations:

- \texttt{next-sibling/prev-sibling} - returns next/previous sibling of a node. If there is no such sibling, null is returned.
- \texttt{next-siblings/prev-siblings} - returns list of next/previous siblings of a node. These operations have an optional parameter that specifies whether to include the current node.
- \texttt{siblings} - returns list of all siblings of a node. These operations have an optional parameter that specifies whether to include the current node.

18.5.10 Ancestors

During model manipulation, it’s common to find all ancestors (parent, parent of a parent, parent of a parent of a parent, etc) of a specified node. For such cases we have two operations:

- \texttt{ancestor} - return a single ancestor of the node
- \texttt{ancestors} - returns all ancestors of the node
  
  Both of them have the following parameters to narrow down the list:
  
  - concept type constraint: \texttt{concept=Concept, concept in [ConceptList]}
  - a flag indicating whether to include the current node: +

E.g. \texttt{myNode.ancestors<\texttt{concept} = \texttt{InstanceMethodDeclaration}, +>}

18.5.11 Descendants

It’s also useful to find all descendants (direct children, children of children etc) of a specified node. We have the \texttt{descendants} operation for such purposes. It has the following parameters:

- concept type constraint: \texttt{concept=Concept, concept in [ConceptList]}
- a flag indicating whether to include current node: +

E.g. \texttt{myNode.descendants<\texttt{concept} = \texttt{InstanceMethodDeclaration}>}

18.5.12 Containing root and model

To access top-most ancestor node of a specified node you can make use of \texttt{containing root} operation. Containing model is available as a result of the \texttt{model} operation.

For example,

- \texttt{node<> containingRoot = myNode.containing root}
- \texttt{model owningModel = myNode.model}

18.5.13 Model queries

The \texttt{model-ptr/.../} expression retrieves a resolvable reference to a model. With a repository it can be resolved into the \texttt{model<>} type.

Often we want to find all nodes in a model which satisfy a particular condition. We have several operations that are applicable to expressions of the \texttt{model<>} type:

- \texttt{roots(Concept)} - returns all roots in a model, which are instances of the specified Concept
- \texttt{nodes(Concept)} - returns all nodes in a model, which are instances of the specified Concept

E.g. \texttt{model.roots(<\texttt{all}>) or model.nodes(IfStatement)}
18.5.14 Search scope queries

In some situations, we want to find out, which references can be set on a specified node. For such cases we have the search scope operation. It can be invoked with the following syntax:

\[
\text{<node expression>.search scope(link, operationContext)}
\]

18.5.15 The Concept literal

Often we want to have a reference to a specified concept. For this task we have the `concept` literal. It has the following syntax:

\[
\text{concept/ConceptName/}
\]

E.g. `concept<IfStatement> concept = concept/IfStatement/`

18.5.16 Concept operation

If you want to find the concept of a specified node, you can call the `concept` operation on the node.

E.g. `concept<IfStatement> concept = myNode.concept`

18.5.17 Migrating away from deprecated types

The `conceptNode<>` type as well as the `conceptNode` operation have been deprecated. The `asConcept` operation will convert a `conceptNode<>` to a `concept<>`. The `asNode` operation, on the other hand, will do the opposite conversion and will return a `node<AbstractConceptDeclaration>` for a `concept<>`.

The `conceptNode<>` type was called `concept<>` in MPS 3.1. The `conceptNode` operation was called `concept` in MPS 3.1.

18.5.18 Concept hierarchy queries

We can query super/sub-concepts of expression with the concept type. The following operations are at your disposal:

- **super-concepts/all** - returns all super-concepts of the specified concept. There is an option to include/exclude the current concept - `super-concepts/all<+>`
- **super-concepts/direct** - returns all direct super-concepts of the specified concept. Again, there is an option to include/exclude the current concept - `super-concepts/direct<+>`
- **sub-concepts** - returns sub-concepts

For example:

```java
concept<IfStatement> concept = myNode.concept;
list<concept<> superConceptsAll = concept.super-concepts/all;
concept.super-concepts/direct<+>;
concept.sub-concepts(model);
```

18.5.19 The hasRole operation

Sometimes we may want to check whether a node has a particular role. For this we have the following syntax:

\[
\text{<node expression>.hasRole(Concept : child)}
\]

For example,

```java
myNode.hasRole(IfStatement : elsifClauses)
```
18.5.20 Link queries

The `link`, `linkName` and `linkNode` operations give you access to the details of a link between nodes.

```java
node<LinkDeclaration> decl = linkNode/ClassCreator : constructorDeclaration/;
decl.sourceCardinality;
decl.metaClass;
decl.role;
decl.target;
decl.unordered;
...
string name = linkName/ClassCreator : constructorDeclaration/;

SReferenceLink link = link/ClassCreator : constructorDeclaration/;
link.isOptional();
link.getDeclarationNode();
link.getOwner();
link.getName();
link.getTargetConcept();
link.getScope(decl/);
...
```

18.5.21 Containing link queries

If one node was added to another one (parent) using the following expression:

```java
parent.childLinkRole.add(node)
```

then you can call the following operations to access the containment relationship information:

- **containingRole** - returns a string representing the child role of the parent node containing this node ("childLinkRole" in above case)
- **containingLink** - returns node<LinkDeclaration> representing a link declaration of the parent node containing this node
- **index** - returns int value representing index of this node in a list of children with corresponding role. Identical to the following query upon the model represented above:

```java
parent.childLinkRole.indexOf(node)
```

18.5.22 Reference operations

Accessing references

Following operations were created to access reference instance representing a reference from source node to target one. Operations are applicable on source node:

- **reference< >** - returns an instance of reference type representing specified reference. This operation requires "linkQualifier" parameter used as reference specification. Parameter can be either link declaration of source node's concept or expression returning node<LinkDeclaration> as a result
- **references** - returns sequence<reference> representing all references specified in source node.

Working with

Having an instance of reference type you can call the following operations on it:

- **linkDeclaration** - returns node<LinkDeclaration> representing this reference
- **resolveInfo** - returns string resolve info object
- **role** - returns reference role - similar to reference.linkDeclaration.role;
- **target** - returns node<> representing reference target is it was specified and located in model(s)
18.5.23  
18.5.24  Downcast to lower semantic level

SModel language generates code that works with raw MPS classes. These classes are quite low-level for the usual work, but in some exceptional cases we may still need to access them. To access the low-level objects, you should use the downcast to lower semantic level construct. It has the following syntax:

```
<node expression>/
```

For example,

```
myNode/.getConcept().findProperty("name")
```

18.6  Modification operations

18.6.1  Feature changes

The most commonly used change operation in SModel is the act of changing a feature. In order to set a value of a property, or assign a child or reference node of 0..1 or 1 cardinality, you can use straight assignment (with =) or the set operation. In order to add a child to 0..n or 1..n children collection, you can either use the.add operation from the collection language or call `add next-sibling`/`add prev-sibling` operations on a `node<>` passing another node as a parameter.

For example,

- `classDef.name = "NewClassName";`
- `classDef.name.set("NewClassName");`
- `myNode.condition = trueConstant;`
- `node<InstanceMethodDeclaration> method = classDef.member.add new initialized(InstanceMethodDeclaration);`

When setting a target to a reference link, there is no need to access a target node. Having a pointer to the target is enough. This is possible with the set ptr operation, which is applicable to reference link access expressions.

```
node.instanceMethodDeclaration.set ptr(Object->equals)
```

Similarly, to check that a node is a specific one, there is the "is" operation available.

```
node.parent.is(Object->equals);
```

18.6.2  New node creation

There are several ways to create a new node:

- new operation: `new node<Concept>()`
- new instance operation on a model: `model.newInstance()`
- new instance operation on a concept: `concept.newInstance()`
- add new(Concept) and set new(Concept) operations applied to feature expressions
- replace with new(Concept) operation
- new root node(Concept) operation applied to a model. In this case the concept should be rootable
- new next-sibling<Concept>}/new prev-sibling<Concept> operations adding new sibling to an existing node
Note that the `jetbrains.mps.lang.actions` language adds the possibility to initialize the newly created nodes using the rules specified in **NodeFactories**. Upon importing the `jetbrains.mps.lang.actions` language you are able to call:

- `new initialized node<Concept>()`
- `model.new initialized node(Concept)`
- `node.new initialized next/previous sibling(Concept)`
- `add new initialized(Concept)`
- `set new initialized(Concept)`
- `replace with new initialized(Concept)`
- `replace with initialized next/previous-sibling(Concept)`

### 18.6.3 Copy

To preserve the tree shape of the model, a node can only have at most one parent. As soon as you add a node as a child to a parent node, the node is detached automatically from any previous parent node that it may have had. Creating copies of nodes, including their sub-trees, may thus come handy. To create a copy of an existing node, you can use the copy operation. E.g.,

```
node<> yourNode = myNode.copy
```

### 18.6.4 Replace with

To replace a node in the AST with an instance of another node, you can use the ‘replace with’ operation. If you want to replace and create at the same time, there is a shortcut operation ‘replace with new(Concept)’, which takes a concept as a parameter.

### 18.6.5 Delete and detach operations

If you want to delete a node from the model, you can use the **detach** operation. You can still add the detached node to the model later.

### 18.7 smodel.query language

The `jetbrains.mps.lang.smodel.query` language enables the same type of queries that the **MPS Console** uses:

```java
with this.model do {
    sequence<IfStatement> ifS = instances<IfStatement>
    where (it -> it.condition.isInstance<BoolConstant>());
    collection<Reference> myUsages = usages(this);
}
```

This language is allowed to be used inside **with**-statement, which constraints the scope, on which queries are performed.

```java
with myProject do {
    #instances<SimpleLanguageAspectDescriptor>.forEach(it ->
        // do something
    );
}
```

The scope can be constrained to a project, module, model or a sequence of these.

### 18.7.1 Operation parameters

The behavior of smodel.query operations can be slightly changed using operation parameters, which can be specified after the operation name.

```java
with myProject do {
    #instances<SimpleLanguageAspectDescriptor>.forEach(it ->
        // do something
    );
}
```

Possible parameters include:
**scope** - Each command operates in the scope specified in the surrounding `with` statement. The `scope` parameter changes the operating scope for a single command.

**exact** - can be used in `#instances` operations to find instances of the concept specified, excluding instances of descendant concepts

### 18.7.2 Scope specification

Scope on which queries are performed can be specified explicitly for single query using `scope` parameter. This option overrides scope derived from `with` statement.

The scope parameter can consist of sequence of models or modules, or can be an arbitrary expression of type `SearchScope`.

```
with myProject do {
  sequence<model> structureModels = #models.ofAspect<structure>;
  collection<node<ConceptDeclaration>> concepts = #instances<scope = models: [structureModels]> (ConceptDeclaration);
}
```

When the `scope` parameter of a query is not specified, the query is performed on the scope constructed by the enclosing `with` statement. To avoid confusion, nested `with` statements are not supported. `With` statements can construct a scope from a model, a module, a sequence of those, from a project or from an existing `SearchScope` object.

Scope constructed by the `with` statement include only editable models so that modifying operations can be safely performed within that scope.

```
with myProject do {
  #nodes.where({it -> it.concept.isInvalid(); }).forEach({it ->
    it.$invalidConcept.setNew(<default>);
  });
}
```

When scope is set explicitly as query parameter, query uses it as is and does not throw away read only models, so the two following queries show different behavior:

```
with module/jetbrains.mps.samples.Baja/ do {
  // without read-only models
  @nodes:

  // including read-only models
  @nodesScope = modules: [module/jetbrains.mps.samples.Baja/];
}
```

### Using smodel queries in console plugin

Queries can be used in **console** without the wrapping `with` statement. Default scope there for queries that do not specify it explicitly contains all editable models in the current project. Also, some additional options are available when specifying the scope explicitly.

### 18.7.3 Queries defined in the smodel.query language

- `#instances` - fast search for instances of a specified concept
- `#usages` - fast search for usages of a specified node
- `#modules` - all modules in scope
- `#models` - all models in scope
- `#nodes` - all nodes in scope
- `#references` - all references in scope
Chapter 19

Open API - accessing models from code

The language repository, project modules, languages and models can be conveniently accessed programmatically through Open API. Open API gives you controlled access to the model and also allows you to provide your own implementations for some aspects, such as persistence. These two usage types will be discussed individually.

*Note: This document is meant to provide a general high-level overview of the Open API philosophy and give you useful starting links to the API. For technical details on how to use it please consult the API on GitHub.*

For hands-on code examples of using the API please check out [Using model & module dependencies FAQ](#).

19.1 Using Open API

The API is located under the `org.jetbrains.mps.openapi` package and is divided into several sub-packages:

- **event** - contains event classes for changes to the model
- **language** - provides a set of interfaces to gain access to compiled languages and inspect their structure
- **model** - gives you ways to inspect and modify the models (ASTs)
- **module** - abstracts repositories and modules as means to logically organize models
- **persistence** - holds the interfaces necessary to extend and customize the persistence mechanism for models
- **repository** - holds listeners to repository-specific events
- **util** - contains utility classes, such as `ProgressMonitor` to hook long-lasting actions to UI progress indicators

The API recognizes these logical elements, with the ones above containing the ones below in the list:

- Repository
- Module
- Model
- Node
- Properties, References

Open API also recognizes meta-structure, which is orthogonal to the elements above. The meta-structure consists of the following key elements:

- Language
- Concept, Enumeration
- Members (such as properties, links and enum literals)

Each node has an associated concept. A concept belongs to a Language. Languages may keep a pointer to the source module that they originated from to give the language user a way to investigate the language in detail. The API enables you to browse the whole repository and investigate its modules, their models as well as the nodes that these models are built from. Additionally the API has capabilities to search for element’s usages or find any element by its name irrespective of its location within the repository. You can also modify all of these elements, save your changes or reload them from a persistent storage. The API will detect colliding modifications to the model in memory and its persistent storage.
19.2 Some API details

19.2.1 Meta-model level

SAbstractConcept
- a common super-class to both SConcept and SConceptInterface
- gives access to concept’s properties, containment links and reference links
- use the getProperties(), getContainmentLinks() and getReferenceLinks() methods to obtain concept’s properties, containment and reference links, respectively

SProperty
- represents a property

SContainmentLink
- represents a containment (parent-child) relationship between concepts

SReferenceLink
- represents an explicit (reference) relationship between concepts

19.2.2 Model level

SNode
- represents individual nodes in the model

SReference
- represents references between nodes

SNodeReference
- a unique global reference to a node that can be persisted and used repeatedly to obtain a node from a repository

19.2.3 Open API usage patterns

Setting a property

```java
Iterable<SProperty> properties = concept/Shape/.getProperties();
list<SProperty> props = new arraylist<SProperty>(copy: properties);
currentNode/.setProperty(props.findFirst({~it => it.getName() : eq: "foo"; }), "value");
```

Adding a child

```java
Iterable<SContainmentLink> containmentLinks = concept/Shape/.getContainmentLinks();
list<SContainmentLink> containments = new arraylist<SContainmentLink>(copy: containmentLinks);
currentNode/.addChild(containmentLinks.findFirst(...), childNode);
```

19.3 Using commands

Open API provides means to alter the models, as well. Modifications need to be performed as commands that are passed to the repository for processing. Typical model-changing/editing actions can be un-done/re-done, while actions performing major changes to the module structure cannot. There are three types of changes:

1. models and nodes can be changed through **undoable actions**
2. modules, their properties and dependencies can be performed through **repository commands**
3. radical changes to the project, such as a VCS update or a complete project reload, need an external update action to be performed - no node-level notifications are fired in such cases, only model replaced or module changed notifications are triggered.

The commands depending on their type will have all the necessary read or write permission assigned automatically before they start changing the model. Change notifications are fired to the registered listeners on the node, model, module or repository levels.

19.4 Concurrent access

Open API is designed for concurrent access and will correctly handle multiple threads accessing the models through Open API simultaneously, provided the supplied synchronization mechanisms are correctly utilized by the calling code. Open API will reject all improperly synchronized requests and thus preserve the integrity of the models.

In a more concrete terms, you need to obtain a read or write action before you start performing your operations, otherwise you’d get exceptions fired from the code.

Use $SRepository.getRepositoryAccess().applyChanges()$ to have your changes applied to the whole repository $SRepository.getModelAccess().runXXXAction()$ to run a read/write action and $SRepository.getModelAccess().executeCommand()$ to run a command. Commands get all write permissions automatically and so always gain exclusive access to the repository. Both methods offer an asynchronous variant that runs the supplied action asynchronously in the EDT.

19.4.1 Model-locking example

$SRepository$ is the right place to start locking the model. You can obtain an $SRepository$ reference from context objects, such as $EditorContext$. Your code can than obtain $ModelAccess$ and get the lock:

```java
(node, editorContext)->JComponent {
    //get ModelAccess from the context
    final ModelAccess modelAccess = editorContext.getRepository().getModelAccess();

    //read the model
    final boolean[] active = new boolean[]{false};
    modelAccess.runReadAction(new Runnable() {
        public void run()
        {
            active[0] = node.showActive;
        }
    });

    final JButton button = new JButton(active[0] ? "Show all" : "Show active");
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent p0)
        {
            //update the model in an action, that can be undone
            modelAccess.executeCommand(new Runnable() {
                public void run()
                {
                    node.showActive = !node.showActive;
                }
            });
            return button;
        }
    });

    return button;
}
```

19.5 Custom persistence

By default MPS stores models as flat files in an XML or binary format. To allow you to customize the way your models are persisted, Open API provides gives you several options.

19.5.1 Alternative file format

Changing the format of model data stored in a flat file is the simplest way to customize model persistence. You simply register your own $ModelFactory$ with the file extension of your choice (through $PersistenceFacade$) and MPS will use that
factory to instantiate your custom SModel implementations whenever that file extension is discovered. You’ll also need to provide and register your own implementations of SModelIdFactory and SNodeIdFactory.

### 19.5.2 Alternative storage

If you’re more adventurous and want, for example, to load your models from a database or other non-file storage, you need to additionally provide a ModelRootFactory, which can create custom ModelRoot instances. These model roots will then handle all the specifics of your chosen storage in order to load/save models. You may typically also need to bundle UI that would allow the users to configure data source details, such as database location, user name or other.

The **xmlPersistence** sample project that comes bundled with MPS shows a non-trivial example of implementing custom persistence. Check out the description of the Custom Persistence Cookbook for practical details on how to provide your own persistence to MPS models in MPS as well as IDEA.

### 19.5.3 Custom Find Usage and Navigation participants

Providing a custom implementation of FindUsagesParticipant will allow you to optimize FindUsages in models using your custom persistence. Similarly, custom implementations of NavigationParticipant will have a chance to optimize Go To Root/Class/Symbol actions. Instead of letting the default find usages and navigation implementations load all models into memory and process them in a standard way, by providing custom participants to PersistenceFacade you have the option to access the persistent storage directly and thus speed-up search as well as navigation.
Chapter 20

Quotations

20.1 Quotations

A quotation is a language construct that lets you easily create a node with a required structure. Of course, you can create a node using the smodelLanguage and then populate it with appropriate children, properties and references by hand, using the same smodelLanguage. However, there’s a simpler - and more visual - way to accomplish this.

The two following constructs will build identical nodes, the first one uses quotation, the second plan model API:

```java
node<IntegerType> node = <int>;
node<IntegerType> node2 = new node<IntegerType>();
```

When creating a quotation, type the left angle bracket symbol ‘<’ and pick ‘quotation’ from the completion menu.

Alternatively, typing ‘<q’ followed by Control + Space will be even faster.

When specifying the top-most concept for the quotation, use the concept name, e.g. `IntegerType`, not its alias, e.g. `int`.

Quotations as well as light quotations are defined int the jetbrains.mps.lang.quotation language. You need this language to be set as a used language in order to be able to use it in your models.

A quotation is an expression, whose value is the MPS node written inside the quotation. Think about a quotation as a "node literal", a construction similar to numeric constants and string literals. That is, you write a literal if you statically know what value do you mean. So inside a quotation you don’t write an expression, which evaluates to a node, you rather write the node itself. For instance, an expression 2 + 3 evaluates to 5, an expression < 2 + 3 > (angled braces being quotation braces) evaluates to a node PlusExpression with leftOperand being an IntegerConstant 3 and rightOperand being IntegerConstant 5.

The following two constructs again create the same AST, now the quotation approach yields clear benefits in code brevity:
20.2 Antiquotations

For it is a literal, a value of quotation should be known statically. On the other hand, in cases when you know some parts (i.e. children, referents or properties) of your node only dynamically, i.e. those parts that can only be evaluated at runtime and are not known at design time, then you can’t use just a quotation to create a node with such parts.

The good news, however, is that if you know the most part of a node statically and you want to replace only several parts by dynamically- evaluated nodes you can use antiquotations. An antiquotation can be of 4 types: child, reference, property and list antiquotation. They all contain an expression, which evaluates dynamically to replace a part of the quoted node by its result. Child and referent antiquotations evaluate to a node, property antiquotation evaluates to string and list antiquotation evaluates to a list of nodes.

For instance, you want to create a ClassifierType with the class ArrayList, but its type parameter is known only dynamically, for instance by calling a method, say, "computeMyTypeParameter()".

Thus, you write the following expression: `< ArrayList < %( computeMyTypeParameter() )% > >`. The construction `%(...)%` here is a node antiquotation.

You may also antiquotate reference targets and property values, with `ˆ(...)ˆ` and `$(...)$`, respectively; or a list of children of one role, using `*(...)*`.

a) If you want to replace a node somewhere inside a quoted node with a node evaluated by an expression, you use node antiquotation, that is `%( )%`. As you may guess there’s no sense to replace the whole quoted node with an antiquotation with an expression inside, because in such cases you could instead write such an expression directly in your program. So node antiquotations are used to replace children, grandchildren, great-grandchildren and other descendants of a quoted node. Thus, an expression inside of antiquotation should return a node. To write such an antiquotation, position your caret on a cell for a child and type `%`

b) If you want to replace a target of a reference from somewhere inside a quoted node with a node evaluated by an expression, you use reference antiquotation, that is `ˆ(...)ˆ`. To write such an antiquotation, position your caret on a cell for a referent and type `ˆ`

c) If you want to replace a child (or some more deeply located descendant), which is of a multiple-cardinality role, and if for that reason you may want to replace it not with a single node but rather with several ones, then use child list (simply list for brevity) antiquotations, `*( )*`. An expression inside a list antiquotation should return a list of nodes, that is of type `**nlist<..>*` or compatible type (i.e. `{list<node<..>>} is ok, too, as well as some others). To write such an antiquotation, position your caret on a cell for a child inside a child collection and type `***`. You cannot use it on an empty child collection, so before you press `***` you have to enter a single child inside it.

d) If you want to replace a property value of a quoted node by a dynamically calculated value, use property antiquotation `$( )$`. An expression inside a quotation should return string, which will be a value for an antiquoted property of a quoted node. To write such an antiquotation, position your caret on a cell for a property and type `$$`

20.3 Light quotations (quotation builders)

Using quotations has its downsides, though. For example, if you’re creating quotations of a language in its own definition, you’re creating a bootstrapping cycle that needs special treatment during language compilation (See Removing bootstrapping dependency problems). Light quotations provide an alternative approach that will help you eliminate such issues.
Unlike direct model API usage, *light quotations* remain conveniently usable for deeper hierarchies:

```plaintext
node<PlusExpression> node3 = <2 + 3>;
node<PlusExpression> node4 = <PlusExpression(
  leftExpression: IntegerConstant(value: 2),
  rightExpression: IntegerConstant(value: 3))>;
```
Chapter 21

Pattern

21.1 The Pattern language

The pattern language has a single purpose - to define patterns of model structures. Those patterns form visual representations of nodes you want to match. A pattern matches a node if the node’s property values are equal to those specified in the pattern, node’s references point to the same targets that the ones of the pattern do and the corresponding children match the appropriate children of the pattern. Also patterns may contain variables for nodes, references and properties, which then match any node/reference/property. On top of that the variables will hold the actual values upon a successful match.

21.2 PatternExpression

The single most important concept of the pattern language is PatternExpression. It contains a pattern as its single arbitrary node. Also, the node can specify the following variables:

- #name - a node variable, a placeholder for a node. Stores the matching node
- #name - a reference variable, a placeholder for a reference. Stores the reference’s target, i.e. a node.
- $name - a property variable, a placeholder for a property value. Stores the property value, i.e. a string.
- *name - a list variable, a placeholder for nodes in the same role. Stores the list of nodes.

Antiquotations may be in particular useful when used inside a pattern, just like inside quotations (see Typesystem).

21.3 Examples

1. The following pattern matches against any InstanceMethodDeclaration without parameters and a return type:

```plaintext
public void $methodName() {
    # statementList
}
```

Captured variables:

| $methodName | string | method’s name |
| #statementList | node<StatementList> | statements |

2. The following pattern matches against a ClassifierType with the actual classifier specified inside an antiquotation expression and with any quantity of any type parameters:

```plaintext
^\( classifierType.classifier )^<* 1 [# ignored]> ^
```

Captured variables:

| *l | nlist<Type> | class type’s parameters |
| #ignored | node<Type> | used as wildcard, its contents is ignored. Means that parameters are arbitrary |
21.4 Using patterns

21.4.1 Match statement

Patterns are typically used as conditions in match statements. Pattern variables can be referenced from inside of the match statement.

For example:

```java
public void bar(n) {
    match n with
    | int $name : as var1 -> {
        System.out.println(var1.type + " " + $name);
    }
    | Integer i : as var2 -> {
        System.err.println("error: " + var2);
    }
    default -> {
        <no ifFalseStatement>
    }
}
```

this piece of code examines a node n and checks whether it satisfies the first or the second condition. Then the statement in the corresponding (matching) block is executed. A pattern variable $name is used in a first block to print out the name of a node. In our case the node holds a variable declaration.

21.4.2 Other usages

Patterns are also used in several other language constructs in MPS. They may appear:

- as conditions on applicable nodes of typesystem/replacement/subtyping/other rules of typesystem language (See Typesystem)
- as supertype patterns in coerce statement and coerce expression (See Using_typesystem)
- as conditions on node in generator rules
- as pattern in TransformStatement used to define language migrations (See Migrations)

You can also use patterns in your own languages. Basically what happens is that a class is generated from a PatternExpression and the expression itself is reduced to a constructor of this class. This class extends GeneratedMatchingPattern and has a boolean method `match(SNode)`, which returns a boolean value indicating whether the node matches the pattern. It also holds a method `getFieldValue(String)` to get the values stored in pattern variables after a successful match.

So to develop your own language constructs using patterns, you can call these two methods in the generator template for your constructs.
Chapter 22

Using model & module dependencies FAQ

// Entry point
// accessing model of the node
model model = node.model;
// "down-"casting it to the interface from MPS open API
SModel sModel = model;
// non-open API interface for MPS model
SModelInternal internalSModel = (SModelInternal) sModel;
// accessing module owning the model
SModule sModule = sModel.getModule();
// Repository or modules
// accessing repository of modules
SRepository sRepository = sModel.getRepository();
// accessing modules from the repository
Iterable<SModule> modules = sRepository.getModules();
SModule module = sRepository.getModule(sModule.getModuleId());
SModule closuresRuntimeModule = sRepository.getModule(module reference/closures.runtime/.getModuleId());
// Languages are represented as a special module type
if (module instanceof Language) {
    Language language = (Language) module;
}
// Getting language by "direct" reference:
// Note: no repository is mentioned here. In the MPS we actually have single repository now,
// so you can use this expression to get module from the single repository
Language mpsBaseLanguage = (Language) module/jetbrains.mps.baseLanguage/

// Module dependencies
// accessing module dependencies
Sequence<SDependency> dependencies = module.getDeclaredDependencies();
// adding used language to a module
AbstractModule internalModule = (AbstractModule) sModule;
internalModule.getModuleDescriptor().getUsedLanguages().add(module reference/jetbrains.mps.baseLanguage/);
// modify module dependencies
Sequence<Dependency> internalDependencies = internalModule.getModuleDescriptor().getDependencies();
internalDependencies = internalDependencies.select({it => it.getModuleRef();});
internalModule.getModuleDescriptor().getDependencies().add(new Dependency(module reference/closures.runtime/SDependencyScope.DEFAULT, false));
// working with DevKits:
internalModule.getModuleDescriptor().getUsedDevkits().add(module reference/jetbrains.mps.devkit.general-purpose);

// Models
// accessing models from the module
Iterable<SModel> models = sModule.getModels();
// working with imported models:
internalSModel.importedModels();
internalSModel.addModelImport(sModel.getReference(), false);
internalSModel.deleteModelImport(sModel.getReference());
// working with imported languages:
internalSModel.importedLanguages();
internalSModel.addLanguage(mpsBaseLanguage.getModuleReference());
internalSModel.addLanguage(module reference/jetbrains.mps.baseLanguage/);
// working with imported DevKits
internalSModel.importedDevkits();
internalSModel.addDevKit(module reference/jetbrains.mps.devkit.general-purpose/);
internalSModel.deleteDevKit(module reference/jetbrains.mps.devkit.general-purpose/);
Chapter 23

Constraints

The *Structure Language* may sometimes be insufficient to express advanced constraints on the language structure. The *Constraints* aspect gives you a way to define such additional constraints.

### 23.1 Can be child/parent/ancestor/root

These are the first knobs to turn when defining constraints for a concept. They determine whether instances of this concept can be hooked as children (parents, ancestors) nodes of other nodes or root nodes in models. You specify them as *boolean-returning* closures, which MPS invokes each time when evaluating allowed position for a node in the AST.

#### Languages to import

You will most likely need at least two languages imported in the *constraints* aspect in order to be able to define constraints - the *j.m.baselanguage* and *j.m.lang.smodel* languages.

**can be child**

Return false if an instance of the concept is not allowed to be a child of specific nodes.

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>operationContext</td>
<td>IOperationContext</td>
</tr>
<tr>
<td>node</td>
<td>the child node we are checking (instance of this concept)</td>
</tr>
<tr>
<td>parentNode</td>
<td>the parent node we are checking</td>
</tr>
<tr>
<td>childConcept</td>
<td>concept of the child node (can be a subconcept of this concept)</td>
</tr>
<tr>
<td>link</td>
<td>LinkDeclaration of the child node (child role can be taken from there)</td>
</tr>
</tbody>
</table>

**can be parent**

Return false if an instance of concept is not allowed to be a parent of specific concept node (in a given role).

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
</table>

```java
concepts constraints Constant {
  can be child
  (childConcept, node, link, parentNode, operationContext) -> boolean {
    parentNode.isEmptyInstanceOf(Constraints) && !parentNode : Constants.constants.contains(node));
  }

can be parent <none>

can be ancestor <none>

property {name}
  get:<default>
  set:<default>
  isValid:(propertyValue, node) -> boolean {
    propertyValue.length() > 5;
  }
}
```
CHAPTER 23. CONSTRANTS

<table>
<thead>
<tr>
<th>operationContext</th>
<th>IOperationContext</th>
</tr>
</thead>
<tbody>
<tr>
<td>childNode</td>
<td>the child node we are checking</td>
</tr>
<tr>
<td>node</td>
<td>the parent node we are checking (instance of this concept)</td>
</tr>
<tr>
<td>childConcept</td>
<td>the concept of the child node we are checking</td>
</tr>
<tr>
<td>link</td>
<td>LinkDeclaration of the child node</td>
</tr>
</tbody>
</table>

**can be ancestor**

Return false if an instance of the concept is not allowed to be an ancestor of specific nodes.

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>operationContext</td>
<td>IOperationContext</td>
</tr>
<tr>
<td>childNode</td>
<td>the child node we are checking</td>
</tr>
<tr>
<td>node</td>
<td>the ancestor node we are checking (instance of this concept)</td>
</tr>
<tr>
<td>childConcept</td>
<td>the concept of the descendant node</td>
</tr>
</tbody>
</table>

**can be root**

This constraint is available only for rootable concepts (*instance can be root* is true in the concept structure description). Return false if instance of concept cannot be a root in the given model.

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>operationContext</td>
<td>IOperationContext</td>
</tr>
<tr>
<td>model</td>
<td>model of the root</td>
</tr>
</tbody>
</table>

### 23.2 Property constraints

Technically speaking, "pure" concept properties are not properties in its original meaning, but only public fields. Property constraints allow you to make them real properties. Using these constraints, the behavior of concept’s properties can be customized. Each property constraint is applied to a single specified property.

**property** - the property to which this constraint is applied.

**get** - this method is executed to get property value every time property is accessed.

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>node to get property from</td>
</tr>
</tbody>
</table>

**set** - this method is executed to set property value on every write. The property value is guaranteed to be valid.

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>node to set property</td>
</tr>
<tr>
<td>propertyValue</td>
<td>new property value</td>
</tr>
</tbody>
</table>

**is valid** - this method should determine whether the value of the property is valid. This method is executed every time before changing the value, and if it returns false, the set() method is not executed.

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>node to check property</td>
</tr>
<tr>
<td>propertyValue</td>
<td>value to be checked</td>
</tr>
</tbody>
</table>

#### 23.2.1 Example - customizing the description in the completion menu

The completion menu lists available nodes together with some additional descriptive information:
In order to customize the additional information and provide more details on the individual options listed in the completion menu, you can override the getter of the `shortDescription` property of the target concept:

23.3 Referent constraints

Constraints of this type help to add behavior to concept’s links and make them look more properties-like.

**Referent set handler** - if specified, this method is executed on every set of this link.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>referenceNode</td>
<td>node that contains link.</td>
</tr>
<tr>
<td>oldReferentNode</td>
<td>old value of the reference.</td>
</tr>
<tr>
<td>newReferentNode</td>
<td>new value of the reference.</td>
</tr>
</tbody>
</table>

**Scope** - defines the set of nodes to which this link can point. The method returns a `Scope` instance. Please refer to the Scopes for more information on scoping. There are two types of scope referent constraint:

- **inherited**
- **reference**

While **inherited** scope simply declares the target concept, the **reference** scope provides a function that calculates the scope on the fly from the parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>referenceNode</td>
<td>the node that contains the actual link. It can be null when a new node is being created for a concept with smart reference. In this situation smart reference is used to determine what type of node to create in the context of enclosingNode, so the search scope method is called with a null referenceNode.</td>
</tr>
<tr>
<td>contextNode</td>
<td>node with the reference or the closest not-null context node</td>
</tr>
</tbody>
</table>
If scope is not set for the reference then default scope from the referenced concept is used. If the default scope is also not set then "global" scope is used: all instances of referent concept from all imported models.

**presentation** (deprecated - the editor aspect now specifies presentation of references, see Editor) - here you specify how the reference will look like in the editor and in the completion list. Sometimes it is convenient to show reference differently depending on context. For example, in Java all references to an instance field \( f \) should be shown as \( \text{this}.f \), if the field is being shadowed by the local variable declaration with the same name. By default, if no presentation is set, the name of the reference node will be used as its presentation (provided it is an INamedConcept).

---

<table>
<thead>
<tr>
<th>parameter</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>containedLink</td>
<td>SContainmentLink describing parent-child relationship between the contextNode and its non-existent child that is being created (if referenceNode exists then this parameter has no meaning)</td>
</tr>
<tr>
<td>linkTarget</td>
<td>the concept that this link can refer to. Usually it is a concept of the reference, so it is known statically. If we specialize reference in subconcept and do not define search scope for specialized reference, then linkTarget parameter can be used to determine what reference specialization is required.</td>
</tr>
<tr>
<td>position</td>
<td>the target index in contextRole</td>
</tr>
</tbody>
</table>

---

### 23.4 Default scope

Suppose we have a link pointing to an instance of concept C and we have no Constraints defined for this link in referent constraints. When you edit this link, all instances of concept C from all imported models are visible by default. If you want to restrict set of visible instances for all links to concept C you can set default scope for the concept. As in referent constraint you can set search scope, validator and presentation methods. All the parameters are the same. Please refer to the Scopes for more information on scoping.
Chapter 24

Scopes

We are going to look at two ways to define scopes for custom language elements - the inherited (hierarchical) and the referential approaches. 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.

24.1 Two ways

All references need to know the set of allowed targets. This enables MPS to populate the completion menu whenever the user is about to supply a value for the reference. Existing references can be validated against that set and marked as invalid, if they refer to elements out of the scope.

MPS offers two ways to define scopes:

- Inherited scopes
- Reference scopes

Reference scope offers lower ceremony, while Inherited scopes allow the scope to be built gradually following the hierarchy of nodes in the model.

The oldest type of scopes in MPS is called Search scope and it has been deprecated in favor of the two types mentioned above, because the scoping API has changed significantly since its introduction. The Reference scope can be viewed as the closest replacement for Search scope compatible with the new API.

24.2 Inherited scopes

We will describe the new hierarchical (inherited) mechanism of scope resolution first. This mechanism delegates scope resolution to the ancestors, who implement ScopeProvider.

1. MPS starts looking for the closest ancestor to the reference node that implements ScopeProvider and who can provide scope for the current kind.

2. If the ScopeProvider returns null, we continue searching for more distant ancestors.

3. Each ScopeProvider can
   - build and return a Scope implementation (more on these later)
   - delegate to the parent scope
   - add its own elements to the parent scope
   - hide elements from parent scope (more on how to work with scopes will be discussed later)

Our InputFieldReference thus searches for InputField nodes and relies on its ancestors to build a list of those.
Once we have specified that the scope for InputFieldReference when searching for an InputField is inherited, we must indicate that Calculator is a ScopeProvider. This ensures that Calculator will have say in building the scope for all InputFieldReferences that are placed as its descendants.
The Calculator in our case should return a list of all its InputFields whenever queried for scope of InputField. So in the Behavior aspect of Calculator we override (Control + O) the getScope() method:
If Scope remains unresolved, we need to import the model (Control + R) that contains it (jetbrains.mps.scope):
The `getScope()` method takes two parameters:

- **kind** - the concept of the possible targets for the reference
- **child** - the child node of the current (this) `ScopeProvider`, from which the request came, so the actual reference is among descendants of the `child` node

We also need `BaseLanguage` since we need to encode some functionality. The `jetbrains.mps.lang.smodel` language needs to be imported in order to query nodes. These languages should have been imported for you automatically. If not, you can import them using the `Control + L` shortcut.
Now we can complete the scope definition code, which, in essence, returns all input fields from within the calculator:

```java
concept behavior Calculator {
    constructor {
        <no statements>
    }

    public Scope getScope(conceptNode<> kind, node<> child) {
        if (kind.isSubConceptOf(InputField) && come from outputField) {
            return SimpleRoleScope.forNamedElements(this, link/Calculator : inputField/); }
        return parent scope;
    }
}
```

A quick tip: Notice the use of SimpleRoleScope class. It is one of several helper classes that can help you build your own custom scopes. Check them out by Navigating to SimpleRoleScope (Control + N) and opening up the containing package structure (Alt + F1).

### 24.2.1 Scope helper implementations

MPS comes with several helper `Scope` implementations that cover many possible scenarios and you can use them to ease the task of defining a scope:

- **ListScope** - represents the nodes passed into its constructor
- **DelegatingScope** - delegates to a Scope instance passed into its constructor, typically to be extended by scopes that need to add functionality around an existing scope, e.g. LazyScope
- **CompositeScope** - delegates to a group of (wrapped) Scope instances
- **FilteringScope** - delegates to a single Scope instance, filtering its nodes with a predicate (the `isExcluded` method)
- **FilteringByNameScope** - delegates to a single Scope instance, filtering its nodes by a name blacklist, which it gets as a constructor parameter
- **EmptyScope** - scope with no nodes
- **SimpleRoleScope** - a scope providing all child nodes of a node, which match a given role
- **ModelsScope** - a scope containing all nodes of a given concept contained in the supplied set of models
- **ModelPlusImportedScope** - like ModelsScope, but includes all models imported by the given model

For example, the `getScope()` method could be rewritten using `ListScope` this way:

```java
public Scope getScope(conceptNode<> kind, node<> child) {
    if (kind.isSubConceptOf(InputField) && come from outputField) {
        return new ListScope(this.inputField) {
            public string getName(node<> child) {
                child : InputField.name;
            };
        }
    }
    return parent scope;
}
```

### 24.2.2 VariableReference

A slightly more advanced example can be found in `BaseLanguage`. `VariableReference` uses inherited scope for its `variableDeclaration` reference.
Concepts such as `ForStatement`, `LocalVariableDeclaration`, `BaseMethodDeclaration`, `Classifier` as well as some others add variable declarations to the scope and thus implement `ScopeProvider`.

```java
public Scope getScope(conceptNode<> kind, node<> child) {
    overrides ScopeProvider.getScope {
        if (kind.isExactly(VariableDeclaration)) {
            nlist<LocalVariableDeclaration> variables = new nlist<LocalVariableDeclaration>;
            if (lcome from variable) {
                variables.add(this.variable);
                if (come from additional1Var) {
                    for (node<AdditionalForLoopVariable> variable : this.additionalVar) {
                        if (variable == child) { break; }
                        variables.add(variable);
                    }
                } else {
                    variables.addAll(this.additionalVar);
                }
            }
            return Scopes.forVariables(kind, variables, parent scope);
        }
    }
    super.getScope(kind, child);
}
```
For example, `ForStatement` uses the `Scopes.forVariables` helper function to build a scope that enriches the parent scope with all variables declared in the for loop, potentially hiding variables of the same name in the parent scope. The `come from` expression detects whether the reference that we’re currently resolving the scope for lies in the given part of the sub-tree.

- The **parent scope** construct will create an instance of `LazyParentScope()` and effectively delegate to an ancestor in the model, which implements `ScopeProvider`, to supply the scope.
- The **come from** construct will delegate to `ScopeUtils.comeFrom()` in order to check whether the scope is being calculated for a direct child of the current node in the given role.
- The **composite with** construct (used as `composite <expr> with parent scope`) will create a combined scope of the supplied scope expression and the parent scope.

### 24.3 Using reference scope

Scopes can alternatively be implemented in a faster but less scalable way - using the reference scope:

```java
InputFieldReference Concepts constraints InputFieldReference {
  can be child <none>
  can be parent <none>
  can be ancestor <none>
  <property constraints>>
  link {field} referent set handler:<none>
  scope: $referenceScope$ creates scope for the reference (jetbrains.mps.scope.ScopeScope)
  default scope <default scope>
}
```

Instead of delegating to the ancestors of type `ScopeProvider` to do the resolution, you can insert the scope resolution code right into the constraint definition.

You may need to import (`Control/Cmd + R`) the `jetbrains.mps.scope` model in order to be able to use `SimpleRoleScope`.

```java
InputFieldReference link {field} referent set handler:<none>
  scope: $referenceScope$ creates scope for the reference (jetbrains.mps.scope.ScopeScope)
  default scope <default scope>
}
```

Instead of the code that originally was inside the `Calculator.getScope()` method, it is now `InputFieldReference` itself that defines the scope. The function for reference scope is supposed to return a `Scope` instance, just like the `ScopeProvider.getScope()` method. `Scope` is essentially a list of potential reference targets together with logic to resolve these targets with textual values.

To remind you, there are several predefined `Scope` implementations and related helper factory methods ready for you to use:

- **SimpleRoleScope** - simply adds all nodes connected to the supplied node and being in the specified role
- **ModelPlusImportedScope** - provides reference targets from imported models. Allows the user add targets to scope by `ctrl + R / cmd + R` (import containing model).
- **FilteringScope** - allow you to exclude some elements from another scope. Subclasses of `FilteringScope` with override the `isExcluded()` method.
- **DelegatingScope** - delegates to another scope. Meant to be overridden to customize the behavior of the original scope.
You may also look around yourself in the scope model:
Chapter 25

Behavior

During syntax tree manipulation, common operations are often extracted to utility methods in order to simplify the task and reuse functionality. It is possible to extract such utilities into static methods or create node wrappers holding the utility code in virtual methods. However, in MPS a better solution is available: the behavior language aspect. It makes it possible to create virtual and non-virtual instance methods, static methods, and concept instance constructors on nodes.

25.1 Concept instance methods

A Concept instance method is a method, which can be invoked on any specified concept instance. They can be both virtual and non-virtual. While virtual methods can be overridden in extending concepts, non-virtual ones cannot. Also a virtual concept method can be declared abstract, forcing the inheritors to provide an implementation. Concept instance methods can be implemented both in concept declarations and in concept interfaces. This may lead to some method resolution issues. When MPS needs to decide, which virtual method to invoke in the inheritance hierarchy, the following algorithm is applied:

- If the current concept implements a matching method, invoke it. Return the computed value.
- Invoke the algorithm recursively for all implemented concept interfaces in the order of their definition in the implements section. The first found interface implementing the method is used. In case of success return the computed value.
- Invoke the algorithm recursively for an extended concept, if there is one. In case of success return the computed value.
- Return failure.

25.1.1 Overriding behavior methods

In order to override a method inherited from a super-concept, use the Control/Cmd + O keyboard shortcut to invoke the Override dialog. There you can select the method to override. By typing the name of the desired method to override you narrow down the list of methods.

25.2 Concept constructors

When a concept instance is created, it is often useful to initialize some properties/references/children to the default values. This is what concept constructors can be used for. The code inside the concept construction is invoked on each instantiation of a new node of a particular concept.

• The node’s constructor is invoked before the node gets attached to the model. Therefore it is pointless to investigate the node’s parent, ancestors, children or descendants in the behaviour constructor. These calls will always evaluate to null. You should define NodeFactories (Editor Actions) in order to have your nodes initialized with values depending on their context within the model.

25.3 Concept static methods

Some utility methods do not belong to concept instances and so should not be created as instance methods. For concept-wide functionality, MPS provides static concept methods. See also Constraints
Chapter 26

Editor

Once the structure for your language is defined, you will probably go and create the means to allow developers to conveniently build ASTs with it. Manipulating the ASTs directly would not be very intuitive nor productive. To hide the AST and offer the user comfortable and intuitive interaction is the role for language editors.
There are sometimes situations when manipulating the AST directly is necessary. For example, when the available editor definition does not give you access to all the properties of a node. The reflective editor gives you the power to cease the editor for a selected node and instead access the AST directly. Hit \texttt{F5} to revert back to the default editor.

```java
public class Sample {
    private int value;
    public Sample(int value) {
        this.value = value;
    }

    public void incrementValue(int increment) {
        this.value = increment;
    }

    static method declaration main {
        resolve info : main
        is final : false
        is deprecated : false
        is synchronized : false
        is native : false
        annotation :
            \texttt{\<... \>}
        visibility :
            \texttt{public}
        type variable declaration :
            \texttt{\<... \>}
        return type :
            \texttt{void}
        modifiers :
            \texttt{\<... \>}
        throws item :
            \texttt{\<... \>}
        body :
            final sequence<int> numbers = new int[]{4, 8, 12};
            parallel for (final int a in numbers) {
                Logger.log("Current value: " + a);
            }
        parameter :
            string[] args
    }

    \@thread safe
    public static class Logger {
        private static synchronized void log(string msg) {
            System.out.println(msg);
        }
    }
}
```

26.1 Editor Overview

An editor for a node serves as its view as well as its controller. An editor displays the node and lets the user modify, replace, delete it and so on. Nodes of different concepts have different editors. A language designer should create an editor for every concept in his/her language.

In MPS, an editor consists of cells, which themselves contain other cells, some text, or a UI component. Each editor has its concept for which it is specified. A concept may have no more than one editor declaration (or can have none). If a concept does not have an editor declaration, its instances will be edited with an editor for the concept’s nearest ancestor that has an editor declaration.
To describe an editor for a certain concept (i.e. which cells have to appear in an editor for nodes of this concept), a language designer will use a dedicated language simply called editor language. You see, MPS applies the Language Oriented Programming principles to itself.

The description of an editor consists of descriptions for cells it holds. We call such descriptions "cell models." For instance, if you want your editor to consist of a unique cell with unmodifiable text, you create in your editor description a constant cell model and specify that text. If you want your editor to consist of several cells, you create a collection cell model and then, inside it, you specify cell models for its elements. And so on.

For a quick how-to document on the MPS editor please check out the Editor cookbook.

### 26.2 Types Of Cell Models

<table>
<thead>
<tr>
<th>Cell Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Constant cell</strong></td>
<td>This model describes a cell which will always contain the same text. Constant cells typically mirror &quot;keywords&quot; in text-based programming languages.</td>
</tr>
<tr>
<td><strong>Collection cell</strong></td>
<td>A cell which contains other cells. Can be horizontal (cells in a collection are arranged in a row), vertical (cells are on top of each other) or have so-called &quot;indent layout&quot; (cells are arranged horizontally but if a line is too long it is wrapped like text to the next line, with indent before each next line). In Inspector, you can specify whether the resulting cell collection will use folding or not, and whether it will use braces or not. Folding allows your cell list to contract into a single cell (fold) and to expand from it (unfold) when necessary. It is useful for a programmer writing in your language when editing a large root: he/she is able to fold some cells and hide all the information that is not necessary for the current task. For instance, when editing a large class, one can fold all method bodies except the method he/she is editing at the moment. The <strong>collapse by default</strong> property when set to true will ensure, that the collection shows up folded when displayed for the first time, unless the user unfolds it manually. Collection cells can also specify a Context assistant, which will provide intuitive visual actions to the user. Check out the Context Assistant for details.</td>
</tr>
<tr>
<td><strong>Property cell</strong></td>
<td>This cell model describes a cell which will show a value of a certain property of a node. The value of a property can be edited in a property cell, therefore, a property cell serves not only as a view also but as a controller. In an inspector, you can specify whether the property cell will be read-only or will allow its property value to be edited.</td>
</tr>
<tr>
<td><strong>Child cell</strong></td>
<td>This cell model contains a reference to a certain link declaration in a node’s concept. The resulting cell will contain an editor for the link’s target (almost always for the child, not the referent). For example if you have a binary operation, say &quot; + &quot;, with two children, &quot;leftOperand&quot; and &quot;rightOperand&quot;, an editor model for your operation will be the following: a indent collection cell containing a referenced node cell for the left operand, a constant cell with &quot; + &quot;, and a referenced node cell for the right operand. It will be rendered as an editor for the right operand, then a cell with &quot; + &quot;, and then an editor for the left operand, arranged in a row. As we have seen, and as follows from its name, this type of cell model is typically used to show editors for children.</td>
</tr>
</tbody>
</table>
### Referent cell

|%reference% -> {%name%}|

Used mainly to show reference targets.

The main difference between a referent cell and a child cell is that we don’t need, or don’t want, to show the whole editor for a reference target. For example, when a certain node, say, a class type, has a reference to a Java class, we don’t want to show the whole editor for that class with its methods, fields, etc - we just want to show its name. Therefore child cells cannot be used for such a purpose. One should use referent cells.

Referent cell allows you to show a different inlined editor for a reference target, instead of using target’s own editor. In most cases it’s very simple: a cell for a reference target usually consists only of a property cell with the target’s name.

### Child list cell

[empty cell:

This cell is a collection containing multiple child cells for a node’s children of the same role. For instance, an editor for a method call will contain a child list cell for rendering its actual arguments. Child list can be indent (text like), horizontal or vertical.

The cell generated from this cell model supports insertion and deletion of the children of the role given, thus serving both as a view and as a controller. The default keys for insertion are Insert and Enter (to insert a child before or after the selected one, respectively), and the default key for deletion is Delete. You also can specify a separator for your list.

A separator is a character which will be shown in constant cells between cells for the children. When you are inside the cell list and you press a key with this character, a new child will be inserted after the selected child. For instance, a separator for a list representing actual parameters in a method call is a comma.

In Inspector, you can specify whether the resulting cell list will use folding or not, and whether it will use braces or not. Folding allows your cell list to contract into a single cell (fold) and to expand from it (unfold) when necessary. It is useful for a programmer writing in your language when editing a large root: he/she is able to fold some cells and hide all the information in editor that is not necessary for the current task at the moment. For instance, when editing a large class, one can fold all method bodies except the method he/she is editing at the moment.

### Indent cell

[empty cell:

An indent cell model will be generated into a non-selectable constant cell containing a whitespace. The main difference between a cell generated from an indent cell and one generated from a constant cell model containing whitespaces as its text is that the width of an indent cell will vary according to user-defined global editor settings. For instance, if a user defines an indent to be 4 spaces long, then every indent cell will occupy a space of 4 characters; if 2 spaces long, then every indent cell will be 2 characters.

### UI component cell

[empty cell:

This cell model allows a language designer to insert an arbitrary UI component inside an editor for a node. A language designer should write a function that returns a JComponent, and that component will be inserted into the generated cell. Note that such a component will be re-created every time an editor is rebuilt, so don’t try to keep any state inside your component. Every state should be taken from and written into a model (i.e. node, its properties and references) - not a view (your component).

A good use case for such a cell model is when you keep a path to some file in a property, and your component is a button which activates a modal file chooser. The default selected path in a file chooser is read from the above-mentioned property, and the file path chosen by the user is written to that property.
Model access

A model access cell model is a generalization of a property cell and, therefore, is more flexible. While a property cell simply shows the value of a property and allows the user to change that value, a model access cell may show an arbitrary text based on the node’s state and modify the node in an arbitrary way based on what changes the user has made to the cell’s text.

While making a property cell work requires you only to specify a property to access via that cell, making a model access cell work requires a language designer to write three methods: "get," "set," and "validate." The latter two are somewhat optional.

A "get" method takes a node and should return a String, which will be shown as the cell’s text. A "set" method takes a String - the cell’s text - and should modify a node according to this String, if necessary. A "validate" method takes the cell’s text and returns whether it is valid or not. If a text in a cell becomes invalid after a user change, then it is marked red and is not passed to the "set" method.

If a "validate" method is not specified, a cell will always be valid. If a "set" method is not specified, no changes in a cell’s text will affect its node itself.

Next applicable editor

A more specific editor may reuse a less specific editor of the same concept through the new next applicable editor editor cell. The next applicable editor cell is used as a place holder, which will re-apply the logic for finding the less specific editor and insert the found editor in its place. For example, an editor specific to a particular context hint may provide some visual ceremony around the next applicable editor cell. By removing the context hint on the next applicable editor cell, MPS will reevaluate the editor-discovery logic and supply the found editor into the next applicable editor cell.

Custom cell

If other cell models are not enough for a language designer to create the editor he/she wants, there’s one more option left for him/her: to create a cell provider which will return an arbitrary custom cell. The only restriction is that it should implement an "EditorCell" interface.

26.2.1 Editor Components and editor component cells

Sometimes two or more editor declarations for different concepts have a common part, which is duplicated in each of those editors. To avoid redundancy, there’s a mechanism called editor components. You specify a concept for which an editor component is created and create a cell model, just as in concept editor declaration. When written, the component could then be used in editor declarations for any of the specified concept’s descendants. To use an editor component inside your editor declarations, one will create a specific cell model: editor component cell model, and set your editor component declaration as the target of this cell model’s reference.

26.3 Cell layouts

Each collection cell has property "cell layout", which describes how child nodes will be placed. There is several layouts:
CHAPTER 26. EDITOR

- indent layout - places cells like text.
- horizontal layout - places cells horizontally in row.
- vertical layout - places cells vertically.

26.4 Styles

Styling the editor cells gives language designers a very powerful way to improve readability of the code. Having keywords, constants, calls, definitions, expressions, comments and other language elements displayed each in different colors or fonts helps developers grasp the syntax more easily. You can also use styling to mask areas of the editor as read-only, so that developers cannot edit them.

Each cell model has some appearance settings that determine the cell’s presentation. They are, for instance, font color, font style, whether a cell is selectable, and some others. Those settings are combined into an entity called stylesheet. A stylesheet could be either inline, i.e. be described together with a particular cell model, or it could be declared separately and used in many cell models. Both inline stylesheet and style reference are specified for each cell in its Inspector View.
The settings do not have to be specified by a single value. A query option is also available for all settings, in which case a concept function needs to be implemented by the developer, which returns the desired value:

```java
font-family : (editorContext, node) -> string {
    if (node.someCondition) {
        return fontFamily/URW Chancery L/;
    } else {
        return fontFamily/Comfortaa/;
    }
}
```

It is a good practice to declare a few stylesheets for different purposes. Another good practice is to have a style guideline in mind when developing an editor for your language, as well as when developing extensions for your language. For example, in BaseLanguage there are styles for keywords (applied to those constant cells in the BaseLanguage editor, which correspond to keywords in Java), static fields (applied to static field declarations and static field references), instance fields, numeric literals, string literals, and so forth. When developing an extension to BaseLanguage, you should apply keyword style to new keywords, field style to new types of fields, and so forth.

A stylesheet is quite similar to CSS stylesheets; it consists of a list of style classes, in which the values for some style properties are specified. MPS additionally provides a mechanism for extending styles as well as for property value overriding.

### 26.5 Style properties

#### 26.5.1 Boolean style properties

- **selectable** - whether the cell can be selected. True by default.
- **read-only** - whether one can modify the cell and the nested cells or not. False by default. Designed for freezing fragments of cell tree.
- **editable** - whether one can modify text in a cell or not. By default is false for constant cell models, true for other cell models.
- **draw-border** - whether border will be drawn around a cell
- **draw-brackets** - whether brackets will be drawn around a cell
- **first-position-allowed / last-position-allowed** - for text-containing cells, specifies whether it is allowed that a caret is on the first/last position (i.e. before/after the whole text of a cell)

You can either choose a property value from a completion menu or specify a query i.e. a function which returns a boolean value.

```java
Style:
Keyword {
    selectable : false
    read-only : true
    first-position-allowed : (editorContext, node) -> boolean {
        if (node.parent.isInstanceOf(Script)) {
            return node.isInstanceOf(RoutineDefinition);
        }
        return false;
    }
}
```

#### 26.5.2 Padding properties.

- **padding-left/right/top/bottom** - a floating point number, which specifies the padding of a text cell, i.e. how much space will be between cell’s text and cell’s left and right sides, respectively.
26.5.3 Punctuation properties.

All cells in a collection are separated with one space by default. Sometimes we need cells placed together.

- **punctuation-left** - if this property is true, space from left side of the cell is deleted and first position in cell becomes not allowed.
- **punctuation-right** - if this property is true, space from right side of the cell is deleted and last position in cell becomes not allowed.
- **horizontal-gap** - specifies gap size between cells in collection. Default value is 1 space.

For example in code
\[(1 + 1)\]

we don’t want spaces between "(" and "1", and between "1" and "). So we should add property punctuation-right to the cell "(" and property punctuation-left to the cell ")".

26.5.4 Color style properties

- **Text foreground color** - cell text’s color (affect text cells only)
- **Text background color** - cell text’s background color (affects text cells only)
- **Background color** - the background color of a cell. Affects any cell. If a text cell has non-zero padding and some text background color, the cell’s background color will be the color of its margins. You can either choose a color from the completion menu or specify a query i.e. a function which returns a color.

26.5.5 Indent layout properties

- **indent-layout-indent** - all lines will be placed with indent. This property can be used for indent in code block.

- **indent-layout-new-line** - after this cell there will be a new line marker.
• `indent-layout-on-new-line` - this cell will be placed on a new line

• `indent-layout-new-line-children` - all children of collection will be placed on new line

• `indent-layout-no-wrap` - the line won’t be wrapped before this cell

26.5.6 Other style properties

• `font family`

• `font size`

• `font style` - can be either plain, bold, italic, or bold italic.

• `layout constraint` -
– For flow layout
  - none - default behavior
  - punctuation - means that previous item in flow layout should always be placed on the same line as the item, which this constraint is assigned to.
  - noflow - excludes a cell from flow layout. Current line is finished and item is placed below it. After this item a new line is started and normal flow layout is applied. This style can be used to embed a picture inside of text.

- **underlined** - Can be either underlined, not underlined, or as is ('as is' means it depends on properties of the enclosing cell collection).

### 26.5.7 Style properties propagation

While some style properties affect only the cell to which they are applied, values of other properties are pushed down the cell subtree (nested cells) and applied to them until some of the child cells specifies its own value for the property. Such inheritable properties that are pushed down the cell hierarchy include `text-foreground-color`, `text-background-color`, `background-color`, `font-style`, `font-size` and many others.

### 26.5.8 Custom styles

Language designers can define their own style attributes in style sheets and then use them in the editor. This increases the flexibility of the language editor definition. The attributes may hold values of different types and can optionally provide default values.

```
simple style attribute myFlag
  of type boolean
default false

inherited style attribute myValue
  of type string
default <no defaultValue>
```

There are two types of custom style attributes:

- **simple** - applied to a single editor cell only
- **inherited** - applied to a cell and all its descendant cells recursively

In order to use the style attribute in an editor definition, your language has to import the language defining the attribute and the `editor` aspect has to list the defining language among the used languages.

To refer to the custom attribute from within `BaseLanguage` code, you need to import `jetbrains.mps.lang.editor` to get access to the `StyleAttributeReferenceExpression` concept.

```
EditorCell cell = ...  
cell.getStyle().get(styleAttribute/myValue/);
```

### 26.5.9 Style inheritance

To be truly usable, style classes need an extension mechanism in order to describe that a particular style class inherits values of all style properties, which are not overridden explicitly. We can use a special style property `apply` to copy values of all properties specified in the parent style class into our style class. Using the `apply` property is semantically equivalent to copy-pasting all of the properties from the parent style class. An `apply-if` variant is also available to apply a style property value conditionally. Unlike traditional style-extension, the apply mechanism allows multiple classes to be inherited from. The `unapply` property allows style classes to cease the effect of selected inherited properties. For example, a style class for commented-out code will push down styles that make code elements look all gray. Yet, links may need to be rendered in their usual colors so that the user can spot them and potentially click on them.

Potential conflicts between properties specified in parent styles and/or the ones defined explicitly in the inheriting cell are resolved on the order basis. The last specified value overrides all previous values of the same style property. For example, the `ConsoleRoot` concept provides a read-only editor with only a single point (the `commandHolder` cell), where edits are allowed. First the `readOnly` style class is set on the editor:
and then the `readOnly` style class is unapplied for the `commandHolder` cell:

The `readOnly` style class is defined as follows:

```plaintext
stylesheets Console_Styles {  
  style readOnly dominates over <all> {  
    read-only : true  
  }  
}
```
26.5.10 Style priorities

A style class can be declared to take precedence over some other style class or multiple classes.

1. If a style class does not dominate over anything, it is a low-level style class.
2. If a style class declares to dominate, but does not specifies a style class that it dominates over (no style class is specifies but words dominate over present), the style class is considered dominating over all low-level style classes.
3. The domination relation is transitive, cycles are not allowed.

The domination relation makes sense only for styles with inheritable attributes. When one value of some style property is pushed down from parent and another value for the same property is specified in the style class applied to the current cell, the resulting behavior depends on the relationship between the two style classes:

1. If both style classes are low-level, the value pushed from parent will be ignored and replaced with value from style class of current cell.
2. If one of style classes dominates over the other, both values are kept and pushed down, but values from the style class, which dominates, hides the values from the other style class.
3. If, however, in some child cell the style class that dominates is unapplied (with special style property unapply), values from the other style class will become resulting values for this property.

For example, a comment containing the word TODO should be styled more prominently then a plain comment. Thus the language concept representing a comment needs to apply a TODO-aware style (TODO_Style), which declares its dominance over a plain Comment.Style. The actual styling properties are, however, only applied if the comment really contains the TODO text (isToDo()), otherwise the plain Comment_Style properties are used.

```java
style TODO_Style dominates over Comment_Style {
    apply-if TODO condition: {editorContext, node} -> boolean {
        return node.isInstanceOf(TextCommentPart) && node as TextCommentPart.isToDo();
    }
}
```

Use the “Add Dominance” intention to append the dominates over clause to a style:

26.6 Cell actions

Every cell model may have some actions associated with it. Such actions are meant to improve usability of editing. You can specify them in an inspector of any cell model.

26.6.1 Key maps

You may specify a reference to a key map for your cell model. A key map is a root concept - a set of key map items each consisting of a keystroke and an action to perform. A cell generated from a cell model with a reference to a certain key map will execute appropriate actions on keystrokes.

In a key map you must specify a concept for which a key map is applicable. For instance, if you want to do some actions with an expression, you must specify Expression as an applicable concept; then you may specify such a key map only for those cell models which are contained inside editor declarations for descendants of Expression, otherwise it is a type error.

If a key map property "everyModel" is "true," then this key map behaves as if it is specified for every cell in the editor. It is useful when you have many descendants of a certain concept which have many different editors, and your key map is applicable to their ancestor. You need not specify such a key map in every editor if you mark it as an "every model" key map.

A key map item consists of the following features:

- A function which is executed when a key map item is triggered (returns nothing)
- A set of keystrokes which trigger this key map item
A boolean function which determines if a key map item is applicable here (if not specified, then it’s always applicable). If a key map item is not applicable the moment it is triggered, then it will not perform an action.

You may specify caret policy for a key map item. Caret policy says where in a cell a caret should be located to make this key map item enabled. Caret policy may be either first position, last position, intermediate position, or any position. By default, caret policy is “any position.” If a caret in a cell does not match the caret policy of a key map item the moment it is triggered, then this key map item will not perform an action.

### 26.6.2 Action maps

A cell model may contain a reference to an action map. An action map overrides some default cell actions for a certain concept. An action map consists of several action map items. In an action map, you must specify a concept for which the action map is applicable.

An action map item contains:

- an action description which is a string,
- and a function which performs an action (returns nothing).

An action map item may override one of the default actions (see Actions). For instance, when you have a return statement without any action maps in its editor, and you press Delete on a cell with the keyword “return,” the whole statement is deleted. But you may specify an action map containing a delete action map item, which instead of just deleting return statement replaces it with an expression statement containing the same expression as the deleted return statement.

```java
action DELETE
description : <no description>
execute : (node, editorContext)->void {
    node < ExpressionStatement > expressionStatement = node . replace with new ( ExpressionStatement
    expressionStatement . expression . set ( node . expression ) ;
}
```

The SELECT_ALL action, which selects the whole contents of the editor and is triggered by Control/Cmd + A, can also be customised through action maps. The jetbrains.mps.nodeEditor.selection.SelectUpUtil class with executeWhile method can be leveraged to specify an upper selection boundary for this action.

```java
action SELECT_ALL
description : <no description>
can execute : (editorContext, node)->boolean {
    SelectUpUtil.canExecute(editorContext);
}
execute : (editorContext, node)->void {
    SelectUpUtil.executeWhile(editorContext, { =>
        SelectionManager m = editorContext . getSelectionManager();
        Selection selection = m . getSelection();
        selection . getSelectedNodes().get(0) . inst: node . command;
    });
}
```

### 26.7 Cell menus

One may specify a custom completion menu for a certain cell. Open an inspector for your cell declaration, find a table named Common, find a row named menu, create a new cell menu descriptor. Cell menu descriptor consists of menu parts, which are of different kinds, which are discussed below.

#### 26.7.1 Property values menu part

This menu part is available on property cells, it specifies a list of property values for your property which will be shown in completion. One should write a function which returns a value of type list<String>.

#### 26.7.2 Property postfix hints menu part

This menu part is available on property cells, it specifies a list of strings which serve as “good” postfixes for your property value. In such a menu part one should write a function which returns a value of type list<String>. Such a menu is useful if you want MPS to “guess” a good value for a property. For instance, one may decide that it will be a good variable name which is a variable type name but with the first letter being lowercased, or which ends with its type name: for a variable of type "Foo" good names will be "foo", "aFoo", "firstFoo", "goodFoo", etc. So one should write in a variable declaration’s editor in a menu for property cell for variable name such a menu part:
property postfix hints
postfixes : (scope, operationContext, node)->list<String> {
    list < String > result ;
    node < Type > nodeType = node . type ;
    if ( nodeType != null ) {
        result = MyUtil.splitByCamels( nodeType . getPresentation() );
    } else {
        result = new list < String > { empty } ;
    }
    return result ;
}

where splitByCamels() will be a function which returns a list of postfixes of a string starting with capitals (for instance MyFooBar -> MyFooBar, FooBar, Bar).

26.7.3 Primary replace child menu
It’s a cell menu part which returns primary actions for child (those by default, as if no cell menu exists).

26.7.4 Primary choose referent menu
It’s a cell menu part which returns primary actions for referent (those by default, as if no cell menu exists).

26.7.5 Replace node menu (custom node’s concept)
This kind of cell menu parts allows to replace an edited node (i.e. node on which a completion menu is called) with instances of a certain specified concept and its subconcepts. Such a cell menu part is useful, for example, when you want a particular cell of your node’s editor to be responsible for replacement of a whole node. For instance, consider an editor for binary operations. There’s a common editor for all binary operations which consists of a cell for left operand, a cell for operation sign which is a cell for concept property "alias" and a cell for right operand.

\[
> \% \text{leftExpression} \% ^{{ \{ \text{alias} \}}} \% \text{rightExpression} \% <
\]

It is natural to create a cell menu for a cell with operation sign, which will allow to replace an operation sign with another one, (by replacing a whole node of course). For such a purpose one will write in the cell for operation sign a replace node menu part:

replace node (custom node concept)
.replace with : BinaryOperation

The former left child and right child are added to newly created BinaryOperation according to Editor Actions for BinaryOperation concept.

26.7.6 Replace child menu (custom child’s concept)
Such a cell menu part is applicable to a cell for a certain child and specifies a specific concept which and subconcepts of which will be shown in completion menu (and instantiated when chosen and the instance will be set as a child). To specify that concept one should write a function which returns a value of a type node<ConceptDeclaration>.

26.7.7 Replace child menu (custom action).
This kind of cell menu parts is applicable to a cell for a certain child and allows one to customize not only child concept, but the whole replace child action: matching text (text which will be shown in completion menu), description text (a description of an action, shown in the right part of completion menu), and the function which creates a child node when the action is selected from completion menu. Hence, to write such a menu one should specify matching text, description text and write a function returning a node (this node should be an instance of a target concept specified in a respective child link).

26.7.8 Generic menu item
This kind of cell menu part allows one to make MPS perform an arbitrary action when a respective menu item will be selected in a completion menu. One should specify matching text for a menu item and write a function which does what one wants. For instance, one may not want to show a child list cell for class fields if no class fields exist. Hence one can’t use its default actions to create a new field. Instead, one can create somewhere in a class’ editor a generic menu item with matching text "create field" which creates a new field for a class.
26.7.9 Action groups

An action group is a cell menu part which returns a group of custom actions. At runtime, during the menu construction, several objects of a certain type, which are called parameter objects, are collected or created. For that parameter object type of an action group functions, which return their matching text and description text, are specified. A function which is triggered when a menu item with a parameter object is chosen is specified also.

Thus, an action group description consists of:

• a parameter object type;
• a function which returns a list of parameter objects of a specified type (takes an edited node, scope and operation context);
• a function which takes a parameter object of a specified type and returns matching text (a text which will be shown in a completion menu);
• a function which takes a parameter object of a specified type and returns description text for a parameter object;
• a function which performs an action when parameter object is chosen in a completion menu.

A function which performs an action may be of different kinds, so there are three different kinds of cell action group menu parts:

• Generic action group. Its action function, given a parameter object, performs an arbitrary action. Besides the parameter object, the function is provided with edited node, its model, scope and operation context.

• Replace child group. It is applicable to child cells and its action function, given a parameter object, returns a new child, which must have a type specified in a respective child link declaration. Besides the parameter object, the function is provided with edited node, its model, current child (i.e. a child being replaced), scope and operation context.

• Replace node group. Its action function, given a parameter object, returns a node. Usually it is some referent of an edited node (i.e. node on which a completion menu is called). Besides the parameter object, the function is provided with edited node, its model, scope and operation context.

26.7.10 Cell menu components

When some menu parts in different cells are equal one may want to extract them into a separate and unique entity, to avoid duplications. For such a purpose cell menu components are meant. A cell menu component consists of a cell menu descriptor (a container for cell menu parts) and a specification of an applicable feature. A specification of applicable feature contains a reference to a feature (i.e. child link declaration, reference link declaration or property declaration), to which a menu is applicable. For instance if your menu component will be used to replace some child its child link declaration should be specified here; etc.

When a cell menu component is created, it can be used in cell menus via cell menu component menu part, which is a cell menu part which contains a reference to a certain menu component.

26.8 Customizing reference presentation

Specification of the matching text and in-editor textual presentation for references can be done directly in the editor aspect. The ref. presentation cell can have the displayed text customized.
The cell menu can customise the text displayed in the completion menu:

This functionality was previously achieved through Constraints.

### 26.9 Migration of presentation query in reference constraints

The design of the reference presentation part in the constraints aspect has been showing its age and so has been replaced with the new functionality described above. Most of the code will be migrated automatically. Some code that produced with migration can be simplified so consider to review it.

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

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

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

### 26.10 Two-step deletion

In projectional editor it is sometimes hard to predict what part of the code will be deleted when you press Delete or Backspace. For example, when the caret is on the semicolon of the baseLanguage statement and you press Backspace, the whole statement will be deleted. With two step deletion, you now can see what part of the code which will be deleted. Here how it works: you press Delete or Backspace and the part of the code, which is to be deleted, becomes highlighted. If it suits you, you press Delete or Backspace again and the code will be deleted. If after highlighting you realize that you don’t want to delete this piece of code you can press Escape or just move the caret and the highlighting will disappear.

Let’s see the example:

Put the caret to the statement semicolon.

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

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

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

Press Backspace again. The statement is deleted.

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

The same works by default for other nodes.

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

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

### 26.10.1 Invoking two-step deletion from code

The language designer may include the two step deletion scenario in her custom delete actions. The `ApproveDelete_Operation` in `jetbrains.mps.lang.editor` is introduced for that purpose. This operation is applied to the node:

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

This operation returns true iff it succeed and the node has not been approved for deletion before. More formally all the following conditions need to be met:

1) The two-step deletion preferences option is checked.
2) The node has not been fully selected.
3) The node has not been approved for deletion already.

When all of these conditions are met, the node approved for deletion gets highlighted and the custom delete action may stop at this point.

If the same custom delete action is called immediately after approving the deletion, the `approveDelete` operation will return false (because the node has been approved already) and the action will proceed with the deletion.
Let's see the typical scenario from the baseLanguage:

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

This is the part of the delete action for the Dot_Expression's operation. This action first tries to approve the operation for the deletion and if it succeed the action stops. If it does not succeed, that means that either node's operation has already been approved (= highlighted), or the node has been selected by the user or the "two step deletion" preferences option is turned off. In this case, we delete the operation and replace it with the node of the abstract concept.

### 26.10.2 More complex cases

Sometimes the customized delete action needs to be more complicated than just deleting the current node. Let's see an example scenario: we press delete on the "final" keyword on the IncompleteMemberDeclaration. There is a custom action, which sets the final property to false. In the editor, there is the cell, which is shown only if the final property of the node is true, so after the action, the cell wouldn't be shown.

If we want to highlight the final keyword before hiding it (by setting the final property to false), we approve it for the deletion this way:

```java
if (node.approveDelete [in: editorContext, cell: finalKeyword]) { return; }
node.final.set(false);
```
Chapter 27

Editor Actions

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

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

Node Factories
When a node is replaced with another one, it may be useful to parametrize the process of creation of the replacing node with values held by the node that is being replaced, or perhaps also to reflect the future position of the replacing node in the model. Node Factories give you exactly that. You write a set of handlers that get invoked whenever a new node needs to be created in a substitution action or through one of the new initialized node<> , set new initialized node<> , add new initialized node<> , replace with new initialized node<> and new initialized instance<> methods.
In brief, Node Factories allow you to customize instantiation of new nodes. In order to create node factory, you first have to create a new Node Factories root node. Inside of this root you can create node factories for concepts. Each node factory consists of node creation block which has the following parameters: newNode (the created node), sampleNode (the currently substituted node; can be null), enclosing node (a node which will be the parent of newNode in the model), and a model. The node factory handler is invoked before the new node gets inserted into the model.
You can leverage the concept inheritance hierarchy in Node Factories to reduce repetition.

<table>
<thead>
<tr>
<th>node concept: Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>description: &lt;none&gt;</td>
</tr>
<tr>
<td>set-up : (newNode, sampleNode, enclosingNode, model)→void</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>node concept: Circle</th>
</tr>
</thead>
<tbody>
<tr>
<td>description: &lt;none&gt;</td>
</tr>
<tr>
<td>set-up : (newNode, sampleNode, enclosingNode, model)→void</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

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

27.1 Paste wrappers

These allow you to customize pasting of nodes into other contexts. For example, if you copy a LocalVariableDeclaration in BaseLanguage and paste it into a ClassConcept to make it a field of the class, a simple transformation must be triggered that will create a new FieldDeclaration out of the LocalVariableDeclaration.
CHAPTER 27. EDITOR ACTIONS

27.2 Copy-paste handlers

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

```java
paste wrappers

paste wrapper Expression -> Statement
(sourceNode) node(nodeStatement) {
    result = new initialized node(nodeStatement);
    return result;
}

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

paste wrapper LocalVariableDeclarationStatement -> LocalVariableDeclaration
(sourceNode) node(nodeLocalVariableDeclaration) {
    return sourceNode.localhostVariableDeclaration;
}

paste wrapper LocalVariableDeclarationStatement -> Statement
(sourceNode) node(nodeStatement) {
    statement = new initialized node(nodeLocalVariableDeclarationStatement);
    return statement;
}

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

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

```java
copy pre processor VariableReference
(copy, original) -> void {
    node = VariableReference
    qualifiedReference = copy.variableDeclaration.get(qualifiedReference);
    if (qualifiedReference != null) {
        copy.replaceWith(qualifiedReference);
    }
}
```

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

```java
paste post processor StaticFieldReference
(sourceNode) -> void {
    if (scope.parent != null) {
        scope.getScope(scope.parent, passedNode, sourceNode, "VariableDeclaration");
        if (surroundingScope != null) {
            surroundingScope.getScope(passedNode, getVariables());
            passedNode.replaceWith(new initialized(sourceNode));
            variableReference.variableDeclaration = passedNode.getVariable();
        }
    }
}
```
Chapter 28

Diagramming Editor

The diagramming support in MPS allows the language designers to provide graphical editors to their concepts. The diagrams typically consist of blocks, represented by boxes, and connectors, represented by lines connecting the boxes. Both blocks and connectors are visualization of nodes from the underlying model.

Ports (optional) are predefined places on the shapes of the blocks, to which connectors may be attached to. MPS allows for two types of ports - input and output ones. Optionally, a palette of available blocks may be displayed on the side of the diagram, so the user could quickly pick the type of the box they need to add to the diagram.

28.0.1 Adding elements

Blocks get added by double-clicking in a free area of the editor. The type of the block is chosen either by activating the particular block type in the palette or by choosing from a pop-up completion menu that shows up after clicking in the free area. Connectors get created by dragging from an output port of a block to an input port of another or the same block.
28.0.2 Samples

MPS comes with bundled samples of diagramming editors. You can try the componentDependencies or the mindMaps sample projects for initial familiarization with how diagrams can be created.
This document uses the `componentDependencies` sample for most of the code examples. The sample defines a simple language for expressing dependencies among components in a system (a component set). Use the "Push Editor Hints" option in the pop-up menu to activate the diagramming editor.
28.1 Dependencies

In order to be able to define diagramming editors in your language, the language has to have the required dependencies and used languages properly set:

- `jetbrains.mps.lang.editor.diagram` - the language for defining diagrams
- `jetbrains.mps.lang.editor.figures` (optional) - a language for defining custom visual elements (blocks and connectors)
- `jetbrains.jetpad` and `jetbrains.mps.lang.editor.diagram.runtime` - runtime libraries that handle the diagram rendering and behavior
28.2 Diagram definition

Let’s start from the concept that should be the root of the diagram. The diagramming editor for that node will contain the diagram editor cell:

```java
<diagram editor for concept "ComponentSet">
    node cell layout:
    {<diagram (content: this.component, this.component.selectMany({it => it.dep; })>)
        <no paletteDeclaration>
    }

    inspected cell layout:
    <choose cell model>
</diagram editor for concept "ComponentSet">
```

Note that the diagram editor cell does not have to be the root of the editor definition. Just like any other editor cell it can be composed with other editor cells into a larger editor definition.

The diagram cell needs its `content` parameter to hold all the nodes that should become part of the diagram. In our case we pass in all the components (will be rendered as blocks) and their dependencies (will be rendered as connectors). The way these nodes are rendered is defined by their respective editor definitions, as explained later.

Down in the Inspector element creation handlers can be defined. These get invoked whenever a new visual block is to be created in the diagram. Each handler has several properties to set:

- `name` - an arbitrary name to represent the option of creating a new element in the completion menu and in the palette
- `container` - a collection of nodes that the newly created node should be added to
- `concept` - the concept of the node that gets created through the handler, defaults to the type of the nodes in the container, but allows sub-types to be specified instead
- `on create` - a handler that can manipulate the node before it gets added to the model and rendered in the diagram. Typically the name is set to some meaningful value and the position of the block on the screen is saved into the model.

There can be multiple element creation handlers defined.

Similarly, connector creation handlers can be defined for the diagram cell to handle connector creation. On top of the attributes already described for element creation handlers, connector creation handlers have these specific attributes:

- `can create` - a concept function returning a boolean value and indicating whether a connector with the specified properties can be legally constructed and added to the diagram.
- `on create` - a concept function that handles creation of a new connector.
• the from and to parameters to these functions specify the source and target nodes (represented by a Block or a Port) for the new connection.

• the fromId and toId parameters to these functions specify the ids of the source and target nodes (represented by a Block or a Port) for the new connection.

Elements get created when the user double-clicks in the editor. If multiple element types are available, a completion pop-up menu shows up. Connectors get created when the user drags from the source block or its output port to a target block or its input port.

28.2.1 Palette

The optional palette will allow developers to pick a type of blocks and links to create whenever double-clicking or dragging in the diagram. The palette is defined for diagram editor cells and apart from specifying the creation components allows for visual grouping and separating of the palette items.

28.2.2 Blocks

The concepts for the nodes that want to participate in diagramming as blocks need to provide properties that will preserve useful diagramming qualities, such as x/y coordinates, size, color, title, etc.

Additionally, the nodes should provide input and output ports, which connectors can visually connect to.
The editor will then use the diagram node cell:

![Diagram Node Cell Example]

The diagram node cell requires a figure to be specified. This is a reference to a figure class that defines the visual layout of the block using the jetpad framework. MPS comes with a set of pre-defined graphical shapes in the jetbrains.mps.lang.editor.figures.library solution, which you can import and use. Each figure may expose several property fields that hold visual characteristics of the figure. All the figure parameters should be specified in the editor definition, most likely by mapping them to the node’s properties defined in the concept:

```java
diagram editor for concept Component
node cell layout:
   "[BlockView {text.name, POSITION_X=x, POSITION_Y=y} inputPorts: #this.in outputPorts: #this.out ]"
inspected cell layout:
   <choose cell model>
```

The values for parameters may either be references to the node’s properties, or BaseLanguage expressions prepended with the # character. You can use this to refer to the edited node from within the expression.

If the node defines input and output ports, they should also be specified as parameters here so that they get displayed in the diagram. Again, to specify ports you can either refer to the node’s properties or use a BaseLanguage expression prepended with the # character.

As all editor cells, diagramming cells can have Action Maps associated with them. This way you can enable the Delete key to delete a block or a connector.

### 28.2.3 Custom figures

Alternatively you can define your own figures. These are BaseLanguage classes implementing the jetbrains.jetpad.projectional.view.View interface (or its descendants) and annotated with the @Figure annotation. Use the @FigureParameter annotation to demarcate property fields, such as width, height etc.

```java
@Figure
public class BlockView extends CenterVerticalLayoutView implements MovableContentView {

    private TextCell myCell = new TextCell();

    public BlockView() {
        super(false);
        Col1View col1View = new Col1View();
        col1View.backgroundColor().set(Color.BLUE);
        col1View.cellView.backgroundColor().set(Color.LIGHT_BLUE);
        col1View.cellView.set(myCell);
        col1View.cellView.setSize(100, 20);
        myCell.add Trait(TextEditing, textEditing());
        children().add(col1View);
        RectView bottomRect = new RectView();
        bottomRect.backgroundColor().set(Color.GRAY);
        bottomRect.dimension().set(new Vector(100, 30));
        children().add(bottomRect);
        initSynchronizers();
    }

    private void initSynchronizers() {
        new MapperBlockView, BlockView>(this, this) {
            @Override
            protected void registerSynchronizers(Mapper.SynchronizersConfiguration configuration) {
            super.registerSynchronizers(configuration);
            }
            }
            .attachRoot();
    }

    @FigureParameter
    public Property<String> text() { return myCell.text(); }  
}
```
The `MovableContentView` interface provides additional parameters to the figure class:

```java
public interface MovableContentView {
    @FigureParameter
    public static final ViewPropertySpec<Integer> POSITION_X = new ViewPropertySpec("position_x");
    @FigureParameter
    public static final ViewPropertySpec<Integer> POSITION_Y = new ViewPropertySpec("position_y");
}
```

By studying `jetbrains.mps.lang.editor.figures.library` you may get a better understanding of the `jetpad` library and its inner workings.

### 28.2.4 Connectors

The nodes that will be represented by connectors do not need to preserve any diagramming properties. As of version 3.1 connectors cannot be visually customized and will be always rendered as a solid black line. This will most likely change in one of the following versions of MPS.

The editor for the node needs to contain a **diagram connector** cell:

```diagram

diagram editor for concept Dependency
node cell layout:

N diagram connector
N diagram node (EditorCellModel in j.m.l.editor.diagram)
N diagram port (EditorCellModel in j.m.l.editor.diagram)

```

The cell requires a source and a target for the connector. These can either be **ports**:

```diagram

\{
  connector [source this.parent : Component.out.first # <no pointID> target: this.to.in.first # <no pointID>]
\}
```

or **nodes** themselves:

```diagram

\{
  connector [source this.parent : Component # <no pointID> target: this.to # <no pointID>]
\}
```

The values may again be direct references to node’s properties or `BaseLanguage` expressions prepended with the `#` character.
28.2.5 Rendering ports
Input and output ports should use the \textit{input port} and \textit{output port} editor cells, respectively. The rendering of ports cannot be customized in MPS 3.1, but will be most likely enabled in later versions.

\textbf{Use the T key to rotate the ports of a selected block by 90 degrees. This way you can easily switch between the left-to-right and top-to-bottom port positions.}

28.2.6 Using implicit ports
In some situations you will not be able to represent ports directly in the model. You’ll only want to use blocks and connectors, but ports will have to be somehow derived from the model. This case can easily be supported:

1. Decide on the representation of ports. Each port will be represented by a unique identifier, such as number or a string

2. Have the concept for the blocks define behavior methods that return collections of identifiers - separately for input and output ports

\begin{code}
concept behavior Component { 
  constructor { 
    this.x = 100; 
    this.y = 100; 
  } 

  public list<Object> retrieveInPorts() { 
    new arraylist<Object>\{\textsc{1}, \textsc{2}, \textsc{3}\}; 
  } 

  public list<Object> retrieveOutPorts() { 
    new arraylist<Object>\{\textsc{10}, \textsc{20}, \textsc{30}\}; 
  } 
}
\end{code}

3. Use the methods to provide the \texttt{inputPorts} and \texttt{outputPorts} parameters to the \texttt{DiagramNode} editor cell

\begin{quote}
\texttt{Diagram editor for concept Component} \\
\texttt{node cell layout:} \\
\{\texttt{BlockView (txt.name, POSITION_X, POSITION_Y)} \texttt{inputPorts: this.retrieveInPorts()} \texttt{outputPorts: this.retrieveOutPorts()}\}
\end{quote}

4. In the connector editor cell refer to the block’s node as \texttt{source} and \texttt{target}. Append the requested id after the \# symbol

\begin{quote}
\texttt{Diagram editor for concept Dependency} \\
\texttt{node cell layout:} \\
\{\texttt{connector (source this.parent \# \"1\" target: this.to \# \"10\")}\}
\end{quote}

\begin{quote}
\texttt{inspected cell layout:} \\
<choose cell model>
\end{quote}
Chapter 29

Context Assistant

29.1 Overview

MPS provides several mechanisms for performing an action in a context: completion, intentions, refactorings, and various other popup menus. These mechanisms have in common that they are not immediately visible to new, inexperienced users. They also usually offer many possible choices and reveal the entire available functionality, which helps advanced users but may overwhelm the beginners.

To better guide the new users through the process of creating a script in your DSL, MPS 3.4 introduces a new UI mechanism, the context assistant. A context assistant shows a dynamically constructed menu with actions that are the most appropriate for a given context. The language author specifies where the menu should be shown by putting placeholders in the editor definition. The placeholders reserve screen space for the menu in advance so that the edited content does not shift around as the menu is being shown and hidden.

As an example consider the RobotKaja sample language which is bundled with MPS. The initial editor for a new script looks as follows (without a context assistant):

```
Script Test runs as
|   end
```

With a context assistant this initial UI might look like this:

```
Script Test runs as
| Step forward | Turn left | Define a routine
|   end
```

and the menu now suggests several possible next steps to the user.

We’ve also shot a short video illustrating use and definition of Context Assistant. You might also like to check out a screen-cast on how context assistant is being utilized for the language definition languages.

29.2 Using Context Assistant UI

- Jump to the context assistant by pressing Ctrl+Alt+Enter (Cmd+Option+Enter on Mac OS X).
- Navigate through the menu by using arrow keys.
- Invoke the selected menu item using Space or Enter.
- Press Escape to jump back to the editor.

29.3 Using Context Assistant Framework

To add context assistant to a language, you as the language author have to do two things:

1. Place context assistant placeholders (a special kind of cell) at appropriate spots in the editor. The context assistant menus will be shown in these placeholders.
2. Define the menu hierarchy using the Transformation Menu Language (specifying location context assistant).

The MPS editor runtime will take care of building the appropriate menu at the appropriate point in time and showing it in the appropriate context assistant placeholder.

### 29.3.1 Placeholder Cells

Placeholder cells are added by choosing "context assistant menu placeholder" from the substitution menu when adding a new cell:

The placeholder cell reserves a certain amount of vertical screen space, about the size of one empty line, so that a menu can be shown in its place without shifting surrounding cells around. However, it doesn’t reserve any horizontal space. It is therefore best to put the placeholder on a separate line or at the end of a short (or empty) line of text. For example, in the RobotKaja sample the placeholder is added after an empty line in the editor for the EmptyLine concept:

![Placeholder example](image)

### 29.3.2 Menu Lookup

The menu to display is looked up by traversing the cell hierarchy from the currently selected cell to the top. You may specify the menu to show explicitly for a given cell (by setting the transformation menu property in the Inspector). In this case that menu is used. Otherwise, if the cell is a big cell (a cell that has no parent or whose parent is associated with a different node), an attempt is made to look up the menu based on the cell’s node. The node’s concept inheritance hierarchy is traversed in breadth-first order. If a non-empty menu is defined for the node’s concept or one of its super-concepts and super-interfaces, this menu is used.

For example, consider a BaseLanguage PlusExpression which extends BinaryOperation, which in turn extends Expression and implements IBinaryLike. If during the traversal we reach the big cell of a PlusExpression, then menus of PlusExpression, BinaryOperation, Expression, IBinaryLike, and finally BaseConcept are checked, in that order, and the first non-empty menu definition is used. If all menu definitions are empty, the search continues from the parent cell of the big cell (if any).

Note that a non-empty menu definition, although chosen, may still produce an empty menu. This may happen if none of its menu parts produce any items (for example if no defined actions are currently applicable).

### 29.3.3 Placeholder Lookup

The place where a menu should be displayed is looked up by traversing the cell hierarchy from the currently selected cell to the root until a collection cell is reached that contains a context assistant placeholder cell, either directly or indirectly (but only belonging to the same node as the collection). The first cell found during this search is chosen and the menu is displayed in this placeholder cell.
Chapter 30

Context actions tool

The Context Actions Tool accommodates for the preference of some DSL users towards mouse navigation. It lists the actions applicable in the given context in a sidebar, potentially hierarchically organized.

By selecting an option in the tool window the user triggers the associated action, which typically modifies the model in the place of the current focus.

The content of the sidebar is specified by the language designer through the new Transformation Menu Language.

The language supports modularization of the menus and their easy reuse, so combining menus of super- or sub-concepts should be quite straightforward. Once you import the jetbrains.mps.editor.contextActionsTool.lang.menus language you’ll be able to specify the context action tool location for your actions, groups and other relevant elements. The language also gives you the possibility to specify an icon and a tooltip for the given entries.

Since the tool window offers enough vertical space and since entries can be grouped into collapsable submenus, you may also consider including existing "substitute" menus into the tool window with the submenu action.

For full details on the actual language syntax see the Transformation Menu Language page.
Chapter 31

Transformation Menu Language

31.1 Overview

Transformation menu language is used to define transformation menus that describe a hierarchical structure of submenus and actions that will appear in various locations in the editor. Currently there are several possible locations where transformation menus are shown: Editor Actions, Editor Actions, Context Assistant, and Context actions tool. Language designers and plugin authors can define additional locations and specify required or optional features for each location (such as an icon or a tooltip), as documented in Extending the Transformation Menu Language.

The Transformation Menu Language provides a way to describe side-transforms and substitute actions in MPS. The core capabilities of the new approach include:

- the ability to explicitly specify side-transform / substitute menu content for a particular cell in the editor
- no need for the "remove defaults" instruction, which often failed to work reliably in the past
- easy mixing substitute actions from the substitute DSL with arbitrary /low-level/ UI actions
- the same action can be visible in different places (parts) of the editor
- supporting different action sets for alternative presentations (editors/projections)
- actions are now defined in the editor aspect of the language

31.2 Defining a Menu

Transformation menus define UI actions that will be shown in various locations. At design time a menu is specified as a list of sections, each section contains a list of menu parts for a particular set of locations. At runtime the menu parts and the locations are used to generate the contents of the menus (menu items). Menu definitions come in two flavors: default and named. Menu definitions can also be extended through menu contributions.

31.2.1 Default Menu

Each concept has a default transformation menu associated. If the language designer does not provide one explicitly, a transformation menu defined for the closest super-concept is assumed. If none is specified for any of the super-concepts, the one defined on BaseConcept is used, which contains the substitute actions suitable for that position (see below the section on substitute actions).

A default menu is used in situations where the language designer hasn’t specified which menu to display.
31.2.2 Named Menu

A named menu is an additional menu for a concept. Like the default menu it also specifies an applicable concept and contains a list of sections. As the term suggests, a named menu has an explicitly set name. A named menu is meant to be set as the transformation menu of a cell or included into another menu via the **Include Menu menu part**.

```plaintext
transformation menu OtherCommands for concept EmptyLine

section(context assistant)

  action "Drop";

  action "Pick";

  ---------

variables

  <\ ...

  condition (editorContext, node)->boolean {
    node.ancestor<concept = CommandList>.parent (eq: node.containing root);
  }

action text (editorContext, node, pattern)->string {
  "Define a routine";
}

can execute <always>

execute (editorContext, node, pattern)->void {
  node<RuntimeDefinition> inserted = node.add prev-sibling(new initialized node<RoutineDefinition>());
  editorContext.selectWHYFocusPolicy(inserted);
}

<no additional features>

---------

submenus

  "Control flow"

  items

    action "Repeat";

    ---------

    action "While";

    -----------

    action "If";

    ---------

submenus

  "Other actions"

  items

    include named menu OtherCommands for current node

}
```

Attaching a named menu to an editor cell:
31.2.3 Menu Contributions

A menu contribution extends a given menu by contributing additional menu parts to it. This is in particular useful, when an extending language needs to add entries into a menu defined in the extended language. Contributions can actually only be defined in languages other than the one with the menu being contributed to. When a menu is requested at runtime the original definition and all contributions are merged and the menu is created using the combined definition. A few important notes on contributions:

1. The order, in which the individual definitions are merged is currently unspecified.
2. A contribution cannot remove menu parts from the menu it contributes to.
3. It is possible to define a contribution to the implicit default menu for a concept.

31.2.4 Section locations

By specifying location for a section within the menu you indicate, into which part of the UI the actions should be inserted:

- **completion** - the completion menu
- **context actions tool** - the Context actions tool (requires import of `jetbrains.mps.editor.contextActionsTool.lang.menus` language)
- **context assistant** - the Context Assistant
• side transform - left or right transformations

31.2.5 Menu Parts

The following standard menu parts are available:

• action – a simple menu item specifying an action to be performed, its corresponding menu text and applicability.

• group - a collection of menu items.

• include - include a specific default or named menu (together with its contributions, if any). Inclusion cycles are detected at runtime and an error message is produced.

• include substitute menu - include a default or named substitute menu to use as part of this menu.

• parametrized - an action that is parametrized with multiple values.

• submenu – a submenu containing further parts.

• superconcepts menu – includes the default menus of the superconcepts of the applicable concept since these are not included by default.

• wrap substitute menu - wraps a specified concept using the provided handler

Language jetbrains.mps.lang.editor.menus.extras contains adapters to include various action-like entities from transformation menus:

• intention – wraps an intention (a subconcept of BaseIntentionDeclaration from jetbrains.mps.lang.intentions).

• refactoring – wraps a refactoring (Refactoring from jetbrains.mps.lang.refactoring).

• plugin Action – wraps a plugin action (ActionDeclaration from jetbrains.mps.lang.plugin).

You may also like to check out a video on Transformation Menu Language.
31.3 Side transformations

When you edit code in a text editor, you can type it either from left to right:

```
1 <caret> press +
1+<caret> press 2
1+2<caret>
```

or from right to left:

```
<caret>1 press 2
2<caret>-1 press +
2+<caret>1
```

In order to emulate this behavior, MPS has side transform actions: left and right transforms. They allow you to create actions which will be available when you type on left or right part of your cell. For example, in MPS you can do the following:

```
1<caret> press + (red cell with + inside appears)
1+<caret> press 2 (red cell disappear)
1+2<caret>
```

or the following:

```
<caret>1 press + (red cell with + inside appears)
+<caret>1 press 2 (red cell disappear)
2<caret>+1
```

The first case is called right transform. The second case is called left transform.

You define side transformations in the Transformation menus by choosing the `side transform` location for the section:

```java
action
  text (editorContext, "filled";
  
  can execute (editorContext, node, model, pattern)->boolean {
    node.filled;
  }
  
  execute (editorContext, node, model, pattern)->void {
    node.filled = true;
  }
  
  icon (editorContext, node, model, pattern)->IconResource {
    iconResource {
      rect fillColor=d0d0d0, border=<default>, size=big
      text color=#010101 , value=1
    }
  }
}
```

The location of the included menu can be specified using the intention "Specify Location".

31.4 Substitute menus

Substitute actions define user-invoked transformations to some parts of the model, during which one node is substituted by another node. The actually mapping of these substitute actions to the visual UI elements (completion menu, etc.) is then done through Transform menus (see the section above).

Typically substitute actions are triggered by pressing `Ctrl + Space` in the editor. The completion menu that shows up contains options that, when selected by the user, will replace the node under caret. Unlike side-transformations, context assistant or context action tool, substitutions have default behavior, which takes effect, unless the language author defines otherwise.
31.4.1 Default behavior for code-completion

Without any menus implemented explicitly by the language author, MPS will still provide a completion menu with substitutions for the current node in either of these two cases:

- The cursor is positioned at the front of a single-cell editor
- The user has selected the whole editor of a node

In these cases pressing Control + Space will show a menu with all concepts from the imported languages applicable in the given context, which can substitute the current node in the model.

MPS follows these steps to populate the default completion menu:

1. If your selection is inside of a position which allows concept A, then all enabled subconcepts of A will be available in the completion menu.
2. All abstract concepts are excluded
3. All concepts, for which the ‘can be a child’ constraint returns false, are excluded
4. All concepts, for which the ‘can be a parent’ constraint of a parent node returns false, are excluded
5. If a concept contains a 1:1 reference, then it is not added to the completion menu itself. Instead, an item is added for each element in scope for that reference. We use a name smart reference for such items.

To customize node substitutions, the substitute menus are used.

31.4.2 Substitute menu (default)

By defining a default substitute menu for a concept you may customize the contents of the completion menu, which displays when the user presses Control + Space. It also has effect on sub-concepts of the concept, unless these sub-concepts define their own default substitute menus.

If you define a default substitute menu for a concept and leave it empty, the concept will not be included in the completion menu. This is the equivalent of using the IDontSubstituteByDefault interface in the previous versions of MPS.
However, if you want to assign the substitute menu to a particular cell of an editor, you will need to include your substitute menu in a transformation menu, because only transformation menus can be attached to editor cells.

```plaintext
default transformation menu for concept Command
section(completion) {
    include default substitute menu for Command for link: <default>
}
```

31.4.3 Substitute menu (named)

Named substitute menus give you the flexibility to create multiple substitute menus and use them in different contexts. Named substitute menus must be first included in another substitute menus or transformation menus to take effect.

31.4.4 Substitute menu contribution

Just like with Transform menus, Substitute menu contributions contribute new entries into substitute menus defined in an extended language.

31.4.5 Options

- add concept - adds a single concept to the menu
- concept list - adds a collection of concepts
- group - adds a group of entries, if a condition is met
- include - includes a specified menu
- parameterized - adds a parametrized substitute action
- reference actions - includes and customises the appearance of the possible targets of a reference
- subconcepts menu - includes all subconcepts of the concept
- substitute action - adds a single substitute action
- wrap substitute menu - wraps a specified concept using the provided handler

31.5 Interaction of Cell Menu and Cell-specific Transformation Menu

If a cell has both "menu" and "transformation menu" specified the applicable entries from both menus are combined. Some cell menu parts (descendants of `CellMenuPart_Abstract` such as `CellMenuPart_PropertyPostfixHints`) do not yet have an equivalent transformation/substitution menu part.

31.6 Include transformation menu for the property/reference

The customization of the menu for the property/reference cells can be done with the `property/reference transformation menu parts`. Suppose you want to customize the completion menu of the reference/property cell. Previously, it could have been done by defining the "inline menu" in the inspector. Now it can be done also via the `Transformation Menu`. We have introduced the property and the reference transformation menu parts. The `reference transformation menu part` includes the actions, which set the target node to the specific reference. The target nodes come from the scope for that reference. The same goes for the property menu part: it takes the values of the property type and includes actions, which set the value to the specific properties.

31.6.1 Example on references

Suppose we want to build the completion menu for a reference cell. The menu should contain the usual targets of the reference that are in scope, plus some other custom action. That could be done by attaching the following named transformation menu to the reference cell:
CHAPTER 31. TRANSFORMATION MENU LANGUAGE

So we will receive all the standard reference actions as well as our custom action in the completion:

```java
transformation menu ClassCreator_CompletionActions for concept ClassCreator
section(completion) {
    include reference menu for the reference constructorDeclaration
text (editorContext, node, model, targetNode)->string {
    return targetNode.name;
}
visible text (editorContext, node, model, targetNode)->string {
    return targetNode.getPresentationInContext(node.parent);
}
can execute <always>
<< ... >>

action
text (editorContext, node, model, pattern)->string {
    "replace with array creator";
}
can execute <always>
execute (editorContext, node, model, pattern)->void {
    node.replace with new(ArrayCreator);
}
<no additional features>
```

As you can see the reference menu part is the analog of the "primary choose referent menu" part in the "inline menu", but more customizable. Also, if no menu is attached to a cell, its reference menu will be used by default.

31.6.2 Example on properties

One more example - suppose you want to see all the possible values for a specific property in the context assistant when the cursor is on a label cell next to that property cell. Let's take a look at a piece of code in the Kaja language:

```kaja
if looking <no direction> do
    pick turnAround Library call
end else do
    step fetch Library call
    step
end
```

We want to see all the variants of the looking direction when we put
The advantages of these menu parts over the "inline menu" include:

1. They can be attached to any cell, not just reference/property cells.
2. They can be used in any menu location (context assistant, context menu), not only the completion.
3. The reference menu is more customizable (the property menu will also be customizable soon).

The menu discovery algorithm Understanding the process of how MPS picks the transformation menu will help you design menus with more confidence.

The built-in behavior for discovery of transformation menus is to include the menu(s) of superconcept of the current concept, so by default MPS will look for a transform for the current concept, named `<CurrentConcept>_TransformationMenu`, then its super-concept’s menu and so on, until BaseConcept_TransformationMenu.

If BaseConcept_TransformationMenu really exists in MPS and contains an instruction to include the appropriate concept’s substitute menu for the current link. Thanks to this a substitute menu that you define for a concept gets included and becomes available without any need to define a transformation menu explicitly.

Substitute menus are similar to transformation menus, but their discovery works in the opposite direction: where the transformation menu for a concept A includes the menus of its superconcepts (up to BaseConcept), i.e. walks up the hierarchy, the substitute menu for A includes menus for its subconcepts, because only the sub-concepts can safely replace A in the model. Note that substitute menus are a bit different from transformation menus, because they are looked up not based on the concept of an existing node, but instead based on the link’s target concept.

### 31.7 Show Item Trace

To track, which transformation or substitute menu contributed a particular action to the completion menu or to the context assistant, users just press Control/Cmd + Alt + B on the completion menu entry and an interactive trace report will show up. Sometimes it is hard to track how the action appeared in the completion or context assistant because of there are many substitute and transformation menus including each other. Now you may select some action in completion (by arrows) or in the context assistant (by pressing cmd/ctrl+alt+Enter) and then press cmd/ctrl+alt+B. You will see the trace in the project tool. This is the trace of the menu and menu part declaration which include each other starting from the top-level menu and ending with the action declaration. If the menu or the menu part declaration is explicit and is in the project, then it has bold style in tool and you can click on it and go to the declaration.

Here how it looks like when we put the caret on the statement, show completion for the variable reference and then invoke
"Show Item Trace": We see that what we see in completion is the default transformation menu for the Statement which includes the menu for superconcept which is the BaseConcept. It in its turn includes the substitute menu for the Statement, which in its turn wraps the menu for the Expression. Then it comes to subconcepts of Expression, one of which is the VariableReference. VariableReference is the "smart reference" concept, so it tries to find all visible target of the Variable concept. So that's how the variable reference appears in the menu for the statement.
Chapter 32

Extending the Transformation Menu Language

32.1 Overview

The menu language may be extended by adding a new location with location-specific features, or by adding a new menu part type.

32.2 Adding Locations and Features

To add a location, define a concept extending TransformationLocation. Required and optional features are specified using behavior methods:

- Override getAvailableFeatures() to return all feature concepts available for the location (both optional and required).
- Override getRequiredFeatures() to return all feature concepts required for the location.

To add a feature, define a concept extending TransformationFeature. Menu parts that implement IExtensibleMenuPart will have to specify features that are required by the location(s) they are in. Such parts are (generally) generated into a class implementing interface ActionItem and additional interfaces as generated by switch_TransformationLocation_actionItemInterfaces. The features are generated into class members of the generated menu part class.

You should also extend switch_TransformationLocation_asStringArray to specify the identifier(s) that you will use to query items for the location at run time.

32.3 Adding Menu Part Types

Extend the TransformationMenuPart concept and define the structure, editor, etc. of the custom menu part. A menu part may implement two additional interfaces: IParameterizableMenuPart, for parts that support being used inside TransformationMenuPart_Parameterized, and IExtensibleMenuPart, for parts that have location-specific features or additional parameters for the standard features. For example, TransformationMenuPart_Action is both an IParameterizableMenuPart and an IExtensibleMenuPart, while TransformationMenuPart_Intention is only extensible (to provide the wrapped intention as a parameter to its queries) but not parameterizable (since a parameterized intention can be used instead).

Each menu part specified in a section should ultimately be generated into a list of zero or more expressions of type interface MenuPart (placeholder parts are generated into zero menu parts).

To generate code for your menu part you need to extend one or two template switches: switch_TransformationMenuPart_declare and possibly switch_TransformationMenuPart_create.

By default, switch_TransformationMenuPart_declare will declare a (static) inner class and attach the generatedClass mapping label to it. Next, switch_TransformationMenuPart_create generates into a new expression creating an instance of the class specified by generatedClass mapping label.

If the default behavior fits your use case, it is enough to extend only switch_TransformationMenuPart_declare to generate a class implementing MenuPart, attaching the generatedClass mapping label to it. In case of more complex scenarios you may gain more flexibility by extending both switches.
Chapter 33

Intensions

Intensions are a very good example of how MPS enables language authors to smoothen the user experience of people using their language. Intensions provide fast access to the most used operations with syntactical constructions of a language, such as "negate boolean", "invert if condition," etc. If you've ever used IntelliJ IDEA's intensions or similar features of any modern IDEs, you will find MPS intensions very familiar.

33.0.1 Using intensions

Like in IDEA, if there are available intensions applicable to the code at the current position, a light bulb is shown. To view the list of available intensions, press Alt+Enter or click the light bulb. To apply an intension, either click it or select it and press Enter. This will trigger the intension and alter the code accordingly.

Example: list of applicable intensions

33.0.2 Intension types

All intensions are "shortcuts" of a sort, bringing some operations on node structure closer to the user. Two kinds of intensions can be distinguished: regular intensions (possibly with parameters) and "surround with" intensions.

Generally speaking, there is no technical difference between these types of intensions. They only differ in how they are typically used by the user.

**regular intensions** are listed on the intensions list (the light bulb) and they directly perform transformations on a node without asking the user for parameters customizing the operations.

**"surround with" intensions** are used to implement a special kind of transformation - surrounding some node(s) with another construct (e.g. "surround with parenthesis"). These intensions are not offered to the users unless they press ctrl-alt-T (the surround with command) on a node. Neither they are shown in general intensions pop-up menu.

**Universal Intension** is a new experimental feature introduced in 3.4, which allows to unify intensions and parameterized intentions. In addition, it allows to add methods and other class members to intention and has a more java-like editor. As the feature is still in an experimental stage, we decided not to replace the old functionality fully. We still recommend you to use the old intensions, but those, who like the new editor better, can experiment with the new ones. The structure of the universal intension is very similar to the old intensions and using them is very straightforward.

33.0.3 Common Intension Structure
### Regular Intentions

**is error intention** - This flag is responsible for an intention's presentation. It distinguishes two types of intentions - "error" intentions which correct some errors in the code (e.g. a missing 'cast') and "regular" intentions, which are intended to help the user perform some genuine code transformations. To visually distinguish the two types, error intentions are shown with a red bulb, instead of an orange one, and are placed above regular intentions in the applicable intentions list.

**Parameterized regular intentions** - Intentions can sometimes be very close to one another. They may all need to perform the same transformation with a node, just slightly differently. E.g. all "Add ... macro" intentions in the generator ultimately add a macro, but the added macro itself is different for different intentions. This is the case when parameterized intention is needed. Instead of creating separate intentions, you create a single intention and allow for its parametrization. The intention has a parameter function, which returns a list of parameter values. Based on the list, a number of intentions are created, each with a different parameter value. The parameter values can then be accessed in almost every intention's method.

**Note**

You don’t have an access to the parameter in the isApplicable function. This is because of performance reasons. As isApplicable is executed very often and delays would quickly become noticeable by the user, you should perform only base checks in isApplicable. All parameter-dependent checks should be performed in the parameter function, and if a check was not passed, this parameter should not be returned.

### Surround With - Intentions

This type of intentions is very similar to regular intentions and all the mentioned details apply to these intentions as well.

### 33.0.4 Where to store my intentions?

You can create intentions in any model by importing the intentions language. However, MPS collects intentions only from the Intentions language aspects. If you want your intentions to be used by the MPS intentions subsystem, they must be stored in the Intentions aspect of your language.
Chapter 34
Generator

For a quick how-to document on the MPS generator please check out the Generator cookbook.

34.1 Introduction

Generator is a part of language specification that defines the denotational semantics for the concepts in the language. MPS follows the model-to-model transformation approach. The MPS generator specifies translation of constructions encoded in the input language into constructions encoded in the output language. The process of model-to-model transformation may involve many intermediate models and ultimately results in an output model, in which all constructions are in a language whose semantics are already defined elsewhere.

For instance, most concepts in baseLanguage (classes, methods etc) are "machine understandable", therefore baseLanguage is often used as the output language.

The target assets are created by applying model-to-text transformation, which must be supported by the output language. The language aspect to define model-to-text transformation is called TextGen and is available as a separate tab in concept’s editor. MPS provides destructive update of generated assets only.

For instance, baseLanguage’s TextGen aspect generates *.java files at the following location:

<generator output path>\<model name>\<ClassName>.java

where:

- Generator output path - is specified in the module, which owns the input model (see MPS project structure).
- Model name - is a path segment created by replacing ‘.’ with the file separator in the input model’s name.

For a quick how-to document on the MPS generator please check out the Generator cookbook.

34.2 Overview

34.2.1 Generator Module

Unlike any other language aspect, the generator aspect is not a single model. Generator specification can comprise many generator models as well as utility models. A Generator Model contains templates, mapping configurations and other constructions of the generator language.

A Generator Model is distinguished from a regular model by the model stereotype - 'generator' (shown after the model name as <name>@generator).

The screenshot below shows the generator module of the SModel language as an example.
You can research the *smodel* (and any other) language generator by yourself:

- download MPS [here](#);
- create new project (can be empty project);
- use the Go To -> Go to Language command in the main menu to navigate to the smodel language (its full name is `jetbrains.mps.lang.smodel`).

Creating a New Generator

A new generator is created by using the New -> Generator command in the language's popup menu. Technically, it is possible to create more than one generator for one language, but at the time of writing MPS does not provide full support for this feature. Therefore, languages normally have only one (or none) generator. For that reason, the generator’s name is not important. Everywhere in the MPS GUI a generator module can be identified by its language name. When creating a new generator module, MPS will also create the generator model ‘main@generator’ containing an empty mapping configuration node.
Generator Properties

As a module, generator can depend on other modules, have used languages and used devkits (see MPS project structure). The generator properties dialog also has two additional properties:

- **depends on generators** - specifies the dependencies on other generators; this allows the dependent generator to make references to templates defined in another generator;
- **mapping constraints** - priority relationships between mapping rules can be specified. If such a relationship involves other generator rules, then declaring a dependency on that generator is also required. For details on mapping constraints, see Generator, Generator, Generator User Guide Demo6.

Generating Generator

MPS generator engine (or the Generator language runtime) uses mixed compilation/interpretation mode for transformation execution. Templates are interpreted and filled at runtime, but all functions in rules, macros, and scripts must be pre-compiled.

To avoid any confusion, always follow this rule: after any changes made to the generator model, the model must be re-generated (Shift+F9). Even better is to use Ctrl+F9, which will re-generate all modified models in the generator module.

34.2.2 Transformation

The transformation is described by means of templates. Templates are written using the output language and so can be edited with the same cell editor that would normally be used to write ‘regular code’ in that language. Therefore, without any additional effort the ‘template editor’ has the same level of tooling support right away - syntax/error highlighting, auto-completion, etc. The templates are then parametrized by referencing into the input model. The applicability of individual templates is defined by Generator Overview, which are grouped into Generator Overview.

Mapping Configurations  A Mapping Configuration is a minimal unit, which can form a single generation step. It contains Generator Overview, defines mapping labels and may include pre- and post-processing scripts.

Generator Rules  Applicability of each transformation is defined by generator rules. There are six types of generator rules:

- conditional root rule
- root mapping rule
- weaving rule
- reduction rule
- pattern rule
- abandon root rule
- drop attribute rule (new in 3.3)

Each generator rule consists of a premise and a consequence (except for the abandon root rule and drop attribute rule, with predefined consequence that cannot be specified by the user).

All rules except for the conditional root rule contain a reference to the concept of the input node (or just input concept) in its premises. All rule premises also contain an optional condition function.

Rule consequence commonly contains a reference to an external template (i.e. a template declared as a root node in the same or different model) or so-called in-line template (conditional root rule and root mapping rule can only have reference to an external template). There are also several other versions of consequences.
The following screenshot shows the contents of a generator model and a mapping configuration example.

Macros  
The code in templates can be parameterized through macros. The generator language defines three kinds of macros:

- **property macro** - computes a property value;
- **reference macro** - computes the target (node) of a reference;
- **node macro** - is used to control template filling at generation time. There are several versions of node macro - LOOP-macro is an example.

Macros implement a special kind of so-called Structure and can wrap property, reference or node cells (depending on the kind of macro) in the template code.

Code wrapping (i.e. the creation of a new macro) is done by pressing Ctrl+Shift+M or by applying the 'Create macro' intention.

The following screenshot shows an example of a property macro.

Macro functions and other parameterization options are edited in the inspector view. Property macro, for instance, requires specifying the value function, which will provide the value of the property at generation time. In the example above, output class node will get the same name that the input node has.

The node parameter in all functions of the generator language always represents the context node to which the transformation is currently being applied (the input node).

Some macros (such as LOOP and SWITCH-macro) can replace the input node with a new one, so that subsequent template code (i.e. code that is wrapped by those macros) will be applied to the new input node.
External Templates  External templates are created as a root node in the generator model. There are two kinds of external templates in MPS. One of them is root template. Any root node created in generator model is treated as a root template unless this node is a part of the generator language (i.e. mapping configuration is not a root template). Root template is created as a normal root node (via Create Root Node menu in the model’s popup). The following screenshot shows an example of a root template.

This root template will transform input node (a Document) into a class (baseLanguage). The root template header is added automatically upon creation, but the concept of the input node is specified by the user.

It is a good practice to specify the input concept, because this allows MPS to perform static type checking in the code of the macro function.

A Root template (reference) can be used as a consequence in conditional root rules and root mapping rules. When used in a conditional root rule, the input node is not available.

The second kind of template is defined in the generator language and its concept name is 'TemplateDeclaration'. It is created via the 'template declaration' action in the Create Root Node menu.

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

**Template declaration** can have parameters, declared in the header. Parameters are accessible through the Generator Overview.

**Template declaration** is used in consequence of weaving, reduction and pattern rules. It is also used as an included template in INCLUDE-macro (only for templates without parameters) or as a callee in CALL-macro.

**Template Switches** A template switch is used when two or more alternative transformations are possible in a certain place in template code. In that case, the template code that allows alternatives is wrapped in a SWITCH-macro, which has reference to a Template Switch. Template Switch is created as a root node in the generator model via the Create Root Node menu (this command can be seen in the 'menu' screenshot above).

The following screenshot shows an example of a template switch.
34.3 Generator Language Reference

34.3.1 Mapping Configuration

Mapping Configuration is a container for generator rules, mapping label declarations and references to pre- and post-processing scripts. A generator model can contain any number of mapping configurations - all of them will be involved in the generation process, if the owning generator module is involved. Mapping configuration also serves as a minimal generator unit that can be referenced in the mapping priority rules (see Generator).

34.3.2 Generator Rule

Generator Rule specifies a transformation of an input node to an output node (except for the conditional root rule which doesn’t have an input node and simply creates a new node in the output model). All rules consist of two parts - a premise and a consequence (except for the abandon root rule, which doesn’t have a consequence and simply ignores the input node). Any generator rule can be tagged by a mapping label.

All generator rules’ functions have the following parameters:

- **node** - the current input node (all except the condition-function in conditional root rule)
- **genContext** - generation context - allows searching for output nodes, generating of unique names and others (see Generator Language)

Generator Rules:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
<th>Premise</th>
<th>Consequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>conditional root rule</td>
<td>Generates a root node in the output model. Applied only one time (max) during a single generation step.</td>
<td>condition function (optional), missing condition function is equivalent to a function always returning true.</td>
<td>root template (ref)</td>
</tr>
</tbody>
</table>
| root mapping rule        | Generates a root node in the output model.                                  | concept - applicable concept (concept of the input node)
  inheritors - if true then the rule is applicable to the specified concept and all its sub-concepts. If false (default) then the sub-concepts are not applicable.
  condition function (optional) - see conditional root rule above.
  keep input root - if false then the input root node (if it’s a root node) will be dropped. If true then input root will be copied to the output model. | root template (ref)  |
| weaving rule             | Allows to insert additional child nodes into the output model. Weaving rules are processed at the end of a generation micro-step just before map_src and reference resolving. The rule is applied on each input node of the specified concept. The parent node for insertion should be provided by the context function. (see Generator Language) | concept - same as above
  inheritors - same as above
  condition function (optional) - same as above | • external template (ref)
  • weave-each
  • context - function that computes the (parent) output node into which the output node(s) generated by this rule will be inserted.
  • anchor (available in Inspector) - specifies a node within the context collection, in front of which the nodes should be inserted, null means insert at the end of the collection |
<table>
<thead>
<tr>
<th>Consequence</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>root template (ref)</td>
<td>• conditional root rule  • (root) mapping rule</td>
<td>Applies the root template</td>
</tr>
<tr>
<td>external template (ref)</td>
<td>• weaving rule          • reduction rule</td>
<td>Applies an external template. Parameters should be passed if required, they can be one of:</td>
</tr>
<tr>
<td></td>
<td>• pattern rule</td>
<td>- pattern captured variable (starting with # sign)</td>
</tr>
<tr>
<td></td>
<td>• integer or string literal</td>
<td>- null, true, false</td>
</tr>
<tr>
<td></td>
<td>• query function</td>
<td></td>
</tr>
<tr>
<td>weave-each</td>
<td>weaving rule</td>
<td>Applies an external template to a set of input nodes. Weave-each consequence consists of:</td>
</tr>
<tr>
<td></td>
<td>• foreach function - returns a sequence of input nodes</td>
<td>- foreach function - returns a sequence of input nodes</td>
</tr>
<tr>
<td></td>
<td>• reference on an external template</td>
<td>- reference on an external template</td>
</tr>
</tbody>
</table>
### 34.3.3 Root Template

Root Template is used in conditional root rules and (root) mapping rules. The Generator language doesn’t define specific concept for root template. The generator language defines a special kind of annotation - root template header, which is automatically added to each new root template. The root template header is used for specifying an expected input concept (i.e. concept of input node). MPS uses this setting to perform a static type checking of code in various macro-functions, which are used in the root template.

| Technically, any root node in the output language can be considered to be a Root Template, when created in the generator model. However, root nodes without annotations are treated by the generator as auxiliary, utility nodes e.g. to specify reference targets and thus are not considered for target language evaluation. A warning is displayed and a quick fix is offered to add the root template header annotation to the root template that miss one. We highly recommend language designers to fix their templates and to always create new root templates with the annotation included. |

### 34.3.4 External Template

External Template is a concept defined in the generator language. It is used in weaving rules and reduction rules. In external templates the user specifies the template name, input concept, parameters and a content node. The content node can be any node in the output language. The actual template code in external templates is surrounded by template fragment ‘tags’ (the template fragment is also a special kind of annotation concept). The code outside template fragment serves as a framework (or context) for the real template code (template fragment) and is ignored by the generator. In external template for weaving rule, the template’s context node is required (it is a design-time representation of the rule’s context node), while the template for reduction rule can be just one context-free template fragment. An external template for a reduction rule must contain exactly one template fragment, while a weaving rule’s template can contain more than one template fragments. Template fragment has a mapping label property, which is edited in Inspector view.

### 34.3.5 Mapping Label

Mapping Labels are declared in a mapping configuration and references stored to this declaration are used to label generator rules, macros and template fragments. Such marks allow finding of an output node by a known input node (see Generator Language).

Properties:

- name
• input concept (optional) - expected concept of the input node of the transformation performed by the tagged rule, macro or template fragment

• output concept (optional) - expected concept of the output node of the transformation performed by the tagged rule, macro or template fragment

MPS makes use of the input/output concept settings to perform static type checking in `get output ...` operations (see Generator Language).

### 34.3.6 Export Label

Export Labels are declared in mapping configuration. Export labels resemble mapping labels in many ways. They add a persistence mechanism that enables access to the labels from other models.

Each export label specifies:

• a **name** to identify it in the macros

• **input and output concepts** indicating the concept before and after the generation phase

• a **keeper** concept, instance of which will be used for storing the exported information

• a **marshal** function, to encode the `inputNode` and the generated `outputNode` into the keeper

• an **unmarshal** function, to decode the information using the original `inputNode` and the keeper to correctly initialize the `outputNode` in the referring model

### 34.3.7 Macro

Macro is a special kind of an annotation concept which can be attached to any node in template code. Macro brings dynamic aspect into otherwise static template-based model transformation.

Property- and reference-macros are attached to property- and reference-cells respectively, while node-macros (which come in many variations - LOOP, IF etc.) are attached to cells representing the whole node in the cell-editor. All properties of a macro are edited using the inspector view.

All macros have the **mapping label** property - reference to a mapping label declaration. Additionally, all macros can be parameterized by various macro-functions - depending on the type of the macro. Any macro-function has at least the three following parameters:

• node - the current input node;

• genContext - generation context - allows searching for output nodes, generating of unique names and others;

• operationContext - instance of `jetbrains.mps.smodel.IOperationContext` interface (used rarely).

Many types of macros have the **mapped node** (or **mapped nodes**) function. This function computes the new input node - a substitution for the current input node. If the **mapped node** function returns null or the **mapped nodes** function returns an empty sequence, then the generator will skip this macro altogether. i.e. no output will be generated in this place.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
<th>Properties (if not mentioned above)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property macro</td>
<td>Computes the value of a property.</td>
<td>value function:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• return type - string, boolean or int - depending on the property type.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• parameters - standard + <code>templateValue</code> - value in the template code wrapped by the macro.</td>
</tr>
<tr>
<td>Macro</td>
<td>Description</td>
<td>Function/Argument</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Reference macro | Computes the referent node in the output model. Normally executed at the end of a generation micro-step, when the output model (tree) is already constructed. Can also be executed earlier, if the user code is trying to obtain the target of the reference. Reference macro supports SNodeReference as specification of a new target. With that, templates don’t need access to a target’s node model the moment they are applied. | referent function:
- return type
  - node (type depends on the reference link declaration)
  - SNodeReference
  - string identifying the target node (see Generator Language).
| IF      | The wrapped template code is applied only if the condition is true. Otherwise the template code is ignored and an ‘alternative consequence’ (if any) is applied.                                               | condition function
alternative consequence (optional) - any of:
- external template (ref)
- in-line template
- abandon input
- dismiss top rule |
| LOOP    | Computes new input nodes and applies the wrapped template to each of them.                                                                                                                                   | mapped nodes function                                                             |
| INCLUDE | The wrapped template code is ignored (it only serves as an anchor for the INCLUDE-macro), a reusable external template will be used instead. Null input makes INCLUDE effectively a no-op. | mapped node function (optional)  
include template - reference to a reusable external template |
| CALL    | Invokes template and replaces wrapped template code with the result of template invocation. Supports templates with parameters. Null input node is tolerated, and the template is ignored altogether in this case, i.e. CALL yields empty collection of nodes as a result when input/mapped node is null. | mapped node function (optional)  
call template - reference to a reusable external template  
argument - one of
- pattern captured variable
- integer or string literal
- null, true, false
- query function |
| SWITCH  | Provides a way to many alternative transformations in the given place in the template code. The wrapped template code is applied, if none of switch cases is applicable and no default consequence is specified in Generator Language. For null input node, SWITCH may react with a message (specified along with its rules), anchor template node is ignored, and SWITCH macro yields no results. | mapped node function (optional)  
template switch - reference to a template switch |
| COPY-SRC | Copies an input node to the output model. The wrapped template code is ignored.                                                                                                                                 | mapped node function - computes the input node to be copied.                     |
| COPY-SRCL| Copies input nodes to the output model. The wrapped template code is ignored. Can be used only for children with multiple aggregation cardinality.                                                               | mapped nodes function - computes the input nodes to be copied.                  |
### Chapter 34. Generator

#### MAP-SRC

Multifunctional macro, can be used for:

- marking a template code with a mapping label;
- replacing the current input node with a new one;
- perform a non-template based transformation;
- accessing the output node for some reason.

The MAP-SRC macro is executed at the end of a generator micro-step - after all node- and property-macros, but before any reference-macro is run.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parentOutputNode</td>
<td>parent node in the output model.</td>
</tr>
</tbody>
</table>

#### MAP-SRCL

Same as MAP-SRC but can handle many new input nodes (similar to the LOOP-macro)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>outputNode</td>
<td>give access to the output node.</td>
</tr>
</tbody>
</table>

#### WEAVE

Allows to insert additional child nodes into the output model in a similar way Weaving rules are used. The node wrapped in the WEAVE macro (or provided by the use input function) will have the supplied template applied to it and the generated nodes will be inserted.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>use input</td>
<td>a function returning a collection of nodes to apply the macro to</td>
</tr>
<tr>
<td>weave</td>
<td>reference to a template to weave into the nodes supplied as the input</td>
</tr>
</tbody>
</table>

#### EXPORT

Saves a node for cross-model reference, so it can be retrieved when generating other models

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parentOutputNode</td>
<td>parent node in the output model.</td>
</tr>
</tbody>
</table>

**Note**

Reference resolving by identifier is only supported in BaseLanguage. The identifier string for classes and class constructors may require (if class is not in the same output model) package name in square brackets preceding the class name: `{package.name}ClassName`

### 34.3.8 Template Switch

A template switch is used in pair with the SWITCH-macro (the TemplateSwitchMacro concept). A single template switch can be re-used in many different SWITCH-macros. A template switch consists of set of cases and one default case. Each switch case is a reduction rule, i.e. a template switch contains a list of reduction rules (see Generator Language).

The **default case** consequence can be one of:

- external template (ref)
- in-line template
- abandon input
- dismiss top rule
  .. or can be omitted. In this case the template code surrounded by corresponding SWITCH-macro will be applied.

A template switch can inherit reduction rules from other switches via the `extends` property. When the generator is executing a SWITCH-macro it tries to find most specific template switch (available in scope). Therefore the actually executed template switch is not necessarily the one that is defined in the `template switch` property of the SWITCH-macro.

Through the `null-input message` property the user can specify an `error`, `warning` or `info` message, which will be shown in the MPS messages view in case when the `mapped node` function in SWITCH-macro returns null (by default no messages are shown and the macro is skipped altogether).

A template switch can accept parameters, the same way as template declarations. A use of parametrized switch mandates arguments to be supplied in the `SWITCH` macro. The TemplateSwitchMacro concept supports switches both with and without arguments.
The old macro concept (TemplateSwitch) has been deprecated in 3.1. Note, visually both macros look the same, *SWITCH* and *SWITCH*, respectively. There’s migration script to replace old macro instances with the new one; you need to invoke the script manually to update the concepts.

### 34.3.9 Generation Context (operations)

Generation context (the genContext parameter in macro- and rule-functions) allows finding of nodes in the output model, generating unique names and provides other useful functionality.

Generation context can be used not only in the generator models, but also in utility models - as a variable of type gencontext. Operations of genContext are invoked using the familiar dot-notation: genContext.operation

#### Finding Output Node

- **get output <mapping label> for model ( <model> )**
  - Returns the output node generated by a labeled conditional root rule in a specified model.
  - Issues an error, if there is more than one matching output node.

- **get output <mapping label> for ( <input node> )**
  - Returns the output node generated from the input node by a labeled generator rule, a macro or a template fragment.
  - Issues an error if there is more than one matching output node.

- **pick output <mapping label> for ( <input node> )**
  - only used in the context of the referent function in a reference-macro and only if the required output node is a target of the reference, which is being resolved by that reference-macro.
  - Returns the output node generated from the input node by a labeled generator rule, a macro or a template fragment. The difference between this and the previous operation is that this operation can automatically resolve the many-output-nodes conflict - it picks the output node, which is visible in the given context (see Constraints).

- **get output list <mapping label> for ( <input node> )**
  - Returns a list of output nodes generated from the input node by a labeled generator rule, a macro or a template fragment.

- **get copied output for ( <input node> )**
  - Returns the output node, which has been created by copying of an input node. If during the copying, the input node has been reduced, but the concept of the output node is the same (i.e. it wasn’t reduced into something totally different), then this is still considered ‘copying’.
  - Issues an error if there is more than one matching output node.

#### Generating Unique Name

- **unique name from <base name> in context <node>**
  - The uniqueness is secured throughout the whole generation session.
  - Clashing with names that weren’t generated using this service is still possible.
  - The context node is optional, though we recommend to specify it to guarantee generation stability. 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 the input model or in the generator model), this won’t affect other names outside the scope.

#### Template Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#patternvar</td>
<td>Value of the captured pattern variable</td>
</tr>
<tr>
<td>param</td>
<td>Value of the template parameter</td>
</tr>
</tbody>
</table>

#### Getting Contextual Info

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputModel</td>
<td>The current input model</td>
</tr>
<tr>
<td>originalModel</td>
<td>The original input model</td>
</tr>
<tr>
<td>outputModel</td>
<td>The current output model</td>
</tr>
</tbody>
</table>
### Invocation Context

Operation context (`jetbrains.mps.smodel.IOperationContext` Java interface) associated with the module - the owner of the original input model.

*This operation has been deprecated in MPS 3.3 and will be removed in the next release along with other activities to eliminate IOperationContext.*

### Scope

Scope - `jetbrains.mps.smodel.IScope` Java interface.

*This operation has been deprecated since MPS 3.1 and is removed in MPS 3.3.*

### Template Node

The template code surrounded by the macro.

*This operation has been deprecated in MPS 3.3.*

The primary flaw is that this operation implies interpreted templates. There is no template model when templates are generated.

Besides, contract of the operation is vague (i.e. what does it give in a context of argument query for a template call).

*get prev input `<mapping label>`*

Returns the input node that has been used for enclosing the template code surrounded by the labeled macro.

*It is only used in macro-functions.*

### Transferring User Data

During generation MPS maintains three maps of user objects, each of which has different life span:

- **Session Object** - kept throughout the whole generation session;
- **Step Object** - kept through a single generation step;
- **Transient Object** - only alive during a micro step.

The developer can access the user object maps using the array (square brackets) notation:

- `session object [ <key> ]`
- `step object [ <key> ]`
- `transient object [ <key> ]`

The key can be any object (`java.lang.Object`).

*Binding user data with particular node*

The session- and step-object cannot be used to pass data associated with a particular input node across steps and micro-steps, because neither an input node nor its id can serve as a key (output nodes always have a different id).

To pass such data you should use the `putUserObject` and `getUserObject` methods defined in class `jetbrains.mps.smodel.SNode`.

The data will be transferred to all output copies of the input node. The data will be also transferred to the output node, if a slight reduction (i.e. without changing of the node concept) took place while the node was being copied.

### Logging

- `show info <message text> -> <node>`
- `show error <message text> -> <node>`
- `show warning <message text> -> <node>`

Creates message in the MPS message view. If the `node` parameter is specified then clicking on the message will navigate to that node. In case of an `error` message, MPS will also output some additional diagnostic info.

### Utilities (Re-usable Code)

If you have duplicated code (in rules, macros, etc.) and want to say, extract it to re-usable static methods, then you must create this class in a separate, non-generator model.

If you create an utility class in the generator model (i.e. in a model with the `generator` stereotype), then it will be treated as a root template (unused) and no code will be generated from it.
### 34.3.10 Mapping Script

A Mapping script is user code, which is executed either before a model transformation (pre-processing script) or after it (post-processing script). It should be referenced from Generator Language to be invoked as a part of it’s generation step. Mapping scripts provide the ability to perform non-template based model transformations.

Pre-processing scripts are also commonly used for collecting certain information from input model that can be later used in the course of template-based transformation. The information collected by script is saved as a transient-, step- or session-object (see Generator).

Script sample:

```plaintext
mapping script myPreProcessingScript

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

(model, genContext, operationContext)->void {
    for (node<MyConcept> te : model.nodes(MyConcept)) {
        genContext.transient object [ MyConceptUtil.KEY_ID ] = ....;
    }
}
```

Properties:

| script kind                  | • pre-process input model - the script is executed at the beginning of a generation step, before the template-based transformation;  
|                             | • post-process output model - the script is executed at the end of a generation step, after the template-based transformation. |
| modifies model               | only available if script kind = pre-process input model  
|                             | If set true and the input model is the original input model, then MPS will create a transient input model before applying the script.  
|                             | If set false but the script tries to modify the input model, then MPS will issue an error. |

Code context:

<table>
<thead>
<tr>
<th>model</th>
<th>The current model</th>
</tr>
</thead>
<tbody>
<tr>
<td>genContext</td>
<td>The generation context to give access to transient/session or step objects.</td>
</tr>
</tbody>
</table>
| invocation context         | Operation context (jetbrains.mps.smodel.IOperationContext java interface) associated with the module - the owner of the original input model  
|                           | Operation context has been deprecated will be removed in the next release, please don’t use. |

### 34.4 The Generator Algorithm

The process of generation of target assets from an input model (generation session) includes 5 stages:

- Defining all generators that must be involved
- Defining the order of priorities of transformations
- Step-by-step model transformation
- Generating text and saving it to a file (for each root in output model)
- Post-processing assets: compiling, etc.

We will discuss the first three stages of this process in detail.

### 34.4.1 Defining the Generators Involved

To define the required generators, MPS examines the input model and determines which languages are used in it. Doing this job MPS doesn’t make use of ‘Used Languages’ specified in the model properties dialog. Instead MPS examines each node in the model and gathers languages that are actually used.
From each 'used language' MPS obtains its generator module. If there are more than one generator module in a language, MPS chooses the first one (multiple generators for the same language are not fully supported in the current version of MPS). If any generator in this list depends on other generators (as specified in the 'depends on generators' property), then those generators are added to the list as well.

After MPS obtains the initial list of generators, it begins to scan the generator's templates in order to determine what languages will be used in intermediate (transient) models. The languages detected this way are handled in the same manner as the languages used in the original input model. This procedure is repeated until no more 'used languages' can be detected.

**Explicit Engagement**

In some rare cases, MPS is unable to detect the language whose generator must be involved in the model transformation. This may happen if that language is not used in the input model or in the template code of other (detected) languages. In this case, you can explicitly specify the generator engagement via the Languages Engaged on Generation section in the input model’s properties dialog (Advanced tab).

### 34.4.2 Dependency scope/kind - 'Generation Target' and 'Design'.

'Generation Target' replaces 'Extends' relation between two languages (L2 extends L1), when one needed to specify that Generator of L2 generates into L1 and thus needs its runtime dependencies. Now, when a language (L2) is translated to another language (L1), and L1 has runtime dependencies, use L1 as 'Generation Target' of L2. Though this approach is much better than 'Extends', it's still not perfect as it’s rather an attribute of a generator than of a language. Once Generators become fully independent from their languages, we might need to fix this approach (different generators may target different languages, thus target has to be specified for a generator, not the source language).

'Design' dependency replaces 'Extends' between two generators. Use it when you need to reference another generator to specify priority rules (though consider if you indeed need these priorities, see changes in the Generator Plan, below).

### 34.4.3 Defining the Order of Priorities

As we discussed The Generator Algorithm, a generator module contains generator models, and generator models contain mapping configurations. Mapping configuration (mapping for short) is a set of generator rules. It is often required that some mappings must be applied before (or not later than, or together with) some other mappings. The language developer specifies such a relationship between mappings by means of mapping constraints in the generator properties dialog (see also The Generator Algorithm and the Generator User Guide Demo6 demo).

After MPS builds the list of involved generators, it divides all mappings into groups, according to the mapping priorities specified. All mappings for which no priority has been specified fall into the last (least-priority) group.

You can check the mapping partitioning for any (input) model by selecting Show Generation Plan action in the model’s popup menu. The result of partitioning will be shown in the MPS Output View.

### 34.4.4 Optimized Generation Plan

When planning the generation phase, MPS prefers to keep every generator as lonely as possible. Eventually, you’ll see many relatively small and fast to process generation steps. Of course, the generators forced to run together with priority rules still run at the same step. Handling several unrelated generators at the same generation step (MPS prior to 3.2) proved to be inefficient, since it imposed a lot of unnecessary checking for rule applicability across other generators from the same step. With in-place transformation in 3.2 and later, the performance penalty for each extra generation steps is negligible.

### 34.4.5 Ignored priority rules

In addition to conflicting priorities, there are rules that get ignored during the generation plan. This might happen if an input model doesn’t have any concept of a language participating in a priority rule. Since there’s no actual use of a language, the rule is ignored, and the 'Show Generation Plan' action reports them along with conflicting rules. Previous MPS versions used to include generators of otherwise unused languages into the generation process, now these generators get no chance to jump in.

### 34.4.6 Implicit priorities

Target languages (languages produced by templates) are considered as implicit 'not later than' rules. You don’t need to specify these priorities manually. MPS automatically inserts "not later than" rules for all generator models in the source and target languages. It is important to understand that priority rules work on the model granularity level.
This implicit priority rule between two generator models is ignored if an explicit priority rule is defined for these two models, one from the language that generates into the other language and one from the other language.

34.4.7 Model Transformation

Each group of mappings is applied in a separate generation step. The entire generation session consists of as many generation steps as there were mapping groups formed during the mapping partitioning. A generation step includes three phases:

- Executing pre-mapping scripts
- Template-based model transformation
- Executing post-mapping scripts

The template-based model transformation phase consists of one or more micro-steps. A micro-step is a single-pass model transformation of an input model into a transient (output) model.

While executing micro-step MPS follows the next procedure:

1. Apply conditional root rules (only once - on the 1-st micro-step)
2. Apply root mapping rules
3. Copy input roots for which no explicit root mapping is specified (this can be overridden by means of the 'keep input root' option in root mapping rules and by the 'abandon root' rules)
4. Apply weaving rules
5. Apply delayed mappings (from MAP_SRC macro)
6. Revalidate references in the output model (all reference-macro are executed here)

There is no separate stage for the application of reduction and pattern rules. Instead, every time MPS copies an input node into the output model, it attempts to find an applicable reduction (or pattern) rule. MPS performs the node copying when it is either copying a root node or executing a COPY_SRC-macro. Therefore, the reduction can occur at either stage of the model transformation.

MPS uses the same rule set (mapping group) for all micro-steps within the generation step. After a micro-step is completed and some transformations have taken place during its execution, MPS starts the next micro-step and passes the output model of the previous micro-step as input to the next micro-step. The whole generation step is considered completed if no transformations have occurred during the execution of the last micro-step, that is, when there are no more rules in the current rule set that are applicable to nodes in the current input model.

The next generation step (if any) will receive the output model of previous generation step as its input.

Intermediate models (transient models) that are the output/input of generation steps and micro-steps are normally destroyed immediately after their transformation to the next model is completed. To keep transient models, enable the following option:

Settings -> Generator Settings -> Save transient models on generation

See also:

- Generator User Guide Demo6
- Generator User Guide Demo6

34.4.8 Handling of node attributes during generation

Node attributes constitute generic extension mechanism, thus Generator shall preserve attributes along the transformation process (unless attribute designer opts not to keep them) without explicit support in any template. When an input node is transformed to another node, Generator copies attributes of the input node to the output. Copy is controlled by newly introduced drop attribute rule only, and happens regardless of @attribute info specification (i.e. its attributed concept or multiple restrictions). The fact that transformation rule may have produced attribute node itself is not taken into account (i.e. if a reduction rule explicitly copies node attributes to a newly created output node, the attributes would get duplicated due to automatic copy of attributes. However, it’s rare for a reduction rule to copy node attributes, and the issue, if ever shows up, is easy to mitigate with drop rules).

While copying attributes of a node, Generator consults drop attribute rules (newly introduced in MPS 3.3, reside next to abandon root rules) to see if language designer don’t need these attributes to survive transformation process. This rules are quite similar to abandon root rules - when any rule is triggered, attribute is not copied into output model.

With the growing adoption of attributes and their increasing complexity, we enabled the generator to transform the attribute contents using the regular template processing rules:
returns to nodes in the same model get updated to point to the respective nodes in the output model

reduction rules are applied in order to transform children of the attribute node.

### 34.4.9 In-place transformation

Generators for languages employed in a model are applied sequentially (aka *Generation Plan*). Effectively, each generation step modifies just a fraction of original model, and the rest of the model is copied as-is. With huge models and numerous generation steps this approach proves to be quite ineffective. In-place transformation tries to address this with a well-known 'delta' approach, where changes only are collected and applied to original model to alter it in-place.

In version 3.1 in-place transformation is left as an option, enabled by default and configurable through the *Project settings* -> *Generator*. Clients are encouraged to fix their templates that fail in the in-place mode, as in-place generation is likely to become the only generation mode later down the road.

Use of in-place transformation brings certain limitations or might even break patterns that used to work in the previous MPS versions:

- Most notable and important - there’s no output model at the moment when rule’s queries/conditions are executed. To consult the output model during the transformation process is a bad practice, and in-place transformation enforces removing it. Access to the output model from a transformation rule implies certain order of execution, thus effectively limiting the set of optimizations applicable by the MPS generator. The contract of a transformation rule, with a complete input and a fraction of the output that this particular rule is responsible for, is more rigorous than "a complete input model" and "an output model in some uncertain state".

- The output model is indeed there for weaving rules, as their purpose is to deal with output nodes.

- The process of delta building requires the generator to know about the nodes being added to the model. Thus, any implicit changes in the output model that used to work would fail with in-place generation enabled. As an example, consider *MAP-SRC* with a post-process function, which replaces the node with a new one: 
  ```java
  postprocess: if (node.someCondition()) node.replace with new(AnotherNode);
  ```
  Generator records a new node produced by *MAP-SRC*, schedules it for addition, and delays post-processing. Once post-processing is over, there’s no way for the generator to figure out the node it tracks as ‘addition to output model’ is no longer valid and there’s another node which should be used instead. Of course, the post-process can safely alter anything below the node produced by *MAP-SRC*, but an attempt to go out from the sandbox of the node would lead to an error.

- Presence of active weaving rules prevents in-place transformation as these rule require both input and output models.

### 34.4.10 Generation trace

Much like in-place transformation, the updated generation trace is inspired by the idea to track actual changes only. Now it’s much less demanding, as only the transformed nodes are being tracked. Besides, various presentation options are available to give different perspective on the transformation process.

### 34.4.11 Support for non-reflective queries

*Note: This is just a preview of incomplete functionality in 3.1*

Queries in the generator end up in the *QueriesGenerated* class, with methods implementing individual queries. These methods are invoked through Java Reflection. This approach has certain limitations - extra effort is required to ensure consistency of method name and arguments in generated code and hand-written invocation code. Provisional API and a generation option has been added, to expose functionality of *QueriesGenerated* through a set of interfaces. With that, generator consults generated queries through regular Java calls, with compile-time checks for arguments, leaving implementation detail about naming and arguments of particular generated queries to *QueriesGenerated* and its generator.

### 34.5 Mapping Priorities

Mapping priorities are a set of mapping priority rules specifying an order of priority between sets of generator rules (mapping configurations).

Mapping priorities are specified in generator module property dialog.

See also: *Generator Properties*, *Generation Process: Defining the Order of Priorities*, *Demo 6: Dividing Generation Process into Steps*.

Each mapping priority rule consists of left part, right part and priority symbol in the middle.

For instance:

```plaintext
* < [sample.language/generator: *]
```

Left part of priority rule can only refer to mapping configurations in this generator.
### 34.6 Generation plan

Generation plans allow developers to specify the desired order of generation for their models explicitly and thus gain better control over the generation process.

#### 34.6.1 Motivation

Specifying mutual generator priorities may become cumbersome for larger projects. Additionally, in order to specify the priorities the involved languages need to know about one another by declaring appropriate mutual dependencies, which breaks their (sometimes desired) independence. Generation plans put the responsibility of proper ordering of generation steps into a single place - the generation plan. They allow language designers to provide intuitive means for end-user models to be processed in a desired order. Generation plans list all the languages that should be included in the generation process, order them appropriately and optionally specify checkpoints, at which the generator shall preserve the current transient models. These models can then be used for automatic cross-model reference resolution further down the generation process.

The mechanism described here is a preliminary implementation that is likely to be further evolved in the following MPS versions. The general direction is:

- to give more flexibility to the language designers in how they organize builds for users of their languages
- to relieve the end-developers (aka language users) from the need to handle directly build plans, genplan models or build facets

So the actual end-user mechanism is likely to become more dependent on the language designer’s intentions. The approach described here, based on facet and a genplan model, is a mere sample of how to accomplish custom generation.
34.6.2 Defining a generation plan

In order to create a generation plan, you first need to create a model. You may consider giving the model a genplan stereotype to easily distinguish it from ordinary models, but this is not mandatory.

After importing the jetbrains.mps.lang.generator.plan and jetbrains.mps.lang.smodel languages, you can create root node of the Plan concept, which will represent your generation plan:

The generation plan consists of transforms and checkpoints.

It is also possible to specify the required generators explicitly.
Transforms represent generation steps and include languages that should be generated as part of that generation phase. Apply represents an explicit invocation of a particular generator. The `apply with extended` statement applies in a single step the specified generators and those that extend them. This allows the language designer to accommodate for possible extensions.

Checkpoint models represent points during the generation, at which the intermediate models should be preserved. References that will be resolved later in the generation will be able to look-up nodes in the stored intermediate models for their resolution through mapping labels. You can view these checkpoint models in the Project View tool window.
These intermediate checkpoint models are preserved until you shutdown MPS or until you rebuild the models or the models that they depend on. Alternatively you can remove them manually:

Checkpoints provide synchronization points between different plans. The checkpoint models are denoted with a stereotype that matches the name of a checkpoint the model has been created with. Models are persisted alongside the generated sources using the naming scheme of <plan-name>-<checkpoint name>.

Distinct statements allow for capturing different aspects of a checkpoint:

- **declare checkpoint <name>** statement - specify a label that generator plans could share among themselves. This statement does not record/persist the state of the transformed model, it is a mere declaration that other generation plans will be able to refer to.

- **checkpoint <checkpoint>** - records/persists the state of the transformed model. It can either declare a checkpoint in-place or refer to a declared checkpoint.

- **synchronize with <checkpoint>** statements - instructs the generation plan to look up the target nodes in persisted models of the specified checkpoint, but do not persist its own nodes (read-only access to the check-point). This statement doesn’t introduce any new state, but references a checkpoint declared elsewhere.
34.6.3 Specifying a generation plan for models

Modules that should have their models built following a generation plan need to enable the **Custom generation** facet and point to the actual desired generation plan.
34.6.4 Verifying the generation plan

The *Show generation plan* action in the models’ pop-up menu will correctly take the *generation plan* into account when building the outline for the generation:
To view the original, on generator priorities based, generation plan that would be used without the explicit generation plan script, hold the Alt key while clicking the Show Generation Plan menu entry:
Note that the report states in the header that it is not the currently active plan.

34.6.5 Using DevKits to associate a generation plan

DevKits can associate a generation plan, as well.

First add dependencies on languages and solutions that the DevKit should be wrapping. Then specify the Generation plan from within the imported solutions, which will be associated with the DevKit. Any model that imports that DevKit will get the DevKit’s associated generation plan applied to it.
Only one DevKit with a plan is allowed per model at the moment.

### 34.6.6 Cross-model generation

**Model** is the unit of generation in MPS. All entities in a single model are generated together and references between the nodes can be resolved with **reference macros** and **mapping labels**. Mapping labels, however, are not by default accessible from other models. This complicates generation of references that refer to nodes from other models. Fortunately, regular **mapping labels** can support mutually independent generation of models with cross-references, if the models share a **generation plan**. The mechanism leverages checkpoints to capture the intermediate transient models and then use them for reference resolution.

In essence, to preserve cross-model references when generating multiple models, make sure that your models share a **generation plan**. That **generation plan** must define checkpoints at moments, when the **mapping labels** that are used for cross-model reference resolution have been populated. The rest will be taken care of automatically. The reference macros can resolve nodes from **mapping labels** through the usual `genContext.get` output by label and input (for nodes generated through reduction or root mapping rules) or `genContext.get` output for model (for nodes generated through conditional root mapping rules) ways.

### 34.6.7 Linking checkpoint models

Models created at checkpoints now keep a reference to the previous checkpoint model in the sequence. This helps the Generator discover mapped nodes matching input that spans several generator phases.

### 34.6.8 Debug information in the checkpoint models

To ease debug of cross-model generation scenarios, a dedicated root inside each **checkpoint model** lists the mapping label names along with pointers to the stored input and output nodes. Investigation of the mapping labels exposed at each checkpoint can substantially help debugging cross-model generation scenarios and fix unresolved references. Thus, next time your cross-model reference doesn’t resolve, inspect corresponding checkpoint model to see if there’s indeed a label for your input.

A **video** on setting a generation plan for a solution as well as for a DevKit is available.

### 34.7 Generating language descriptor models

Generation plans have been enhanced to generate descriptor models for languages (known as `<language.name>@descriptor`). The structure, textgen, typesystem, dataflow and constraints aspects are now generated with generation plans and they use the new cross-model reference resolution mechanism. Custom aspects defined by language authors can join the generation plan, as well. If you got a custom aspect, you should make sure that its generator extends the generator of `jetbrains.mps.lang.descriptor` language, as this is the way to get custom extensions activated for the plan.
34.8 Generating from Ant

The Ant MPS generator task exposes properties configurable from the build script (parallel, threads, in-place, warnings). The build language uses the Ant Generate task under the hood to transform models during the build process. This task now exposes parameters familiar from the Generator settings page:

- strict generation mode
- parallel generation with configurable number of threads
- option to enable in-place transformation
- option to control generation warnings/errors.

These options are also exposed at build language with the BuildMps_GeneratorOptions concept, so that build scripts have more control over the process.

34.9 Examples

If you’re feeling like it’s time for more practical experience, check out the Generator Demos. The demos contain examples of usage of all the concepts discussed above.
35.1 TextGen language aspect

35.1.1 Introduction

The TextGen language aspect defines a model to text transformation. It comes in handy each time you need to convert your models into the text form directly. The language contains constructs to print out text, transform nodes into text values and give the output some reasonable layout.

35.1.2 Operations

The *append* command performs the transformation and adds resulting text to the output. You can use *found error* command to report problems in the model. The *with indent* command demarcates blocks with increased indentation. Alternatively, the *increase depth* and *decrease depth* commands manipulate the current indentation depth without being limited to a block structure. The *indent buffer* command applies the current indentation (as specified by *with indent* or *increase/decrease depth*) for the current line.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>any number of:</td>
</tr>
<tr>
<td></td>
<td>• {string value}, to insert use the &quot; char, or pick constant from the completion menu</td>
</tr>
<tr>
<td></td>
<td>• \n</td>
</tr>
<tr>
<td></td>
<td>• $list{node.list}$ - list without separator</td>
</tr>
<tr>
<td></td>
<td>• $list{node.list with ,}$ - with separator (intentions to add/remove a separator are available)</td>
</tr>
<tr>
<td></td>
<td>• $ref{node.reference}$, e.g. $ref{node.reference&lt;target&gt;}$ - deprecated and will be removed</td>
</tr>
<tr>
<td></td>
<td>• ${node.child}$</td>
</tr>
<tr>
<td></td>
<td>• ${attributed node}$ - available in attribute nodes, delegates to the attributed node</td>
</tr>
<tr>
<td>found error</td>
<td>error text</td>
</tr>
<tr>
<td>decrease depth</td>
<td>decrease indentation level from now onwards</td>
</tr>
<tr>
<td>increase depth</td>
<td>increase indentation level from now on</td>
</tr>
<tr>
<td>indent buffer</td>
<td>apply indentation to the current line</td>
</tr>
<tr>
<td>with indent { &lt;code&gt; }</td>
<td>increase indentation level for the &lt;code&gt;</td>
</tr>
</tbody>
</table>

The parameters to the *append* command may have the *with indent* flag in the Inspector tool window set to *true* to get prepended with the current indentation buffer.

35.1.3 Indentation

Proper indentation is easy to get right once you understand the underlying principle. TextGen flushes the AST into text. The TextGen commands simply manipulate sequentially the output buffer and output some text to it, one node at a time.
A variable holding the current depth of indentation (**indentation buffer**) is preserved for each root concept. **Indentation buffer** starts at zero and is changed by **increase/decrease depth** and **with indent** commands. The "indentation", however, must be inserted into the output stream explicitly by the **append** commands. Simply marking a block with **with indent** will not automatically indent the text generated by the wrapped TextGen code. The **with indent** block only increases the value of the **indentation buffer**, but the individual **appends** may or may not wish to be prepended with the **indentation buffer** of the current size.

There are two ways to explicitly insert **indentation buffer** into the output stream:

- **indent buffer** command
- **with indent** flag in the inspector for the parameters of the **append** command

For example, to properly indent **Constants** in a list of constants, we call **indent buffer** at the beginning of each emitted line. This ensures that the indentation is inserted only at the beginning of each line.

```plaintext
for each constant:
    indent buffer;
    append $constant.name {=} $constant.initializer
```

Alternatively, we could specify the **with indent** flag in the inspector for the first parameter to the **append** command. This will also insert the indentation only at the beginning of each line.

### 35.1.4 Root concepts

TextGen provides two types of root concepts:

- **text gen component**, represented by the **ConceptTextGenDeclaration** concept, which encodes a transformation of a concept into text. For **rootable** concepts the target file can also be specified.

- **base text gen component**, represented by the **LanguageTextGenDeclaration** concept, which allows definition of reusable textgen operations and utility methods. These can be called from other text gen components of the same language as well as extending languages.

### 35.1.5 TextGen in extended concepts

MPS does not create files for root concept automatically. Even sub-concepts of a concept that has TextGen defined will have no file created automatically. Only exact concept matches are considered. If an extending concept desires to re-use textgen component of an ancestor as is, it shall declare its own **empty** TextGen component, stating the essentials as the file name, encoding and extension, and leaving the body of the component empty.
35.1.6 Layout

There’s provisional mechanism to control layout of output files. The text layout section of ConceptTextGenDeclaration (available only in rootable concepts) allows the authors to define multiple logical sections (with a default one) and then optionally specify for each append, to which section to append the text.

Text generation is not always possible in a sequence that corresponds to lines in a physical file. E.g. for a Java source, one could distinguish 2 distinct areas, e.g. imports and class body, where imports is populated along with the body. A passionate language designer might want to break up the file further, e.g. up to file comment, package statement, imports, and class body that consists of fields and methods, and populate each one independently while traversing a ClassConcept. That’s what we call a layout of an output file, and that’s what we give control over now. MPS veterans might be aware of two buffers (TOP and BOTTOM) that used to be available in TextGen for years. These were predefined, hard-coded values. Now it’s up to language designer to designate areas of an output file and their order.

Note, distinct areas come handy especially when generating text from attributes, as they change order of execution. With them, it’s even more tricky to make sure flow of text gen corresponds to physical text lines, and designated areas make generation a lot more comfortable.

Layout of the file could be specified for a top text gen, the one that produces files.

The support for this mechanism is preliminary and is quite rudimentary now. We utilize it in our BaseLanguage implementation, so this notice is to explain you what’s going on rather than encourage you to put this into production.

35.1.7 Context objects

It is vital for certain model-to-text conversion scenarios to preserve some context information during TextGen. In BaseLanguage, for example, TextGen has to track model imports and qualified class names. The cumbersome and low-level approach of previous versions based on direct text buffer manipulation has been replaced with the possibility to define and use customized objects as part of the concept’s textgen specification.
At the moment, regular Java class (with a no-arg or a single-arg constructor that takes concept instance) are supported as context objects. You reference context objects from code as a regular variable.

### 35.1.8 Handling attributes in TextGen

When nodes are annotated with Attributes, the TextGen for these attributes is processed first. The `$\{attributed node\}$ construct within the attribute’s TextGen will then insert the TextGen of the attributes node itself.

If there are multiple attributes on a single node, they are processed in turn, starting with the last-assigned (top-most) attribute. Attributes without TextGen associated are ignored and skipped.

The top-most attribute is technically the last in the containment (the way editor depicts the attributes is **visually** different from the order one may notice in Node Explorer). I.e. A node with attributes A1 and A2 in the editor looks like A2(A1(N)), and TextGen processes A2 first, then A1, then N. When TextGen is asked to generate text for N, it looks up the last attribute in the containment that has textgen defined (if any), and delegates to it. Let’s assume A2 has TextGen, and A1 not. Then, if the attribute’s component has `$\{attributedNode\}$`, TextGen would check the previous attributes for associated textgen. If there are none (and in our sample A1 has none), text generation for the actual node N is the last to receive control.

### 35.1.9 Examples

Here is an example of the text gen component for the `ForeachStatement` (**jetbrains.mps.baseLanguage**).

```java
text gen component for concept ForeachStatement {
  (node, context, buffer)->void {
    if (node.loopLabel != null) {
      append \n $\{node.loopLabel.name\}$ {:} ;
    } else if (node.label != null) {
      append \n $\{node.label\}$ {:} ;
    }
    append \n ;
    indent buffer ;
    append {for () $\{node.variable\}$ { : } $\{node.iterable\}$ {} } ;
    with indent {
      append $\{node.body\}$ ;
    }
    append \n {}} ;
  }
}
```

This is an artificial example of the text gen:
text gen component for concept CodeBlockConcept {
  (node, context, buffer)->void {
    indent buffer ;
    append {codeBlock {} \n ;
    with indent {
      indent buffer ;
      append {// Begin of codeBlock} \n ;

      indent buffer ;
      append {int i = 0} \n ;

      indent buffer ;
      append {// End of codeBlock} \n ;
    }
    append {}} ;

  }
}

producing following code block containing a number of lines with indentation:

```
text gen component for concept CodeBlockConcept {
  codeBlock {
    // Begin of codeBlock
    int i = 0
    // End of codeBlock
  }
}
```

An example of **TextGen** for attribute that adds extra text to output of attributed node:

```
text gen component for concept MethodDocComment {
  (context, buffer, node)->void {
    append doc comment start node;
    append $list{node.tags};
    append doc comment end node;

    append ${ attributed node }$;
  }
}
```
Chapter 36

Data flow

36.1 Data Flow

A language’s data flow aspect allows you to find unreachable statements, detect unused assignments, check whether a variable might not be initialized before it’s read, and so on. It also allows performing some code transformations, for example the ‘extract method’ refactoring.

Most users of data flow analyses aren’t interested in details of its inner working, but are interested in getting the results they need. They want to know which of their statements are unreachable, and what can be read before it’s initialized. In order to shield a user from the complexities of these analyses, we provide assembly-like intermediate language into which you translate your program. After translation, this intermediate presentation is analyzed and a user can find which of the statements of original language are unreachable etc.

For example here is the translation of a ‘for’ loop from baseLanguage:

```java
for (Object o : list) {
    return o;
}
```

First, we emit a label so we can jump after it. Then we perform a conditional jump after the current node. Then we emit code for node.variable. Finally, we emit code for the loop’s body, and jump to the previously emitted label.

36.1.1 Commands of intermediate language

Here are the commands of our intermediate language:

- read x - reads a variable x
- write x - writes to variable x
- jump before node - jumps before node
- jump after node - jumps after node
- jump label - jumps to label
- ifjump ((before|after)node) | label - conditional jump before/after node / to label
- code for node - insert code for node
- ret - returns from current subroutine

36.1.2 May be unreachable

Some commands shouldn’t be highlighted as unreachable. For example we might want to write some code like this:

```java
for (Object o : list) {
    return o;
}
```
If you generate data flow intermediate code for this statement, the last command: the `jump after condition` command will be unreachable. On the other hand, it’s a legal base language statement, so we want to ignore this command during reachable statement analysis. To do so we mark it as `may be unreachable`, which is indicated by curly braces around it. You can toggle this settings with the appropriate `intention`.

```plaintext
data flow builder for ForEachStatement { (node)->void { 
  code for node.inputSequence
  label condition
  if jump after node
  write node.variable = <unknown>
  code for node.body
  } jump after label condition
}

(data flow builder for ForEachStatement { (node)->void { 
  code for node.inputSequence
  label condition
  if jump after node
  write node.variable = <unknown>
  code for node.body
  } jump after label condition }
```

You may like to try our Dataflow, as well.

### 36.1.3 Links:

Chapter 37

Typesystem

37.1 Defining A Typesystem For Your Language

This page describes the MPS type-system to a great detail. If you would prefer a more lightweight introduction into defining your first type-system rules, consider checking out the Cookbook - Type System. If you would like to get familiar with the ways you can use the type-system from your code, you may also look at the Using_typesystem chapter.

37.2 What is a typesystem?

A typesystem is a part of a language definition assigning types to the nodes in the models written using the language. The typesystem language is also used to check certain constraints on nodes and their types. Information about types of nodes is useful for:

- finding type errors
- checking conditions on nodes’ types during generation to apply only appropriate generator rules
- providing information required for certain refactorings (e.g. for the "extract variable" refactoring)
- and more

37.3 Types

Any MPS node may serve as a type. To enable MPS to assign types to nodes of your language, you should create a language aspect for typesystem. The typesystem model for your language will be written in the typesystem language.

37.4 Inference Rules

The main concept of the typesystem language is an inference rule. An inference rule for a certain concept is mainly responsible for computing a type for instances of that concept. An inference rule consists of a condition and a body. A condition is used to determine whether a rule is applicable to a certain node. A condition may be of two kinds: a concept reference or a pattern. A rule with a condition in the form of concept reference is applicable to every instance of that concept and its subconcepts. A rule with a pattern is applicable to nodes that match the pattern. A node matches a pattern if it has the same properties and references as the pattern, and if its children match the pattern's children. A pattern also may contain several variables which match everything. The body of an inference rule is a list of statements which are executed when a rule is applied to a node. The main kind of statements of typesystem language are statements used for creating equations and inequations between types. An inference rule may define the overrides block, which is a boolean flag that tells the typechecker that, in case there are other inference rules applicable to the superconcepts of the concept specified in the condition, this inference rule takes precedence, and all the rules for superconcepts are ignored. The version 3.3 brings possibility to use a code block instead of a static flag.

Starting with the version 3.3, the inference rules that are applicable to instances of node attributes have additional features that allow for overriding or amending the rules applied to the attributed node. This, for example, can be used to implement alternate type inference based on presence conditions, which can take into account the parameters specified at the project or system level.
In case an inference rule is applicable to a node attribute, there is also a possibility to tell the typechecker that this rule supercedes the rules applicable to the attributed node, which are then ignored. Also, the attributed node is accessible in all rule's code blocks as attributedNode.

### 37.4.1 Inference Methods

To avoid duplications, one may want to extract identical parts of code of several inference rules to a method. An inference method is just a simple Base Language method marked with an annotation "@InferenceMethod". There are several language constructions you may use only inside inference rules and replacement rules and inference methods, they are: typeof expressions, equations and inequations, when concrete statements, type variable declarations and type variable references, and invocations of inference methods. That is made for not to use such constructions in arbitrary methods, which may be called in arbitrary context, maybe not during type checking.

### 37.4.2 Overriding

A type-system rule of a sub-concept can override the rules defined on the super-concepts. If the overrides flag is set to false, the rule is added to the list of rules applied to a concept together with the rules defined for the super-concepts, while, if the flag is set to true, the overriding rule replaces the rules of the super-concepts in the rule engine and so they do not take effect. This applies both to Inference and NonTypeSystem rules.

### 37.5 Equations And Inequations

The main process which is performed by the type-system engine is the process of solving equations and inequations among all the types. A language designer tells the engine, which equations it should solve by writing them in inference rules. To add an equation into an engine, the following statement is used:

```plaintext
expr1 ::= expr2, where expr1 and expr2 are expressions, which evaluate to a node.
```

Consider the following use case. You want to say that the type of a local variable reference is equal to the type of the variable declaration it points to. So, you write `typeof (varRef) ::= typeof (varRef.localVariableDeclaration)`, and that's all. The typesystem engine will solve such equations automatically.

The above-mentioned expression `typeof(expr)` (where `expr` must evaluate to an MPS node) is a language construct, which returns a so-called type variable, which serves as a type of that node. Type variables become concrete types gradually during the process of equation solving.

In certain situations you want to say that a certain type doesn’t have to exactly equal another type, but also may be a subtype or a supertype of that type. For instance, the type of the actual parameter of a method call does not necessarily have to be the same type as that of the method’s formal parameter - it can be its subtype. For example, a method, which requires an Object as a parameter, may be applied also to a String.

To express such a constraint, you may use an inequation instead of an equation. An inequation expresses the fact that a certain type should be a subtype of another type. It is written as follows: `expr1 ::= expr2`.

### 37.6 Weak And Strong Subtyping

A relationship of subtyping is useful for several different cases. You want a type of an actual parameter to be a subtype of a formal parameter type, or you want a type of an assigned value to be a subtype of variable’s declared type; in method calls or field access operations you want a type of an operand to be a subtype of a method’s declaring class.

Sometimes such demands are somewhat controversial: consider, for instance, two types, `int` and `Integer`, which you want to be interchangeable when you pass parameters of such types to a method: if a method is `doSomething(int i)`, it is legal to call `doSomething(1)` as well as `doSomething(new Integer(1))`. But when these types are used as types for operand of, say, a method call, the situation is the different: you shouldn’t be able to call a method of an expression of type `int`, of an integer constant for example. So, we have to conclude that in one sense `int` and `Integer` are subtypes of one another, while in the other sense they are not.

For solving such a controversy, we introduce two relationships of subtyping: namely, weak and strong subtyping. Weak subtyping will follow from strong subtyping: if a node is a strong subtype of another node, then it is it’s weak subtype also.

Then, we can say about our example, that `int` and `Integer` are weak subtypes of each other, but they are not strong subtypes. Assignment and parameters passing require weak subtyping only, method calls require strong subtyping.

When you create an inequation in your typesystem, you may choose it to be a strong or weak inequation. Also subtyping rules, those which state subtyping relationship (see below), can be either weak or strong. A weak inequation looks like ` ::= ;`. A strong inequation looks like ` ::= ;`

In most cases you want to state strong subtyping, and to check weak subtyping. If you are not sure, which subtyping you need, use weak one for inequations, strong one for subtyping rules.
37.7 Subtyping Rules

When the typesystem engine solves inequations, it requires information about whether a type is a subtype of another type. But how does the typesystem engine know about that? It uses subtyping rules. Subtyping rules are used to express subtyping relationship between types. In fact, a subtyping rule is a function which, given a type, returns its immediate supertypes.

A subtyping rule consists of a condition (which can be either a concept reference or a pattern) and a body, which is a list of statements that compute and return a node or a list of nodes that are immediate supertypes of the given node. When checking whether some type A is a supertype of another type B, the typesystem engine applies subtyping rules to B and computes its immediate supertypes, then applies subtyping rules to those supertypes and so on. If type A is among the computed supertypes of type B, the answer is "yes".

By default, subtyping stated by subtyping rules is a strong one. If you want to state only weak subtyping, set "is weak" property of a rule to "true".

37.8 Comparison Inequations And Comparison Rules

Consider you want to write a rule for \textit{EqualsExpression} (operator $==$ in Java, BaseLanguage and some other languages): you want left operand and right operand of \textit{EqualsExpression} to be comparable, that is either type of a left operand should be a (non-strict) subtype of a right operand, or vice versa. To express this, you write a comparison inequation, in a form \texttt{expr1 :\neg : expr2}, where \texttt{expr1} and \texttt{expr2} are expressions, which represent types. Such an inequation is fulfilled if \texttt{expr1} is a subtype of \texttt{expr2} (\texttt{expr1 <: expr2}), or \texttt{expr2} is a supertype of \texttt{expr1}.

Then, consider that, say, any Java interfaces should also be comparable, even if such interfaces are not subtypes of one another. That is because there always can be written a class, which implements both of interfaces, so variables of interface types can contain the same node, and variable of an interface type can be cast to any other interface. hence an \textit{equation}, \textit{cast}, or \textit{instanceof} expressions with both types being interface types should be legal (and, for example, in Java they are). To state such a comparability, which does not stem from subtyping relationships, you should use comparison rules. A comparison rule consists of two conditions for the two applicable types and a body which returns true if the types are comparable or false if they are not.

Here's the comparison rule for interface types:

```plaintext
comparison rule interfaces_are_comparable
applicable for concept = ClassifierType as classifierType1 , concept = ClassifierType as classifierType2
applicable always
overrides false
rule {
  if (classifierType1.classifier.isInstanceOf(Interface) &&
      classifierType2.classifier.isInstanceOf(Interface)) {
    return true;
  } else {
    return false;
  }
}
```

37.9 Quotations

A quotation is a language construct that lets you easily create a node with a required structure. Of course, you can create a node using the \texttt{smodelLanguage} and then populate it with appropriate children, properties and references by hand, using the same \texttt{smodelLanguage}. However, there's a simpler - and more visual - way to accomplish this.

A quotation is an expression, whose value is the MPS node written inside the quotation. Think about a quotation as a "node literal", a construction similar to numeric constants and string literals. That is, you write a literal if you statically know what value do you mean. So inside a quotation you don't write an expression, which evaluates to a node, you rather write the node itself. For instance, an expression $2 + 3$ evaluates to 5, an expression $< 2 + 3 >$ (angled braces being quotation braces) evaluates to a node \textit{PlusExpression} with \texttt{leftOperand} being an \texttt{IntegerConstant 3} and \texttt{rightOperand} being \texttt{IntegerConstant 5}.

(See the Quotations documentation for more details on quotations, anti quotations and light quotations)

37.10 Antiquotations

For it is a literal, a value of quotation should be known statically. On the other hand, in cases when you know some parts (i.e. children, referents or properties) of your node only dynamically, i.e. those parts that can only be evaluated at runtime
and are not known at design time, then you can’t use just a quotation to create a node with such parts. The good news, however, is that if you know the most part of a node statically and you want to replace only several parts by dynamically-evaluated nodes you can use antiquotations. An antiquotation can be of 4 types: child, reference, property and list antiquotation. They all contain an expression, which evaluates dynamically to replace a part of the quoted node by its result. Child and referent antiquotations evaluate to a node, property antiquotation evaluates to string and list antiquotation evaluates to a list of nodes.

For instance, you want to create a ClassifierType with the class ArrayList, but its type parameter is known only dynamically, for instance by calling a method, say, "computeMyTypeParameter()". Thus, you write the following expression: `< ArrayList < %( computeMyTypeParameter() )% > >`. The construction `%(...)%` here is a node antiquotation.

You may also antiquotate reference targets and property values, with `ˆ(...)ˆ` and `$(...)$`, respectively; or a list of children of one role, using `*(...)*`. a) If you want to replace a node somewhere inside a quoted node with a node evaluated by an expression, you use node antiquotation, that is `%(...)%`. As you may guess there’s no sense to replace the whole quoted node with an antiquotation with an expression inside, because in such cases you could instead write such an expression directly in your program. So node antiquotations are used to replace children, grandchildren, great-grandchildren and other descendants of a quoted node. Thus, an expression inside of antiquotation should return a node. To write such an antiquotation, position your caret on a cell for a child and type "%".

b) If you want to replace a target of a reference from somewhere inside a quoted node with a node evaluated by an expression, you use reference antiquotation, that is `ˆ(...)ˆ`. To write such an antiquotation, position your caret on a cell for a referent and type `ˆ`.

c) If you want to replace a child (or some more deeply located descendant), which is of a multiple-cardinality role, and if for that reason you may want to replace it not with a single node but rather with several ones, then use child list (simply list for brevity) antiquotations, `*(...)*`. An expression inside a list antiquotation should return a list of nodes, that is of type `**nlist<..>` or compatible type (i.e. `**`{list<nod<..>` is ok, too, as well as some others). To write such an antiquotation, position your caret on a cell for a child inside a child collection and type "*". You cannot use it on an empty child collection, so before you press "*" you have to enter a single child inside it.

d) If you want to replace a property value of a quoted node by a dynamically calculated value, use property antiquotation `$(...)$`. An expression inside a quotation should return string, which will be a value for an antiquoted property of a quoted node. To write such an antiquotation, position your caret on a cell for a property and type "$".

(See the Quotations documentation for more details on quotations, anti quotations and light quotations)

### 37.11 Examples Of Inference Rules

Here are the simplest basic use cases of an inference rule:

- to assign the same type to all instances of a concept (useful mainly for literals):

```plaintext
applicable to concept = StringLiteral as nodeToCheck
{
  typeof (nodeToCheck) ::=: < String >
}
```

- to equate a type of a declaration and the references to it (for example, for variables and their usages):

```plaintext
applicable to concept = VariableReference as nodeToCheck
{
  typeof (nodeToCheck) ::=: typeof (nodeToCheck.variableDeclaration)
}
```

- to give a type to a node with a type annotation (for example, type of a variable declaration):

```plaintext
applicable to concept = VariableDeclaration as nodeToCheck
{
  typeof (nodeToCheck) ::=: nodeToCheck.type
}
```

- to establish a restriction for a type of a certain node: useful for actual parameters of a method, an initializer of a type variable, the right-hand part of an assignment, etc.
CHAPTER 37. TYPESYSTEM

37.12 Type Variables

Inside the typesystem engine during type evaluation, a type may be either a concrete type (a node) or a so-called type variable. Also, it may be a node which contains some type variables as its children or further descendants. A type variable represents an undefined type, which may then become a concrete type, as a result of solving equations that contain this type variable.

Type variables appear at runtime mainly as a result of the "typeof" operation, but you can create them manually, if you want to. There's a statement called TypeVarDeclaration in the typesystem language to do so. You write it like "var T" or "var X" or "var V", i.e. "var" followed by the name of a type variable. Then you may use your variable, for example, in antiquotations to create a node with type variables inside.

Example: an inference rule for "for each" loop. A "for each" loop in Java consists of a loop body, an iterable to iterate over, and a variable into which the next member of an iterable is assigned before the next iteration. An iterable should be either an instance of a subclass of the Iterable interface, or an array. To simplify the example, we don't consider the case of the iterable being an array. Therefore, we need to express the following: an iterable’s type should be a subtype of an Iterable of something, and the variable’s type should be a supertype of that very something. For instance, you can write the following:

```java
for (String s : new ArrayList<String>(...)) {
    ...
}
```

or the following:

```java
for (Object o : new ArrayList<String>(...)) {
    ...
}
```

Iterables in both examples above have the type ArrayList<String>, which is a subtype of Iterable<String>. Variables have types String and Object, respectively, both of which are subtypes of String.

As we see, an iterable’s type should be a subtype of an Iterable of something, and the variable’s type should be a supertype of that very something. But how to say “that very something" in the typesystem language? The answer is, it’s a type variable that we use to express the link between the type of an iterable and the type of a variable. So we write the following inference rule:

```java
applicable to concept = AssignmentExpression as nodeToCheck
{
    typeof (nodeToCheck.rValue) :<=: typeof (nodeToCheck.lValue)
}
```

37.13 Meet and Join types

Meet and Join types are special types, which are treated differently by the typesystem engine. Technically Meet and Join types are instances of MeetType and JoinType concepts, respectively. They may have an arbitrary number of argument types, which could be any nodes. Semantically, a Join type is a type, which is a supertype of all its arguments, and a node which has a type Join(T1|T2|..Tn) can be regarded as if it had type T1 or type T2 or... or type Tn. A Meet type is a type, which is a subtype of its every argument, so one can say that a node, which has a type Meet(T1&T2&..&Tn) inhabits type T1 and type T2 and.. and type Tn. The separators of arguments of the Join and Meet types (i.e. "|" and "&") are chosen respectively to serve as a mnemonics.

Meet and Join types are very useful at certain situations. Meet types appear even in MPS BaseLanguage (which is very close to Java). For instance, the type of such an expression:

```java
true ? new Integer(1) : "hello"
```

is Meet(Serializable & Comparable), because both Integer (the type of new Integer(1)) and String (the type of "hello") implement both Serializable and Comparable.

Join type is useful when, say, you want some function-like concept return values of two different types (node or list of nodes, for instance). Then you should make type of its invocation be Join(node<> | list<node<}).
You can create Meet and Join types by yourself, if you need to. Use quotations to create them, just as with other types and other nodes. The concepts of Meet and Join types are MeetType and JoinType, as it is said above.

### 37.14 "When Concrete" Blocks

Sometimes you may want not only to write equations and inequations for a certain types, but to perform some complex analysis with type structure. That is, inspect inner structure of a concrete type: its children, children of children, referents, etc.

It may seem that one just may write typeof(some expression), and then analyze this type. The problem is, however, that one can't just inspect a result of typeof expression because it may be a type variable at that moment. Although a type variable usually will become a concrete type at some moment, it can't be guaranteed that it is concrete in some given point of your typesystem code.

To solve such a problem you can use a "when concrete" block.

```plaintext
when concrete ( expr as var ) {
    body
}
```

Here, *expr* is an expression which will evaluate to a mere type you want to inspect (not to a node type of which you want to inspect), and *var* is a variable to which an expression will be assigned. Then this variable may be used inside a body of "when concrete" block. A body is a list of statements which will be executed only when a type denoted by "expr" becomes concrete, thus inside the body of a when concrete block you may safely inspect its children, properties, etc. if you need to. If you have written a when concrete block and look into its inspector you will see two options: "is shallow" and "skip error". If you set "is shallow" to "true", the body of your when concrete block will be executed when expression becomes shallowly concrete, i.e. becomes not a type variable itself but possibly has type variables as children or referents. Normally, if your expression in condition of when concrete block is never concrete, then an error is reported. If it is normal for a type denoted by your expression to never become a concrete type, you can disable such error reporting by setting "skip error" to true.

### 37.15 Overloaded Operators

Sometimes an operator (like +, -, etc.) has different semantics when applied to different values. For example, + in Java means addition when applied to numbers, and it means string concatenation if one of its operands is of type String. When the semantics of an operator depends on the types of its operands, it's called operator overloading. In fact, we have many different operators denoted by the same syntactic construction.

Let's try to write an inference rule for a plus expression. First, we should inspect the types of operands, because if we don't know the types of operands (whether they are numbers or Strings), we cannot choose the type for an operation (it will be either a number or a String). To be sure that types of operands are concrete we'll surround our code with two when concrete blocks, one for left operand's type and another one for right operand's type.

```plaintext
when concrete(typeof(plusExpression.leftExpression) as leftType) {
    when concrete(typeof(plusExpression.rightExpression) as rightType) {
       ...
    }
}
```

Then, we can write some inspections, where we check whether our types are strings or numbers and choose an appropriate type of operation. But there will be a problem here: if someone writes an extension of BaseLanguage, where they want to use the plus expression for addition of some other entities, say, matrices or dates, they won't be able to use plus expression because types for plus expression are hard-coded in the already existing inference rule. So, we need an extension point to allow language-developers to overload existing binary operations.

Typesystem language has such an extension point. It consists of:

- overloading operation rules and
- a construct which provides a type of operation by operation and types of its operands.

For instance, a rule for PlusExpression in BaseLanguage is written as follows:
when concrete(typeof(plusExpression.leftExpression) as leftType) {
    when concrete(typeof(plusExpression.rightExpression) as rightType) {
        node<> opType = operation type(plusExpression, leftType, rightType);
        if (opType.isNotNull) {
            typeof(plusExpression) == opType;
        } else {
            error "+ can’t be applied to these operands" -> plusExpression;
        }
    }
}

Here, "operation type" is a construct which provides a type of an operation according to types of operation’s left operand’s type, right operand’s type and the operation itself. For such a purpose it uses overloading operation rules.

Overloaded Operation Rules

Overloaded operation rules reside within a root node of concept OverloadedOpRulesContainer. Each overloaded operation rule consists of:

- an applicable operation concept, i.e. a reference to a concept of operation to which a rule is applicable (e.g. PlusExpression);
- left and right operand type restrictions, which contain a type which restricts a type of left/right operand, respectively. A restriction can be either exact or not, which means that a type of an operand should be exactly a type in a restriction (if the restriction is exact), or its subtype (if not exact), for a rule to be applicable to such operand types;
- a function itself, which returns a type of the operation knowing the operation concept and the left and right operand types.

Here’s an example of one of overloading operation rules for PlusExpression in BaseLanguage:

```
operation concept: PlusExpression
left operand type: <Numeric>.descriptor is exact: false
right operand type: <Numeric>.descriptor is exact: false
operation type:
<operation, leftOperandType, rightOperandType> -> node<> {
    if (leftOperandType.isInstanceOf(NullType) || rightOperandType.isInstanceOf(NullType)) {
        return null;
    } else {
        return Queries.getBinaryOperationType(leftOperandType, rightOperandType);
    }
}
```

37.16 Replacement Rules

37.16.1 Motivation

Consider the following use case: you have types for functions in your language, e.g. \((a_1, a_2, \ldots, a_N) \rightarrow r\), where \(a_1, a_2, \ldots, a_N\), and \(r\) are types: \(a_K\) is a type of \(K\)-th function argument and \(r\) is a type of a result of a function. Then you want to say that your function types are covariant by their return types and contravariant by their argument types. That is, a function type \(F = (T_1, \ldots, T_N) \rightarrow R\) is a subtype of a function type \(G = (S_1, \ldots, S_N) \rightarrow Q\) (written as \(F <: G\)) if and only if \(R <: Q\) (covariant by return type) and for any \(K\) from 1 to \(N\), \(T_K :> S_K\) (that is, contravariant by arguments types).

The problem is, how to express covariance and contravariance in the typesystem language? Using subtyping rules you may express covariance by writing something like this:

```
nlist < > result = new nlist < > ;
for ( node < > returnTypeSupertype : immediateSupertypes (functionType . returnType ) ) {
    node <FunctionType> ft = functionType . copy;
    ft . returnType = returnTypeSupertype;
    result . add ( ft ) ;
}
return result ;
```

Okay, we have collected all immediate supertypes for a function’s return type and have created a list of function types with those collected types as return types and with original argument types. But, first, if we have many supertypes of return
type, it's not very efficient to perform such an action each time we need to solve an inequation, and second, although now we have covariance by function's return type, we still don't have contravariance by function's arguments' types. We can't collect immediate subtypes of a certain type because subtyping rules give us supertypes, not subtypes.

In fact, we just want to express the abovementioned property: \( F = (T_1, \ldots, T_N) \rightarrow R <: G = (S_1, \ldots, S_N) \rightarrow Q \) (written as \( F <: G \)) if and only if \( R <: Q \) and for any \( K \) from 1 to \( N \), \( T_K :> S_K \). For this and similar purposes the typesystem language has a notion called "replacement rule."

### 37.16.2 What's a replacement rule?

A replacement rule provides a convenient way to solve inequations. While the standard way is to transitively apply subtyping rules to a should-be-subtype until a should-be-supertype is found among the results (or is never found among the results), a replacement rule, if applicable to an inequation, removes the inequation and then executes its body (which usually contains "create equation" and "create inequation" statements).

### 37.16.3 Examples

A replacement rule for above-mentioned example is written as follows:

```plaintext
replacement rule FunctionType_subtypeOf_FunctionType
applicable for concept = FunctionType as functionSubType <: concept = FunctionType as functionSuperType
rule {
  if ( functionSubType . parameterType . count != functionSuperType . parameterType . count ) {
    error " different parameter numbers " -> equationInfo . getNodeWithError ( ) ;
    return ;
  }
  functionSubType . returnType :<=: functionSuperType . returnType ;
  foreach ( node < > paramType1 : functionSubType . parameterType ;
           node < > paramType2 : functionSuperType . parameterType ) {
    paramType2 :<=: paramType1 ;
  }
}
```

Here we say that a rule is applicable to a should-be-subtype of concept FunctionType and a should-be-supertype of concept FunctionType. The body of a rule ensures that the number of parameter types of function types are equal, otherwise it reports an error and returns. If the numbers of parameter types of both function types are equal, a rule creates an inequation with return types and appropriate inequation for appropriate parameter types.

Another simple example of replacement rules usage is a rule, which states that a Null type (a type of null literal) is a subtype of every type except primitive ones. Of course, we can't write a subtyping rule for Null type, which returns a list of all types. Instead, we write the following replacement rule:

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

This rule is applicable to any should-be-supertype and to those should-be-subtypes which are Null types. The only thing this rule does is checking whether should-be-supertype is an instance of PrimitiveType concept. If it is, the rule reports an error. If is not, the rule does nothing, therefore the inequation to solve is simply removed from the typesystem engine with no further effects.

### 37.16.4 Different semantics

A semantic of a replacement rule, as explained above, is to replace an inequation with some other equations and inequations or to perform some other actions when applied. This semantic really doesn't state that a certain type is a subtype of another type under some conditions. It just defines how to solve an inequation with those two types.

For example, suppose that during generation you need to inspect whether some statically unknown type is a subtype of String. What will an engine answer when a type to inspect is Null type? When we have an inequation, a replacement rule can say that it is true, but for this case its abovementioned semantics is unuseful: we have no inequations, we have a
question to answer yes or no. With function types, it is worse because a rule says that we should create some inequations. So, what do we have to do with them in our use case?

To make replacement rules usable when we want to inspect whether a type is a subtype of another type, a different semantic is given to replacement rules in such a case.

This semantic is as follows: each "add equation" statement is treated as an inspection of whether two nodes match; each "add inequation" statement is treated as an inspection of whether one node is a subtype of another; each report error statement is treated as "return false."

Consider the above replacement rule for function types:

```plaintext
replacement rule FunctionType_subtypeOf_FunctionType

applicable for concept = FunctionType as functionSubType <: concept = FunctionType as functionSuperType

rule {
    if (functionSubType . parameterType . count != functionSuperType . parameterType . count) {
        error "different parameter numbers" -> equationInfo . getNodeWithError ( ) ;
        return ;
    }
    functionSubType . returnType : <=: functionSuperType . returnType ;
    foreach (node <> paramType1 : functionSubType . parameterType ; node <> paramType2 : functionSuperType . parameterType ) {
        paramType2 : <=: paramType1 ;
    }
}
```

In a different semantic, it will be treated as follows:

```plaintext
boolean result = true;
if (functionSubType . parameterType . count != functionSuperType . parameterType . count) {
    result = false;
    return result;
}
result = result && isSubtype (functionSubType . returnType <: functionSuperType . returnType ) ;
foreach (node <> paramType1 : functionSubType . parameterType ; node <> paramType2 : functionSuperType . parameterType ) {
    result = result && isSubtype (paramType2 <: paramType1 ) ;
}
return result;
```

So, as we can see, the other semantic is quite an intuitive mapping between creating equations/inequations and performing inspections.

### 37.17 Type-system, trace

MPS provides a handy debugging tool that gives you insight into how the type-system engine evaluates the type-system rules on a particular problem and calculates the types. You invoke it from the context menu or by a keyboard shortcut (Control + Shift + X / Cmd + Shift + X):
The console has two panels. The one on the left shows the sequence or rules as they were applied, while the one on the right gives you a snapshot of the type-system engine’s working memory at the time of evaluating the rule selected in the left panel:

Type errors are marked inside the Type-system Trace panel with red color:
Additionally, if you spot an error in your code, use Control + Alt + Click / Cmd + Alt + Click to navigate quickly to the rule that fails to validate the types.
37.18 Advanced features of typesystem language

37.18.1 Overriding default type node

When type is assigned to a program node as a result of either applying an equation or resolving an inequality, the node to represent type is taken as is by default. That is to say, it may be a node in the program or be created with a quotation. In both cases, the result of evaluating the expression that specifies type to be assigned by either equation or inequality statement, literally represents target type. This feature allows to substitute another node to represent type instead. For example, one might decide to use different types for different program configurations, such as using \texttt{int} or \texttt{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 \texttt{Substitute Type Rule} in the typesystem model.

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

The \texttt{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 overrides 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 \texttt{Substitute Type Rule} applicable to the attribute first. This way one can override the type nodes even for languages, which implementation is sealed.

```java
substitute type rule substituteType_SubstituteAnnotation {
    applicable for concept = SubstituteAnnotation as substituteAnnotation
    substitute {
        if (substituteAnnotation.condition.isSatisfied(attributedNode)) {
            return substituteAnnotation.substitute;
        } else {
            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 \texttt{attributedNode} expression.

One caveat that should be mentioned, concerns the case when a type node just returned from a substitute rule, is itself a subject to another substitution. The typechecker tries to apply all matching substitution rules exhaustively, until no more substitutions are available. Only then the type appears in the internal model of the typechecker. Some precautions are taken to prevent the typechecker from going into an endless cycle of substitutions, such as \texttt{A -> B -> A}, but these are not perfect and one should be careful so as to not to introduce infinite cycles.

37.18.2 Check-only inequations

Basically, inequations may affect nodes’ types, for instance if one of the inequation part is a type variable, it may become a concrete type because of this inequation. But, sometimes one does not want a certain inequation to create types, only to check whether such an inequation is satisfied. We call such inequations check-only inequations. To mark an inequation as a check-only, one should go to this inequation’s inspector and should set a flag "check-only" to "true". To visually distinguish such inequations, the "less or equals" sign for check-only inequation is gray, while for normal ones it is black, so you can see whether an inequation is check-only without looking at its inspector.
37.18.3 Dependencies

When writing a generator for a certain language (see generator), one may want to ask for a type of a certain node in generator queries. When generator generates a model, such a query will make typesystem engine do some typechecking to find out the type needed. Performing full typechecking of a node’s containing root to obtain the node’s type is expensive and almost always unnecessary. In most cases, the typechecker should check only the node given. In more difficult cases, obtaining the type of a given node may require checking its parent or maybe a further ancestor. The typechecking engine will check a given node if the computed type is not fully concrete (i.e. contains one or more type variables); then the typechecker will check the node’s parent, and so on.

Sometimes there’s an even more complex case: the type of a certain node being computed in isolation is fully concrete; and the type of the same node - in a certain environment - is fully concrete also, but differs from the first one. In such a case, the abovementioned algorithm will break, returning the type of an node as if being isolated, which is not the correct type for the given node.

To solve this kind of problem, you can give some hints to the typechecker. Such hints are called dependencies - they express a fact that a node’s type depends on some other node. Thus, when computing a type of a certain node during generation, if this node has some dependencies they will be checked also, so the node will be type-checked in an appropriate environment. A dependency consists of a "target" concept (a concept of a node being checked, whose type depends on some other node), an optional "source" concept (a concept of another node on which a type of a node depends), and a query, which returns dependencies for a node, which is being checked, i.e. a query returns a node or a set of nodes.

For example, sometimes a type of a variable initializer should be checked with the enclosing variable declaration to obtain the correct type. A dependency which implements such a behavior may be written as follows:

```plaintext
target concept: Expression find source: (targetNode)->JOIN(node< | Set<node<>>) {
  if (targetNode/ .getRole().equals("initializer")
      return targetNode .parent;
  }
  return null;
}
source concept(optional): <auto>
```

That means the following: if the typechecker is asked for a type of a certain Expression during generation, it will check whether such an expression is of a role initializer, and if it is, will say that not only the given Expression, but also its parent should be checked to get the correct type for the given Expression.

37.19 Overriding type of literal or expression

In addition to type substitution rules, which are only applicable to types, we introduce support for attributes in the inference rules.

37.19.1 Inference rules

Literals or expressions usually have associated type inference rules that get triggered when the typechecker requires type of the node in question. The rules have a mechanism allowing subconcepts to extend or override the predefined rule.

```plaintext
rule typeof_IntLiteral { applicable for concept = IntLiteral as nodeToCheck
  applicable always
  overrides true
  do {
    typeof(nodeToCheck) :==: <integer>;
  }
}
```

37.19.2 Inference rules for node attributes

If a node has one or more attributes, the inference rules applicable to these attributes are applied before the rules applicable to the node itself. The process of applying inference rules can be described with a pseudo code.

```plaintext
lookup-inference-rules(node) :
```
let skipAttributed = false
foreach a in attributesOf(node) do
  if hasInferenceRuleFor(a) then
    let rule = getInferenceRuleFor(a)
    yield rule
    if isSuperceding(rule) then
      skipAttributed = true
    end if
  end if
end do
if skipAttributed then
  return
end if
/* proceed as usual */

An example of using an inference rule applicable to a node attribute shows how the presence condition can alter the type of a literal. Note that in this example the type of the annotated literal is affected by both this inference rule and any other inference rule applicable to the node.

```
rule typeof_Literal { applicable for concept = PresenceConditionAnnotation as pca
  applicable always
  overrides false
  supercedes attributed false
  do {
    typeof(pca.parent) :<=: pca.alternativeNode
  }
}
```

### 37.19.3 Conditionally overriding type inference

Keeping in mind that the condition under which the user might want to override the type inference via attributes depend on the configuration, we don’t always want to override the default type.

```
rule typeof_Literal {
  applicable for concept = PresenceConditionAnnotation as pea
  applicable always
  supercedes attributed {
    isConditionSatisfied(pca);
  }
  do {
    typeof(attributedNode) :==: pca.replacementType
  }
}
```
37.20 Checking rules

Checking, (or Non-typesystem) rules can inspect the model searching for known error patterns in code and report them to the user. This kind of pre-compilation code inspection is generally known as static code analysis. Error patterns in typical tools for static code analysis can fall into several categories, such as correctness problems, multi-threaded correctness, i18n problems, vulnerability-increasing errors, styling issues, performance problems, etc. The found issues are reported to the user either on-demand through an interactive report:

or in a on-the-fly mode directly in the editor by colorful symbols and code underlines:

37.20.1 Severity

MPS distinguishes problems by severity:

- errors - displayed in red
- warnings - displayed in yellow
- infos - displayed in grayish

The jetbrains.mps.lang.typesystem language offers corresponding statements that emit these problem categories together with their description and the node to highlight. The additional ensure statement gives the user a more succinct syntax to report an error in case a condition is not met:

Checking rules typically check for one or a few related issues in a given node or a small part of the model and report to the user, if a problem is discovered:
37.20.2 Quick-fixes

A **quick-fix** provides a single model-transforming function, which will automatically eliminate the reported problem:

```java
checking rule RepetitionCheck {
    applicable for concept = AbstractCommand as command
    overrides false
    do {
        if (command.concept == command.next-sibling.concept) {
            if (command.isDefined() && command.RoutineCall.definition != command.next-sibling.RoutineCall.definition) {
                warning "Consider using repeat to avoid repetition" -> command.next-sibling;
            }
        }
    }
}
```

A **quick-fix** must provide a **description** to represent it in the **Intentions** context menu, unless it is only ever referred to from callers with **apply immediately** set to **true**. A **quick-fix** may also declare **fields**, to hold reused values, and it can accept **arguments** from the caller.

37.20.3 Invoking quick-fixes

A **quick-fix** may be associated with each reported problem through the **Inspector** tool window using the **intention to fix**:

Normally the user invokes the **quick-fix** through the **Intentions** context menu, which is displayed after pressing the **Alt + Enter** key shortcut. If the **apply immediately** flag is set, however, MPS will run the associated **quick-fix** as soon as the problem is discovered during on-the-fly analysis without waiting for the user trigger.

The two other optional properties configured through the **Inspector** are needed less frequently:

- **node feature to highlight** - specifies a node’s property, child of reference to highlight as the source of the problem, instead of highlighting the whole node

- **foreign message source** - when a user clicks (Control/Cmd + Alt + click) on the reported error in the editor, she is taken to the **Checking rule’s error/warning/info/ensure** command that raised that error. With the **foreign message source** property you can override this behavior and provide your own node that the user will be taken to upon clicking on the error.
Chapter 38

Using_typesystem

38.1 Using a typesystem

If you have defined a typesystem for a language, a typechecker will automatically use it in editors to highlight opened nodes with errors and warnings. You may additionally want also to use the information about types in queries, like editor actions, generator queries, etc. You may want to use the type of a node, or you may want to know whether a certain type is a subtype of another one, or you may want to find a supertype of a type which has a given form.

38.2 Type Operation

You may obtain a type of a node in your queries using the type operation. Just write <expr>.type, where <expr> is an expression which is evaluated to a node.

Do not use type operation inside inference rules and inference methods! Inference rules are used to compute types, and type operation returns an already computed type.

38.3 Is Subtype expression

To inspect whether one type is a subtype of another one, use the isSubtype expression. Write isSubtype( type1 :< type2 ) or isStrongSubtype( type1 :« type2 ), it will return true if type1 is a subtype of type2, or if type1 is a strong subtype of type2, respectively.

38.4 Coerce expression

A result of a coerce expression is a boolean value, which says whether a certain type may be coerced to a given form, i.e. whether this type has a supertype, which has a given form (satisfies a certain condition). A condition could be written either as a reference to a concept declaration, which means that a sought-for supertype should be an instance of this concept; or as a pattern, which a sought-for supertype must match.

A coerce expression is written coerce( type :< condition ) or coerceStrong( type :« condition ), where condition is what has just been discussed above.

38.5 Coerce Statement

A coerce statement consists of a list of statements, which are executed if a certain type can be coerced to a certain form. It is written as follows:

coerce ( type :< condition ) {
    ...
} else {
    ...
}

If a type can be coerced so as to satisfy a condition, the first (if) block will be executed, otherwise the else block will be executed. The supertype to which a type is coerced can be used inside the first block of a coerce statement. If the condition is a pattern and contains some pattern variables, which match parts of the supertype to which the type is coerced, such pattern variables can also be used inside the first block of the coerce statement.
Chapter 39

Typesystem Debugging

For debugging typesystem MPS provides Typesystem Trace - an integrated visual tool that gives you insight into the evaluation process that happens inside the typesystem engine.

39.1 Try it out for yourself

We prepared a dedicated sample language for you to easily experiment with the typesystem. Open the Expressions sample project that comes bundled with MPS and should be available among the sample projects in the user home folder.

The sample language

The language to experiment with is a simplified expression language with several types, four arithmetic operations (+, -, *, /), assignment (:=), two types of variable declarations and a variable reference. The editor is very basic with almost no customization, so editing the expressions will perhaps be quite rough. Nevertheless, we expect you to inspect the existing samples and debug their types more than writing new code, so the lack of smooth editing should not be an issue.
The language can be embedded into Java thanks to the `SimpleMathWrapper` concept, but no interaction between the language and `BaseLanguage` is possible.

The expression language supports six types, organized by subtyping rules into two branches:

1. Element -> Number -> Float -> Long -> Int
2. Element -> Bool

### 39.2 Inspecting the types

If you open the `Simple` example class, you can position the cursor to any part of the expression or select a valid expression block. As soon as you hit `Control/Cmd + Shift + T`, you’ll see the type of the selected node in a pop-up dialog.

The `Main` sample class will give you a more involved example showing how `Type-inference` correctly propagates the suitable type to variables:
39.3 Type errors

The TypeError sample class shows a simple example of a type error. Just uncomment the code (Control/Cmd + /) and check the reported error:

```java
simple math
  var A = 10 + 11
  var B = 100 L + 200 L
  var C = 300 L - 20
  var D = 10 * 2.6 F
  var E = 10

  E := D
```

Since this variable declaration declares its type explicitly to be an Int, while the initializer is of type Float, the type-system reports an error. You may check the status bar at the bottom or hover your mouse over the incorrect piece of code.

39.4 Type-system Trace

When you hit Control/Cmd + Shift + X or navigate through the pop-up menu, you get the Typesystem Trace panel displayed on the right hand-side.

The Trace shows in Panel 2 all steps (i.e. type system rules) that the type-system engine executed. The steps are ordered top-to-bottom in the order in which they were performed. When you have Button 1 selected, the Panel 2 highlights the steps that directly or indirectly influence the type of the node selected in the editor (Panel 1). Panel 3 details the step
selected in Panel 2 - it describes what changes have been made to the type-system engine's state in the step. The actual state of the engine’s working memory is displayed in Panel 4.

39.5 Step-by-step debugging

The Simple sample class is probably the easiest one to start experimenting with. The types get resolved in six steps, following the typesystem rules specified in the language. You may want to refer to these rules quickly by pressing F4 or using the _Control/Cmd + N “Go to Root Node” command. F3 will navigate you to the node, which is being affected by the current rule.

1. The type of a variable declaration has to be a super-type of the type of the initializer. The aValue variable is assigned the a type-system variable, the initializer expression is assigned the b type-system variable and a>=b (b sub-type or equal type to a) is added into the working memory.

2. Following the type-system rule for Arithmetic Expressions, b has to be a sub-type of Number, the value 10 is assigned the c variable, 1.3F is assigned the d variable and a when-concrete handler is added to wait for c to be calculated.

3. Following the rules for float constants d is solved as Float.

4. Following the rules for integer constants c is solved as Int. This triggers the when-concrete handler registered in step 2 and registered another when-concrete handler to wait for d. Since d has already been resolved to Float, the handler triggered and resolves b (the whole arithmetic expression) as Float. This also solves the earlier equation (step 2) that b<=Number.

5. Now a can be resolved as Float, which also solves the step 1 equation that a>=b.

6. If you enable type expansions by pressing the button in the tool-bar, you’ll get the final expansions of all nodes to concrete types as the last step.
CHAPTER 39. TYPESYSTEM DEBUGGING
Chapter 40

Refactoring

40.1 Changes in the Refactoring language

In order to make the structure of MPS core languages more consistent and clear, the Refactoring language has been changed considerably. Several new and easy-to-use constructs have been added and parts of the functionality was deprecated and moved into the Actions language.

The UI for retrieving the refactoring parameters has been removed from the refactoring language. Choosers for parameters are no longer called, it is not allowed to show UI in init (e.g. ask and ask boolean) and keystroke has no effect. All this functionality should be moved to an action corresponding to the refactoring.

The following constructs have been added to the refactoring language. These new constructs are intended to be used from code, typically from within the actions:

- \texttt{is applicable refactoring<Refactoring>(target)}
  returns true if the refactoring target corresponds to the current target (type, single/multiple) and applicable as in refactoring isApplicable method, and there is no refactoring that overrides current refactoring for this target.

- \texttt{execute refactoring<Refactoring>(target : project, parameters )};
  executes the refactoring for the target with parameters

- \texttt{create refcontext<Refactoring>(target : project, parameters )}
  create a refactoring context for the refactoring, target and fill parameters in context, this context then can be used for refactoring execution or for further work with parameters; UI is not shown during this call

It is necessary to manually migrate existing user refactorings. The migration consists of several steps:

- create a UI action for the refactoring (This is a simple action from the plugin language. You can check the Rename action from jetbrains.mps.ide.platform.actions.core as an example of proper refactoring action registration)
  - copy the caption, create context parameters
- add a refactoring keystroke with the newly created action to KeymapChangesDeclaration
- create ActionGroupDeclaration for the refactoring that modifies the jetbrains.mps.ide.platform.actions.NodeRefactoring action group at the default position
  - add an isApplicable clause to the action created; usually it is just \texttt{is applicable refactoring<>() call}
- add an execute clause to the action created; all the parameter preparations that were in init of the refactoring should be moved here; at the end it is necessary to execute the refactoring with the prepared parameters (with \texttt{execute refactoring<>()}; statement)
  - remove all parameter preparation code from init of the refactoring, they are now prepared before the entry to init; you can still validate parameters and return false if the validation fails
Chapter 41

Testing languages

41.1 Testing languages

41.2 Introduction

Testing is an essential part of language designer’s work. To be of any good MPS has to provide testing facilities both for BaseLanguage code and for languages. While the jetbrains.mps.baselanguage.unitTest language enables JUnit-like unit tests to test BaseLanguage code, the Language test language jetbrains.mps.lang.test provides a useful interface for creating language tests.

To minimize impact of test assertions on the test code, the Language test language describes the testing aspects through annotations (in a similar way that the generator language annotates template code with generator macros).

41.3 Quick navigation table

Different aspects of language definitions are tested with different means:

<table>
<thead>
<tr>
<th>Language definition aspects</th>
<th>The way to test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intentions</td>
<td>Use the jetbrains.mps.lang.test language to create EditorTestCases. You set the stage by providing an initial piece of code, define a set of editing actions to perform against the initial code and also provide an expected outcome as another piece of code. Any differences between the expected and real output of the test will be reported as errors. See the Decision table for details.</td>
</tr>
<tr>
<td>Actions</td>
<td></td>
</tr>
<tr>
<td>Side-transforms</td>
<td></td>
</tr>
<tr>
<td>Editor ActionMaps</td>
<td></td>
</tr>
<tr>
<td>KeyMaps</td>
<td></td>
</tr>
<tr>
<td>Constraints</td>
<td>Use the jetbrains.mps.lang.test language to create NodesTestCases. In these test cases write snippets of &quot;correct&quot; code and ensure no error or warning is reported on them. Similarly, write &quot;invalid&quot; pieces of code and assert that an error or a warning is reported in the correct node. See the Decision table for details.</td>
</tr>
<tr>
<td>Scopes</td>
<td></td>
</tr>
<tr>
<td>Type-system</td>
<td></td>
</tr>
<tr>
<td>Dataflow</td>
<td></td>
</tr>
<tr>
<td>Generator</td>
<td>There is currently no built-in testing facility for these aspects. There are a few practices that have worked for us over time:</td>
</tr>
<tr>
<td>TextGen</td>
<td>• Perhaps the most reasonable way to check the generation process is by generating models, for which we already know the correct generation result, and then comparing the generated output with the expected one. For example, if your generated code is stored in a VCS, you could check for differences after each run of the tests.</td>
</tr>
<tr>
<td>Migrations</td>
<td>• You may also consider providing code snippets that may represent corner cases for the generator and check whether the generator successfully generates output from them, or whether it fails.</td>
</tr>
<tr>
<td></td>
<td>• Compiling and running the generated code may also increase your confidence about the correctness of your generator.</td>
</tr>
<tr>
<td></td>
<td>Use the jetbrains.mps.lang.test language to create MigrationTestCases. In these test cases write pieces of code to run migration on them. See the Decision table for details.</td>
</tr>
</tbody>
</table>
41.4 Tests creation

There are two options to add test models into your projects.

41.4.1 1. Create a Test aspect in your language

This is easier to setup, but can only contain tests that do not need to run in a newly started MPS instance. So typically can hold plain baselanguage unit tests. To create the Test aspect, right-click on the language node and choose New->Test Aspect.

Now you can start creating unit tests in the Test aspect.

Right-clicking on the Test aspect will give you the option to run all tests. The test report will then show up in a Run panel at the bottom of the screen.

41.4.2 2. Create a test model

This option gives you more flexibility. Create a test model, either in a new or an existing solution. Make sure the model's stereotype is set to tests.

Open the model's properties and add the jetbrains.mps.baselanguage.unitTest language in order to be able to create unit tests. Add the jetbrains.mps.lang.test language in order to create language (node) tests.
Additionally, you need to make sure the solution containing your test model has a kind set - typically choose Other, if you do not need either of the two other options (Core plugin or Editor plugin).

Right-clicking on the model allows you to create new unit or language tests. See all the root concepts that are available:
41.4.3 Unit testing with BTestCase

As for BaseLanguage Test Case, represents a unit test written in baseLanguage. Those are familiar with JUnit will be quickly at home.

A BTestCase has four sections - one to specify test members (fields), which are reused by test methods, one to specify initialization code, one for clean up code and finally a section for the actual test methods. The language also provides a couple of handy assertion statements, which code completion reveals.

41.4.4 TestInfo

In order to be able to run node tests, you need to provide more information through a TestInfo node in the root of your test model.

Especially the Project path attribute is worth your attention. This is where you need to provide a path to the project root, either as an absolute or relative path, or as a reference to a Path Variable defined in MPS (Project Settings -> Path Variables).
To make the path variable available in Ant scripts, define it in your build file with the `mps.macro.` prefix (see Testing languages).

### 41.5 Testing aspects of language definitions

Different aspects of language definitions are tested with different means:

<table>
<thead>
<tr>
<th>Language definition aspects</th>
<th>The way to test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intentions</td>
<td>Use the <code>jetbrains.mps.lang.test</code> language to create <code>EditorTestCases</code>. You set the stage by providing an initial piece of code, define a set of editing actions to perform against the initial code and also provide an expected outcome as another piece of code. Any differences between the expected and real output of the test will be reported as errors. See the Decision table for details.</td>
</tr>
<tr>
<td>Actions</td>
<td></td>
</tr>
<tr>
<td>Side-transforms</td>
<td></td>
</tr>
<tr>
<td>Editor ActionMaps</td>
<td></td>
</tr>
<tr>
<td>KeyMaps</td>
<td></td>
</tr>
<tr>
<td>Constraints</td>
<td>Use the <code>jetbrains.mps.lang.test</code> language to create <code>NodesTestCases</code>. In these test cases write snippets of “correct” code and ensure no error or warning is reported on them. Similarly, write “invalid” pieces of code and assert that an error or a warning is reported in the correct node. See the Decision table for details.</td>
</tr>
<tr>
<td>Scopes</td>
<td></td>
</tr>
<tr>
<td>Type-system</td>
<td></td>
</tr>
<tr>
<td>Dataflow</td>
<td></td>
</tr>
<tr>
<td>Generator</td>
<td>There is currently no built-in testing facility for these aspects. There are a few practices that have worked for us over time:</td>
</tr>
<tr>
<td>TextGen</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Perhaps the most reasonable way to check the generation process is by generating models, for which we already know the correct generation result, and then comparing the generated output with the expected one. For example, if your generated code is stored in a VCS, you could check for differences after each run of the tests.</td>
</tr>
<tr>
<td></td>
<td>• You may also consider providing code snippets that may represent corner cases for the generator and check whether the generator successfully generates output from them, or whether it fails.</td>
</tr>
<tr>
<td></td>
<td>• Compiling and running the generated code may also increase your confidence about the correctness of your generator.</td>
</tr>
</tbody>
</table>
Use the `jetbrains.mps.lang.test` language to create `MigrationTestCases`. In these test cases write pieces of code to run migration on them. See the Decision table for details.

### 41.5.1 Node tests

A `NodesTestCase` contains three sections:

- **Test case EmptyNodesTestCase**
  - `nodes`:
  - `test methods`:
  - `utility methods`

The first one contains code that should be verified. The section for **test methods** may contain `baseLanguage` code that further investigates nodes specified in the first section. The **utility methods** section may hold reusable `baseLanguage` code, typically invoked from the **test methods**.

#### Checking for correctness

To test that the type system correctly calculates types and that proper errors and warnings are reported, you write a piece of code in your desired language first. Then select the nodes, that you’d like to have tested for correctness and choose the Add Node Operations Test Annotation intention.

This will annotate the code with a `check` attribute, which then can be made more concrete by setting a type of the check:

- **Test case DuplicateCommandGetsReported**
  - `nodes`:
  - `test methods`:
  - `utility methods`

Note that many of the options have been deprecated and should no longer be used.

The **for error messages** option ensures that potential error messages inside the checked node get reported as test failures. So, in the given example, we are checking that there are no errors in the whole `Script`.
Checking for type system and data-flow errors and warnings

If, on the other hand, you want to test that a particular node is correctly reported by MPS as having an error or a warning, use the **has error** / **has warning** option.

This works for both warnings and errors.

You can even tie the check with the rule that you expect to report the error / warning. Hit `Alt + Enter` when with cursor over the node and pick the **Specify Rule References** option:
An identifier of the rule has been added. You can navigate by Control/Cmd + B (or click) to the definition of the rule.

When run, the test will check that the specified rule is really the one that reports the error.

**Type-system specific options**

The `check` command offers several options to test the calculated type of a node.
CHAPTER 41. TESTING LANGUAGES

Multiple expectations can be combined conveniently:

```java
public void bar() {
    System.out.println(<check age has expected type string, has type int>);
}
```

Testing scopes

The **Scope Test Annotation** allows the test to verify that the scoping rules bring the correct items into the applicable scope:

```java
Test case FooNodesTestCases
  nodes
  ([ <check public class Foo {
      private int age = 0;
        
      public void bar() {
          System.out.println(<check age});
      }]
  )
  test methods
  utility methods
```

The **Inspector panel** holds the list of expected items that must appear in the completion menu and that are valid targets for the annotated cell:
Test and utility methods

The **test methods** may refer to nodes in your tests through labels. You assign labels to nodes using intentions:

![Intention for adding test label annotation]

The labels then become available in the test methods.
41.5.2 Editor tests

Editor tests allow you to test the dynamism of the editor - actions, intentions and substitutions.

An empty editor test case needs a name, an optional description, setup the code as it should look before an editor transformation, the code after the transformation (result) and finally the actual trigger that transforms the code in the code section.

For example, a test that an IfStatement of the Robot_Kaja language can be transformed into a WhileStatement by typing while in front of the if keyword would look as follows:
In the code section the `jetbrains.mps.lang.test` language gives you several options to invoke user-initiated actions - use type, press keys, invoke action or invoke intention. Obviously you can combine the special test commands for the plain baseLanguage code.

In order to be able to specify the desired actions and intentions, you need to import their models into the test model. Typically the `jetbrains.mps.ide.editor.actions` model is the most needed one when testing the editor reactions to user-generated actions.

To mark the position of the caret in the code, use the appropriate intention with the cursor located at the desired position:

The cursor position can be specified in both the before and the after code:
The cell editor annotation has extra properties to fine-tune the position of the caret in the annotated editor cell. These can be set in the Inspector panel.

**Inspecting the editor state**

Some editor tests may wish to inspect the state of the editor more thoroughly. The editor component expression gives you access to the editor component under cursor. You can inspect its state as well as modify it, like in these samples:

```code
invoke action -> Backspace
editor.getEditorContext().getRepository().getModelAccess().runReadAction(callback
nodeClassifierType> testNode = test;
EditorCell editorCell = editor.component.findCellWithId(testNode, "ReferencePresentation_91bvs_s0a0");
assert "SmartReferenceUpdate" equals {{EditorCell_Label} editorCell.getText();
}
```

```code

type " "
invoke action -> Complete
assert true editor component.getNodeSubstituteChooser().isVisible();
editor component.getNodeSubstituteChooser().setVisible(false);
invoke action -> MoveUp
```

```code
invoke action -> ShowReflectiveEditor
assert true editor component.getUpdater().
getExplicitEditorHintsForNode(editor component.getSelectedNode().getReference()).asSequence.
contains(concept editor hint/reflectiveEditor/);
invoke action -> ShowRegularEditor
```

```code
invoke action -> Complete
assert editor component.getNodeSubstituteChooser().isVisible() &&
editor component.getNodeSubstituteChooser().getNumberOfActions() == 1;
```

The is intention applicable expression let’s you test, whether a particular intention can be invoked in the given editor context:

```java
before: {
    boolean b = true;
    int a = <cell> b ? 10 : 20>
}
result: {
    boolean b = true;
    int a = !b ? 20 : 10;
}
code:
assert is intention applicable SwapTernaryBranches;
invoke intention SwapTernaryBranches
```

You can also get hold of the model and project using the model and project expressions, respectively.

### 41.5.3 Migration tests

Migrations tests can be used to check that migration scripts produce expected results using specified input.
To create a migration test case you should specify its name and the migration scripts to test. In many cases it should be enough to test individual migration scripts separately, but you can safely specify more than one migration script in a single test case, if you need to test how migrations interact with one another.

Additionally, migration test cases contain nodes to be passed into the migration process and those also nodes that are expected to come out as the output of the migration.

When running, migration tests behave the following way:

1. Input nodes are copied as roots into an empty module with single model.
2. Migration scripts run on that module.
3. Roots contained in that module after migration are compared with the expected output.
4. The check() method of the concerned migration(s) is invoked to ensure that it returns an empty list of problems.
To simplify the process of writing migration tests, the expected output can be generated automatically from the input nodes using the currently deployed migration scripts. To do this, use the intention called ‘Generate Output from Input’.

### 41.6 Running the tests

#### 41.6.1 Inside MPS

To run tests in a model, just right-click the model in the *Project View* panel and choose *Run tests*:
If the model contains any of the `jetbrains.mps.lang.test` tests, a new instance of MPS is silently started in the background (that’s why it takes quite some time to run these compared to plain `baseLanguage` unit tests) and the tests are executed in that new MPS instance. A new run configuration is created, which you can then re-use or customize:

The Run configurations dialog gives you options to tune the performance of tests.

- **Reuse caches** - reusing the old caches of headless MPS instance when running tests cuts away a lot of time that would be needed to setup a test instance of MPS. It is possible to set and unset this option in the run configuration dialog.

- **Save caches in** - specify the directory to save the caches in. By default, MPS choses the temp directory. Thus with the option **Reuse caches** set on, MPS saves its caches in the specified folder and reuses them whenever possible. If the option is unset, the directory is cleared on every run.

- **Execute in the same process** - to speed up testing tests can be run in a so-called in-process mode. It was designed specifically for tests, which need to have an MPS instance running. (For example, for the language type-system tests MPS should safely be able to check the types of nodes on the fly.) The original way was to have a new MPS instance started in the background and run the tests in this instance. This option, instead, allows to have all tests run in the same original MPS process, so no new instance needs to be created. When the option **Execute in the same process** is set (the default setting), the test is executed in the current MPS environment. To run tests in the original way (in a separate process) you should uncheck this option. This way of tests’ execution is applicable to all test kinds in MPS. Thus it works even for the editor tests!

Although the performance is so much better for in-process test execution, there are certain drawbacks in this workflow. Note, that the tests are executed in the same MPS environment that holds the project, so there is a possibility, that the code you write in your test may be potentially dangerous and sometimes cause real harm. For example, a test, which disposes the current project, could destroy the whole project. So the user of this feature needs to be careful when writing the tests. There are certain cases when the test must not be executable in-process. In that case it is possible to switch an option in the inspector to prohibit the in-process execution for that specific test.

The test report is shown in the Run panel at the bottom of the screen:
41.6.2 From a build script

In order to have your generated build script offer the test target that you could use to run the tests using Ant, you need to import the `jetbrains.mps.build.mps` and `jetbrains.mps.build.mps.tests` languages into your build script, declare using the `module-tests` plugin and specify a test modules configuration.

To define a macro that Ant will pass to JUnit (e.g. for use in TestInfo roots in your tests), prefix it with `mps.macro:`.
41.7 Running Editor tests in IDEA Plugin

With the new JUnit test suite (jetbrains.mps.idea.core.tests.PluginsTestSuite) it is possible to execute editor tests for your languages in IntelliJ IDEA, when using the MPS plugin. To make use of this functionality you have to create a simple ANT script that will install all the necessary plugins into the IntelliJ platform and executing the tests by specifying test module name(s).
Chapter 42

Common language patterns

This chapter covers common language design patterns that the beginners learning MPS would frequently come across. We’ve identified these over time as the most frequent questions that MPS users ask in the on-line forum and so we decided to summarise the answers in a single place for easy reference.

42.1 Sample project with language patterns

MPS comes bundled with sample projects. Many of the patterns discussed here have been implemented in the `languagePatterns` sample project. You can open it up in MPS (click on Open sample project in the MPS welcome screen) and check the actual implementation of the patterns of this chapter that refer to the sample project.
The patterns are organised into virtual packages both in the language definition and in the sandbox solution, to help you identify the elements that form the individual patterns.

### 42.2 Initial tips

#### 42.2.1 Naming concepts

Many concepts need to provide a string property to hold the name of the nodes. Since a name of a concept has certain special qualities and needs to be recognised and handled by MPS in a special way (e.g. to label nodes in the code completion menu as well as in the Project View, Node Explorer, Debugger Tool Window and others), it is advisable to inherit the property from the `INamedConcept` concept interface. Many of your concepts and almost all root concepts should thus declare to implement `INamedConcept`.

#### 42.2.2 Valid identifiers

If you are extending `BaseLanguage` and the name of the concept translates directly to a Java identifier, you should consider inheriting the `name` property from the `IValidIdentifier` concept interface, which extends `INamedConcept`. It will ensure through constraints that the name meets the criteria for a valid Java identifier.
42.2.3 Unique names

Frequently you want to ensure uniqueness of names for a certain group of nodes of the same concept. You may use either constraints or non-typesystem rules. Not-typesystem rules give you the option to customise the error message and may even offer quick-fixes to the user for automatic problem resolution. A sample uniqueness check for the name of an InputField in a Calculator of the Calculator tutorial would look like this:

```plaintext
checking rule check_InputField {
  applicable for concept = InputField as inputField
  overrides false
  do {
    if (inputField.parent : Calculator.inputField.any({~it => it.name :eq: inputField.name && it :ne: inputField; })) {
      error "Duplicate name " + inputField.name -> inputField;
    }
  }
}
```

The same constraint specified with constraints would look this way:

```plaintext
concepts constraints InputField {
  can be child <none>
  can be parent <none>
  can be ancestor <none>

  property {name}
    get:<default>
    set:<default>
    is valid:(propertyValue, node)->boolean {
      node.parent : Calculator.inputField.where({~it => it.name :eq: propertyValue; }).size <= 1;
    }

  <<referent constraints>>

  default scope
    <no default scope>
}
```

Please note that constraints are probably less optimal here, for several reasons:

- you cannot specify a quick-fix
- you cannot customise the error message
- while the non-typesystem rule only underlines the duplicate names to indicate the error, constraints prevent invalid values to be inserted into the model and display the invalid values in red font, which makes the errors look more severe

42.2.4 Not seeing type-system errors

The type-system runs in the background and may sometimes be slow to deliver its results into the editor. Hit F5 in order to refresh all error and warning messages in the editor. Also make sure you have the Power Save Mode setting switched off. In Power Safe Mode the type-system only runs on explicit demand from the user (F5).

42.2.5 Regular expressions

By default properties can be of one of three types - integer, boolean and string. If you want a more customized datatype, you have to define one and restrict the allowed values by a regular expression. BaseLanguage defines float types this way, for example. Open the FloatingPointConstant concept (Control/Cmd + N) and see the property value - it has a type of _FPNumber_String. Navigate to the definition of _FPNumber_String (Control/Cmd + B) to see an example of a constrained data type:
42.2.6 Safe operators and operations

BaseLanguage and its core extensions offer several handy shortcuts for common operations that in Java require more efforts to get right - for example, checking for null before comparing, distinguishing empty and null lists, calling equals instead of "==", etc. Here's a list of the most commonly useful ones:

- :eq: - null-safe equals
- :ne: - null-safe not-equals
- isNull
- isNotNull
- size
- isEmpty
- isNotEmpty

42.3 Basic patterns

42.3.1 Container - component

Illustrated as the container-component virtual package in the languagePatterns sample.

A common scenario where a container (FruitPlate in our case) contains elements (Fruit) of multiple kinds (Apples, Oranges). An abstract concept (Fruit) is used as a placeholder in the container and concrete sub-concepts (Apples, Oranges) are then provided by the language user explicitly. The sub-concepts may have completely different visual appearance.

See also the seamless-substitution pattern to learn about how to allow the user to switch Apples to Oranges and back with little effort. The current implementation requires the user to first select the whole node (Apple or Orange) and only then code-completion offers the alternatives.
42.3.2 Customized presentation

Illustrated as the custom-presentation virtual package in the languagePatterns sample. Motivated by the discussion at http://forum.jetbrains.com/thread/Meta-Programming-System-2138?message=Meta-Programming-System-2138-3

Components held within Containers are referred to from ComponentUsages. The reference, however, should contain the name of the containing Container together with the name of the Component - both in the completion menu and in the editor.
Additionally, scoping rules should guarantee that only one reference to any Component may exist and the completion menu is filtered so that already referred Components are not offered.

```csharp
concepts constraints ComponentUsage {
  can be child <use>
  can be parent <use>
  can be ancestor <use>
  <property constraints>
  link (usedComponent)
    referent set handler <use>
    scopes:
      exists, referenceNode, containingLink, linkTarget, operationContext, enclosingNode, model, position, contextRole->Scope {
        sequence<br> Component
          unusedComponents = model.nodes(Component).
          where(-h = u.usedComponent; e: b: ));
          all(h = u.usedComponent; e: x: b: ));
        new ListNodes(unusedComponents) {
          public string getNode(name: child) {
            child = Component.name;
          }
        }
      }
    }
  presentation {
    exists, referenceNode, containingLink, linkTarget, operationContext, enclosingNode, model, position, contextRole, parameterNode, 'inEdi'
    parameterNode.fullName();
  }
}
```

### 42.3.3 Declaration - references
Illustrated as the declaration-references virtual package in the languagePatterns sample.

**Concert SchoolEvent**

**Performers:**
- Singer Mr. Smith (teacher)
- Singer Maria
- Singer Jim
- Singer James
- Singer Ann

**Agenda:**
- Mr. Smith (teacher)
- Jim

A typical pattern on declarations and references to them - Singers at an event can be organised into agendas with their Performances. Different types of Performances are available.

Scoping rules ensure that:

- only singers listed in the current event can be added to its agenda
- each singer can only be listed once in a combined performance
- a handy intention (IntroduceSinger) is available to create a singer from a string typed in the performance (frequently known as "create from usage" or "introduce variable" refactoring).

### 42.3.4 Fluent editing
Illustrated as the fluent-editing virtual package in the languagePatterns sample.

```csharp
Commands My Painting {
  dotted line from: 1 2 to: 3 4
  line from: 1 2 to: 0 0
} + dotted
  solid
```
An example of creating a text-like editing experience and implements many of the recommendations mentioned in the Editor cookbook (https://confluence.jetbrains.com/display/MPSD32/Editor+cookbook). It implements a simple language for specifying drawing commands (line, rectangle):

- an empty line is inserted upon hitting Enter, empty lines can be placed anywhere thanks to a default node factory specified for the child collection
- empty line does not appear as an option in the completion menu thanks to implementing the `IDontSubstituteByDefault` concept interface
- typing on an empty line will insert the desired item (line or rectangle)
- when the body of a block of code is empty, the cursor is positioned on the first line (next to the header), editing can be started on this first line
- `NodeFactories` take care of propagating values to new nodes if a node is replaced with another one
- an optional "line style" child can be specified on the left of the draw commands thanks to left-side transformations
- wrappers allow the desired line style to be typed on an empty line and an intermediate "IncompleteCommand" will be created automatically out of it
- `IncompleteCommand` expects either line or rectangle to be typed and will be substituted into the desired draw command instantly
- draw commands define aliases and editors so that they start with the same word
- an editor in the abstract super-concept of the draw commands is reused by concrete sub-concepts
- typing on the left hand-side of a command will offer the allowed prefixes
- deleting a prefix (solid, dotted) will correctly delete only the prefix

### 42.3.5 Override editor component

Illustrated as the `override-editor-component` virtual package in the `languagePatterns` sample.

An example of using editor components that get overridden in a sub-concept (`Truck`). An editor in `Car` uses an editor component `CarProperties` also defined in `Car`. A sub-concept (`Truck`) overrides the `CarProperties` editor component with the `TruckProperties` editor component to contain its own properties. The editor in `Car` will use the `Truck`'s variant of the editor component for `Trucks` and the `Car` variant for `Cars`.

### 42.3.6 Seamless substitution

Illustrated as the `seamless-substitution` virtual package in the `languagePatterns` sample.
An example that seamlessly switches (substitutes) between different related subconcepts. A Request contains a "description", which may be either a string, a simple form or a complex form. A completion-menu lets the user pick the requested description type. If the user simply types text, the "string"-based description is picked automatically in the "PickTheRight-DescriptionType" substitute action. The first cell of editors for each of the description concepts is sensitive to substitution (set through the "menu" property of the cell) and so offers the option to switch between description types with completion-menu. The "Converters" node factories contain code that preserves parts of the description information and propagates it into the newly instantiated description concept.

```java
Request NewPen
Description:
 requester: Dave age: 26 applying first time: true

<table>
<thead>
<tr>
<th>form description</th>
<th>(AbstractRequestDescription in j.m.s.languagePatterns.Basic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>simplified form description</td>
<td>(AbstractRequestDescription in j.m.s.languagePatterns.Basic)</td>
</tr>
<tr>
<td>string description</td>
<td>(AbstractRequestDescription in j.m.s.languagePatterns.Basic)</td>
</tr>
</tbody>
</table>
```

```java
node factories Converters
node concept: SimplifiedFormDescription
description: (newNode, sampleNode, enclosingNode, model)->void {
  ifInstanceOf (sampleNode is StringDescription original) {
    newNode.detail = original.value;
  }
  ifInstanceOf (sampleNode is FormDescription original) {
    newNode.detail = original.detail;
    newNode.firstTimeApplicant = original.firstTimeApplicant;
    newNode.requestor = original.requestor;
  }
}

node concept: FormDescription
description: (newNode, sampleNode, enclosingNode, model)->void {
  ifInstanceOf (sampleNode is StringDescription original) {
    newNode.detail = original.value;
  }
  ifInstanceOf (sampleNode is SimplifiedFormDescription original) {
    newNode.detail = original.detail;
    newNode.firstTimeApplicant = original.firstTimeApplicant;
    newNode.requestor = original.requestor;
  }
}

node concept: StringDescription
description: (newNode, sampleNode, enclosingNode, model)->void {
  ifInstanceOf (sampleNode is SimplifiedFormDescription original) {
    newNode.value = original.detail;
  }
  ifInstanceOf (sampleNode is FormDescription original) {
    newNode.value = original.detail;
  }
}
```
42.3.7 Hierarchical scopes

If you are defining references of some sort, you typically want to restrict the scope for the references so they can only point to nodes in certain locations or "distance" from the reference. Additionally, definitions may hide other references with the same names that are further away from the reference - think of Java's local variables hiding parameter declarations or fields, for example. In MPS when defining references you provide the the scope of the reference in the Constraints aspect of the language definition. For hierarchical scopes you use the inherited scope type in the reference and then let your code containers (aka classes, methods, block statements, etc.) implement ScopeProvider concept interface. In its getScope() method you implement the logic of retrieving candidate targets of certain kind for references that want to point to them. Check out the Scopes for detailed how-to.

42.3.8 Extending the DotExpression for your own references

Illustrated as the dotexpression virtual package in the languagePatterns sample (bundled in MPS 3.3 and later). De-referencing elements by "dot" notation is a very common practise in many languages. BaseLanguage offers the DotExpression concept to mimics Java's "dot" operator and you can leverage it in your languages, if they're extending BaseLanguage's expressions. Let's assume a simple form with several addresses that need to be validated by validation expressions. The expressions need to refer to the street and zip code of each address in order to include their values in the validation expression:

```
Addresses: home street: abc zip code 1
work street: def zip code 2
Validation: home.zip.isNotEmpty && work.street.length() > 1
```

Each address is represented by a node of Address concept. Notice that it also customises the presentation so that it displays the "kind" property in the code-completion menu:
The `AddressReference` concept allows referring to addresses from within `BaseLanguage` expressions in the validation section.
All `BaseLanguage` expressions are converted to `DotExpression` as soon as you append "dot" to them, with the left operand being the original expression. What remains is the right side of the `DotExpression`, called `operation`. Operations implement the `IOperation` concept interface. We will need two operations, one for street and one for the zip code, thus we'll start with a common abstract super concept for these:

```
abstract concept OperationOnAddress extends BaseConcept
  implements IOperation

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

  properties:
  << ... >>

  children:
  << ... >>

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

The concrete operations then only specify a meaningful alias and a type-system rule so as to participate properly in the whole surrounding expression:
Chapter 43

Accessories

The **Accessories Models** can be stored at two places - either as an aspect of a language (recommended), or as a regular model under a solution. In both cases, the model needs to be added to the *Language Runtime Language Settings* so as it could be used. A typical use case would be a default library of Concept instances to be available at any place the language is used.

### 43.1 Example

Let’s alter the **Shapes** sample project created as part of the introductory **Shapes - an introductory MPS tutorial** and bundled with MPS distributions as a sample project. The project allows the language users to define various colorful shapes and put them on a canvas. The colors of each shape are defined as references to one of the **StaticFieldDeclarations** defined in the **Color** class.

Accessories models allow us to define our own color constants instead of referencing directly the **Color** class and thus impose a dependency on **BaseLanguage** from the user solutions. You’ll get finer control over what colors will be available and how they will get generated.

#### 43.1.1 Define the concept to represent colors

First we need to define the concept that we will then use to define individual colors:

```
concept MyColor extends BaseConcept
  implements INamedConcept

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

  properties:
  << ... >>

  children:
  << ... >>

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

#### 43.1.2 Update **ColorReference**

The **ColorReference** concept should now point to nodes of the **MyColor** concept:

```
concept ColorReference extends BaseConcept
  implements <none>

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

  properties:
  << ... >>

  children:
  << ... >>

  references:
  color: MyColor[1]
```
43.1.3 Define a method to obtain the real \textit{Color}

During generation we will need to replace nodes of \texttt{MyColor} with nodes of \texttt{StaticFieldDeclaration} representing the corresponding color constants defined in the \textit{Color} class:

```java
public node<StaticFieldDeclaration> findColor() {
    string colorName = this.color.name;
    node<StaticFieldDeclaration> declaration = node<Color/>.staticFields().where({-it => it.name eq colorName; }),
    first;
    return declaration;
}
```

43.1.4 Change the generator templates for circle and square

These templates hold a reference macro that inserts a reference to the particular desired static field in the \textit{Color} class. We need to change the macro so that it uses the \texttt{findColor()} behavior method that we defined above:

```java
template reduce Circle
input Circle
parameters << ... >>
content node:
    <E>
    graphics.setColor(Color.+red);
    graphics.drawOval($10, $10, $10, $10);
    </E>
</content>
```

43.1.5 Define colors in the Accessories model

Now the colors can be safely defined:

After rebuilding the language the color constants will be available in the completion menu in your \textit{Canvas} nodes and the generated code will hold correct references to Java colors.
Chapter 44

Scripts

TODO
Chapter 45

Concept Functions

Concept functions allow language designers to leave hooks for their language users, through which the users can provide code to leverage in the generated code. For example, most of the languages that MPS offers for language design, such as Editor, Constraints or Intentions, leverage Concept functions:

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

  can be parent
  {childConcept, node, childNode, operationContext, link}->boolean {
    true;
  }

  can be ancestor <none>
}
```

You can also discover their usages down in the Inspector window:

- **Style**
  - **Keyword**
    - `indent-layout-indent` : (editorContext, node)->boolean {
      node.definations.isNotEmty;
    }
    - `text-foreground-color` : (editorContext, node)->Color {
      if (node.name.contains("Essential") {{
        return Color.red;
      } else {
        return Color.green;
      }
    }

Concept functions are defined in `jetbrains.mps.baselanguage` and they contain BaseLanguage code, which upon generation becomes part of the generated Java code. This option can give your DSLs enormous flexibility.

### 45.1 Example

We'll use the Robot Kaja sample project to experiment with Concept functions. The goal is to allow the Script authors to provide a function that will customize the Trace messages, which are reported to the user through the `trace` command:
The user will be able to customize the trace messages through a function that receives the original message as a parameter and returns a string that should be displayed instead:

```plaintext
Script Sample runs as
trace -> Starting... <-
step
trace -> The robot has made its first step <-
while not heading south do
  turnLeft
end
trace -> Looking south now <-
while not wall ahead do
  step
end
trace -> Finished <-
end

trace customization:
customizeMessage(message)->string {
  return "Robot progress report: *" + message + "*";
}
```

### 45.1.1 Define the concept function `concept`

First, a sub-concept of `ConceptFunction` must be created:

```plaintext
concept MyFunction extends ConceptFunction
    implements <none>

    instance can be root: false
    alias: customizeMessage
    short description: return a customized trace message to show to the user

    properties:
    << ... >>

    children:
    << ... >>

    references:
    << ... >>
```
CHAPTER 45. CONCEPT FUNCTIONS

245

The behavior aspect overrides a few methods inherited from ConceptFunction:

- `getExpectedReturnType()` - declares what type should be returned from the function
- `getApplicableConceptFunctionParameter()` - lists the concepts that will represent parameters to this function
- `showName()` - indicates, whether the name of the function should be displayed in the editor alongside the parameter list and the return type
- `getName()` - the name of the function to display in the editor

Since `MyFunction` requires an argument to hold the original trace message value, we also need to create a concept to represent that parameter, which extends the `ConceptFunctionParameter` concept and specifies its type through an overridden `getType()` behavior method:

```java
concept behavior MyFunction {

  constructor {
    <no statements>
  }

  public node<String> getExpectedReturnType() {
    overrides ConceptFunction.getExpectedReturnType {
      string;
    }
  }

  public static list<node<ConceptDeclaration>> getApplicableConceptFunctionParameter() {
    overrides ConceptFunction.getApplicableConceptFunctionParameter {
      list<node<ConceptDeclaration>> result = super<ConceptFunction>.getApplicableConceptFunctionParameter();
      result.add(node<Message_Parameter>);
      return result;
    }
  }

  public static boolean showName() {
    overrides ConceptFunction.showName {
      true;
    }
  }

  public string getName() {
    overrides ConceptFunction.getName {
      "customizedMessage";
    }
  }
}
```

The behavior aspect overrides a few methods inherited from ConceptFunction:

- `getExpectedReturnType()` - declares what type should be returned from the function
- `getApplicableConceptFunctionParameter()` - lists the concepts that will represent parameters to this function
- `showName()` - indicates, whether the name of the function should be displayed in the editor alongside the parameter list and the return type
- `getName()` - the name of the function to display in the editor

Since `MyFunction` requires an argument to hold the original trace message value, we also need to create a concept to represent that parameter, which extends the `ConceptFunctionParameter` concept and specifies its type through an overridden `getType()` behavior method:

```java
concept Message_Parameter extends ConceptFunctionParameter implements <none>

  instance can be root: false
  alias: message
  short description: the message to customize before printing to the user

  properties:
  << ... >>

  children:
  << ... >>

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

```java
concept behavior Message_Parameter {

  constructor {
    <no statements>
  }

  public node<Type> getType() {
    overrides ConceptFunctionParameter.getType {
      string2;
    }
  }
}
```

### 45.1.2 Add MyFunction to Script

Once defined, the `MyFunction` concept can be added to `Script`:
This will allow us to edit the function in the `Script` editor:

```plaintext
Script Sample runs as

trace -> Starting... <-

step

trace -> The robot has made its first step <-

while not heading south do
    turnLeft
end

trace -> Looking south now <-

while not wall ahead do
    step
end

trace -> Finished <-
end

trace customization: <no myFunction>
```

When you hit enter, the editor will display the signature of the concept function and you will be able to edit its body:
Notice that the Inspector shows the description messages for the function as well as its parameters, when you place the cursor on the concept function signature.

### 45.1.3 Generator adjustment

The last step that remains is to alter the generator so that the trace message customization can happen. We first need to modify the `KajaFrame` class, which is a super-class for all the classes that get generated from Robot Kaja Scripts:

```java
protected abstract void perform();

public void reportError(String msg) {
    JOptionPane.showMessageDialog(canvas, msg, "Error", JOptionPane.ERROR_MESSAGE);
    stop();
}

public void trace(String msg) {
    JOptionPane.showMessageDialog(canvas, customizeMessage(msg), "Trace", JOptionPane.INFORMATION_MESSAGE);
}
protected String customizeMessage(String msg) {
    return msg;
}
```

The `trace()` method needs to call the new `customizeMessage()` method in order to have the original trace message customized. The default implementation of `customizeMessage()` method returns the message without any alteration.

The generator template that defines how a class generated for a Script should look like, now has to generate a extra method that will override the `customizeMessage()` method in `KajaFrame`:
The overriding method only gets generated when the concept function exists in the `Script`. The generator uses the body of `myFunction` as a body of the generated `customizeMessage()` method.

```java
public static void main(String[] args) {
    map_Script script = new map_Script();
    script.initializeAppComponents();
    script.run();
}

@Override
public string customizeMessage(string message) {
    //$COPY_SRC$[";
}
```

Now the concept function for customizing trace messages should be fully functional:
Script Sample runs as
trace -> Starting... <-

    step
    trace -> The robot has made its first step <-

    while not heading south do
        turnLeft
    end

    trace -> Looking south now <-

    while not wall ahead do
        step
    end

    trace -> Finished <-
end

trace customization:
    customizeMessage(message)->string {
        return "Robot progress report: *" + message + ",*";
    }
Part IV

Languages for IDE integration
Chapter 46

Custom Aspect

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.

46.1 What is a custom aspect?

Language definitions in MPS can be thought of as a collection of aspects: structure, editor, type-system, generator. Each aspect of a language is defined in a separate aspect model. For example, the editor aspect of language L is defined in the L.editor model. The aspect can be generated into a language’s aspect runtime, which represents this aspect at runtime, in other words, when the language is being used inside MPS. The generated classes are then obtained by the corresponding subsystem through LanguageRuntime.getAspect() and can be used further.
46.2 Custom Aspect Descriptor

The IDE part of a language aspect is described by a `SimpleLanguageAspectDescriptor`, which can be created in the plugin model of a language.

| aspect name | The name of an aspect. It will be possible to create a model with this name in other languages and the model will be considered as the model of this aspect for the given other language. |
| main languages | Aspect’s "main" languages are automatically imported into models describing this aspect and concepts from these languages are shown on top of the "new node" menu. |
| additional languages | Not currently implemented. These are the languages that the user might want to use in an aspect model of this kind (but might not). They will be shown in some "hint" on an aspect. This is only for discoverability purposes, e.g. `editor.tables` language might be such an "additional language" for the "editor" aspect. |
| help url | This URL is used to show context help for this concept. |
| icon | Icon to show in the logical view near an aspect model of this kind. |

46.3 Cookbook

The `SimpleLanguageAspectDescriptor` concept covers only the IDE integration of this aspect. Please refer to Custom language aspect cookbook to learn how to implement the generator part to generate a runtime for your aspects and how to use the generated code from the corresponding subsystems.
Chapter 47

Generic placeholders and generic comments

47.1 Generic placeholders

A generic placeholder represents the whitespace between two nodes and can be added to any node collection. The Control/Cmd + Shift + Enter key combination inserts the placeholder at the current position within a collection. The placeholder behaves in a transparent way - you may still invoke the completion menu on the placeholder node to replace it nodes or press Enter to add the usual node in the next sibling position. Using the generic placeholder users of any language can insert arbitrary visual separators (empty lines) into code, even if the language does not support such a concept.

47.2 Generic comments

The generic placeholder may itself contain content. MPS provides the text content for the placeholder in the jetbrains.mps.lang.text language or the general-purpose devkit. This will give you fully editable multiline text language with support for basic styling (bold, italic and underlined), clickable hyper-links and embedded nodes (code). After including the jetbrains.mps.lang.text language or the general-purpose devkit press "[" (open square bracket) on the placeholder. You will a node that allows you to enter and edit text. The text is multiline and consists of words. Any word can be made bold (press Control + B), italic (press Control + I) and underlined (press Control + U). To add a link press Alt + Enter and invoke the Add Link intention. To insert an arbitrary node into the text, invoke code completion inside the text.
node and select "node". The node placeholder will appear, so you may input any sample node. The embedded code can use

```java
public class Printer {
    public State currentState = State.Off;

    // This class should implement a simplified printer workflow. Implementation should follow this workflow (see requirement #)
    // tabular workflow classical printer
    //
    // | state | event Turn on | event Turn off | event Error |
    // |-------|--------------|---------------|-------------|
    // | Off   |              | On            |             |
    // | On    |              | Off           | Failure     |
    // | Failure |              | Off           |             |

    public void transit(Event e) {
        // TODO implement
    }
}
```
Chapter 48

Commenting out nodes

48.1 Generic support for commenting out nodes in MPS

MPS provides a universal way to comment out nodes in models. In previous versions this functionality had to be implemented in all languages separately, either through node attributes or dedicated “comment” nodes. Since MPS 3.3 the information about a node being commented out is stored in the model in a generic way. The smodel language ignores commented out nodes by default so that your queries do not have to filter out commented out nodes explicitly. Additionally, actions have been created to comment/uncomment out a node by hitting "Control/Cmd + /".

You can watch a short screen-cast on generic commenting out that explains the feature and describes the customization options.

48.1.1 How to use it

In the previous versions of MPS language authors had to provide their own implementations of the comment-out functionality for their languages. Thus it may happen that the old language-specific functionality will clash with the new generic functionality of MPS 3.3, especially the keyboard shortcut Control/Cmd + / is now taken by the generic comment-out action and will not work for the specific implementations, if they were using this keyboard shortcut before. It is advisable for the language authors to:

1. choose a different key combination to trigger the specific comment out/uncomment functionality
2. deprecate the custom comment-out functionality
3. customize the generic comment-out functionality
4. provide a migration that automatically replaces usages of the custom comment-out functionality with the generic one
5. eventually remove the custom comment-out functionality

A semi-automated migration process is available in MPS 3.3 to help you migrate painlessly. Please check out the Migrating away from your custom commenting out functionality section below.

You can select any node in MPS, except roots, and press Control/Cmd + “/”. That node will be commented out. Let’s watch some examples:
CHAPTER 48. COMMENTING OUT NODES

The node that you select or point the cursor at will get commented out. Every single non-root node can be commented out - irrespective of whether it occupies a whole line, several lines, or whether it is nested deeply in an expression-like hierarchy.

```plaintext
routine sniffAround means
    turnLeft
    while /*(not mark and not wall ahead) */ looking south do
        step
        if wall ahead do
            turnRight Library call
            sniffAround
        end else do
            pick
            trace -> Found a mark <-
        end
    end
end
```

If you comment out a node, it will be physically removed from its place in the model. If the commented out node occupied a requested child link, an empty cell is provided so that the user can fill in a new child value.
In *BaseLanguage*, for example, this gives you possibilities beyond what the Java parsers allows. You can comment out an `IfStatement`'s condition:

```java
if (/*count == 1*/ count == 0) {
}
```

a method parameter:

```java
public static void hello(/*int*/ a/*, boolean b) {
  <no statements>
}
```

or a variable type:

```java
/*int*/ double a = 0.6;
```

To give another example, the editor definition language allows you to comment out an editor cell, for example:

```
[> /*need */ require ( % library % -> { name } ) ]<
```

To uncomment a commented out node you simply press Control/Cmd + "/" while positioned on it.

### 48.1.2 Smart commenting out

The comment and uncomment actions have some intelligence in them in order to decide, which node to comment/uncomment.

- if a node or a set of nodes is selected, this node/nodes will be commented out/uncommented
- if no node is selected, the editor attempts to comment out/uncomment the current "line" - to achieve this a search starts in the node under caret to identify the closest ancestor vertical collection and an ancestor of the node under caret that is a member of this vertical collection will be commented out/uncommented
You may get finer-grained control over the mechanism of detecting the ancestor line of the node under caret - simply define a handler for the `COMMENT` action on a collection of cells as follows:

```
<default> editor for concept StaticMethodDeclaration
node cell layout:
[-] # DeprecatedPart:
  # BaseAnnotation_AnnotationComponent # _Component.Visibility # ? final "static" ? synchronized ? native
  ?(- ? modifiers $ /empty cell: <default> -) ?# GenericDeclaration_TypeVariables_Component # ?# returnType $
  ?# BaseMethodDeclaration_NameCellComponent # { ?# $ parameter $ /empty cell: <default> -) $ ?
  ?(- ? throws $ /empty cell: <default> -) $ ?
  }[-] ?# BaseMethodDeclaration_BodyComponent # ?-
  ]

<table>
<thead>
<tr>
<th>Structure</th>
<th>Editor</th>
<th>Behavior</th>
<th>Typesystem</th>
<th>Actions</th>
<th>Refactorings</th>
<th>Generator</th>
<th>Nextgen</th>
</tr>
</thead>
<tbody>
<tr>
<td>brains.mps.lang.editor.structure.CellModel_RefNodeList</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Style: | <no base style> { 
    | } |
| Common: |
| cell id | <default> |
| keymap | <default> |
| action map | _InstanceMethodDeclaration_CommentParameters |
| menu | <none> |
| attracts | focus noAttraction |
| show if | <no condition> |

**48.1.3 How does it work in the model**

When a node is commented out, it is placed as a child (wrapped) in a special “child attribute”, called `BaseCommentAttribute`. Then the instance of this attribute is attached to the commented node’s link in the former parent of the commented node. A `ChildAttribute` is same as the `LinkAttribute` concept, except that `ChildAttribute` gets attached to aggregation links. So the commented nodes are not stored as the usual children, and they won’t appear in queries like `node.children`, `node.descendants`, etc.
CHAPTER 48. COMMENTING OUT NODES

The MPS editor knows about comments and it will draw children as well as the commented out nodes, in this role.

The BaseCommentAttribute annotation comes from jetbrains.mps.lang.core, so this language needs to be listed among used languages in models that contain commented out nodes.

48.1.4 Querying for commented nodes

The smodel language gives you options to query for the commented nodes. You use the same syntax that works for any attributes, only that the comment attribute allows for parametrization by the containment link. For example, if a node has a child collection named commands, querying whether any of the commands children has been commented out would look like:

```
node.@comment<commands>.isNotEmpty
```

48.1.5 Customization

By default every commented node is drawn surrounded by /* */. You can override the visual appearance of a commented out node by defining a custom commented editor for the concept. Just define the usual editor with the hint “comment”:
For the comment hint to be available, your editor model needs to import the `jetbrains.mps.lang.core.editor` model.

The style of the editor should be changed so that the user can easily visually distinguish commented code.

You can either re-use the pre-defined Comment style, which uses a gray color with italics style, or you may create your own style for commented out nodes.

**Note:** The children of the commented node should be drawn with their usual editor so you need to remove the comment hint in child cells:
**Style:**

```plaintext
<no base style> {
   << ... >>
}
```

**Common:**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cell id</td>
<td>&lt;default&gt;</td>
</tr>
<tr>
<td>action map</td>
<td>&lt;default&gt;</td>
</tr>
<tr>
<td>keymap</td>
<td>&lt;default&gt;</td>
</tr>
<tr>
<td>menu</td>
<td>&lt;none&gt;</td>
</tr>
<tr>
<td>attracts focus</td>
<td>noAttraction</td>
</tr>
<tr>
<td>show if</td>
<td>&lt;no condition&gt;</td>
</tr>
</tbody>
</table>

**Cell_collection:**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cell layout</td>
<td>indent</td>
</tr>
<tr>
<td>uses braces</td>
<td>false</td>
</tr>
<tr>
<td>uses folding</td>
<td>false</td>
</tr>
<tr>
<td>add context hints</td>
<td>&lt;no addHints&gt;</td>
</tr>
<tr>
<td>remove context hints</td>
<td>comment</td>
</tr>
</tbody>
</table>
48.1.6 Easier customization

The next applicable editor cell gives you a more convenient way to customize the look of commented out nodes - you may address several concepts in a hierarchy with a single customized comment editor.

The next applicable editor cell simply removes the comment hint and redirects the request to find the original editor of the concept (IfStatement). This avoids the need for repetition of the editor definition. You may further simplify the task, if you define a single editor bound to the comment hint for a common super-concept - this way all sub-concepts will get the customized comment editor.

48.1.7 Commenting out/uncommenting nodes from code

The CommentUtil class from jetbrains.mps.editor.runtime can be used to comment out and uncomment nodes from code, such as actions, intentions or key maps. This gives you options to further customize the behavior of commented out nodes.

The CellAction_CommentOrUncomment and its inheritors class come from the same package. They give you the way to simply comment the node and restore the selection or uncomment the node if it is currently commented.

48.1.8 The Comment editor action

The response to the comment/uncomment action can also be customized on the node level. You can set the handler for COMMENT action in the cell's action map:

```java
action map DisableCommentingConditionInWhile

applicable concept: while

actions:

  action COMMENT description: <no description>
  can execute: (editorContext, node) -> boolean {
    if (!editorContext.getSelectedNode().eq(node.condition)) { return false; }
    return new CellAction_CommentOrUncommentNode(node).canExecute(editorContext);
  }

  execute: (editorContext, node) -> void {
    new CellAction_CommentOrUncommentNode(node).execute(editorContext);
  }
```

For example, if we want to prevent the user from commenting out conditions in the robot's Kaja While statement, we attach the above action map to the While editor's cell representing the condition:
CHAPTER 48. COMMENTING OUT NODES

Since the `COMMENT` action is customized it will do what is indicated. The action will work only if the `condition` node is selected. Since we create the `CellAction_CommentOrUncommentNode` with the `node` as the parameter, where the `node` is the `While` statement, the action will process the `While` statement:

1) If it is not commented, the action will comment it out.
2) If it is commented out, the action will uncomment it.

Thus the commenting of the `condition` will be disabled.

48.2 Migrating away from your custom commenting out functionality

In versions prior to MPS 3.3 language authors had to implement the comment out functionality themselves for each language individually. In MPS 3.3 the custom functionality may be redundant and should be replaced by the generic functionality provided by MPS, perhaps with some customization as described above. The existing usages of the old custom commenting-out functionality should be migrated to the generic version, which should be done in several steps:

1. Your old concepts used for commenting out should be deprecated
2. Your keyboard shortcuts, actions and intentions for commenting out/uncommenting should be deprecated or removed
3. You may wish to customize the look of commented out nodes by defining custom editors attached to the "comment" editor hint (as described above)
4. You may also wish to disable the generic comment out functionality on some editor cells (as described above)
5. You may need to provide a migration that will automatically translate usages of your old custom commented nodes in user code into nodes commented in the generic way. This can be done either fully manually or with MPS assistance.
48.2.1 MPS-assisted migration

MPS can create a Migration for you, provided you indicate, which concepts represent the old custom comments using the IOldCommentAnnotation and IOldCommentContainer. Since there were two typical ways to create custom comments in the past, there need to be two interface:

- IOldCommentAnnotation - should be implemented by the NodeAttribute that indicates a node is commented out, if attributes were used annotate commented out nodes

- IOldCommentContainer - in case commented nodes were represented by a dedicated concept, such as SingleLineComment, these dedicated concepts should be marked with this concept interface

These marker concept interfaces come from jetbrains.mps.lang.core, so this language needs to extended by your language in order to use them. Once annotated, the generic comment-out functionality will be ceased on the nodes of these concepts in favor of the old custom comment out functionality.

Additionally, the old commenting-out concepts will have warnings reported on them - Old comment container should be migrated or Old comment annotation should be migrated. The quick-fixes for these warnings will create the necessary migrations to convert your old custom commenting-out scheme into the generic one painlessly. Just trigger the quick fixes, check the generated migrations and then migrate your projects.

48.2.2 Fully manual migration

You may create the migration fully manually. Typically all that your migration needs to do is to find all nodes being commented out in the old custom way, uncomment them and call CommentUtil.comment() on each node to get it commented out in the new way. The CommentUtil class comes from jetbrains.mps.editor.runtime.

The generic comment out functionality marks commented out nodes with the BaseCommentAttribute annotation that is attached to the parent of the commented-out node, holds the original role of the commented out node and comes from jetbrains.mps.lang.core, so this language needs to be used in models that contain commented out nodes. An automatic migration should add such language dependency to all altered models. You may take inspiration from The ReplaceSingleLineCommentsWithGenericComments migration, which migrated SingleLineComment nodes in BaseLanguage:

```java
public static void addDependencyOnCoreIfMissing(SModel currentModel) {
  if (!currentModel instanceof SModelInternal) { return; }
  SModelInternal model = (SModelInternal) currentModel;
  SLanguage langCoreLang = language/jetbrains.mps.lang.core; /
  if (!model.importedLanguageIds().contains(langCoreLang)) {
    model.addLanguage(langCoreLang);
  }
}
```
Chapter 49

Plugin

Plugin is a way to integrate your code with the MPS IDE functionality. The `jetbrains.mps.lang.plugin` and `jetbrains.mps.lang.plugin.standalone` languages give you a number of root concepts that can be used in your plugin. This chapter describes all of them.

49.1 Plugin instantiation

While developing a plugin, you have a solution holding the plugin and want the plugin classes to be automatically reloadable so as not to have to restart MPS after each change to see its effect. To set the development phase correctly, do the following:

1. Create a new solution for your plugin
2. Create a model in this solution named `<solution_name>.plugin`
3. Import `j.m.lang.plugin` and `j.m.lang.plugin.standalone` languages into the solution and the model
4. Create a root `StandalonePluginDescriptor` in the model (it comes from the `j.m.lang.plugin.standalone` language)
5. Set the solution's Solution Kind to Other

You can now edit your plugin model and see the changes applied to the current MPS instance just after generation. You can also distribute the solution and have the plugin successfully working for the users.

49.2 Actions and action groups

One can add custom actions to any menu in MPS by using `action` and `action group` entities.
An action describes one concrete action. Action groups are named lists of actions intended for structuring of actions - adding them to other groups and MPS groups (which represent menus themselves) and combining them into popup menus. You can also create groups with dynamically changing contents.

How to add new actions to existing groups?
In order to add new actions to existing groups, the following should be done:

1. actions should be described
2. described actions should be composed into groups
3. these groups should be added to existing groups (e.g. to predefined MPS groups to add new actions to MPS menus).

Predefined MPS groups are stored in the jetbrains.mps.ide.actions model, which is an accessory model to jetbrains.mps.lang.plugin language, so you don’t need to import it explicitly into your model.

Action structure

Action properties

- **Name** - The name of an action. You can give any name you want, the only obvious constraint is that the names must be unique in the scope of the model.

- **Mnemonic** - If mnemonic is specified, the action will be available via the alt+mnemonic shortcut when any group that contains this action is displayed. Note that the mnemonic (if specified) must be one of the chars in action’s caption. Mnemonic is displayed as an underlined symbol in the action’s caption.

- **Execute outside command** - All operations with MPS models are executed within commands. A command is an item in the undo list (you don’t control it manually, MPS does it for you), so the user can undo changes brought into the model by action’s execution. Also, all the code executed in a command, has read-write access to the model. The catch is that if you show visual dialogs to the user from inside of a command, it can cause a deadlock by blocking while holding the read/write locks. It is thus recommended to have the execute outside command option set to false, only if you are not using UI in your action. Otherwise it should be set to true and proper read/write access locking should be performed manually with the read action and command statements within the action.

```java
action ExtractMethod {
    mnemonic: <no mnemonic>
    execute outside command: true
    also available in: << ... >>
    caption: Extract Method
    description: <no description>
    icon: <no icon>

    action context parameters ( always visible = false )
    all nodes key: NODES required
    EditorContext context key: EDITOR_CONTEXT required
    EditorComponent editorComponent key: EDITOR_COMPONENT required
    Project project key: PROJECT required

    isApplicable(event) -> boolean {
        !ReadOnlyUtil.isCellsReadOnlyInEditor(this.editorComponent,
            this.nodes.select({-it => (EditorCell) this.editorComponent.findNodeCell(it); })) &
            ExtractMethodFactory.isRefactoringAvailable(this.nodes);
    }

    execute(event) -> void {
        FeatureUsageTracker.getInstance().triggerFeatureUsed("refactoring.extractMethod");
        ExtractMethodRefactoringParameters params;
        ExtractMethodRefactoring refactoring;
        this.context.getRepository().getModelAccess().runWriteAction("refactoring.extractMethod",
            params = ExtractMethodFactory.createParameters()).runReadAction(Runnable::void member o,
            runread = ExtractMethodFactory.createRefactoring(params.setReturnType(refactoring.getMethodType()),
            runre = runWriteAction(Runnable::void member o,
            runreadInEDT = runWriteInEDT(Runnable::void member o,
            runPost = ExtractMethodDialog dialog = new ExtractMethodDialog(this.project, this.context, params, runPost, runDialog.show());
    }

    additional methods
    << ... >>

```
**Caption** - the string representing the action in menus

**Description** - this string (if specified) will be displayed in the status bar when this action is active (selected in any menu)

**Icon** - this icon will be displayed near the action in all menus. You can select the icon file by pressing the "..." button.

Note that the icon must be placed **near your language** (because it’s stored not as an image, but as a path relative to the language’s root)

**Construction parameters** Each action can be parameterized at construction time using **construction parameters**. This can be any data determining action’s behavior. Thus, a single action that uses construction parameters can represent multiple different behaviors. To manage actions and handle keymaps MPS needs a unique identifier for each concrete behavior represented by an action. So, the **toString** function was introduced for each construction parameter (can be seen in the inspector). For primitive types there is no need to specify this function explicitly - MPS can do it automatically. For more complex parameters, you need to write this function explicitly so that for each concrete behavior of an action there is a different set of values returned from the toString() functions.

**Enable/disable action control** Is always visible flag - if you want your action to be visible even in the disabled state (when the action is not applicable in the current context), set this to true, otherwise to false.

**Context parameters** - specifies which items must be present in the current context for the action to be able to execute. They are extracted from the context before any action’s method is executed. Context parameters have conditions associated with them - **required** and **custom** are the two most frequently used ones. If some **required** parameters were not extracted, the action state is set to **disabled** and the **isApplicable/update/execute** methods are not executed. If all required action parameters were extracted, you can use their values in all the action methods. **Custom** context parameters give you the option to decide whether the context parameter is mandatory on case-by-case basis using the supplied function.

There are 2 types of action parameters - **simple** and **complex** action parameters.

- **Simple** action parameters (represented by **ActionDataParameterDeclaration**) allow to simply extract all available data from the current data context. The data is provided "by key", so you should specify the **name** and the **key** in the declaration. The type of the parameter will be set automatically.

- **Complex** action parameters (represented by **ActionParameterDeclaration**) were introduced to perform some frequently used checks and typecasts. Now there are 3 types available for the context parameter of this type:
  - **node<concept>** - currently selected node, which is an instance of a specified concept. Action won’t be enabled, if the selected node isn’t an instance of this concept.
  - **nlist<concept>** - currently selected nodes. It is checked that all nodes are instances of the **concept** (if specified). As with **node<concept>**, the action won’t be enabled if the check fails.
  - **model** - the current model holding the selected node

The available **keys** that the user can type into the **context parameters** declaration are obtained automatically from all imported models. MPS searches the imported models for subclasses of the **CommonDataKeys** (**com.intellij.openapi.actionSystem**) class. Typical such subclasses are:

- **CommonDataKeys** (**com.intellij.openapi.actionSystem**)
- **PlatformDataKeys** (**com.intellij.openapi.actionSystem**)
- **MPSCommonDataKeys** (**jetbrains.mps.ide.actions**)
- **MPSEditorDataKeys** (**jetbrains.mps.ide.editor**)
- **MPSDataKeys** (**jetbrains.mps.workbench**)

Be sure to import these models (**Control/Cmd + M**) in order to see them in the completion menu for **context parameters**.

**Is Applicable / update** - In cooperation with the context parameters, this method controls the **enabled/disabled** state of the action. You can pick either of the two options:

- The **isApplicable** method returns the new state of an action

- The **update** method is designed to update the state manually. You can also update any of your action’s properties (caption, icon etc.) by accessing action’s presentation via **event.getPresentation()**. Call the the **setEnabledState()** method on an action to enable or disable it manually.

These methods are executed only if all **required** context parameters have been successfully extracted from the context. Note: The **this** keyword refers to the current action, use action<...> to get hold of any visible action from your code.
Note

Do not use the `isApplicable()` method if you want to modify the presentation manually. Although no errors would be reported from within `isApplicable()`, it is not guaranteed to work in all cases properly. The `update()` method is a more suitable place for complex presentation manipulations.

Execute - this method is executed when the action is performed. It is guaranteed that it is executed only if the action’s `update` method for the same event left the action in active state (or `isApplicable` returned `true`) and all the required context parameters are present in the current context and were filled in.

Methods - in this section you can declare utility methods.

Group structure

Group describes a set of actions and provides the information about how to modify other groups with current group.

Presentation Name - The name of the group. You can give any name you want, the only obvious constraint is that the names must be unique in the scope of the model.

is popup - if this is true, the group represents a popup menu, otherwise it represents a list of actions.

When "is popup" is true:

- **Caption** - string that will be displayed as the name of the popup menu
- **Mnemonic** - if mnemonic is specified, the popup menu will be available via the `alt+mnemonic` shortcut when any group that contains it is displayed. Note that the mnemonic (if specified) must be one of the chars in `caption`. Mnemonic is displayed as an underlined symbol in the popup menu caption.
- **Is invisible when disabled** - if set to true, the group will not be shown in case it has no enabled actions or is disabled manually in the `update()` method. Call the `enable()/disable()` methods on an action group to enable or disable it manually.

Contents

There are 3 possibilities to describe group contents:

Element list - this is just a static list of actions, groups and labels (see modifications). The available elements are:

- `->name` - an anchors. Anchors are used for modifying one group with another. See Add statement section for details.
- `<—>` - separator
- `ActionName[parameters]` - an actions.

Build - this alternative should be used in groups, the contents of which is static, but depends on some initial conditions - the group is built once and is not updated ever after. Use the `add` statement to add elements inside build block.

Update - this goes for dynamically changing groups. Group is updated every time right before it is rendered.

Modifications and labels

Add to `<group>` at position `<position>` - this statement adds the current group to a `<group>` at the given position. Every group has a `<default>` position, which tells to add the current group to the end of the target group. Some groups can provide additional positions by adding so-called anchors into themselves. Adding anchors is described in the contents section. The anchor itself is invisible and represents a position, in which a group can be inserted.

Note

In the update/build blocks use the `add` statement to add group members.

Modifications and labels

Add to `<group>` at position `<position>` - this statement adds the current group to a `<group>` at the given position. Every group has a `<default>` position, which tells to add the current group to the end of the target group. Some groups can provide additional positions by adding so-called anchors into themselves. Adding anchors is described in the contents section. The anchor itself is invisible and represents a position, in which a group can be inserted.

Note

- You shouldn’t care about the group creation order and modifications order - this statement is fully declarative.
- If A is added into B, B into C, C will contain A
actionGroup <...> expression

There is a specific expression available in the jetbrains.mps.lang.plugin language to access any registered group - actionGroup<group> expression.

Bootstrap groups  Bootstrap groups are a way to work with action groups that have been defined outside of MPS (e.g. groups contributed by IDEA or some IDEA plugin). In this case, a bootstrap group is defined in MPS and its internal ID is set to the ID of the external group. After having this done, you can work with the bootstrap group just like with a normal one - insert it into your groups and vice versa. A regular user rarely needs to use bootstrap groups.

A quick and simple tutorial by Federico Tomassetti on how to create action and show it in a context menu is available here: http://www.federico-tomassetti.it/tutorial-how-to-add-an-action-to-the-jetbrains-metaprogramming-system/

Please bear in mind that this tutorial uses an older version of MPS and the actual workings in MPS have changed since then. Especially we now recommend to use plugin solutions instead of the plugin aspect of a language to hold your actions. The tutorial may still give you some guidelines and useful insight.

49.2.1 Displaying progress indicators

Long-lasting actions should indicate their activity and progress to the user. Check out the Progress indicators page for details on how to use progress bars, how to allow for cancellation and how to enable actions for running in the background.

49.3 KeyMap Changes

The KeymapChangesDeclaration concept allows the plugin to assign key shortcuts to individual actions and group them into shortcuts schemes. Any action can have a number of keyboard shortcuts. This can be specified using the KeyMapChanges concept. For a parameterized action, which has a number of "instances" (one instance per parameter value), a function can be specified, which returns different shortcut for a different parameter value. In MPS, there are some "default keymaps", which you can see in Settings->Keymaps. The for keymap section allows you to specify a keycap that the KeyMapChanges definition is contributing to. E.g. one can set different shortcuts for the same action in the MacOS and the Windows keymaps.

Default Keymap

If you add a keyboard shortcut to the Default keymap, all keymaps are altered with this shortcut.

MacOs

Note that by default ctrl is changed to cmd in MacOS keymap. If you want your action to have a ctrl + something shortcut on MacOS, you should re-define this shortcut for the MacOS keymap.

All the actions added by plugins are visible in Settings->Keymap and Settings->Menus and Toolbars. This means that any user can customize the shortcuts used for all MPS actions.

keymap changes Mac for Mac OS X

```
GoToBookmark          <ctrl>+<VK_0>
NewElement            <ctrl>+<VK_N>
SafeDelete            <meta>+<VK_DELETE>
ShowClassInHierarchy  <ctrl>+<VK_H>
ShowConceptInHierarchy <ctrl>+<VK_H>
ShowNodeInfo          <ctrl>+<VK_Q>
GoToBookmark          (num) { addKeystroke("ctrl " + num); }  
GoByCurrentReferenceToIDEA  <ctrl>+<VK_B>
```
A KeyMap Change should be given a name unique within the model, it must specify the Keymap that is being altered (or Default to change all keymaps) and then assign a keystroke to actions that should have one. The keystroke can either be SimpleShortcutChange with a directly specified keystroke or ParametrizedShortcutChange, which gives you the ability to handle parametrized actions.

### 49.4 NonDumbAwareActions

If your action uses platform indices (which is very rare), add it to NonDumbAwareActions. Those actions will be automatically disabled while the indices are being build.

### 49.5 Editor Tabs

If you look at any concept declaration you will certainly notice the tabs at the bottom of the editor. You are able to add the same functionality to the concepts from your language. What is the meaning of these tabs? The answer is pretty simple - they contain the editors for some aspects of the "base" node. Each tab can be either single-tabbed (which means that only one node is displayed in it, e.g. editor tab) or multi-tabbed (if multiple nodes can be created for this aspect of the base node, see the Typesystem tab, for example). How the editor for a node is created? When you open some node, call it N, MPS tries to find the "base" node for N. If there isn’t any base node, MPS just opens the editor for the selected node. If the node is found (call it B), MPS opens some tabs for it, containing editors for some subordinate nodes. Then it selects the tab for N and sets the top icon and caption corresponding to B.

When you create tabbed editors, you actually provide rules for:

- finding the base node
- finding subordinate nodes
- optionally an algorithms of subordinate nodes creation

The tabs that match the requested base concept are displayed and organized depending on their relative order rules specified in their respective order constraints sections.

**Editor Tab Structure**

- **Name** - The name of the rule. You can give any name you want, the only obvious constraint is that the names must be unique in the scope of the model.
- **Icon** - this icon will be displayed in the header of the tab. You can select the icon file by pressing the "..." button. Note that the icon must be placed near your language (because it’s stored not as an image, but as a path relative to the language’s root)
- **Shortcut char** - a char to quickly navigate to the tab using the keyboard
- **Order constraints** - an instance of the Order concept. Orders specify an order, in which the current tab should be displayed relative to the other tabs. You can either refer to an external order or specify one in-place.
- **Base node concept** - the concept of the base node for this as well as all the related tabs.
- **Base Node** - this is a rule for searching for the base node given a known node. It should return null, if the base node is not found or this TabbedEditor can’t be applied.
- **Is applicable** - indicates whether the tab can be used for the given base node
- **command** - indicated whether the node creation should be performed as a command, i.e. whether it should be undoable and uses no additional UI interaction with the user.
- **getNode/getNodes** - should return the node or a list of nodes to edit in this tab
- **getConcepts** - return the concepts of nodes that this tab can be used for to edit
- **Create** - if specified, this will be executed when user asks to create a new node from this tab. It is given a requested concept and the base node as parameters.

### 49.6 Tools

Tool is an instrument that has a graphical presentation and aimed to perform some specific tasks. For example, Usages View, Todo Viewer, Model and Module repository Viewers are all tools. MPS has rich UI support for tools - you can move it by drag-and-drop from one edge of the window to another, hide, show and perform many other actions. Tools are created "per project". They are initialized/disposed on class reloading (after language generation, on "reload all" action etc.)
CHAPTER 49. PLUGIN

Tool structure

**Name** - The name of the tool. You can give any name you want, the only obvious constraint is that the names must be unique in the scope of the model.

**Caption** - this string will be displayed in tool’s header and on the tool’s button in tools pane

**Number** - if specified, `alt+number` becomes a shortcut for showing this tool (if it’s available)

**Icon** - the icon to be displayed on the tool’s button. You can select the icon file by pressing "..." button. Note that the icon must be placed near your language (because it’s stored not as an image, but as a path relative to the language’s root)

**Position** - on of `top/bottom/left/right` to add the tool to the desired MPS tool bar

**Init** - initialize the tool instance here

**Dispose** - dispose all the tool resources here

**getComponent** - should return a Swing component (instance of a class which extends JComponent) to display inside the tool’s window. If you are planning to create tabs in your tool and you are familiar with the tools framework in IDEA, it’s better to use IDEA’s support for tabs. Using this framework greatly improves tabs functionality and UI.

**Fields and methods** - regular fields and methods, you can use them in your tool and in the external code.

Tool operation

We added the operation (GetToolInProjectOperation concept) to simply access a tool in some project. Use it as `project.tool<toolName>`, where `project` is an IDEA Project. Do not forget to import the `jetbrains.mps.lang.plugin.standalone` language to be able to use it.

> Be careful

This operation can’t currently be used in the `dispose()` method

49.7 Tabbed Tools

It’s same as tool window, but additionally can contain multiple tabs

Tool structure

**Name** - The name of the tool. You can give any name you want, the only obvious constraint is that the names must be unique in the scope of the model.

**Caption** - this string will be displayed in tool’s header and on the tool’s button in tools pane

**Number** - if specified, `alt+number` becomes a shortcut for showing this tool (if it’s available)

**Icon** - the icon to be displayed on the tool’s button. You can select the icon file by pressing "..." button. Note that the icon must be placed near your language (because it’s stored not as an image, but as a path relative to the language’s root)

**Position** - on of `top/bottom/left/right` to add the tool to the desired MPS tool bar

**Init** - initialize the tool instance here

**Dispose** - dispose all the tool resources here

**Fields and methods** - regular fields and methods, you can use them in your tool and in the external code.

49.8 Preferences components

Sometimes you may want to be able to edit and save some settings (e.g. your tools’ settings) between MPS startups. We have introduced **preferences components** for these purposes.

Each preferences component includes a number of preferences pages and a number of persistent fields. Preferences page is a dialog for editing user preferences. They are accessible through File->Settings.

Persistent fields are saved to the `.iws` files when the project is closed and restored from them on project open. The saving process uses reflection, so you don’t need to care about serialization/deserialization in most cases.

> Note

Only primitive types and non-abstract classes can be used as types of persistent fields. If you want to store some complex data, create a persistent field of type `org.jdom.Element` (do not forget to import the model `org.jdom`), annotate it with `com.intellij.util.xmlb.annotations.Tag` and serialize/deserialize your data manually after read / before write
Preferences component structure

**name** - component name. You can give any name you want, the only obvious constraint is that the names must be unique in the scope of the model. **fields** - these are the persistent fields. They are initialized before **after read** and pages creation, so their values will be correct in every moment they can be accessed. They can have default values specified, as well. **after read** / **before write** - these blocks are used for custom serialization purposes and for applying/collecting preferences, which have no corresponding preferences pages (e.g. tool dimensions) **pages** - preferences pages

Preferences page structure

**name** - the string to be used as a caption in **Settings** page. The name must be unique within a model. **component** - a UI component to edit preferences. **icon** - the icon to show in **Settings** window. The size of the icon can be up to 32x32 **reset** - reset the preferences values in the UI component when this method is called. **commit** - in this method preferences should be collected from the UI component and committed to wherever they are used. **isModified** - if this method returns false, **commit** won’t be executed. This is typically useful for preferences pages with long-running commit method.

PreferenceComponent expression

We added an expression to simply access a PreferenceComponent in some project. You can access it as project.preferenceComponent<componentName>, where project is an IDEA Project. Do not forget to import the jetbrains.mps.lang.plugin.standalone language to use it. **Hint**

The uiLanguage components can be used here

**Be careful**

This operation can’t currently be used in the dispose() method

49.9 Custom plugin parts (ProjectPlugin, ApplicationPlugin)

Custom plugin parts are custom actions performed on plugin initializing/disposing. They behave exactly like plugins. You can create as many custom plugins for your language as you want. There are two types of custom plugins - project and application custom plugins. The project custom plugin is instantiated once per project, while the application custom plugin is instantiated once per application and therefore it doesn’t have a project parameter.
Chapter 50

Find usages

In MPS, any model consists of nodes. Nodes can have many types of relations. These relations may be expressed in a node structure (e.g. "class descendants" relation on classes) or not (e.g. "overriding method" relation on methods). Find Usages is a tool to display some specifically related nodes for a given node.

In MPS, the Find Usages system is fully customizable - you can write your own entities, so-called finders, which represent algorithms for finding related nodes. For every type of relation there is a corresponding finder.

This is how "find usages" result looks like:

![Find Usages Subsystem](image)

50.0.1 Using Find Usages Subsystem

You can press Alt+F7 on a node (no matter where - in the editor or in the project tree) to see what kind of usages MPS can search for.

You can also right-click a node and select "Find Usages" to open the "Find usages" window.
Finders - select the categories of usages you want to search for
Scope - this lets you select where you want to search for usages - in concrete model, module, current project or everywhere.
View Options - additional view options
After adjusting your search, click OK to run it. Results will be shown in the Find Usages Tool as shown above.

50.0.2 Finders

To implement your own mechanism for finding related nodes, you should become familiar with Finders. For every relation there is a specific Finder that provides all the information about the search process.

Where to store my finders?
Finders can be created in any model by importing findUsages language. However, MPS collects finders only from findUsages language aspects. So, if you want your finder to be used by the MPS Find Usages subsystem, it must be stored in the findUsages aspect of your language.

Finder structure

<table>
<thead>
<tr>
<th>name</th>
<th>The name of a finder. You can choose any name you want, the only obvious constraint being that the names must be unique in the scope of the model.</th>
</tr>
</thead>
<tbody>
<tr>
<td>for concept</td>
<td>Finder will be tested for applicability only to those nodes that are instances of this concept and its subconcepts.</td>
</tr>
<tr>
<td>description</td>
<td>This string represents the finder in the list of finders. Should be rather short.</td>
</tr>
<tr>
<td>long description</td>
<td>If it’s not clear from the description string what exactly the finder does, you can add a long description, which will be shown as a tooltip for the finder in the list of finders.</td>
</tr>
<tr>
<td>is visible</td>
<td>Determines whether the finder is visible for the current node. For example, a finder that finds ancestor classes of some class should not be visible when this class has no parent.</td>
</tr>
</tbody>
</table>
Finders that have passed `for concept` are tested for applicability to the current node. If this method returns true, the finder is shown in the list of available finders; otherwise it is not shown. The `node` argument of this method is guaranteed to be an instance of the concept specified in "for concept" or its subconcepts.

Please note the difference between `is visible` and `is applicable`. The first one is responsible only for viewing. The second one represents a "valid call" contract between the finder and its caller. This is important because we have an `execute` statement in findUsagesLanguage, which will be described later. See `execute` section below for details.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>is applicable</code></td>
<td>Finders that have passed <code>for concept</code> are tested for applicability to the current node. If this method returns true, the finder is shown in the list of available finders; otherwise it is not shown. The <code>node</code> argument of this method is guaranteed to be an instance of the concept specified in &quot;for concept&quot; or its subconcepts. Please note the difference between <code>is visible</code> and <code>is applicable</code>. The first one is responsible only for viewing. The second one represents a &quot;valid call&quot; contract between the finder and its caller. This is important because we have an <code>execute</code> statement in findUsagesLanguage, which will be described later. See <code>execute</code> section below for details.</td>
</tr>
<tr>
<td><code>find</code></td>
<td>This method should find given node usages in a given scope. For each found usage, use the <code>add result</code> statement to register it.</td>
</tr>
<tr>
<td><code>searched nodes</code></td>
<td>This method returns nodes for which the finder searched. These nodes are shown in <code>searched nodes</code> subtree in the tool. For each node to display, use the <code>add node</code> statement to register it.</td>
</tr>
<tr>
<td><code>get category</code></td>
<td>There are a number of variants to group found nodes in the tool. One of them is grouping by category, that is given for every found node by the finder that has found it. This method gives a category to each node found by this finder.</td>
</tr>
</tbody>
</table>

What does the MPS Find Usages subsystem do automatically?

- Stores search options between multiple invocations and between MPS runs
- Stores search results between MPS runs
- Automatically handles deleted nodes
- All the visualization and operations with found nodes is done by the subsystem, not by finders

### 50.0.3 Specific Statements

**execute**

Finders can be reused thanks to the `execute` statement. The execution of this statement consists of 2 steps: validating the search query (checking `for concept` and `isApplicable`), and executing the `find` method. That’s where you can see the difference between `isApplicable` and `isShown`. If you use `isApplicable` for cases when the finder should be applicable, but not shown, you can get an error when using this finder in the `execute` statement.

### 50.0.4 Examples

You can see some finder examples in `jetbrains.mps.baseLanguage.findUsages` You can also find all finders by going to the FinderDeclaration concept (Ctrl+N, type "FinderDeclaration", then press ENTER) and finding all instances of this concept (Alt+F7, check instances, then check Global Scope).
Chapter 51

Suppressing Errors

One of very effective ways to maintain high quality of code in MPS is the instant on-the-fly code analysis that highlights errors, warnings or potential problems directly in code. Just like with other code quality reporting tools, it is essential for the user to be able to mark false positives so that they are not reported repeatedly. MPS now provides the language developers with a customizable way to suppress errors in their languages. This functionality was used to implement Suppress Errors intention for BaseLanguage:

One place where this feature is also useful are the generators, since type errors, for example, are sometimes unavoidable in the templates.

If a node is an instance of a concept, which implements the ISuppressErrors interface, all issues on this node and all its children won’t be shown. For example, comments in BaseLanguage implement ISuppressErrors. It is also possible to define child roles, in which issues should be suppressed, by overriding the boolean method suppress(node<> child) of the ISuppressErrors interface.

Additionally, if a node has an attribute of a concept that implements ISuppressErrors, issues in such node will be suppressed too. There is a convenience default implementation of an ISuppressErrors node attribute called SuppressErrorsAttribute. It can be applied to only those nodes that are instances of ICanSuppressErrors.

51.0.5 An example of using the SuppressErrorsAttribute attribute and the corresponding intention.

There is an error in editor:

```java
int a = "";
```

BaseLanguage Statement implements ICanSuppressErrors, so the user can apply the highlighted intention here:

```java
int a = "";
```

Now the error isn’t highlighted any longer, but there is a newly added cross icon in the left pane. The SuppressErrorsAttribute can be removed either by pressing that cross or by applying the corresponding intention.
Chapter 52

Debugger

MPS provides an API for creating custom debuggers as well as integrating with debugger for java. See Using MPS Debugger for a description of the MPS debugger features.

52.0.6 The fundamentals

In order to debug code that gets generated from the user models, MPS needs to:

- track nodes in user models down to the generated code, in order to be able to match the two worlds seamlessly in the debugger
- understand, which types of breakpoints can be created on what nodes
- know the options for starting the debugged code in the debugger
- optionally also have a set of customized viewers to display the current values of data in memory of the debugged program to the user

MPS tries to automate as much of it as possible, however, in some scenarios the language designer also has to do her share of weight-lifting. Suppose you have a language, let’s call it high.level, which generates code in some language low.level, which in turn is generated directly into text (there can be several other steps between high.level and low.level). Suppose that the text generated from low.level consists of java classes, and you want to have your high.level language integrated with MPS java debugger engine. See the following explanatory table:

| high.level extends or generates into BaseLanguage | high.level does not extend nor generates into BaseLanguage |

52.0.7 Debugging BaseLanguage and its extensions - integration with the java debugger

To integrate your BaseLanguage-generated language with the MPS java debugger engine, you rarely need to specify anything. MPS can keep track of the generation trace in the trace.info files, so breakpoints can be set as expected and the debugger correctly steps through your DSL code.

The automatic tracing recognizes situations, when a node gets transformed through a reduction rule, and keeps a tracking record of the transformation in an appropriate model’s trace.info file. For concepts that do not get reduced through their own reduction rules, you may, however, indicate explicitly, which part of the generated code should be preserved in the trace.info file. The $TRACES$ macro serves this purpose.

See Debugger for more details and an example on the $TRACES$ macro usage.

Startup of a run configuration under java debugger

MPS provides a special language for creating run configurations for languages generated into java – jetbrains.mps.baseLanguage.runConfigurations. Those run configurations are able to start under debugger automatically. See Run Configurations for details.

Custom viewers

When one views variables and fields in a variable view, one may want to define one’s own way to show certain values. For instance, collections could be shown as a collection of elements rather than as an ordinary object with all its internal structure.

For creating custom viewers MPS has jetbrains.mps.debugger.java.customViewers language.
The `jetbrains.mps.debugger.java.customViewers` language enables one to write one's own viewers for data of certain form.

A main concept of customViewers language is a custom data viewer. It receives a raw java value (an objects on stack) and returns a list of so-called watchables. A watchable is a pair of a value and its label (a string which categorizes a value, i.e. whether a value is a method, a field, an element, a size etc.) Labels for watchables are defined in custom watchables container. Each label could be assigned an icon.

The viewer for a specific type is defined in a custom viewer root. In the following table custom viewer parts are described:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>for type</td>
<td>A type for which this viewer is intended.</td>
</tr>
<tr>
<td>can wrap</td>
<td>An additional filter for viewed objects.</td>
</tr>
<tr>
<td>get presentation</td>
<td>A string representation of an object.</td>
</tr>
<tr>
<td>get custom watchables</td>
<td>Subvalues of this object. Result of this funtion must be of type watchable list.</td>
</tr>
</tbody>
</table>

Custom Viewers language introduces two new types: watchable list and watchable.

This is the custom viewer specification for `java.util.Map.Entry` class:

```
for type: Map.Entry

can wrap: <no can wrap>

generate presentation:
  (value).string {
    Object key = value.getKey();
    Object entryValue = value.getValue();
    return "E" + (key == null ? "null" : key.toString()) + " = " + (entryValue == null ? "null" : entryValue.toString());
  }

generate custom watchables:
  (value).watchable list {
    watchable list result = new watchables array list;
    Object key = value.getKey();
    Object entryValue = value.getValue();
    result.add(new watchable key ( key ));
    result.add(new watchable value { entryValue });
    return result;
  }
```

And here we see how a map entry is displayed in debugger view:

Note that the JDT-tools solution must be imported into your plugin solution in order to compile your custom viewers.

52.0.8 Creating custom debugger

If generation of your language avoids BaseLanguage, you'll need to take care of node tracing and breakpoint specification manually. Additionally, if you are generating languages other than Java, you'll have to attach the target platform debugger into MPS. The Debugger API provided by MPS allows you to create such non-java debuggers. All the necessary classes are located in the "Debugger API for MPS" plugin. See also Debugger API.

To summarize, when you target a language other than BaseLanguage, you typically need to specify:

- nodes which should be traced (see Debugger)
- on which nodes breakpoints could be created (see Debugger)
- how to compile your generated sources (see HowTo – Integrating into the MPS Make Framework)
- how to start your application under debug (see Run Configurations)

Not all of those steps are absolutely necessary - which of them are depends on the actual language.
The *customizedDebugger* sample project bundled with MPS will give you an easy-to-follow example of a non-BaseLanguage Java-generating language that customizes the breakpoints as well as node traces in order to support debugging.

### Traceable Nodes

This section describes how to specify which nodes require to save some additional information in `trace.info` file (like information about positions text, generated from the node, visible variables, name of the file the node was generated into etc.). `trace.info` files contain information allowing to connect nodes in MPS with generated text. For example, if a breakpoint is hit, java debugger tells MPS the line number in source file and to get the actual node from this information MPS uses information from `trace.info` files.

Specifically, `trace.info` files contain the following information:

- position information: name of text file and position in it where the node was generated;
- scope information: for each "scope" node (such that has some variables, associated with it and visible in the scope of the node) – names and ids of variables visible in the scope;
- unit information: for each "unit node" (such that represent some unit of a language, for example a class in java) – name of the unit the node is generated into.

```xml
<debug-info>
  <concept fn="jetbrains.mps.baseLanguage.structure.BlockStatement" />
  <concept fn="jetbrains.mps.baseLanguage.structure.ExpressionStatement" />
  <concept fn="jetbrains.mps.baseLanguage.structure.FieldDeclaration" />
  <concept fn="jetbrains.mps.baseLanguage.structure.InstanceMethodDeclaration" />
  <concept fn="jetbrains.mps.baseLanguage.structure.LocalVariableDeclarationStatement" />
  <concept fn="jetbrains.mps.baseLanguage.structure.Statement" />
  <concept fn="jetbrains.mps.baseLanguage.structure.StaticMethodDeclaration" />
</debug-info>
```

Ranges in the generated files where variables are in scope

Two generated files

Concepts `TraceableConcept`, `ScopeConcept` and `UnitConcept` of language `jetbrains.mps.lang.traceable` are used for that purpose. To save some information into `trace.info` file, user should derive from one of those concepts and implement the specific behavior method. The concepts are described in the table below.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
<th>Behavior method to implement</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>TraceableConcept</td>
<td>Location in the generated file</td>
<td>trace.info file</td>
<td><code>&lt;concept fn=&quot;jetbrains.mps.baseLanguage.structure.BlockStatement&quot;</code></td>
</tr>
<tr>
<td>ScopeConcept</td>
<td>Name and ids of variables visible in the scope</td>
<td>trace.info file</td>
<td><code>&lt;concept fn=&quot;jetbrains.mps.baseLanguage.structure.FieldDeclaration&quot;</code></td>
</tr>
<tr>
<td>UnitConcept</td>
<td>Name of the unit the node is generated into</td>
<td>trace.info file</td>
<td><code>&lt;concept fn=&quot;jetbrains.mps.baseLanguage.structure.Statement&quot;</code></td>
</tr>
</tbody>
</table>
CHAPTER 52. DEBUGGER

<table>
<thead>
<tr>
<th>TraceableConcept</th>
<th>Concepts for which location in text is saved and for which breakpoints could be created.</th>
<th>getTraceableProperty – some property to be saved into trace.info file.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScopeConcept</td>
<td>Concepts which have some local variables, visible in the scope.</td>
<td>getScopeVariables – variable declarations in the scope.</td>
</tr>
<tr>
<td>UnitConcept</td>
<td>Concepts which are generated into separate units, like classes or inner classes in Java.</td>
<td>getUnitName – name of the generated unit.</td>
</tr>
</tbody>
</table>

trace.info files are created on the last stage of generation – while generating text. So the described concepts are only to be used in languages generated into text. The entries are filled in automatically, whenever a TraceableConcept, ScopeConcept or UnitConcept are being generated through a reduction rule.

When automatic tracing is impossible, the $TRACE$ macro can be used in order to match the desired input node of a concept from language.high with the generated code explicitly.

Breakpoint Creators

TODO update screenshots with a non-Java debugger used

To specify how breakpoints are created on various nodes, root breakpoint creators is used. This is a root of concept BreakpointCreator from jetbrains.mps.debugger.api.lang language. The root should be located in the language plugin model. It contains a list of BreakpointableNodeItem, each of them specify a list of concept to create breakpoint for and a method actually creating a breakpoint. jetbrains.mps.debugger.api.lang provides several concepts to operate with debuggers, and specifically to create breakpoints. They are described below.

- DebuggerReference – a reference to a specific debugger, like java debugger;
- CreateBreakpointOperation – an operation which creates a location breakpoint of specified kind on a given node for a given project;
- DebuggerType – a special type for references to debuggers.
On the following example breakpoint creators node from baseLanguage is shown.

```java
breakpoint creators
for concepts:
  Statement
create breakpoint:
  (debuggableNode, project)->ILocationBreakpoint throws DebuggerNotPresentException {
    return debugger<Java>.create(Java Line Breakpoint, debuggableNode, project);
}
for concepts:
  FieldDeclaration
  StaticFieldDeclaration
create breakpoint:
  (debuggableNode, project)->ILocationBreakpoint throws DebuggerNotPresentException {
    return debugger<Java>.create(Java Field Breakpoint, debuggableNode, project);
}
```

In order to provide more complex filtering behavior, instead of a plain complex list breakpoint creators can use isApplicable function. There is an intention to switch to using this function.
Chapter 53

HowTo – Integrating into the MPS Make Framework

53.1 Build Facets

53.2 Overview

Like basically any build or make system, the MPS make executes a sequence of steps, or targets, to build an artifact. A global ordering of the necessary make steps is derived from relative priorities specified for each build targets (target A has to run before B, and B has to run before C, so the global order is A, B, C).

A complete build process may address several concerns, for example generating models into text, compiling these models, deploying them to the server, and/or generating .png files from graphviz source files. In MPS, such different build aspects are implemented with build facets. A facet is a collection of targets that address a common concern.

```plaintext
facet Sample extends <none> {  
  Required: Generate, TextGen

  <no optional facets>
  Targets:

  target target1 overrides <none> weight default {  
    resources policy: transform <no input> -> <no output>
    <no dependencies>
    <no properties>
    <no queries>
    <no config>
    (progressMonitor, input)-->void {
      <no statements>
    }
  }
}
```

Avoiding unnecessary file overwrites
The make process does not overwrite generated files that hold identical content to the one just generated. You can rely on the fact that only the modified files get updated on disk.
53.3 The targets within a facet can exchange configuration parameters. For example, a target that is declared to run early in the overall make process may collect configuration parameters and pass them to the second facet, which then uses the parameters. The mechanism to achieve this intra-facet parameter exchange is called properties. In addition, targets can use queries to obtain information from the user during the make process.

```java
facet Sample extends <none> {
    Required: Generate, TextGen
    <no optional facets>
    Targets:
    target target1 overrides <none> weight default {
        resources policy: transform <no input> -> <no output>
        Dependencies:
            before make
            • after checkParameters
            • after cleanup
            • after configure
            • after configure
            • after generate
            • after make
            • after preloadModels
            • after reconcile
            • after target1
            • after textGen
            • after textGenToMemory
}
```

The overall make process is organized along the pipes and filters pattern. The targets act as filters, working on a stream of data being delivered to them. The data flowing among targets is called resources. There are different kinds of resources, all represented as different Java interfaces and tuples:

- **MResource** contains MPS models created by users, those that are contained in the project’s solutions and languages
- **GResource** represents the results of the generation process, which includes the output models, that is the final state of the models after generation has completed. These are transient models, which may be inspected by using the **Save Transient Models** build option
- **TResource** represents the result of text-gen
- **CResource** represents a collection of Java classes
- **DResource** represents a collection of delta changes to models (**IDelta**)

The overall make process is organized along the pipes and filters pattern. The targets act as filters, working on a stream of data being delivered to them. The data flowing among targets is called resources. There are different kinds of resources, all represented as different Java interfaces and tuples:
• TextGenOutcomeResource represents the text files generated by textgen

These resources interfaces have been deprecated:

• IMResource contains MPS models created by users, those that are contained in the project’s solutions and languages
• IGResource represents the results of the generation process, which includes the output models, that is the final state of the models after generation has completed. These are transient models, which may be inspected by using the Save Transient Models build option
• ITResource represents the text files generated by textgen towards the end of the make process
• FResource

Build targets specify an interface. According to the pipes and filters pattern, the interface describes the kind of data that flows into and out of a make target. It is specified in terms of the resource types mentioned above, as well as in terms of the kind of processing the target applies to these resources. The following four processing policies are defined:

• transform is the default. This policy consumes instances of the input resource type and produces instances of the output resource type (e.g. it may consume MResources and produce TResources.)
• consume consumes the declared input, but produces no output. * produce consumes nothing, but produces output
• pass through does not access any resources, neither produce nor consume.

Note that the make process is more coarse grained than model generation. In other words, there is one facet that runs all the model generators. If one needs to "interject" additional targets into the MPS generation process (as opposed to doing something before or after model generation), this requires refactoring the generate facets. This is beyond the scope of this discussion.

53.4 Building an Example Facet

As part of the mbeddr.com project to build a C base language for MPS, the actual C compiler has to be integrated into the MPS build process. More specifically, programs written in the C base language contain a way to generate a Makefile. This Makefile has to be executed once it and all the corresponding .c and .h files have been generated, i.e. at the very end of the MPS make process.

To do this, we built a make facet with two targets. The first one inspects input models and collects the absolute paths of the directories that may contain a Makefile after textgen. The second target then checks if there is actually a file called Makefile in this directory and then runs make there. The two targets exchange the directories via properties, as discussed in the overview above.

The sampleFacet sample project that comes bundled with MPS distributions provides a simple facet definition that you can take as a starting point for your adventure with make facets.

53.4.1 The first target: Collecting Directories

Facets live in the plugins aspect of a language definition. Make sure you include the {{jetbrains.mps.make.facets}} language into the plugins model, so you can create instances of FacetDeclaration. A facet is executed as part of the make process of a model if that model uses the language that declares the facet.

The facet is called runMake. It depends on TextGen and Generate. The dependencies to those two facets has to be specified so we can then declare our targets’ priorities relative to targets in those facets.

facet runMake extends <none>
Required: TextGen, Generate

The first target is called collectPaths. It is specified as {{transform IMResource -> IMResource}} in order to get in touch with the input models. The
facet specifies, as priorities, after configure and before generate. The latter is obvious, since we want to get at the models before they are generated into text. The former priority essentially says that we want this target to run after the make process has been initialized (in other words: if you want to do something "at the beginning", use these two priorities.)

```java
#define collectPathes

The latter is obvious, since we want to get at the models before they are generated into text. The former priority essentially says that we want this target to run after the make process has been initialized (in other words: if you want to do something "at the beginning", use these two priorities.)

```java
#define collectPathes

We then declare a property pathes which we use to store information about the modules that contain make files, and the paths to the directories in which the generated code will reside.

```java
Properties:
    list<[string, string]> pathes;
```

Let's now look at the implementation code of the target. Here is the basic structure. We first initialize the pathes list. We then iterate of the input (which is a collection of resources) and do something with each input (explained below). We then use the output statement to output the input data, i.e. we just pass through whatever came into out target. We use the success statement to finish this target successfully (using success at the end is optional, since this is the default). If something goes wrong, the failure statement can be used to terminate the target unsuccessfully.

```java
(input)->void {
    pathes = new linkedlist<[string, string]>;
    input.forEach({~inpt =>
        (module, models) res = ((module, models)) inpt;
        // do stuff. See below.
    });
    output input;
    success;
}
```

The actual processing is straight forward Java programming against MPS data structures:

```java
res.models.forEach({-model =>
    string path = res.module.getGeneratorOutputPath() + "/" +
        model.getLongName().replaceAll("\.", "/");
    string locationInfo = res.module.getModuleName() + "/" +
        model.getLongName();
    pathes.add([path, locationInfo]);
});
```

We use the getGeneratorOutputPath method to get the path to which the particular module generates its code (this can be configured by the user in the model properties). We then get the model’s dotted name and replace the dots to slashes, since this is where the generated files of a model in that module will end up (inspect any example MPS project to see this). We then store the module’s name and the model’s name, separated by a slash, as a way of improving the logging messages in our second target (via the variable locationInfo). We add the two strings to the pathes collection. This pathes property is queried by the second target in the facet.

### 53.4.2 The second Target: Running Make

This one uses the pass through policy since it does not have to deal with resources. All the input it needs it can get from the properties of the collectPathes target discussed above. This second target runs after collectPathes, {{after textGen and before reconcile. It is obvious that is has to run after collectPaths}, since it uses the property data populated by it. It has to run after {{textGen}, otherwise the make files aren't there yet. And it has to run before {{reconcile}}.
Let us now look at the implementation code. We start by grabbing all those entries from the collectPathes.pathes property that actually contain a Makefile. If none is found, we return with success.

```java
sequence<[string, string]> modelDirectoriesWithMakefile = collectPathes.pathes.
    where({~it => new File(it[0] + "/Makefile").exists(); });
if (modelDirectoriesWithMakefile.isEmpty) { success; }
```

We then use the progress indicator language to set up the progress bar with as many work units as we have directories with make files in them.

```java
begin work "run make" covering ALL units of total work left,
    expecting modelDirectoriesWithMakefile.size units;
```

We then iterate over all the entries in the `modelDirectoriesWithMakefile` collection. In the loop we advance the progress indicator and then use Java standard APIs to run the make file.

```java
foreach dirInfoTuple in modelDirectoriesWithMakefile {
    try {
        advance 1 units of "run make"
            with comment "running make for " + dirInfoTuple[1];
        Process process = Runtime.getRuntime().
            exec("make", new string[0], new File(dirInfoTuple[0]));
        if (process.waitFor() > 0) {
            error "make failed with exit code " + process.exitValue() + " for " +
                dirInfoTuple[1];
        } else {
            info "make finished successfully for " + dirInfoTuple[1];
        }
    } catch (Exception ex) {
        error ex.getMessage(), ex;
    }
}
```

To wrap up the target, we use the `finish` statement to clean up the progress bar.

```java
finish "run make";
```
Chapter 54

Extension support

Extensions provide a possibility to extend certain aspects of a solution or a language, which are not covered by the standard language aspects and the plugin mechanisms. Typically you may need your language to slightly alter its behavior depending on the distribution model - MPS plugin, IntelliJ IDEA plugin or a standalone IDE. In such cases you define your extension points as interfaces to which then different implementations will be provided in different distributions.

Support for extensions exists in

- languages
- plugin solutions

54.1 Quick howto

1. Create an extension point
2. Create one or more extensions
3. Both the extension point and the extension must be in the plugin model
   (a) Each extension must provide a get method, returning an object
   (b) Each extension may opt to receive the activate/deactivate notifications
   (c) An extension may declare fields, just like classes can

54.2 Extension language

The language jetbrains.mps.lang.extension declares concepts necessary for building extensions.

Extension point

The ExtensionPoint concept represents an extension point. The extension object type must be specified as a parameter.
Extension

The Extension concept is used to create a concrete extension.

```java
Extension of testExtensionPoint

TestObject myTestObject

  on activate ()->{
    myTestObject = new TestObject();
    myTestObject.value = "foobar";
  }

  on deactivate ()->{
    myTestObject.shutDown = true;
    myTestObject = null;
  }

  extension object ()->{
    myTestObject;
  }
```

Accessing extension point

An extension point can be accessed by reference using extension point expression.

```
extensionPoint/testExtensionPoint/
```

Accessing extension objects

An extension point includes a way to access all objects provided by its extensions.

```
extensionPoint/multiExtensionPoint/.objects.sortBy(it => it; , desc).join(" ");
```

Be Careful

Objects returned by the extensions have transient nature: they may become obsolete as soon as a module reloading event happens. It is not recommended to e.g. cache these objects. Instead is it better to get a fresh copy each time.
54.3 Java API

```java
public interface Extension<T> {
    String getExtensionPointId ();
    T get ();
    void activate ();
    void deactivate ();
}

public class ExtensionPoint<T> {
    public ExtensionPoint (String id, Class<T> type) { ... }
    public Class<T> getType () { ... }
    public Iterable<Extension<T>> getExtensions () { ... }
    public Iterable<T> getObjects () { ... }
    public String getNamespace () { ... }
    public String getId () { ... }
}
```

Extension points and extensions are managed by the `ExtensionRegistry` core component.
Chapter 55

Custom Persistence Cookbook

This document will use the xmlPersistence sample bundled with MPS to teach you how to define, deploy and use your own persistence formats.

55.1 What is custom persistence?

MPS normally saves models in its own XML-based format. However, there are cases when you may want to load or save model files in your own format. Suppose, for example, that we want to leverage MPS to describe one of the existing languages. BaseLanguage, for instance, is a good description of Java. In such a case, the libraries written for that original language should be accessible from within MPS. It is the stubs aspect of a language that helps to create stub models for such libraries. Stub models are then used to reference the code outside of MPS.

Another useful example - if all you need from MPS is to merely edit files of your own DSL using the MPS editor, it would be useful to store the model in text format so that it could be edited in any text editor.

55.2 Look around the sample project

If you open the xmlPersistence sample project, you will get three main solutions, each of which fulfills a separate role in the puzzle.
CHAPTER 55. CUSTOM PERSISTENCE COOKBOOK

The xmlPersistence module defines the actual persistence logic, xmlPersistence.build contains a build script and xmlPersistence.ideaPlugin contains a customized plugin descriptor. Although the build script would normally provide a reasonable plugin descriptor by itself, this time we need to customize the descriptor, thus we include it in the project explicitly.

55.3 Define the persistence format

The xmlPersistence module implements the persistence logic. The persistence type in MPS is set on the per-model level. In our simplified case, the sample can store a model, which is restricted to a single XMLFile root element of the jetbrains.mps.core.xml language, into plain XML documents. The actual XML parsing logic resides in the XmlConverter class, while the XmlModelPersistence class implements the essential interfaces for hooking into the internal workings of MPS.

```java
@override
public String getFileExtension() {
    return XML_EXTENSION;
}
@override
public String getFormatTitle() {
    return "XML file";
}
```

The getFormatTitle() method is worth a special mention here, since the string it returns will be used to represent the storage format to the future users.

Additionally, the getFileExtension() method will register our custom persistence for .xml files.

55.3.1 Plugin ID

Also notice that the module has a Plugin ID set in the Idea Plugin tab of its module properties:
The identifier must match the plugin identifier declared in the plugin (xml) descriptor inside the xmlPersistence.ideaPlugin solution. We add an IDEA plug-in identifier to the properties of the xmlPersistence solution in order to specify that the solution is part of the plugin and thus can reference (or load at runtime) any plugin’s classes.

55.4 Build the MPS plugin

The xmlPersistencePlugin build script is a standard build script that zips the three modules into an IDEA/MPS plugin.
Make sure the paths specified in the macros section correctly point to your MPS and IntelliJ IDEA installation folders.

The plugin descriptor provides the standard plugin information plus it registers our `jetbrains.mps.persistence.ModelPersistence` class as `mps.ModelFactoryProvider`. This is the additional bit that requires us to provide an explicit plugin descriptor.

After rebuild you should be able to run the build script and get the plugin generated:
This will create the plugin so we can distribute it to the users.

### 55.5 Build the IntelliJ IDEA plugin

Building an IntelliJ IDEA plugin isn’t really that much different from building an MPS plugin. You only need to change the dependency from `mps` to `mpsPlugin` in the build script and set the artifacts location to wherever your MPS IDEA plugin has been deployed.

```groovy
build xmlPersistencePlugin generates build.xml

base directory: ./.
/Users/vaclav/MPSSamples.3.0/xmlPersistence

use plugins:
java
mps

macros:
folder mps_plugin = ./.../Library/Application Support/IntelliJIDEA12
folder idea_home = ./.../ Applications/IntelliJ IDEA 12 CE.app

dependencies:
mpsPlugin (artifacts location $mps_plugin)
IDEA (artifacts location $idea_home)
```

After rebuilding the project and running the build script, you get a plugin to deploy into IDEA.
55.6 Using the custom persistence plugin in MPS

After installing the generated plugin into MPS, when you are creating new models you are able to specify the XML file persistence provider for them:

Just as we specify in the `XmlModelPersistence.createEmpty()` method, the new model will have the `jetbrains.mps.core.xml` language added as a used language and an empty Root Node will be added to it:

Whatever you type into the root node, will be persisted immediately into a corresponding xml file. Changes to the underlying xml file will be reflected in the model upon opening in the editor.
55.7 Using the custom persistence plugin in IntelliJ IDEA

The IDEA plugin also allows for XML documents to be edited with an MPS-based projectional editor and yet persisted into plain xml files. Since .xml files are associated with the XML editor in IDEA, we get the default IDEA’s editor open when we click on an XML file.

However, since the sample.xml file is located under the configured MPS models root, MPS will invoke our custom persistence plugin and have it build a model out of it. When you hit Control + N / Cmd + N to open a class by name, you get the option to open the sample model:

Then you can edit it in a projectional editor and have your changes persisted into the original xml file:

At the moment MPS does not allow you to customize the persistence storage type of newly created models in IntelliJ IDEA. New models always receive the default MPS persistence type. MPS can, however, open existing models with any supported persistence type, if they are copied into the configured models root directory.

55.8 Debugging the plugin

MPS can also give you a hand when you want to test your fresh persistence implementation right away, directly in MPS. You simply create a Run Configuration off the Deploy Plugins template:

Then you specify your plugin (after rebuilding the project and re-running the build script) to be deployed by the Run Configuration:
Finally you run the configuration to get the plugin installed into MPS (MPS will restart):

After this step you will be able to use the custom persistence provider for models and test that it behaves as expected. The plugin will be installed into the `.MPS3x` sub-directory of your home folder on Windows, or `~/.HOME/Library/Application Support/MPS3x` on Mac. You may uninstall or disable the plugin through the Plugin Manager UI or by deleting the plugin manually.
Part V

IDE tools
Chapter 56

Dependencies Analyzer - analyze model dependencies

The Dependencies Analyzer can report dependencies among modules or models. It can be called from the main menu or from the popup menu of modules/models:
The interactive report, shown in a panel at the bottom, allows the user to view usages of modules, models and nodes by other modules, models and nodes. The panel on the right side displays modules and models that the element selected in the left-hand side list depends on. The bottom panel lists the actual places that demand the currently selected dependency.

The \textbf{L} icon enables the user to toggle between analyzing the model dependencies and the languages used in the models. Unlike the \textit{Module Dependencies Tool}, which simply visualizes the dependency information specified in model properties, the Analyzer checks the actual code and performs dependency analysis. It detects and highlights the elements that you really depend on.
Chapter 57

Module Dependencies Tool

The Module Dependencies Tool allows the user to overview all the dependencies and used languages of a module or a set of modules, to detect potential cyclic dependencies as well as to see detailed paths that form the dependencies. The tool can be invoked from the menu as well as from the project pane when one or more modules are selected.

Module Dependency Tool shows all transitive dependencies of the modules in the left panel. Optionally, it can also display all directly or indirectly used languages. It is possible to expand any dependency node and get all dependencies of the expanded node as children. These will again be transitive dependencies, but this time for the expanded node. Select one or more of the dependency nodes in the left panel. The right panel will show paths to each of the selected modules from its "parent" module. You can see a brief explanation of each relation between modules in the right tree. The types of dependencies can be one of: depends on, uses language, exports runtime, uses devkit, etc. For convinience the name of the target dependent module is shown in bold.
There are two types of dependency paths: **Dependency** and **Used Language**. The L button in the toolbar enables/disables displaying of **Used Languages** in the left tree panel. When you select a module in the **Used Language** folder in the left tree, the right tree shows only the dependency paths that introduce the **used language** relation for the given module. To show "ordinary" dependencies on a language module, you should select it outside of the **Used Languages** folder (e.g. the `jetbrains.mps.lang.core` language in the picture below). It is also possible to select multiple nodes (e.g. the same language dependency both inside and outside of the **Used Language** folder). In that case you get a union of results for both paths.

When you are using a language that comes with its own libraries, those libraries are typically not needed to compile your project. It is the runtime when the libraries must be around for your code to work. For tracking runtime dependencies in addition to the "compile-time visible" ones, you should check the **Runtime** option in the toolbar. The runtime dependencies are marked with a "runtime" comment.

The modules in the left tree that participate in dependency cycles are shown in red color. It is possible to expand the tree node to see the paths forming the cycle:

For some types of dependencies the pop-up menu offers the possibility to invoke convenience actions such as **Show Usages** or **Safe Delete**. For the "depends on" dependencies (those without re-export) Dependencies Analyzer will be invoked for the **Show Usages** action.
Chapter 58

Run Configurations

58.0.1 Introduction

Run configurations allow users to define how to execute programs written in their language. An existing run configuration can be executed either from run configurations box, located on the main toolbar, by the "Run" menu item in the main menu

or through the run/debug popup (Alt+Shift+F10/Alt+Shift+F9).

Also run configurations could be executed/created for nodes, models, modules and projects. For example, the JUnit run configuration could run all tests in a selected project, module or model. See Run Configurations on how to implement such behavior for your own run configurations.
To summarize, run configurations define the following things:

- In the creation stage:
  - the configurations name, caption, icon;
  - the configurations kind;
  - how to create a configuration from node(s), model(s), module(s), project.

- In the configuration stage:
  - persistent parameters;
  - an editor for persistent parameters;
  - a checker of persistent parameters validity.

- In the execution stage:
  - the process, which is actually executed;
  - a console with all its tabs, action buttons and actual console window;
  - the things required for debugging this configuration (if it is possible).

The following languages have been introduced to support run configurations in MPS.

- `jetbrains.mps.execution.common` (common language) – contains concepts utilized by the other execution* languages;
- `jetbrains.mps.execution.settings` (settings language) – a language for defining different setting editors;
- `jetbrains.mps.execution.commands` (command languages) – processes invocations from java;
- `jetbrains.mps.execution.configurations` (configurations language) – the run configurations definition;

### 58.0.2 Settings

The Settings language allows to create setting editors and integrate them into one another. What we need from a settings editor is the following:

- the fields to edit;
- validation of fields’ correctness;
- an editor UI component;
- apply/reset functions to apply settings from the UI component and to reset settings in the UI component to the saved state;
- a dispose function to destroy the UI component when it is no longer needed.
As you can see, settings have UI components. Usually, one UI component is created for multiple instances of settings. In the settings language settings are usually called "configurations" and their UI components are called "editors".

The main concept of settings language is PersistentConfigurationTemplate. It has the following sections:

- **persistent properties** - This section describes the actual settings we are editing. Since we want also to persist these settings (i.e. to write to xml/read from xml) and to clone our configurations, there is a restriction on their type: each property should be either Cloneable or String or any primitive type. There is also a special kind of property named template persistent property, but they are going to be discussed later.

- **editor** - This section describes the editor of the configuration. It holds the following functions: create, apply to, reset from, dispose. A section can also define fields to store some objects of the editor. A create function should return a swing component — the main UI component of the editor. apply to/reset from functions apply or reset settings in the editor to/from configuration given as a parameter. dispose function disposes the editor.

- **check** - In this section persistent properties are checked for correctness. If some properties are not valid, a report error statement can be used. Essentially, this statement throws RuntimeException.

- **additional methods** - This section is for methods, used in the configurations. Essentially, these methods are configuration instance methods.

**Persistent properties**

It was mentioned above that persistent properties could be either Cloneable or String or any primitive type. But if one uses the Settings language inside run configurations, those properties should also support xml persistence. Strings and primitives are persisted as usual. For objects the persistence is more complicated. Two types of properties are persisted for an object: public instance fields and properties with setXXX and getXXX methods. So, if one wish to use some complex type for a persistent property, he should either make all important fields public or provide setXXX and getXXX methods for whatever he wants to persist.

**Integrating configurations into one another**

One of the two basic features of the Settings language is easy integration of one configuration into another. For that template persistent properties are used.

**Template parameters**

The second basic feature of the Settings language is template parameters. These somewhat resemble constructor parameters in java. For example, if one creates a configuration for choosing a node, he may want to parametrize the configuration with nodes concept. The concept is not a persistent parameter in this case: it is not chosen by the user. This is a parameter specified at configuration creation.

### 58.0.3 Commands

The Commands language allows you to start up processes from the code in a way it is done from a command line. The main concept of the language is CommandDeclaration. In the declaration, command parameters and the way to start process with this parameters are specified. Also, commands can have debugger parameters and some utility methods.

**Execute command sections**

Each command can have several execute sections. Each of this sections defines several execution parameters. There are parameters of two types: required and optional. Optional parameters can have default values and could be ommited when the command is started, while required cannot have default values and they are mandatory. Any two execute sections of the command should have different (by types) lists of required parameters. One execute section can invoke another execute section. Each execute section should return either values of process or ProcessHandler types.

**ProcessBuilderExpression**

To start a process from a command execute section ProcessBuilderExpression is used. It is a simple list of command parts. Each part is either ProcessBuilderPart, which consists of an expression of type string or list<string>, or a ProcessBuilderKeyPart, which represents a parameter with a key (like "-classpath /path/to/classes"). When the code generated from ProcessBuilderExpression is invoked, each part is tested for being null or empty and ommited if so. Then, each part is split into multiple parts by spaces. So if you intent to provide a command part with a space in it and do not wish it to be split (for example, you have a file path with spaces), you have to put it into double quotes ("). The working directory of created process could be specified in the Inspector.
Debugger integration

To integrate a command with the debugger, two things are required to be specified:

- the specific debugger to integrate with;
- the command line arguments for a process.

To specify a debugger you can use DebuggerReference – an expression of debugger type in jetbrains.mps.debug.apiLang – to reference a specific debugger. Debugger settings must be an object of type IDebuggerSettings.

58.0.4 Configurations

The Configurations language allows to create run configurations. To create a run configuration, one should create an instance of RunConfiguration (essentially, configuration from the settings language) and provide a RunConfigurationExecutor for it. One also may need a RunConfigurationKind to specify a kind of this configuration, RunConfigurationProducer to provide a way of creating this configuration from nodes, models, modules etc and a BeforeTask to specify, how to prepare the configuration before execution.

Executors

Executor is a node, which describes how a process is started for this run configuration. It takes the settings that the user entered and creates a process from it. So, the executor’s execute methods should return an instance of type Process. This is done via StartProcessHandlerStatement. Anything that has a type Process or ProcessHandler could be passed to it. A Process could be created in three different ways:

1. via command;
2. via ProcessBuilderExpression (recommended to use in commands only);
3. by creating new instance of the ProcessHandler class; this method is recommended only if the above two do not fit you, for example when you are creating a run configuration for remote debugging and you do not really need to start a process.

The executor itself consists of the following sections:

1. "for" section where the configuration this executor is for and an alias for it is specified;
2. "can" section where the ability of run/debug this configuration is specified; if the command is not used in this executor, one must provide an instance of DebuggerConfiguration here;
3. "before" section with the call of tasks, which could be executed before this configuration run, such as Make;
4. "execute" section where the process itself is created.

Debugger integration

If a command is used to start a process, nothing should be done apart from specifying a configuration as debuggable (by selecting "debug" in the executor). However, if a custom debugger integration is required, it is done the same way as in the command declaration.

Producers

Producers for a run configuration describe how to create this configuration for various nodes or groups of nodes, models, modules or a project. This makes run configurations easily discoverable for users since for each producer they will see an action in the context menu suggesting to run the selected item. Also this simplifies configuring because it gives a default way to execute something without seeing the editing dialog first.

Each producer specifies one run configuration that it creates. It can have several produce from sections for each kind of source the configuration can be produced from. This source should be one of the following: node<>>, nlist<>>, model, module, project. Apart from source, each produce from section has a create section – a concept function parametrized with a source. The function should return either the created run configuration or null if it cannot be created for some reason.

58.0.5 Useful examples

In this section you can find some useful tips and examples of run configurations usages.
Person Editor

In this example an editor for a "Person" is created. This editor edits two properties of a person: name and e-mail address.

```java
persistent configuration template PersonEditor <>

  persistent properties:
  string myName;
  string myEmail;

  check() -> void {
    if (!this.myName.matches("(\w|\s)*")) {
      report error "Name should contain only letters and spaces.";
    }
    if (!this.myEmail.matches("(\w|\.)*\@(\w|\.)*")) {
      report error "Email should be valid.";
    }
  }

editor:
  JTextField myNameField;
  JTextField myEmailField;

create() -> JComponent {
  JPanel panel = new JPanel(new GridBagLayout());
  myNameField = new JTextField();
  myEmailField = new JTextField();
  panel.add(new JLabel("Name"), grid bag constraints/label, 0/);
  panel.add(myNameField, grid bag constraints/field, 1/);
  panel.add(new JLabel("Email"), grid bag constraints/label, 2/);
  panel.add(myEmailField, grid bag constraints/field, 3/);
  return panel;
}
reset from(configuration) -> void {
  myNameField.setText(configuration.myName);
  myEmailField.setText(configuration.myEmail);
}
apply to(configuration) -> void {
  configuration.myName = myNameField.getText();
  configuration.myEmail = myEmailField.getText();
}
<no dispose>
```

PersonEditor could be used from java code in the following way:
public class SettingsEditorDialogTest {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                // create PersonEditor
                template configuration<PersonEditor> personEditor = new PersonEditor();
                // create dialog
                SettingsEditorDialog dialog = new SettingsEditorDialog(personEditor);
                // show dialog
                dialog.setVisible(true);
                // print name and email, specified by user
                System.out.println("name = " + personEditor.myName);
                System.out.println("email = " + personEditor.myEmail);
            }
        });
    }
}

**Exec command**

This is an example of a simple command, which starts a given executable with programParameters in a given workingDirectory.

```java
command exec
destructor: <no debuggerConfiguration>

    execute(File executable (required)
    File workingDirectory = new File(SystemProperties.getUserHome());
    String programParameters
    {
        return this.protect(executable.getAbsolutePath()) programParameters in workingDirectory;
    }

    private String protect(String rawString) {
        if ([rawString.contains(" ") { return "\\" + rawString + "\\"; }
        return rawString;
    }
```

**Compile with gcc before task**

This is an example of a BeforeTask which performs compilation of a source file with the gcc command. It also demonstrates how to use commands outside of run configurations executors.
Note that this is just a toy example, in a real-life scenarios the task should show a progress window while compiling, for example.

**Java Executor**

This is an actual executor for the Java run configuration from MPS.
CHAPTER 58. RUN CONFIGURATIONS

Java Producer

This is a producer for Java run configuration from MPS.

```java
def Java as myRunConfiguration
can: run, debug

before:
Make(new arraylist<node>(myRunConfiguration.myNode.getNode());)

eexecute(project)-<join(process | ProcessHandler) {
    // create a console
    console console = new console(project, false);
    // add a message filter to parse stack traces
    console.addMessageFilter(new JavaStackTraceFilter());
    // start java command process with tool console
    start java(node = myRunConfiguration.myNode.getNode(),
    runParameters = myRunConfiguration.runParameters.myJavaRunParameters) with tool console;
}
```

Running a node, generated into java class

Let’s suppose you have a node of a concept which is generated into a java class with a main method and you wish to run this node from MPS. Then you do not have to create a run configuration in this case, but you should do the following:

1. The concept you wish to run should implement IMainClass concept from jetbrains.mps.execution.util language. To specify when the node can be executed, override isNodeRunnable method.

2. Unit information should be generated for the concept. Unit information is required to correctly determine the class name which is to be executed. You can read more about unit information, as well as about all trace information, in Debugger section of MPS documentation. To ensure this check that one of the following conditions are satisfied:

   (a) a ClassConcept from jetbrains.mps.baseLanguage is generated from the node;
   (b) a node is generated into text (the language uses textGen aspect for generation) and the concept of the node implements UnitConcept interface from jetbrains.mps.traceable;
   (c) a node is generated into a concept for which one of the three specified conditions is satisfied.
Chapter 59

Changes highlighting

Changes highlighting is a handy way to show changes which were made since last update from version control system. The changes in models are highlighted in the following places:

59.0.6 Project tree view

Models, nodes, properties and references are highlighted. green means new items, blue means modified items, brown means unversioned items.

59.0.7 Editor tabs

Highlighting appears for all of the editor tabs: for language aspect tabs of a concept and also for custom tabbed editors declared in plugin aspect of a language (see Plugin).
59.0.8 Editor

Every kind of changes are highlighted in MPS editor: changing properties and references, adding, deleting and replacing nodes.

```java
private void reinitMessageGroups() {
    myMessageGroups = new ArrayList<MessageGroup>);
    EditorComponent editorComponent = getEditorComponent();

    list<EditorComponentChangesHighlighter.ChangeEditorMessage> messagesWithCells =
        myEditorComponentChangesHighlighter.getEditorMessages().where({~m =
            toList;
            messagesWithCells = messagesWithCells.sort({~aMsg, ~bMsg =>
                Rectangle a = getMessageMessageBounds(editorComponent, aMsg);
                Rectangle b = getMessageMessageBounds(editorComponent, bMsg);
                // First compare by y, then by x, and, after all, by height
                if (a.y == b.y) {
                    if (a.x == b.x) {
                        return a.height - b.height;
                    } else {
                        return a.x - b.x;
                    }
                } else {
                    return a.y - b.y;
                }
            }, asc).toList;
```

If you hover mouse cursor over the highlighter’s strip on the left margin of editor, the corresponding changes become highlighted in editor pane.

If you want to have your changes highlighted in editor pane all the time (not only on hovering mouse cursor over highlighter’s strip), you can select "Highlight Nodes With Changes Relative to Base Version" option in IDE Settings → Editor.
private void reinitMessageGroups() {
    myMessageGroups = new ArrayList<MessageGroup>;
    EditorComponent editorComponent = getEditorComponent();

    list<EditorComponentChangesHighlighter.ChangeEditorMessage> messagesWithCells
        = myEditorComponentChangesHighlighter.getEditorMessages().where({m =
            toList;

    messagesWithCells = messagesWithCells.sort((aMsg, bMsg) =>
        Rectangle a = getMessageBounds(editorComponent, aMsg);
        Rectangle b = getMessageBounds(editorComponent, bMsg);
        // First compare by y, then by x, and, after all, by height

    If you click on highlighter’s strip on the left margin, there appears a panel with three buttons: "Go to Previous Change", "Go to Next Change" and "Rollback". 

    If you click “Rollback”, all the corresponding changes are reverted.
    This feature allows you to freely make any changes to the MPS model in the editor without fear, because at any moment you can revert your changes conveniently right from the editor.
# Chapter 60

## Default Keymap Reference

### Core of editing

<table>
<thead>
<tr>
<th>Windows/Linux</th>
<th>MacOS</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl + Space</td>
<td>Ctrl + Space</td>
<td>Code completion</td>
</tr>
<tr>
<td>Alt + Enter</td>
<td>Alt + Enter</td>
<td>Show contextual intention actions</td>
</tr>
<tr>
<td>Ctrl + Z</td>
<td>Cmd + Z</td>
<td>Undo</td>
</tr>
<tr>
<td>Ctrl + Shift + Z</td>
<td>Cmd + Shift + Z</td>
<td>Redo</td>
</tr>
<tr>
<td>Tab</td>
<td>Tab</td>
<td>Move to the next cell</td>
</tr>
<tr>
<td>Shift + Tab</td>
<td>Shift + Tab</td>
<td>Move to the previous cell</td>
</tr>
</tbody>
</table>

### General editing

<table>
<thead>
<tr>
<th>Windows/Linux</th>
<th>MacOS</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl + Alt + T</td>
<td>Cmd + Alt + T</td>
<td>Surround with...</td>
</tr>
<tr>
<td>Ctrl + X / Shift + Delete</td>
<td>Cmd + X</td>
<td>Cut current line or selected block to buffer</td>
</tr>
<tr>
<td>Ctrl + C / Ctrl + Insert</td>
<td>Cmd + C</td>
<td>Copy current line or selected block to buffer</td>
</tr>
<tr>
<td>Ctrl + V / Shift + Insert</td>
<td>Cmd + V</td>
<td>Paste from buffer</td>
</tr>
<tr>
<td>Ctrl + D</td>
<td>Cmd + D</td>
<td>Duplicate current line or selected block</td>
</tr>
<tr>
<td>Shift + F5</td>
<td>Shift + F5</td>
<td>Clone root</td>
</tr>
<tr>
<td>Ctrl + Up/Down</td>
<td>Cmd + Up/Down</td>
<td>Expand/Shrink block selection region</td>
</tr>
<tr>
<td>Ctrl + Shift + Up/Down</td>
<td>Cmd + Shift + Up/Down</td>
<td>Move statements up/down</td>
</tr>
<tr>
<td>Shift + Arrows</td>
<td>Shift + Arrows</td>
<td>Extend the selected region to siblings</td>
</tr>
<tr>
<td>Ctrl + W</td>
<td>Cmd + W</td>
<td>Select successively increasing code blocks</td>
</tr>
<tr>
<td>Ctrl + Shift + W</td>
<td>Cmd + Shift + W</td>
<td>Decrease current selection to previous state</td>
</tr>
<tr>
<td>Ctrl + Y</td>
<td>Cmd + Y</td>
<td>Delete line</td>
</tr>
<tr>
<td>Alt + X</td>
<td>Control + X</td>
<td>Show note in AST explorer</td>
</tr>
<tr>
<td>F5</td>
<td>F5</td>
<td>Refresh the error messages in the editor</td>
</tr>
<tr>
<td>Ctrl + -</td>
<td>Cmd + -</td>
<td>Collapse</td>
</tr>
<tr>
<td>Ctrl + Shift + -</td>
<td>Cmd + Shift + -</td>
<td>Collapse all</td>
</tr>
<tr>
<td>Ctrl + +</td>
<td>Cmd + +</td>
<td>Expand</td>
</tr>
<tr>
<td>Ctrl + Shift + +</td>
<td>Cmd + Shift + +</td>
<td>Expand all</td>
</tr>
<tr>
<td>Ctrl + Shift + 0-9</td>
<td>Cmd + Shift + 0-9</td>
<td>Set bookmark</td>
</tr>
<tr>
<td>Ctrl + 0-9</td>
<td>Ctrl + 0-9</td>
<td>Go to bookmark</td>
</tr>
<tr>
<td>Insert</td>
<td>Ctrl + N</td>
<td>Create Root Node (in the Project View)</td>
</tr>
<tr>
<td>Ctrl + Alt + click</td>
<td>Cmd + Alt + click</td>
<td>Show descriptions of error or warning at caret</td>
</tr>
<tr>
<td>Ctrl + Shift + T</td>
<td>Cmd + Shift + T</td>
<td>Show type of node</td>
</tr>
<tr>
<td>Ctrl + Alt + T</td>
<td>Cmd + Alt + T</td>
<td>Surround with...</td>
</tr>
</tbody>
</table>

### Set dependencies on models, import used languages

<table>
<thead>
<tr>
<th>Windows/Linux</th>
<th>MacOS</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl + M</td>
<td>Cmd + M</td>
<td>Import model</td>
</tr>
<tr>
<td>Ctrl + L</td>
<td>Cmd + L</td>
<td>Import language</td>
</tr>
<tr>
<td>Ctrl + R</td>
<td>Cmd + R</td>
<td>Import model by a root name</td>
</tr>
</tbody>
</table>
## Find usages and Search

<table>
<thead>
<tr>
<th>Windows/Linux</th>
<th>MacOS</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alt + F7</td>
<td>Alt + F7</td>
<td>Find usages</td>
</tr>
<tr>
<td>Alt + F6</td>
<td>Alt + F6</td>
<td>Find concept instances</td>
</tr>
<tr>
<td>Ctrl + Alt + Shift + F7</td>
<td>Cmd + Alt + Shift + F7</td>
<td>Highlight cell dependencies</td>
</tr>
<tr>
<td>Ctrl + Shift + F6</td>
<td>Cmd + Shift + F6</td>
<td>Highlight instances</td>
</tr>
<tr>
<td>Ctrl + Shift + F7</td>
<td>Cmd + Shift + F7</td>
<td>Highlight usages</td>
</tr>
<tr>
<td>Ctrl + F</td>
<td>Cmd + F</td>
<td>Find text</td>
</tr>
<tr>
<td>F3</td>
<td>F3</td>
<td>Find next</td>
</tr>
<tr>
<td>Shift + F3</td>
<td>Shift + F3</td>
<td>Find previous</td>
</tr>
</tbody>
</table>

## Navigation

<table>
<thead>
<tr>
<th>Windows/Linux</th>
<th>MacOS</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl + B / Ctrl + click</td>
<td>Cmd + B / Cmd + click</td>
<td>Go to declaration</td>
</tr>
<tr>
<td>Ctrl + N</td>
<td>Cmd + N</td>
<td>Go to root node by name</td>
</tr>
<tr>
<td>Ctrl + Shift + N</td>
<td>Cmd + Shift + N</td>
<td>Go to file by name</td>
</tr>
<tr>
<td>Ctrl + G</td>
<td>Cmd + G</td>
<td>Go to node by id</td>
</tr>
<tr>
<td>Ctrl + Shift + A</td>
<td>Cmd + Shift + A</td>
<td>Go to action by name</td>
</tr>
<tr>
<td>Ctrl + Alt + Shift + M</td>
<td>Cmd + Alt + Shift + M</td>
<td>Go to model</td>
</tr>
<tr>
<td>Ctrl + Alt + Shift + S</td>
<td>Cmd + Alt + Shift + S</td>
<td>Go to solution</td>
</tr>
<tr>
<td>Ctrl + Shift + S</td>
<td>Cmd + Shift + S</td>
<td>Go to concept declaration</td>
</tr>
<tr>
<td>Ctrl + Shift + E</td>
<td>Cmd + Shift + E</td>
<td>Go to concept editor declaration</td>
</tr>
<tr>
<td>Alt + Left/Right</td>
<td>Control + Left/Right</td>
<td>Go to next/previous editor tab</td>
</tr>
<tr>
<td>Esc</td>
<td>Esc</td>
<td>Go to editor (from tool window)</td>
</tr>
<tr>
<td>Shift + Esc</td>
<td>Shift + Esc</td>
<td>Hide active or last active window</td>
</tr>
<tr>
<td>Shift + F12</td>
<td>Shift + F12</td>
<td>Restore default window layout</td>
</tr>
<tr>
<td>Ctrl + Shift + F12</td>
<td>Cmd + Shift + F12</td>
<td>Hide all tool windows</td>
</tr>
<tr>
<td>F12</td>
<td>F12</td>
<td>Jump to the last tool window</td>
</tr>
<tr>
<td>Ctrl + E</td>
<td>Cmd + E</td>
<td>Recent nodes popup</td>
</tr>
<tr>
<td>Ctrl + Alt + Left/Right</td>
<td>Cmd + Alt + Left/Right</td>
<td>Navigate back/forward</td>
</tr>
<tr>
<td>Alt + F1</td>
<td>Alt + F1</td>
<td>Select current node in any view</td>
</tr>
<tr>
<td>Ctrl + H</td>
<td>Cmd + H</td>
<td>Concept/Class hierarchy</td>
</tr>
<tr>
<td>F4 / Enter</td>
<td>F4 / Enter</td>
<td>Edit source / View source</td>
</tr>
<tr>
<td>Ctrl + F4</td>
<td>Cmd + F4</td>
<td>Close active editor tab</td>
</tr>
<tr>
<td>Alt + 2</td>
<td>Alt + 2</td>
<td>Go to inspector</td>
</tr>
<tr>
<td>Ctrl + F10</td>
<td>Cmd + F10</td>
<td>Show structure</td>
</tr>
<tr>
<td>Ctrl + Alt + ]</td>
<td>Cmd + Alt + ]</td>
<td>Go to next project window</td>
</tr>
<tr>
<td>Ctrl + Alt + [</td>
<td>Cmd + Alt + [</td>
<td>Go to previous project window</td>
</tr>
<tr>
<td>Ctrl + Shift + Right</td>
<td>Ctrl + Shift + Right</td>
<td>Go to next aspect tab</td>
</tr>
<tr>
<td>Ctrl + Shift + Left</td>
<td>Ctrl + Shift + Left</td>
<td>Go to previous aspect tab</td>
</tr>
<tr>
<td>Ctrl + Alt + Shift + R</td>
<td>Cmd + Alt + Shift + R</td>
<td>Go to type-system rules</td>
</tr>
<tr>
<td>Ctrl + Shift + T</td>
<td>Cmd + Shift + T</td>
<td>Show type</td>
</tr>
<tr>
<td>Ctrl + H</td>
<td>Ctrl + H</td>
<td>Show in hierarchy view</td>
</tr>
<tr>
<td>Ctrl + I</td>
<td>Cmd + I</td>
<td>Inspect node</td>
</tr>
</tbody>
</table>

## BaseLanguage Editing

<table>
<thead>
<tr>
<th>Windows/Linux</th>
<th>MacOS</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl + O</td>
<td>Cmd + O</td>
<td>Override methods</td>
</tr>
<tr>
<td>Ctrl + I</td>
<td>Cmd + I</td>
<td>Implement methods</td>
</tr>
<tr>
<td>Ctrl + /</td>
<td>Cmd + /</td>
<td>Comment/uncomment with block comment</td>
</tr>
<tr>
<td>Ctrl + F12</td>
<td>Cmd + F12</td>
<td>Show nodes</td>
</tr>
<tr>
<td>Ctrl + P</td>
<td>Cmd + P</td>
<td>Show parameters</td>
</tr>
<tr>
<td>Ctrl + Q</td>
<td>Ctrl + Q</td>
<td>Show node information</td>
</tr>
<tr>
<td>Alt + Insert</td>
<td>Ctrl + N</td>
<td>Create new ...</td>
</tr>
<tr>
<td>Ctrl + Alt + B</td>
<td>Cmd + Alt + B</td>
<td>Go to overriding methods / Go to inherited classifiers</td>
</tr>
<tr>
<td>Ctrl + U</td>
<td>Cmd + U</td>
<td>Go to uverriden method</td>
</tr>
</tbody>
</table>
## BaseLanguage refactoring

<table>
<thead>
<tr>
<th>Windows/Linux</th>
<th>MacOS</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>F6</td>
<td>F6</td>
<td>Move</td>
</tr>
<tr>
<td>Shift + F6</td>
<td>Shift + F6</td>
<td>Rename</td>
</tr>
<tr>
<td>Alt + Delete</td>
<td>Alt + Delete</td>
<td>Safe Delete</td>
</tr>
<tr>
<td>Ctrl + Alt + N</td>
<td>Ctrl + Alt + N</td>
<td>Inline</td>
</tr>
<tr>
<td>Ctrl + Alt + M</td>
<td>Ctrl + Alt + M</td>
<td>Extract Method</td>
</tr>
<tr>
<td>Ctrl + Alt + V</td>
<td>Ctrl + Alt + V</td>
<td>Introduce Variable</td>
</tr>
<tr>
<td>Ctrl + Alt + C</td>
<td>Ctrl + Alt + C</td>
<td>Introduce constant</td>
</tr>
<tr>
<td>Ctrl + Alt + F</td>
<td>Ctrl + Alt + F</td>
<td>Introduce field</td>
</tr>
<tr>
<td>Ctrl + Alt + P</td>
<td>Ctrl + Alt + P</td>
<td>Extract parameter</td>
</tr>
<tr>
<td>Ctrl + Alt + M</td>
<td>Ctrl + Alt + M</td>
<td>Extract method</td>
</tr>
<tr>
<td>Ctrl + Alt + N</td>
<td>Ctrl + Alt + N</td>
<td>Inline</td>
</tr>
</tbody>
</table>

## Generation, compilation and run

<table>
<thead>
<tr>
<th>Windows/Linux</th>
<th>MacOS</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl + F9</td>
<td>Cmd + F9</td>
<td>Generate current module</td>
</tr>
<tr>
<td>Ctrl + Shift + F9</td>
<td>Cmd + Shift + F9</td>
<td>Generate current model</td>
</tr>
<tr>
<td>Shift + F10</td>
<td>Shift + F10</td>
<td>Run</td>
</tr>
<tr>
<td>Shift + F9</td>
<td>Shift + F9</td>
<td>Debug</td>
</tr>
<tr>
<td>Ctrl + Shift + F10</td>
<td>Cmd + Shift + F10</td>
<td>Run context configuration</td>
</tr>
<tr>
<td>Alt + Shift + F10</td>
<td>Alt + Shift + F10</td>
<td>Select and run a configuration</td>
</tr>
<tr>
<td>Ctrl + Shift + F9</td>
<td>Cmd + Shift + F9</td>
<td>Debug context configuration</td>
</tr>
<tr>
<td>Alt + Shift + F9</td>
<td>Alt + Shift + F9</td>
<td>Select and debug a configuration</td>
</tr>
<tr>
<td>Ctrl + Alt + Shift + F9</td>
<td>Cmd + Alt + Shift + F9</td>
<td>Preview generated text</td>
</tr>
<tr>
<td>Ctrl + Shift + X</td>
<td>Cmd + Shift + X</td>
<td>Show type-system trace</td>
</tr>
</tbody>
</table>

## Debugger

<table>
<thead>
<tr>
<th>Windows/Linux</th>
<th>MacOS</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>F8</td>
<td>F8</td>
<td>Step over</td>
</tr>
<tr>
<td>F7</td>
<td>F7</td>
<td>Step into</td>
</tr>
<tr>
<td>Shift + F8</td>
<td>Shift + F8</td>
<td>Step out</td>
</tr>
<tr>
<td>F9</td>
<td>F9</td>
<td>Resume</td>
</tr>
<tr>
<td>Alt + F8</td>
<td>Alt + F8</td>
<td>Evaluate expression</td>
</tr>
<tr>
<td>Ctrl + F8</td>
<td>Cmd + F8</td>
<td>Toggle breakpoints</td>
</tr>
<tr>
<td>Ctrl + Shift + F8</td>
<td>Cmd + Shift + F8</td>
<td>View breakpoints</td>
</tr>
</tbody>
</table>

## VCS/Local History

<table>
<thead>
<tr>
<th>Windows/Linux</th>
<th>MacOS</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl + K</td>
<td>Cmd + K</td>
<td>Commit project to VCS</td>
</tr>
<tr>
<td>Ctrl + T</td>
<td>Cmd + T</td>
<td>Update project from VCS</td>
</tr>
<tr>
<td>Ctrl + V</td>
<td>Ctrl + V</td>
<td>VCS operations popup</td>
</tr>
<tr>
<td>Ctrl + Alt + A</td>
<td>Cmd + Alt + A</td>
<td>Add to VCS</td>
</tr>
<tr>
<td>Ctrl + Alt + E</td>
<td>Cmd + Alt + E</td>
<td>Browse history</td>
</tr>
<tr>
<td>Ctrl + D</td>
<td>Cmd + D</td>
<td>Show differences</td>
</tr>
</tbody>
</table>

## General

<table>
<thead>
<tr>
<th>Windows/Linux</th>
<th>MacOS</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alt + 0-9</td>
<td>Alt + 0-9</td>
<td>Open the corresponding tool window</td>
</tr>
<tr>
<td>Ctrl + S</td>
<td>Cmd + S</td>
<td>Save all</td>
</tr>
<tr>
<td>Ctrl + Alt + F11</td>
<td>N/A</td>
<td>Toggle full screen mode</td>
</tr>
<tr>
<td>Ctrl + Shift + F12</td>
<td>N/A</td>
<td>Toggle maximizing editor</td>
</tr>
<tr>
<td>Ctrl + BackQuote (')</td>
<td>Control + BackQuote (')</td>
<td>Quick switch current scheme</td>
</tr>
<tr>
<td>Ctrl + Alt + S</td>
<td>Cmd + .</td>
<td>Open Settings dialog</td>
</tr>
<tr>
<td>Ctrl + Alt + C</td>
<td>Cmd + Alt + C</td>
<td>Model Checker</td>
</tr>
</tbody>
</table>
Chapter 61

Module Cloning

You may run into situations when you need to create a copy of a language or a solution. Module cloning gives you a quick way to create copies of modules.

If you want to clone a module, you select it in the project menu and click on the Clone Solution/Language action in the context menu.

In the dialog that pops up you then choose the name and the location for the new module.
After pressing OK, the module will be cloned and ready to use. The new module will contain all the code and properties from the old one. If you are cloning a language then its generator will be also cloned.

As you know every module contains lots of references to other instances (model/module dependencies, generator priority rules, node references in the code, etc). So if an instance and a reference to it are both cloned, it’s preferred that the new reference refers to the new instance. The cloning engine takes care about such situations and so you do not have to do it manually.

There are some cases when you cannot clone a module. First of all, you can clone only solutions and languages. It also matters, how models are stored in a module. In short, if all model roots in the module support cloning then the module can be cloned.

Currently, there are three model root types that are provided by MPS out of the box: default, javaclasses and java-
source_stubs. All these model roots support cloning, except for one case - when the model files are stored outside of the module directory. By default, this is not the case, so you rarely encounter any obstacles in module cloning.
Part VI

Platform Languages
Chapter 62

Base Language

The BaseLanguage is an MPS’ counterpart to Java, since it shares with Java almost the same set of constructs. BaseLanguage is the most common target of code generation in MPS and the most extensively extended language at the same time.

In order to simplify integration with Java, it is possible to specify the classpath for all modules in MPS. Classes found on the classpath will then be automatically imported into @java_stub models and so can be used directly in programs that use the BaseLanguage.

The frequently extended concepts of MPS include:

- **Expression.** Constructs, which are evaluated to some results like 1, "abc", etc.
- **Statement.** Constructs, which can be contained on a method level like if/while/synchronized statement.
- **Type.** Types of variables, like int, double.
- **IOperation.** Constructs, which can be placed after a dot like in node.parent. The parent element is a IOperation here.
- **AbstractCreator.** Constructs, which can be used to instantiate various elements.

BaseLanguage was created as a copy of Java 6. Extensions to BaseLanguage for Java 7 and 8 compatibility have been gradually added.

- Java 7 language constructs are contained in the jetbrains.mps.baselanguage.jdk7 language
- Java 8 language extensions are contained in the jetbrains.mps.baselanguage.jdk8 language
- You may like to check out a documentation dedicated to MPS Java compatibility
Chapter 63

Base Language Extensions Style Guide

*Base Language* is by far the most widely extended language in MPS. Since it is very likely that a typical MPS project will use a lot of different extensions from different sources or language vendors, the community might benefit from having a unified style across all languages. In this document we describe the conventions that creators should apply to all *Base Language* extensions.

### 63.1 Quick Reference

<table>
<thead>
<tr>
<th>If you use...</th>
<th>Set its style to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Dot</td>
</tr>
<tr>
<td>[</td>
<td>LeftBracket</td>
</tr>
<tr>
<td>]</td>
<td>RightBracket</td>
</tr>
<tr>
<td>{</td>
<td>LeftBrace</td>
</tr>
<tr>
<td>}</td>
<td>RightBrace</td>
</tr>
<tr>
<td>Operator</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td></td>
</tr>
<tr>
<td>%</td>
<td></td>
</tr>
<tr>
<td>++</td>
<td></td>
</tr>
<tr>
<td>--</td>
<td></td>
</tr>
<tr>
<td>!</td>
<td></td>
</tr>
<tr>
<td>==</td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?: (ternary)</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>:eq:</td>
<td></td>
</tr>
<tr>
<td>:ne:</td>
<td></td>
</tr>
</tbody>
</table>
@interface
abstract
assert
boolean
break
byte
case
catch
char
class
const
continue
default
do
double
double
else
elem
extends
false
final
finally
float
for
goto
if
implements
import
instanceof
int
interface
long
native
new
null
package
private
protected
public
return
set
short
static
strictfp
super
switch
synchronized
this
throw
throws
transient
try
true
void
volatile
while
63.2 Keywords

A keyword is a widely used string, which identifies important concepts from a language. For example, all the primitive types from Base Language are keywords. Also names of statements such as ifStatement, forStatement are keywords. Use the KeyWord style from base language’s stylesheet for keywords.

63.3 Curly braces

Curly braces are often used to demarcate a block of code inside of a containing construction. If you create an if-like construct, place opening curly brace on the same line as the construct header. I.e. use:

```plaintext
abc {
  code
}
```

instead of

```plaintext
abc {
  code
}
```

Use the LeftBrace and RightBrace styles to set correct offsets. Make sure that the space between a character which is to left to opening curly brace and the curly brace itself is equal to 1 space. You can do so with a help of padding-left/padding-right styles.

63.4 Parentheses

When you use parentheses, set the LeftParen/RightParen styles to the left/right parenthesis. If a parenthesis cell’s sibling is a named node’s property, disable the first/last position of a parenthesis with first/last-position-allowed style.

63.5 Identifiers

When you use named nodes: methods, variables, fields, etc, it’s advisable to make their name properties have 0 left and right padding. Making identifier declaration and reference holding the same color is also a good idea. For example, in Base Language, field declarations and references have the same color.

63.6 Punctuation

If you have a semicolon somewhere, set its style to Semicolon. If you have a dot, use the Dot style. If you have a binary operator, use the Operator style for it.
Chapter 64

MPS Java compatibility

64.1 Configuration

The Java Compiler configuration tab in the preferences window only holds a single setting – “Project bytecode version”. This setting defines the bytecode version of all Java classes compiled by MPS. These classes include classes generated from language’s aspects, classes of the runtime solutions, classes of the sandbox solutions, etc. By default, the bytecode version is set to “JDK Default”. This means that the version of the compiled classes will be equal to the version of Java, which MPS is running under. E.g. if you run MPS under JDK 1.8 and “JDK Default” is selected, the bytecode version will be 1.8. The other options for project bytecode version are 1.6, 1.7 and 1.8.

Note that MPS since version 3.4 can only run on JDK 1.8 and higher, so when compiling languages or MPS plugins you have to set the bytecode version to 1.8, otherwise your languages/plug-ins won’t be loaded. Setting the bytecode version to earlier JDK versions is only useful for solution-only projects, which are generated into Java sources that you then compile and use outside of MPS.

64.2 Build scripts

Also, don’t forget to set java compliance level in the build scripts of your project. It should be the same as the project bytecode version.
64.3 Using java classes compiled with JDK 1.8

In the MPS modules pool you can find the JDK solution, which holds the classes of the running Java. So when you start MPS under JDK 1.8, the latest Java Platform classes will be available in the JDK solution.

You can also use any external Java classes, compiled under JDK 1.8 by adding them as Java stubs. Since version 1.8, Java interfaces can contain default and static methods. At present, MPS does not support creating them in your BaseLanguage code, but you can call static and default methods defined in external Java classes, e.g. classes of the Java Platform.

64.3.1 Static interface method call

```java
public class TestClass {

    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(4, 8, 15, 16, 23, 42);
        Collections.sort(list, Comparator.reverseOrder());
        System.out.println(list);
    }
}
```

In the example, we sort a list with the `Comparator.reverseOrder()`. `Comparator` is an interface from `java.util`, and `reverseOrder()` is its static method, which was introduced in Java 1.8.

64.3.2 Default interface methods

Java 8 introduced also default methods. These are methods implemented directly in the interface. You can read about default methods here: [http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html](http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html)

These methods can be called just like the usual instance methods. Sometimes, however, you need to call the default method directly from an interface that your class is implementing. E.g. in case of multiple inheritance when a class implements several interfaces, each containing a default method with the same signature.

```java
public interface A {
    public default void foo() {
        System.out.println("A");
    }
}

public interface B {
    public default void foo() {
        System.out.println("B");
    }
}
```

In that case `foo()` can be called explicitly on one of the interfaces via a `SuperInterfaceMethodCall` construction, which is newly located in the `jetbrains.mps.baseLanguage.jdk8language`. 
64.4 Using Java platform API

Java 8 introduced lambda expressions, of which you can learn more here: [http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html](http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html).

MPS doesn’t yet have a language that would be generated into lambda-expressions. Instead, it has its own closure language, which is compatible with the new Java API!

Here’s the example of an interaction with the new JDK 8 Collections API:

```java
public class TestClass implements A, B {
    public void foo() {
        super.foo();
    }
}
```

The `forEach()` method is the new default method of `java.lang.Iterable`. It takes a `Consumer` interface as a parameter. `Consumer` is a functional interface as it only has one method. In Java 8 it would be possible to pass a lambda expression to `forEach()`. In MPS you can pass the MPS closure. A closure knows the type of the parameter taken by `forEach()` while generating and it will be generated exactly to the correct instance of the `Consumer`.

```java
public class TestClass {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add("Hello ");
        list.add("World!");
        list.forEach((string element) -> System.out.println(element));
    }
}
```
Chapter 65

Closures

65.1 Introduction

Closures are a handy extension to the base language. Not only they make code more concise, but you can use them as a vehicle to carry you through the lands of functional paradigm in programming. You can treat functions as first-class citizens in your programs - store them in variables, pass them around to methods as arguments or have methods and functions return other functions. The MPS Closures Support allows you to employ closures in your own languages. In fact, MPS itself uses closures heavily, for example, in the collections language.

This language loosely follows the "BGGA" proposal specification for closures in Java\(^1\). However, you don’t need Java 7 to run code with MPS closures. The actual implementation uses anonymous inner classes, so any recent version of Java starting with 1.5 will run the generated code without problems. Only the closures runtime jar file is required to be on the classpath of the generated solutions.

65.2 Function type

\(\{ \text{Type}_1, \text{Type}_2... \Rightarrow \text{ReturnType} \}\)

Let’s start with a trivial example of function type declaration. It declares a function that accepts no parameters and returns no value.

\(\Rightarrow \text{void}\)

Subtyping rules

A function type is covariant by its return type and contravariant by parameter types.

For example, given we have defined a method that accepts \(\{\text{String} \Rightarrow \text{Number}\}\):

```java
public void accept_Number_from_String ({String => Number} function) {
    ...
}
```

we can pass an instance of \(\{\text{Object} \Rightarrow \text{Integer}\}\) (a function that accepts Object and returns int) to this method:

```java
this.accept_Number_from_String ({Object o => o.hashCode();});
```

Simply put, you can use different actual types of parameters and the return value so long as you keep the promise made in the super-type’s signature.

Notice the int type automatically converted to boxed type Integer.

65.3 Closure literal

Closure literal is created simply by entering a following construct: \(\{ <\text{parameter decls}> \Rightarrow <\text{body}> \}\). No "new" operator is necessary.

The result type is calculated following one or more of these rules:

- last statement, if it’s an ExpressionStatement;
- return statement with an expression;
• yield statement.

Note: it's impossible to combine return and yield within a single closure literal.

## 65.4 Closure invocation

The `invoke` operation is the only method you can call on a closure. Instead of entering

```java
closure.invoke(p1,p2);
```

To invoke a closure, it is recommended to use the simplified version of this operation - parentheses enclosing the parameter list.

```java
closure(p1,p2);
```

Invoking a closure then looks like a regular method call.

Some examples of closure literal definitions.

```java
{int => int} fib = { int n => n <= 1 ? n : invoke(n-1) + invoke(n-2); }
{int => int} fact = { int n => int res = 1;
    while (n > 1) {
        res = res * n--;
    }
    res;

{=> sequence<int>} closure = {=> yield 1;};
```

### 65.4.1 Recursion

Functional programming without recursion would be like making coffee without water, so obviously you have a natural way to call recursively a closure from within its body:

```java
{int => long} fact = {int n =>
    if (n == 1) {
        return 1L;
    } else {
        return n * invoke(n - 1);
    }
};
println("Factorial of 10 = "+ fact(10));
```

A standalone `invoke` within the closure's body calls the current closure.

## 65.5 Closure conversion

For practical purposes a closure literal can be used in places where an instance of a single-method interface is expected, and vice versa³.

```java
public interface Worker {
    String doWork (int amount);
}
...
Worker worker = { int amount => "Done "+ amount + " work";};
```

The generated code is exactly the same as when using anonymous class:

```java
Worker worker = new Worker() {
    public String doWork(int amount) {
        return "Done "+ amount + " work";
    }
};
```
Think of all the places where Java requires instances of `Runnable`, `Callable` or various observer or listener classes:

```java
println("Reported from the main thread " + Thread.currentThread());
final Runnable runnable = { => println("Reported from a thread " + Thread.currentThread()); }
Thread t = new Thread(runnable);
t.start();
```

As with interfaces, an abstract class containing exactly one abstract method can also be adapted to from a closure literal. This can help, for example, in smooth transition to a new API, when existing interfaces serving as functions can be changed to abstract classes implementing the new interfaces.

### 65.6 Yield statement

The yield statement allows closures populate collections. If a yield statement is encountered within the body of a closure literal, the following are the consequences:

- if the type of yield statement expression is Type, then the result type of the closure literal is `sequence<Type>`;
- all control statements within the body are converted into a switch statement within an infinite do-while loop at the generation;
- usage of return statement is forbidden and the the value of last ExpressionStatement is ignored.

```java
sequence<int> sequence = new sequence<int> (){ => yield 1;};
yield 2;
yield 3;
```

### 65.7 Functions that return functions

A little bit of functional programming for the functional hearts out there:

```java
{ int , int => int } add = { int x , int y => x + y ; };
{ int => int } plusThree = { int x => x + 3 ; };
{ int => int } curriedPlusThree = this . curry ( add , 3 ) ;
assert plusThree . invoke ( 1 ) equals curriedPlusThree . invoke ( 1 ) ;
```

The `curry()` method is defined as follows:

```java
public { int => int } curry ( final { int , int => int } fun , final int y ) {
    return { int x => fun . invoke ( x , y ) ; } ;
}
```

### 65.8 Runtime

In order to run the code generated by the closures language, it’s necessary to add to the classpath of the solution the closures runtime library. This jar file contains the synthetic interfaces needed to support variables of function type and some utility classes. It’s located in:

```text
%MPS_HOME%/core/baseLanguage/jetbrains.mps.baseLanguage.closures.runtime.jar
```

### 65.9 Differences from the BGGA proposal

- No messing up with control flow. This means no support for control flow statements that break the boundaries of a closure literal.
- No "early return" problem: since MPS allows return to be used anywhere within the body.
• The `yield` statement.

[Closures] Closures for the Java Programming Language

[Closures] Version 0.5 of the BGGA closures specification is partially supported

[Closures] This is no longer true: only closure literal to interface conversion is supported, as an optimization measure.
Chapter 66

Collections language

An extension to the Base Language that adds support for collections.

66.1 Introduction

Collection language provides a set of abstractions that enable the use of a few most commonly used containers, as well as a set of powerful tools to construct queries. The fundamental type provided by the collections is `sequence`, which is an abstraction analogous to Iterable in Java, or IEnumerable in .NET. The containers include `list` (both array-based and linked list), `set` and `map`. The collections language also provides the means to build expressive queries using Closures, in a way similar to what LINQ does.

66.2 Null handling

Collections language has a set of relaxed rules regarding null elements and null sequences.

Null sequence is still a sequence
Null is a perfectly accepted value that can be assigned to a sequence variable. This results simply in an empty sequence.

```java
sequence<Type> nullSeq = null;
```

Null is returned instead of exception throwing
Whereas the standard collections framework would have to throw an exception as a result of calling a method that cannot successfully complete, the collection language's sequence and its subtypes would return `null` value. For example, invoking `first` operation on an empty sequence will yield a `null` value instead of throwing an exception.

```java
sequence<Type> nullSeq = null;
Type nullValue = nullSeq.first;
```

66.3 Skip and stop statements

`skip`
Applicable within a `selectMany` or `forEach` closure. The effect of the `skip` statement is that the processing of the current input element stops, and the next element (if available) is immediately selected.

`stop`
Applicable within a `selectMany` closure or a sequence initializer closure. The stop statement causes the construction of the output sequence to end immediately, ignoring all the remaining elements in the input sequence (if any).

66.4 Collections Runtime

Collections language uses a runtime library as its back end, which is designed to be extensible. Prior to version 1.5, the collections runtime library was written in Java and used only standard Java APIs. The release 1.5 brings a change: now the runtime library is available as an MPS model and uses constructs from `jetbrains.mps.baseLanguage.closures` language to facilitate passing of function-type parameters around.
Important change!
In order to make the transition from Java interfaces to abstract function types possible, several of the former Java interfaces in the collections runtime library have been changed into abstract classes. While no existing code that uses collections runtime will be broken, unfortunately this breaks the so called binary compatibility, which means that a complete recompilation of all the generated code is required to avoid incompatibility with the changed classes in the runtime.

The classes which constitute the collections runtime library can be found in the collections.runtime solution, which is available from the jetbrains.mps.baseLanguage.collections language.

66.5 Sequence
Sequence is an abstraction of the order defined on a collection of elements of some type. The only operation that is allowed on a sequence is iterating its elements from first to last. A sequence is immutable. All operations defined in the following subsections and declared to return a sequence, always return a new instance of a sequence or the original sequence. Although it is possible to create a sequence that produces infinite number of elements, it is not recommended. Some operations may require one or two full traversals of the sequence in order to compute, and invoking such an operation on an infinite sequence would never yield result.

Sequence type

sequence<Type>

<table>
<thead>
<tr>
<th>Subtypes</th>
<th>Supertypes</th>
<th>Comparable types</th>
</tr>
</thead>
<tbody>
<tr>
<td>list&lt;Type&gt;</td>
<td>none</td>
<td>java.lang.Iterable&lt;Type&gt;</td>
</tr>
<tr>
<td>set&lt;Type&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Creation

new sequence

\[
\{ \Rightarrow \text{sequence}\langle\text{Type}\rangle \}
\]

Result type

sequence<Type>

Sequence can be created with initializer.

sequence<Type> seq = new sequence<Type>(\{\Rightarrow /\ast \text{code with yield keywords }*/\})

closure invocation

Result type

sequence<Type>

A sequence may be returned from a closure (see Closures).

sequence<Type> seq = \{\Rightarrow /\ast \text{code with yield keywords producing a sequence }*/\}.invoke;

array as a sequence

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type[]</td>
<td>none</td>
<td>sequence&lt;Type&gt;</td>
</tr>
</tbody>
</table>

An array can be used as a sequence.

T[] array0fT;
sequence<T> seq = T.asSequence;

A list, a set and a map are sequences, too. All operations defined on a sequence are also available on an instance of any of these types. Sequence type is assignable to a variable of type java.lang.Iterable. The opposite is also true.
## Operations on sequence

### Iteration and querying

**foreach statement** Loop statement

```java
foreach foo in bar {
    ...
}
```

is equivalent to

```java
for (Type foo: bar) {
    ...
}
```

### forEach

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sequence&lt;Type&gt;</code></td>
<td><code>{ Type =&gt; void }</code></td>
<td><code>void</code></td>
</tr>
</tbody>
</table>

The code passed as a parameter (as a closure literal or by reference) is executed once for each element.

```
seq.forEach({=> /* arbitrary code*/});
```

### size

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sequence&lt;Type&gt;</code></td>
<td><code>none</code></td>
<td><code>int</code></td>
</tr>
</tbody>
</table>

Gives number of elements in a sequence. **isEmpty**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sequence&lt;Type&gt;</code></td>
<td><code>none</code></td>
<td><code>boolean</code></td>
</tr>
</tbody>
</table>

Test whether a sequence is empty, that is its size is 0. **isNotEmpty**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sequence&lt;Type&gt;</code></td>
<td><code>none</code></td>
<td><code>boolean</code></td>
</tr>
</tbody>
</table>

Test whether a sequence contains any elements. **indexOf**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sequence&lt;Type&gt;</code></td>
<td><code>Type</code></td>
<td><code>int</code></td>
</tr>
</tbody>
</table>

Gives the index of a first occurrence in a sequence of an element that is passed to it as a parameter.

```java
int idx = seq.indexOf(elm);
```

### contains

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sequence&lt;Type&gt;</code></td>
<td><code>Type</code></td>
<td><code>boolean</code></td>
</tr>
</tbody>
</table>

Produces boolean value, indicating whether or not a sequence contains the specified element.

```java
if (seq.contains(elm)) {
    ...
}
```

### any / all

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sequence&lt;Type&gt;</code></td>
<td><code>{ Type =&gt; boolean }</code></td>
<td><code>boolean</code></td>
</tr>
</tbody>
</table>
CHAPTER 66. COLLECTIONS LANGUAGE

Produces boolean value that indicates whether any (in case of any operation) or all (in case of all) of the elements in the input sequence match the condition specified by the closure.

```java
if (seq.any({~it=> /* condition */ })) {
  ...
}
```

**iterator**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>none</td>
<td>iterator&lt;Type&gt;</td>
</tr>
</tbody>
</table>

Produces a Sequence.  

**enumerator**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>none</td>
<td>enumerator&lt;Type&gt;</td>
</tr>
</tbody>
</table>

Produces an Sequence.

**Selection and filtering**  

**first**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>none</td>
<td>Type</td>
</tr>
</tbody>
</table>

Yields the first element.  

**last**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>none</td>
<td>Type</td>
</tr>
</tbody>
</table>

Yields the last element.  

**take**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>int</td>
<td>sequence&lt;Type&gt;</td>
</tr>
</tbody>
</table>

Produces a sequence that is sub-sequence of the original one, starting from first element and of size count.

```
seq.take (count)
```

**skip**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>int</td>
<td>sequence&lt;Type&gt;</td>
</tr>
</tbody>
</table>

Produces a sequence that is sub-sequence of the original one, containing all elements starting with the element at index count.

```
seq.skip (count)
```

**cut**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>int</td>
<td>sequence&lt;Type&gt;</td>
</tr>
</tbody>
</table>

Produces a sequence that is a sub-sequence of the original one, containing all elements starting with first and up to (but not including) the element at index size minus count. In other words, this operation returns a sequence with all elements from the original one except the last count elements.

```
seq.cut (count)
```

**tail**
### OPERATOR: seq.tail (count)

**Operand type**
- `sequence<Type>`

**Parameter type**
- `int`

**Result type**
- `sequence<Type>`

Produces a sequence that is a sub-sequence of the original one, containing all elements starting with the element at index `size` minus `count`. In other words, this operation returns a sequence with `count` elements from the end of the original sequence, in the original order.

### OPERATOR: seq.page (start, end)

**Operand type**
- `sequence<Type>`

**Parameter type**
- `int`
- `int`

**Result type**
- `sequence>Type`

Results in a sequence that is a sub-sequence of the original one, containing all elements starting with the element at index `start` and up to (but not including) the element at index `end`. It is a requirement that `start` is no greater than `end`.

This is equivalent to

```plaintext
seq.skip (skip).take (count)
```

Where `skip = start`, `count = end - start`.

### OPERATOR: seq.where ({ => ... })

**Operand type**
- `sequence<Type>`

**Parameter type**
- `{ Type => boolean }`

**Result type**
- `sequence<Type>`

Produces a sequence that is a sub-sequence of the original one, with all elements for which the code passed as a parameter returns true.

### OPERATOR: seq.findFirst ({ => ... })

**Operand type**
- `sequence<Type>`

**Parameter type**
- `{ Type => boolean }`

**Result type**
- `Type`

Results in the first element that matches the parameter closure.

### OPERATOR: seq.findLast ({ => ... })

**Operand type**
- `sequence<Type>`

**Parameter type**
- `{ Type => boolean }`

**Result type**
- `Type`

Results in the last element that matches the parameter closure.

### OPERATOR: select

**Operand type**
- `sequence<Type>`

**Parameter type**
- `{ Type => Type2 }`

**Result type**
- `sequence>Type2`

Results in a sequence consisting of elements, each of which is the result of applying the parameter function to each element of the original sequence in turn.
seq.select ({ Type element => /* code returning single element of same or some other type */ })

selectMany

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>{ Type =&gt; sequence&lt;Type2&gt; }</td>
<td>sequence&lt;Type2&gt;</td>
</tr>
</tbody>
</table>

Produces a sequence that is a concatenation of all sequences, which are all the results of applying the parameter closure to each element of the original sequence in turn. The statements skip and stop are available within the parameter closure.

seq.selectMany ({ <Type> element => /* code returning a sequence. May utilize yield keyword.*/ })

distinct

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>none</td>
<td>sequence&lt;Type&gt;</td>
</tr>
</tbody>
</table>

Produces a sequence, which contains all elements from the original sequence in the original order, with all the elements having cardinality exactly 1. Of all occurrences of an element in the original sequence only the first occurrence is included in the resulting sequence. sortBy

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>{ Type =&gt; Type2 }</td>
<td>sequence&lt;Type&gt;</td>
</tr>
</tbody>
</table>

Produces a sequence with all elements from the original one in the order, which corresponds to an order induced by an imaginary sequence produced by applying the selector function to each element in the original sequence in turn. The selector function can be thought of as returning a key, which is used to sort elements in a sequence. The ascending parameter controls the sort order.

seq.sortBy ({ => /* code returning a sorting key */ }, ascending)

alsoSortBy

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>{ Type =&gt; Type2 }</td>
<td>sequence&lt;Type&gt;</td>
</tr>
</tbody>
</table>

Equivalent to sortBy, unless used as a chain operation immediately following sortBy or another alsoSortBy. The result is a sequence sorted with a compound key, with the first component taken from previous sortBy or alsoSortBy (which is also a compound key), and the last component taken from this operation.

seq.sortBy ({ => /* code returning a sorting key */ }, ascending).alsoSortBy ({ => /* code returning another key */ }, descending)

sort

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>{ Type, Type =&gt; int }</td>
<td>boolean sequence&lt;Type&gt;</td>
</tr>
</tbody>
</table>

Produces a sequence containing all elements from the original one in the order produced by applying the comparator function (passed as a closure literal or by reference) to a list with elements from the original sequence. The ascending parameter controls the sort order (order is reversed if the value is false).

seq.sort ({ <Type> a, <Type> b => /* code returning -1, 0 or 1 according to java.util.Comparator */ }, ascending)

Binary operations  All operations in this section have the semantics of operations on multisets. This means that the elements of both the operand and the parameter sequences are taken with their respective cardinalities (that is, how many times an element appears in the sequence), and then the cardinality of matched elements in the output is the result of an arithmetic operation on these values. These operations keep the order of elements (first operand’s, then the parameter’s). intersect

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**CHAPTER 66. COLLECTIONS LANGUAGE**

<table>
<thead>
<tr>
<th>sequence&lt;Type&gt;</th>
<th>sequence&lt;Type&gt;</th>
<th>sequence&lt;Type&gt;</th>
</tr>
</thead>
</table>

Produces a sequence containing elements contained both by the original sequence and the parameter sequence. More precisely, the cardinality in the output of an element appearing in both the operand and the parameter sequences, is the minimum of its cardinalities in the input.

```scala
seq.intersect (anotherSeq)
```

**except**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>sequence&lt;Type&gt;</td>
<td>sequence&lt;Type&gt;</td>
</tr>
</tbody>
</table>

Produces a sequence containing all elements from the original sequence that are not also members of the parameter sequence. More precisely, the cardinality in the output of an element appearing in both the operand and the parameter sequences, is the maximum of 0 and the result of subtracting the parameter's cardinality from the operand's cardinality.

```scala
seq.except (anotherSeq)
```

**union**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>sequence&lt;Type&gt;</td>
<td>sequence&lt;Type&gt;</td>
</tr>
</tbody>
</table>

Produces a sequence containing elements either from the original sequence or the one passed as a parameter. More precisely, the cardinality in the output of an element appearing in both the operand and the parameter sequences, is the maximum of its cardinalities in the input.

```scala
seq.union (anotherSeq)
```

**disjunction**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>sequence&lt;Type&gt;</td>
<td>sequence&lt;Type&gt;</td>
</tr>
</tbody>
</table>

Produces exclusive disjunction (symmetric difference) of the original sequence and the one passed as a parameter. This operation equivalent to:

```scala
(operand.except(param)).concat(parameter.except(operand))
```

```scala
seq.disjunction (anotherSeq)
```

**concat**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>sequence&lt;Type&gt;</td>
<td>sequence&lt;Type&gt;</td>
</tr>
</tbody>
</table>

Produces a sequence, which is a concatenation of the original one with the sequence passed as a parameter.

```scala
seq.concat (otherSeq)
```

**Conversion reduceLeft / reduceRight**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>{ Type, Type =&gt; Type }</td>
<td>Type</td>
</tr>
</tbody>
</table>

Operation reduceLeft/reduceRight applies the combinator function passed as a parameter to all elements of the sequence in turn. One of the function parameters is a sequence element, and another is the result of previous application of the function. Operation reduceLeft takes first two sequence elements and applies the function to them, then takes the result of the first application and the third sequence element, etc. Operation reduceRight does the same, but moving from the
sequence’s tail backwards.

```java
sequence<int> int = ...
int sum = in.reduceLeft({-a,-b => a + b; });
```

- **reduceLeft**

```java
{Type,Type => Type} f = ... // represented as *
sequence<Type> seq = new sequence<Type>{A,B,C,D,E,F}; A B C D E F
seq.reduceLeft(f) == f(f(f(f(A,B),C),D),E),F); \ | | | | |
\ | | | | */
\ | | | /*
\ | | */
\ | */
\ |*/
result of reduceLeft operation --> *
```

- **reduceRight**

```java
{Type,Type => Type} f = ... // represented as *
sequence<Type> seq = new sequence<Type>{A,B,C,D,E,F} A B C D E F
seq.reduceRight(f) == f(A,f(B,f(C,f(D,f(E,F)))))); | | | | | / 
| | | | /*
| | | */
| | */
| |/*
| */
result of reduceRight operation --> *
```

**foldLeft / foldRight**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
<th>Applicable for</th>
</tr>
</thead>
<tbody>
<tr>
<td>seed</td>
<td>{ Z, Type =&gt; Z }</td>
<td>Z</td>
<td>foldLeft</td>
</tr>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>{ Type, Z =&gt; Z }</td>
<td>Z</td>
<td>foldRight</td>
</tr>
</tbody>
</table>

Operation foldLeft/foldRight behaves similarly to reduceLeft/reduceRight, with the difference that it also accepts a seed value. Also the combinator function is asymmetric (it takes a Type and a Z parameters and returns a Z value). The result of the operation is of type Z.

```java
sequence<int> int = ...
string concat = in.foldLeft("",{string s,-it => "" + s + it; });
```

- **foldLeft**
{Z, Type => Z} \( f = \ldots \) // represented as *
Z \( z = \ldots \) // the "seed" value

```
sequence<Type> seq = new sequence<Type>{A,B,C,D,E,F};
seq.reduceLeft(f) == f(f(f(f(f(f(z,A),B),C),D),E),F);
```

result of foldLeft operation --> *

• foldRight

```
{Type, Z => Z} \( f = \ldots \) // represented as *
Z \( z = \ldots \) // the "seed" value
sequence<Type> seq = new sequence<Type>{A,B,C,D,E,F};
```

seq.foldRight(f) == f(A,f(B,f(C,f(D,f(E,f(F,z))))));

result of foldRight operation --> *

join

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt; ? extends string &gt;</td>
<td>string (optional)</td>
<td>string</td>
</tr>
</tbody>
</table>

This operation is only available on a sequence of strings. The result is a string that is produced by concatenating all elements with the optional separator. The default separator is " " (single space).

```
seq.join ("; ");
```

toList

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>none</td>
<td>list&lt;Type&gt;</td>
</tr>
</tbody>
</table>

Returns new list containing all the elements from the original sequence.

```
seq.toList
```

toArray

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>none</td>
<td>Type*[*]</td>
</tr>
</tbody>
</table>

Returns new array containing all the elements from the original sequence.
66.6 List

A basic list container backed by either array list or linked list.

### List type

```
list<
```

#### Subtypes

<table>
<thead>
<tr>
<th>Supertypes</th>
<th>Comparable types</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>java.util.List&lt;Type&gt;</td>
</tr>
<tr>
<td><code>sequence&lt;Type&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>

#### List creation

- new arraylist
- new linkedlist

<table>
<thead>
<tr>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Type...</code></td>
<td><code>list&lt;Type&gt;</code></td>
</tr>
<tr>
<td><code>{*}sequence&lt;? extends *Type&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>

Creates an empty list. Optionally, initial values may be specified right in the new list creation expression.

```
list<Type> = new arraylist<Type> { /* initial values */ };
```

Alternatively, a sequence may be specified that is used to copy elements from.

```
list<Type> = new arraylist<Type> (sequence);
```

### Operations on list

#### iterator

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sequence&lt;Type&gt;</code></td>
<td>none</td>
<td><code>modifying_iterator&lt;Type&gt;</code></td>
</tr>
</tbody>
</table>

This operation is redefined for list to return a `modifying_iterator`.

#### get

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>list&lt;Type&gt;</code></td>
<td><code>int</code></td>
<td><code>Type</code></td>
</tr>
</tbody>
</table>

Yields the element at **index** position.

```
list.get (index);
```

- **indexed access**

```
list[index];
```

#### set

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>list&lt;Type&gt;</code></td>
<td><code>Type</code></td>
<td><code>list&lt;Type&gt;</code></td>
</tr>
</tbody>
</table>
The collection language provides several operations to manipulate lists:

- **list.set(index, value)**: Sets the element at `index` position to the specified value. Yields the new value.

- **indexed access**
  
  ```java
  list[index] = value;
  ```

- **add**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list&lt;Type&gt;</td>
<td>Type</td>
<td>Type</td>
</tr>
</tbody>
</table>

  Adds an element to the list.

  ```java
  list.add(value);
  ```

- **addFirst**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list&lt;Type&gt;</td>
<td>Type</td>
<td>Type</td>
</tr>
</tbody>
</table>

  Adds an element to the list as the first element.

  ```java
  list.addFirst(value);
  ```

- **addLast**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list&lt;Type&gt;</td>
<td>Type</td>
<td>Type</td>
</tr>
</tbody>
</table>

  Adds an element to the list as the last element.

  ```java
  list.addLast(value);
  ```

- **insert**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list&lt;Type&gt;</td>
<td>int</td>
<td>Type</td>
</tr>
</tbody>
</table>

  Inserts an element into the list at the position `index`.

  ```java
  list.insert(index, value);
  ```

- **remove**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>list&lt;Type&gt;</td>
<td>Type</td>
<td>Type</td>
</tr>
</tbody>
</table>
Removes an element from the list.

```java
list.remove (value);
```

**removeFirst**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>list&lt;Type&gt;</code></td>
<td>none</td>
<td><code>Type</code></td>
</tr>
</tbody>
</table>

Removes the first element from the list.

```java
list.removeFirst;
```

**removeLast**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>list&lt;Type&gt;</code></td>
<td>none</td>
<td><code>Type</code></td>
</tr>
</tbody>
</table>

Removes the last element from the list.

```java
list.removeLast;
```

**removeAt**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>list&lt;Type&gt;</code></td>
<td><code>int</code></td>
<td><code>Type</code></td>
</tr>
</tbody>
</table>

Removes an element from the list located at the position `index`.

```java
list.removeAt (index);
```

**addAll**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>list&lt;Type&gt;</code></td>
<td><code>sequence&lt;Type&gt;</code></td>
<td><code>list&lt;Type&gt;</code></td>
</tr>
</tbody>
</table>

Adds all elements in the parameter sequence to the list.

```java
list.addAll (seq);
```

**removeAll**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>list&lt;Type&gt;</code></td>
<td><code>sequence&lt;Type&gt;</code></td>
<td><code>list&lt;Type&gt;</code></td>
</tr>
</tbody>
</table>

Removes all elements in the parameter sequence from the list.

```java
list.removeAll (seq);
```

**clear**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>list&lt;Type&gt;</code></td>
<td>none</td>
<td><code>void</code></td>
</tr>
</tbody>
</table>
Clears all elements from the list.

```
list.clear;
```

**reverse**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>list&lt;Type&gt;</code></td>
<td>none</td>
<td><code>list&lt;Type&gt;</code></td>
</tr>
</tbody>
</table>

Produces a list with all elements from the original list in the reversed order.

```
list.reverse;
```

---

**Important**

The `reverse` operation does not modify the original list, but rather produces another list.

### 66.7 Set

A basic set container backed by either hash set or linked hash set.

**Set type**

```
set<Type>
```

<table>
<thead>
<tr>
<th>Subtypes</th>
<th>Supertypes</th>
<th>Comparable types</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sorted_set&lt;Type&gt;</code></td>
<td><code>sequence&lt;Type&gt;</code></td>
<td><code>java.util.Set&lt;Type&gt;</code></td>
</tr>
</tbody>
</table>

**Set creation**

- `new HashSet<Type>`
- `new LinkedHashSet<Type>`

<table>
<thead>
<tr>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Type...</code></td>
<td><code>set&lt;Type&gt;</code></td>
</tr>
<tr>
<td><code>Sequence&lt;? extends Type&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>

Creates an empty set. Optionally, initial values may be specified right in the new set creation expression.

```
set<string> test = new HashSet<string> {"A", "B", "C"};
```

Alternatively, a sequence may be specified that is used to copy elements from.

```
set<Type> = new HashSet<Type> (sequence);
```

**Operations on set**

**iterator**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Sequence&lt;Type&gt;</code></td>
<td>none</td>
<td><code>modifying_iterator&lt;Type&gt;</code></td>
</tr>
</tbody>
</table>

This operation is redefined for `set` to return a `modifying_iterator`. 

**add**
### CHAPTER 66. COLLECTIONS LANGUAGE

#### 66.8 Sorted Set

A subtype of `set` that provides iteration over its elements in the natural sorting order, backed by a tree set.

**Sorted Set type**

<table>
<thead>
<tr>
<th>Subtypes</th>
<th>Supertypes</th>
<th>Comparable types</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td><code>set&lt;Type&gt;</code></td>
<td><code>java.util.SortedSet&lt;Type&gt;</code></td>
</tr>
</tbody>
</table>

---

**Operand type** | **Parameter type** | **Result type**
---|---|---
`set<Type>` | `Type` | `Type`

Adds an element to the set.

```java
set.add (value);
```

**addAll**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set&lt;Type&gt;</code></td>
<td><code>sequence&lt;Type&gt;</code></td>
<td><code>set&lt;Type&gt;</code></td>
</tr>
</tbody>
</table>

Adds all elements in the parameter sequence to the set.

```java
set.addAll (seq);
```

**remove**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set&lt;Type&gt;</code></td>
<td><code>Type</code></td>
<td><code>Type</code></td>
</tr>
</tbody>
</table>

Removes an element from the set.

```java
set.remove (value);
```

**removeAll**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set&lt;Type&gt;</code></td>
<td><code>sequence&lt;Type&gt;</code></td>
<td><code>set&lt;Type&gt;</code></td>
</tr>
</tbody>
</table>

Removes all elements in the parameter sequence from the set.

```java
set.removeAll (seq);
```

**clear**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set&lt;Type&gt;</code></td>
<td>none</td>
<td><code>void</code></td>
</tr>
</tbody>
</table>

Clears all elements from the set.

```java
set.clear;
```
Sorted set creation

new treeset

<table>
<thead>
<tr>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>set&gt;Type</td>
</tr>
</tbody>
</table>

Creates an empty set. Optionally, initial values may be specified right in the new set creation expression.

```java
sorted_set<string> test = new treeset<string> {"A", "B", "C"};
```

Alternatively, a sequence may be specified that is used to copy elements from.

```java
sorted_set<Type> = new treeset<Type> (sequence);
```

Operations on sorted set

headSet

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorted_set&lt;Type&gt;</td>
<td>Type</td>
<td>sorted_set&lt;Type&gt;</td>
</tr>
</tbody>
</table>

Results in a sorted_set that is a subset of all elements from the original set in the original sorting order, starting with the first element and up to but not including the specified element.

```java
set.headSet (upTo);
```

tailSet

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorted_set&lt;Type&gt;</td>
<td>Type</td>
<td>sorted_set&lt;Type&gt;</td>
</tr>
</tbody>
</table>

Results in a sorted_set that is a subset of all elements from the original set in the original sorting order, starting with the specified element.

```java
set.tailSet (from);
```

subSet

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sorted_set&lt;Type&gt;</td>
<td>Type, Type</td>
<td>sorted_set&lt;Type&gt;</td>
</tr>
</tbody>
</table>

Results in a sorted_set that is a subset of all elements from the original set in the original sorting order, starting with the first specified element and up to but not including the second specified element.

```java
set.tailSet (from,upTo);
```

66.9 Map

A map container backed by either a hash map or a linked hash map.

Map type

map<KeyType, ValueType>
The map type is retrofitted to be a subtype of `sequence`.

**Map creation**

- `new hashmap`
- `new linked_hashmap`

```
Parameter type                                      Result type
(KeyType => ValueType)...
map<KeyType, ValueType>
```

Creates an empty map. Optionally, initial values may be specified right in the new map creation expression.

```
map<string, int> map = new hashmap<string, int> {"A" = 1, "B" = 2, "C" = 3};
```

**Operations on map**

**get value by key**

```
Operand type   Parameter type   Result type
map<KeyType, ValueType>   KeyType   ValueType
```

```
int value = map["key"];
```

**keys**

```
Operand type   Parameter type   Result type
map<KeyType, ValueType>   none   sequence<KeyType>
```

Results in a sequence containing all the keys in the map.

```
sequence<string> keys = map.keys;
```

**containsKey**

```
Operand type   Parameter type   Result type
map<KeyType, ValueType>   KeyType   boolean
```

Returns true if the map contains a mapping for the specified key, false otherwise.

```
map.containsKey ("key");
```

**values**

```
Operand type   Parameter type   Result type
map<KeyType, ValueType>   none   sequence<ValueType>
```

Results in a sequence containing all the values in the map.

```
sequence<string> values = map.values;
```

**containsValue**

---

### Table: Subtypes, Supertypes, Comparable types

<table>
<thead>
<tr>
<th>Subtypes</th>
<th>Supertypes</th>
<th>Comparable types</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sorted_map&lt;KeyType, ValueType&gt;</code></td>
<td><code>sequence&lt;KeyType, ValueType&gt;</code></td>
<td><code>java.util.Map&lt;KeyType, ValueType&gt;</code></td>
</tr>
</tbody>
</table>
Returns true if the map contains a mapping with the specified value, false otherwise.

```java
map.containsValue ("value");
```

### mappings

Results in a set of mappings contained by this map. The mappings can be removed from the set, but not added.

```java
set<mappings<K,V>> = map.mappings;
```

#### assign value to a key

```java
map["key"] = value;
```

#### remove

Removes the specified key and the associated value from the map.

```java
map.remove ("key");
```

#### clear

Clears all key-value pairs from the map.

```java
map.clear;
```

#### putAll

Puts all mappings from the map specified as a parameter into this map, replacing existing mappings.

```java
map.putAll (anotherMap);
```

### 66.10 Sorted Map

A subtype of `map` that provides iteration over keys conforming to the natural sorting order, backed by a tree map.

**Sorted map type**
map<KeyType, ValueType>
### Subtypes, Supertypes, Comparable types

<table>
<thead>
<tr>
<th>Subtypes</th>
<th>Supertypes</th>
<th>Comparable types</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>map&lt;KeyType, ValueType&gt;</td>
<td>java.util.SortedMap&lt;KeyType, ValueType&gt;</td>
</tr>
</tbody>
</table>

### Sorted map creation

**new treemap**

<table>
<thead>
<tr>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(KeyType =&gt; ValueType) ...</td>
<td>map&lt;KeyType, ValueType&gt;</td>
</tr>
</tbody>
</table>

Creates an empty tree map. Optionally, initial values may be specified right in the new map creation expression.

```java
sorted_map<string, int> map = new treemap<string, int> {"A" => 1, "B" => 2, "C" => 3};
```

#### Operations on sorted map

**headMap**

```java
map.headMap (upTo);
```

**tailMap**

```java
map.tailMap (from);
```

**subMap**

```java
map.tailMap (from,upTo);
```

### 66.11 Stack

A simple stack abstraction, backed by linked list.

**Stack type**

```java
stack<Type>
```
### Stack creation

**new linkedlist**

<table>
<thead>
<tr>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt; ? extends Type &gt;</td>
<td>stack&lt; Type &gt;</td>
</tr>
</tbody>
</table>

Creates an empty stack. Optionally, initial values may be specified right in the new linked list creation expression.

```
stack<string> test = new linkedlist<string> {"A", "B", "C"};
```

Alternatively, a sequence may be specified that is used to copy elements from.

```
stack<Type> = new linkedlist<Type> (sequence);
```

### Operations on stack

**iterator**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt; Type &gt;</td>
<td>none</td>
<td>modifying_iterator&lt; Type &gt;</td>
</tr>
</tbody>
</table>

This operation is redefined for stack to return a `modifying_iterator`.

**addFirst / push**

```
set.addFirst (value);
set.push (value);
```

**removeFirst / pop**

```
set.removeFirst (value);
set.pop (value);
```

**first / peek**

```
set.first (value);
set.peek (value);
```

Retrieves the first element at the head of the stack without removing it.
set.first();
set.peek();

---

## 66.12 Queue

A simple queue abstraction, backed by linked list or priority queue.

### Queue type

```java
queue<Type>
```

<table>
<thead>
<tr>
<th>Subtypes</th>
<th>Supertypes</th>
<th>Comparable types</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>deque&lt;Type&gt;</code></td>
<td><code>sequence&lt;Type&gt;</code></td>
<td><code>java.util.Deque&lt;Type&gt;</code></td>
</tr>
</tbody>
</table>

### Queue creation

- `new linkedlist`  
- `new priority_queue`

<table>
<thead>
<tr>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sequence&lt;? extends Type&gt;</code></td>
<td><code>queue&lt;Type&gt;</code></td>
</tr>
</tbody>
</table>

Creates an empty queue. Optionally, initial values may be specified right in the new linked list creation expression.

```java
queue<string> test = new linkedlist<string> {"A", "B", "C"};
```

Alternatively, a sequence may be specified that is used to copy elements from.

```java
queue<Type> = new linkedlist<Type> (sequence);
```

### Operations on queue

#### iterator

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sequence&lt;Type&gt;</code></td>
<td>none</td>
<td><code>modifying_iterator&lt;Type&gt;</code></td>
</tr>
</tbody>
</table>

This operation is redefined for `queue` to return a `modifying_iterator`.

#### addLast

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>queue&lt;Type&gt;</code></td>
<td><code>Type</code></td>
<td><code>Type</code></td>
</tr>
</tbody>
</table>

Appends an element to the tail of the queue.

```java
set.addLast (value);
```

#### removeFirst

| Operand type       | Parameter type | Result type | |
|--------------------|----------------|-------------|
| `queue<Type>`      |                | `Type`      | |

Removes an element from the head of the queue.

```java
set.removeFirst (value);
```
set.removeFirst (value);

first

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue&lt;Type&gt;</td>
<td></td>
<td>Type</td>
</tr>
</tbody>
</table>

Retrieves the first element at the head of the queue without removing it.

set.first ();

last

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue&lt;Type&gt;</td>
<td></td>
<td>Type</td>
</tr>
</tbody>
</table>

Retrieves the first element at the tail of the queue without removing it.

set.first ();

—

66.13 Deque

A simple double-linked queue abstraction, backed by linked list.

Deque type

queue<Type>

<table>
<thead>
<tr>
<th>Subtypes</th>
<th>Supertypes</th>
<th>Comparable types</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence&lt;Type&gt;</td>
<td>queue&lt;Type&gt;</td>
<td>java.util.Deque&lt;Type&gt;</td>
</tr>
<tr>
<td>stack&lt;Type&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Deque creation

new LinkedList

<table>
<thead>
<tr>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type...</td>
<td></td>
</tr>
<tr>
<td>sequence&lt;? extends Type&gt;</td>
<td>deque&lt;Type&gt;</td>
</tr>
</tbody>
</table>

Creates an empty deque. Optionally, initial values may be specified right in the new linked list creation expression.

deque<string> test = new LinkedList<string> {"A", "B", "C"};

Alternatively, a sequence may be specified that is used to copy elements from.

deque<Type> = new LinkedList<Type> (sequence);

Operations on deque

iterator

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**sequence**<Type> | none | **modifying_iterator**<Type>

This operation is redefined for deque to return a **modifying_iterator**.

### addFirst / push

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>deque&lt;Type&gt;</td>
<td>Type</td>
<td>Type</td>
</tr>
</tbody>
</table>

Appends an element to the head of the deque.

```
set.addFirst (value);
```

### addLast

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>deque&lt;Type&gt;</td>
<td>Type</td>
<td>Type</td>
</tr>
</tbody>
</table>

Appends an element to the tail of the deque.

```
set.addLast (value);
```

### removeFirst / pop

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>deque&lt;Type&gt;</td>
<td></td>
<td>Type</td>
</tr>
</tbody>
</table>

Removes an element from the head of the deque.

```
set.removeFirst (value);
```

### first

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue&lt;Type&gt;</td>
<td></td>
<td>Type</td>
</tr>
</tbody>
</table>

Retrieves the first element at the head of the deque without removing it.

```
set.first ();
```

### last

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue&lt;Type&gt;</td>
<td></td>
<td>Type</td>
</tr>
</tbody>
</table>

Retrieves the first element at the tail of the deque without removing it.

```
set.first ();
```

---

### 66.14 Iterator

A helper type that is analogous to java.util.Iterator. An instance of the iterator can be obtained with the `iterator` operation on a sequence.
**Iterator type**

**iterator**<Type>

<table>
<thead>
<tr>
<th>Subtypes</th>
<th>Supertypes</th>
<th>Comparable types</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>none</td>
<td>java.util.Iterator&lt;Type&gt;</td>
</tr>
</tbody>
</table>

**Operations on iterator**

**hasNext**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterator&lt;Type&gt;</td>
<td>none</td>
<td>boolean</td>
</tr>
</tbody>
</table>

Tests if there is an element available.

**next**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterator&lt;Type&gt;</td>
<td>none</td>
<td>Type</td>
</tr>
</tbody>
</table>

Returns the next element.

### 66.15 Modifying Iterator

A subtype of **iterator** that supports **remove** operation.

**Modifying Iterator type**

**modifying_iterator**<Type>

<table>
<thead>
<tr>
<th>Subtypes</th>
<th>Supertypes</th>
<th>Comparable types</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>iterator&lt;Type&gt;</td>
<td>java.util.Iterator&lt;Type&gt;</td>
</tr>
</tbody>
</table>

**Operations on modifying iterator**

**remove**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>modifying_iterator&lt;Type&gt;</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

Removes the element this iterator is currently positioned at.

### 66.16 Enumerator

An alternative to the iterator, a helper type that works similar to .NET’s IEnumerater. An instance of the enumerator can be obtained with the **enumerator** operation on a sequence.

**Enumerator type**

**enumerator**<Type>

<table>
<thead>
<tr>
<th>Subtypes</th>
<th>Supertypes</th>
<th>Comparable types</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

**Operations on enumerator**

**moveNext**

<table>
<thead>
<tr>
<th>Operand type</th>
<th>Parameter type</th>
<th>Result type</th>
</tr>
</thead>
</table>
CHAPTER 66. COLLECTIONS LANGUAGE

enumerator<Type> none boolean

Moves to the next element. Returns true if there is an element available.

current

Returns the element this enumerator is currently positioned at.

66.17 Mapping

A helper type used by map and sorted_map.

Mapping type

class mapping<KeyType, ValueType>

Operations on mapping

generate value

Operand type Parameter type Result type

type mapping<KeyType, ValueType> none ValueType

set value

Operand type Parameter type Result type

type mapping<KeyType, ValueType> ValueType ValueType

type mapping.value = newValue;

generate key

Operand type Parameter type Result type

type map<KeyType, ValueType> none Key Type

type mapping.key;

66.18 Custom Containers

Custom containers is a simple way to provide own implementation of standard container types, thus allowing for easy extensibility of the collections language.

Example: weakHashMap

Provided the following declaration is reachable from the model currently being edited...
custom containers WeakCollections {

    public container weakHashMap<K, V> specifies map<K, V> {
        runtime type: WeakHashMap<K, V>
        factory: <no factory>
    }

    public container weakHashSet<E> specifies set<E> {
        runtime type: WeakSet<E>
        factory: <no factory>
    }

}

... one can use the weak version of hashmap thusly:

map<Object, int> myMap= new weakHashMap<Object, int>;

Custom Containers Declaration

A root node of concept CustomContainers may have one or more declarations.

[public|protected|private] container containerName<TypeVar, ...> specifies container_type<TypeVar, ...> {
    runtime type: Classifier<TypeVar, ...>
    factory: expression of type Classifier<> (optional)
}

<table>
<thead>
<tr>
<th>declaration part</th>
<th>allowed contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>containerName</td>
<td>any valid identifier</td>
</tr>
<tr>
<td>container_type</td>
<td>one of the existing container types of the collections language</td>
</tr>
<tr>
<td>runtime type</td>
<td>Java classifier which represent implementation of the container</td>
</tr>
<tr>
<td>factory</td>
<td>(optional) container creation expression; the classifier's default constructor used if undefined</td>
</tr>
</tbody>
</table>

66.19  Primitive Containers

Collections framework include a set of custom containers designed to work with primitive data types. Using primitive types helps optimize speed and/or size of the containers. These containers are available with a separate language jetbrains.mps.baseLanguage.collections.trove.

Primitive list containers

list<?,?>

<table>
<thead>
<tr>
<th>list&lt;byte&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>byteArrayList</td>
</tr>
<tr>
<td>doubleArrayList</td>
</tr>
<tr>
<td>floatArrayList</td>
</tr>
<tr>
<td>intArrayList</td>
</tr>
<tr>
<td>longArrayList</td>
</tr>
<tr>
<td>shortArrayList</td>
</tr>
</tbody>
</table>

Primitive set containers

set<?,?>

<table>
<thead>
<tr>
<th>set&lt;byte&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>byteHashSet</td>
</tr>
<tr>
<td>doubleHashSet</td>
</tr>
<tr>
<td>floatHashSet</td>
</tr>
<tr>
<td>intHashSet</td>
</tr>
<tr>
<td>longHashSet</td>
</tr>
<tr>
<td>shortHashSet</td>
</tr>
</tbody>
</table>
## Primitive maps

### `map<byte,?>`

<table>
<thead>
<tr>
<th>Key Type</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>byteByteHashMap</code></td>
<td><code>map&lt;byte, byte&gt;</code></td>
</tr>
<tr>
<td><code>byteDoubleHashMap</code></td>
<td><code>map&lt;byte, double&gt;</code></td>
</tr>
<tr>
<td><code>byteFloatHashMap</code></td>
<td><code>map&lt;byte, float&gt;</code></td>
</tr>
<tr>
<td><code>byteIntHashMap</code></td>
<td><code>map&lt;byte, int&gt;</code></td>
</tr>
<tr>
<td><code>byteLongHashMap</code></td>
<td><code>map&lt;byte, long&gt;</code></td>
</tr>
<tr>
<td><code>byteShortHashMap</code></td>
<td><code>map&lt;byte, short&gt;</code></td>
</tr>
</tbody>
</table>

### `map<double,?>`

<table>
<thead>
<tr>
<th>Key Type</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>doubleByteHashMap</code></td>
<td><code>map&lt;double, byte&gt;</code></td>
</tr>
<tr>
<td><code>doubleDoubleHashMap</code></td>
<td><code>map&lt;double, double&gt;</code></td>
</tr>
<tr>
<td><code>doubleFloatHashMap</code></td>
<td><code>map&lt;double, float&gt;</code></td>
</tr>
<tr>
<td><code>doubleIntHashMap</code></td>
<td><code>map&lt;double, int&gt;</code></td>
</tr>
<tr>
<td><code>doubleLongHashMap</code></td>
<td><code>map&lt;double, long&gt;</code></td>
</tr>
<tr>
<td><code>doubleShortHashMap</code></td>
<td><code>map&lt;double, short&gt;</code></td>
</tr>
</tbody>
</table>

### `map<float,?>`

<table>
<thead>
<tr>
<th>Key Type</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>floatByteHashMap</code></td>
<td><code>map&lt;float, byte&gt;</code></td>
</tr>
<tr>
<td><code>floatDoubleHashMap</code></td>
<td><code>map&lt;float, double&gt;</code></td>
</tr>
<tr>
<td><code>floatFloatHashMap</code></td>
<td><code>map&lt;float, float&gt;</code></td>
</tr>
<tr>
<td><code>floatIntHashMap</code></td>
<td><code>map&lt;float, int&gt;</code></td>
</tr>
<tr>
<td><code>floatLongHashMap</code></td>
<td><code>map&lt;float, long&gt;</code></td>
</tr>
<tr>
<td><code>floatShortHashMap</code></td>
<td><code>map&lt;float, short&gt;</code></td>
</tr>
</tbody>
</table>

### `map<int,?>`

<table>
<thead>
<tr>
<th>Key Type</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>intByteHashMap</code></td>
<td><code>map&lt;int, byte&gt;</code></td>
</tr>
<tr>
<td><code>intDoubleHashMap</code></td>
<td><code>map&lt;int, double&gt;</code></td>
</tr>
<tr>
<td><code>intFloatHashMap</code></td>
<td><code>map&lt;int, float&gt;</code></td>
</tr>
<tr>
<td><code>intIntHashMap</code></td>
<td><code>map&lt;int, int&gt;</code></td>
</tr>
<tr>
<td><code>intLongHashMap</code></td>
<td><code>map&lt;int, long&gt;</code></td>
</tr>
<tr>
<td><code>intShortHashMap</code></td>
<td><code>map&lt;int, short&gt;</code></td>
</tr>
</tbody>
</table>

### `map<long,?>`

<table>
<thead>
<tr>
<th>Key Type</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>longByteHashMap</code></td>
<td><code>map&lt;long, byte&gt;</code></td>
</tr>
<tr>
<td><code>longDoubleHashMap</code></td>
<td><code>map&lt;long, double&gt;</code></td>
</tr>
<tr>
<td><code>longFloatHashMap</code></td>
<td><code>map&lt;long, float&gt;</code></td>
</tr>
<tr>
<td><code>longIntHashMap</code></td>
<td><code>map&lt;long, int&gt;</code></td>
</tr>
<tr>
<td><code>longLongHashMap</code></td>
<td><code>map&lt;long, long&gt;</code></td>
</tr>
<tr>
<td><code>longShortHashMap</code></td>
<td><code>map&lt;long, short&gt;</code></td>
</tr>
</tbody>
</table>

### `map<short,?>`

<table>
<thead>
<tr>
<th>Key Type</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>shortByteHashMap</code></td>
<td><code>map&lt;short, byte&gt;</code></td>
</tr>
<tr>
<td><code>shortDoubleHashMap</code></td>
<td><code>map&lt;short, double&gt;</code></td>
</tr>
<tr>
<td><code>shortFloatHashMap</code></td>
<td><code>map&lt;short, float&gt;</code></td>
</tr>
<tr>
<td><code>shortIntHashMap</code></td>
<td><code>map&lt;short, int&gt;</code></td>
</tr>
<tr>
<td><code>shortLongHashMap</code></td>
<td><code>map&lt;short, long&gt;</code></td>
</tr>
<tr>
<td><code>shortShortHashMap</code></td>
<td><code>map&lt;short, short&gt;</code></td>
</tr>
</tbody>
</table>

### `<K> map<K,?>`

<table>
<thead>
<tr>
<th>Key Type</th>
<th>Value Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ObjectByteHashMap&lt;K&gt;</code></td>
<td><code>map&lt;K, byte&gt;</code></td>
</tr>
<tr>
<td><code>ObjectDoubleHashMap&lt;K&gt;</code></td>
<td><code>map&lt;K, double&gt;</code></td>
</tr>
<tr>
<td><code>ObjectFloatHashMap&lt;K&gt;</code></td>
<td><code>map&lt;K, float&gt;</code></td>
</tr>
<tr>
<td><code>ObjectIntHashMap&lt;K&gt;</code></td>
<td><code>map&lt;K, int&gt;</code></td>
</tr>
<tr>
<td>Class</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td><code>ObjectLongHashMap&lt;K&gt;</code></td>
<td><code>map&lt;K, long&gt;</code></td>
</tr>
<tr>
<td><code>ObjectShortHashMap&lt;K&gt;</code></td>
<td><code>map&lt;K, short&gt;</code></td>
</tr>
</tbody>
</table>
Chapter 67

Tuples

Tuples give you a way to group related data of different types into small collection-like data structures. In MPS, tuples are available within the `jetbrains.mps.baseLanguage.tuples` language.

67.1 Indexed tuples

Indexed tuple is a structure, which can contain several elements of arbitrary types and elements of which can be accessed by an index. The MPS implementation represents a tuple instance by a Java object. The usual meaning of '==' and '!=' operations on Java objects within MPS remains unchanged.

```java
[int, string] pair = [1, "a"];
int first = pair[0];
string second = pair[1];
```

67.2 Named tuples

Named tuples are similar to indexed tuples, with the difference that elements are accessed by name instead of by index. To use named tuples in the model you first need to explicitly define them in your model (new->jetbrains.mps.baseLanguage.tuples/tuple).

```java
(first, second)<string, int> pair = (first = "abcd", second = 1);
```

Declaration of `Pair`:
```
public tuple Pair<S, T> {
    S first;
    T second;
}
```

67.2.1 Named tuple declaration

A root node of concept `NamedTupleDeclaration` contains a single declaration.

```
[public|protected|private] tuple tupleName<TypeVar, ...> {
    elementType elementName;
}
```

<table>
<thead>
<tr>
<th>declaration part</th>
<th>allowed contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>tupleName</td>
<td>any valid identifier</td>
</tr>
<tr>
<td>elementType</td>
<td>either a primitive type or any type that reduces to Java classifier</td>
</tr>
<tr>
<td>elementName</td>
<td>any valid identifier</td>
</tr>
</tbody>
</table>
Chapter 68

Lightweight DSL

Years of evolving the MPS core languages led us to recognizing recurring patterns in our language-descriptive DSLs for IDE integration. Conceptually they looked and behaved like higher-level constructions expressed by plain classes that "implement" some higher-level interfaces. In the good tradition of language-oriented programming we decided to reflect these patterns in the languages and re-implement them as a thin abstraction on top of plain $BaseLanguage$ classes.

The new Lightweight DSL language has been created to enable such abstractions. The $jetbrains.mps.baselanguage.lightweightdsl$ language enables internal DSLs to be embedded inside $BaseLanguage$ classes. Internal DSLs in general are easier and faster to develop than full-blown external DSLs, they typically reuse the syntax of the host language and tightly integrate with the surrounding non-DSL code. Similarly, lightweight DSLs in MPS can be created by defining a single node and then weaving the node into a $BaseLanguage$ $ClassConcept$ or its subconcepts.

MPS itself leverages this mechanism in several places:

- $MigrationScript$ concept, which is a mere $BaseLanguage$ class, is enhanced by the $Migration$ DSLDescriptor that adds a few extra properties, members and custom members
- Find Usages
- Intentions
- Custom language aspects

You can have your lightweight DSL weaved into a plain $BaseLanguage$ class or into your own concept extending $ClassConcept$.

68.1 Enhancing plain classes

The core idea behind the Lightweight DSL language is to allow the DSL designer to define interfaces with constraints, such as optional methods and properties, types depending on constraints, optional method parameters, custom class members and others. The $DSLDescriptor$ concept represents nodes that codify such constrained interfaces that should be enforced on implementing $Class$. For example, the following $DSLDescriptor$ instance will weave a $numberOfFrames$ property into classes as well as a $calculateFoo()$ method and a custom BuilderMember node:

dslclass SwingBuilder for concept $ClassConcept$ {
  property $numberOfFrames$ : integer ; placeholder<one frame>
  method $calculateFoo(int value): int$ ; required
  custom member BuilderMember ; <modifiers>

  initializer:
  {node<$ClassConcept$> node, model model => }
}

In order to have a particular $DSLDescriptor$ take effect on a class, it needs to be annotated with a $DSLAnnotation$, which is done through intentions:
As soon as the annotation is added, the required elements defined in the DSLDescriptor or the ones with placeholders specified are added to the class. Deleting required elements will result in an error indicated on the class:

An intention can be used to re-add the required elements as well as the placeholders in one step:
The standard **Implement method** action \((\text{Control/Cmd} + I)\) will also work for required methods.

### 68.2 Defining a DSLDescriptor

There are four types of elements a **DSLDescriptor** can add to classes:

- properties
- methods
- custom members
- an initializer

Each of these can be **required**, can have a **placeholder** defined to represent a missing member and **custom members** can be marked as **multiple** indicating that more than one node of this kind can be a member of the class.

#### 68.2.1 Properties

Properties have a **name** and can be of type **string**, **int** or **boolean**. Properties become full blown members of the weaved class and can be accessed from other members, including the ones weaved in through this or other DSLDescriptor.

#### 68.2.2 Methods

Methods provide a very convenient way to let users of your lightweight DSL inject code - the **DSLDescriptor** specifies method signatures of methods that the enhanced classes can or must implement. Unlike for normal methods, the return type as well as the types of any parameters of these weaved methods can be decided based on the actual usage in the host (enhanced) class.

**Dependent types**

The **dependent type**, when specified for a method’s return type or a parameter type, specifies way to calculate the actual type of the method for a given class.
Conditional method parameters

Methods can have some parameters marked as conditional, so that they only become visible in user code when the provided condition is satisfied. To mark a parameter as conditional you can use the corresponding intention:

```
DataSource for concept Pipeline {
  method getData(string p): list<int> ; required
  custom member Transformers ; required
  initializer:

  [condition]
```

The actual condition is specified in the Inspector:

```
DataSource for concept Pipeline {
  method getData(cond string p): list<int> ; required
  custom member Transformers ; required
  initializer:

  [node<MethodInstance> method => method.ancestor<concept = ClassConcept>.isFinal; ]
```

68.2.3 Custom members

If neither properties nor methods provide the required level of abstraction, custom members can be used to weave in any arbitrary concept with just a little bit of extra work.

```
SwingBuilder for concept ClassConcept {
  property numberOfFrames : integer ; placeholder<one frame>

  method calculateFoo(int value): int ; required

  custom member BuilderMember ; multiple
```

The CustomMemberDescriptor points to a concept (BuilderMember in our case) that will be weaved into the class as a new member. For this to be possible the concept has to implement the MemberInstance and ClassifierMember concept interfaces and point back to the CustomMemberDescriptor from the overriding getDeclaration() method.
A node (or multiple nodes, if the CustomMemberDescriptor is defined as multiple) of the specified concept can now be instantiated as a member of the host class and edited right inside it:
68.2.4 Initializer

The initializer function gets a chance to enhance the host class programmatically as soon as it is being created or annotated with the `DSLAnnotation`. The node (the host class) itself may not have been added to the model when the initializer is called, thus the model parameter is provided as well as the node. If you import the `smodel` language you can manipulate the class and the model. Typically the initializer would set the imports and used languages or provide default values and implementations for the weaved in members, if needed.
68.3 Enhancing custom ClassConcept sub-concepts

There are some specifics if you’re enhancing not BaseLanguage classes directly, but specific sub-concepts of the ClassConcept class. This allows you to combine the benefits of external and internal DSLs. You typically create ClassConcept sub-concepts instead of using plain BaseLanguage classes once your intended DSL requires customized editing experience, dedicated generator or specific type-system rules. So you create a sub-concept of ClassConcept and make it implement the AutoInitDSLClass.

Implementing AutoInitDSLClass ensures that whenever a node of your concept gets created, the weaved in DSLs get properly initialized by calling the initializer defined in their DSLDescriptors.
concept behavior MigrationScript {

    constructor {
        <no statements>
    }

    public node<DSLDescriptor> getDescriptor() {
        overrides AutoInitDSLClass.getDescriptor {
            return node/Migration/;
        }
    }

    The AutoInitDSLClass interface also mandates the implementing concepts to override the getDescriptor() method that should return the the particular DSLDescriptor instance to weave in.

68.4 Samples

Samples illustrating basic usages of the Lightweight DSL can be found in the lightweightDSL sample project bundled with MPS distributions.
Chapter 69

Dates language

An extension to the Base Language that adds support for dates.

• Dates language
• Dates language
  – Dates language
  – Dates language
• Dates language
• Dates language
  – Dates language
  – Dates language
  – Dates language
  – Dates language

69.1 Introduction

The Dates Language provides dedicated facilities to operate with date and time. Dates creation, comparison or adding and subtracting periods are done in a natural way, using common conventions and standard operators. As the backend implementation vehicle the Joda Time library is used.

69.2 Types

The following types are defined:

• **instant** represents the number of milliseconds passed since the epoch, 1970-01-01T00:00Z. It’s represented with a long. Thus an instant can be exchanged freely for a long value and vice versa.

• **datetime** represents a date and time with time zone.

• **duration** represents an interval of time measured in milliseconds.

• **period** is a representation of a time interval defined in terms of fields: seconds, minutes, etc. A period is constructed using + (plus) operator.

• **timezone** represents a time zone.

⚠ Old datetime type is renamed to instant. New datetime type contains timezone information. Use in expression to convert instant to datetime.
Predefined values

A special reserved keyword `now` represents the current time.

```java
instant times = now;
```

A reserved keyword `never` represents an instant before the beginning of times. All instants are placed after `never` on the time axis. Its actual representation is `null`.

```java
instant happened = never;
```

**Period constant** contains of a number and a property. They can be summed up together to form more complicated periods.

```java
period workdays = 5 days;
period weekend = 2 days + 1 hours;
```

**Time zone** can be created in different ways. All predefined values are available in completion menu. `default timezone` is a shortcut to the computer local time zone.

```java
timezone msk = Europe/Moscow;
timezone coordinated = UTC timezone;
timezone tz05 = Etc/GMT+5;
timezone localtz = default timezone;
```

Converting between types

Datetime can be obtained from an `instant` by providing a `time zone`. For reverse conversion simply get the datetime's `instant` property.

```java
datetime dt = now in (default timezone);
instant inst = dt.instant;
```

`Period` can be converted to `duration` using `toDuration` operation. It converts the `period` to a `duration` assuming a 7 day week, 24 hour day, 60 minute hour and 60 second minute.

```java
period p = 5 days;
duration workdays = p.toDuration;
```

For compatibility there are ways to convert `datetime` or `instant` types to `java.util.Date` and `java.util.Calendar`.

```java
datetime dt = ....;
Calendar calender = dt.javaCalendar;
Date date = dt.javaDate;
```

Reverse conversion:

```java
Calendar calendar = Calendar.getInstance();
instant dt = calendar.instant;
// -and-
Date date = new Date();
instant dt = date.instant;
```

69.3 Properties

Each individual `datetime` field can be queried using `dot` expression.

```java
datetime local = now in (default timezone);
int month = local.month;
```

To replace a field, use `with` expression.

```java
datetime dt = now in (default timezone) with (hours = 0) with (minutes = 0) with (seconds = 0);
```

Each period can be re-evaluated in terms of any field (with rounding if needed) leveraging the `in` operator.

```java
period p = ....;
int seconds = 20 days in seconds;
boolean foo = p in days < 7;
```


### 69.4 Operations

**Arithmetic**

A period can be added to or subtracted from a datetime. The result is a datetime.

```plaintext
datetime now1 = now in (CET);
datetime now2 = now1 + 5 hours;
datetime now3 = now1 - 1 year;
```

Two datetimes can be subtracted, the result is a period.

```plaintext
datetime start = now in (Europe/Berlin);
period executionTime = now in (Europe/Berlin) - start;
```

A duration can be added to or subtracted from an instant. The result is an instant.

```plaintext
instant firstdate = now;
instant nextdate = firstdate + 3 hours.toDuration;
```

Two instants can be subtracted, the result is a duration.

```plaintext
instant launch = now;
duration lifetime = now - launch;
```

**Comparison**

Two values of the same type (instant, datetime, period or duration) can be compared using standard operators: <, >, ==, etc.

```plaintext
instant endtime = now + 5 minutes.toDuration;
while (now < endtime) {
    // ...
}
```

Another form of comparison can be used for datetime by adding a keyword by following the field specification. In this case values are compared rounded to the field (see Dates language).

```plaintext
datetime justasec = now in (CET) + 1 second;
boolean mostlyTrue = now in (CET) == justasec by days;

datetime fromdb = ....;
if(fromdb < now in (default timezone) by month) {
    // ... previous month or earlier
}
```

Minimum and maximum operations are defined for instant and datetime types.

```plaintext
datetime fromdb = ....;
datetime result = max fromdb, now in (CET) - 2 years;
// -or-
instant lastaccess = max fromdb.instant, now - 1 month.toDuration;
```

**Rounding**

Datetime values can be rounded to the nearest whole unit of the specified field (second, minute, hour, day, month, etc). There are several ways of rounding:

- **round** returns the datetime that is closest to the original
- **round down to** returns the largest datetime that does not exceed the original
- **round up to** returns the smallest datetime that is not less than the original

```plaintext
datetime tomorrowStart = (now in (default timezone)) round up to day;
datetime currentHour = (now in (default timezone)) round down to hour;
```
Printing and Parsing

To print or parse datetime value we use date format describing its textual representation. The following formats are available by default:

- defaultFormat, rssDate
- shortDate, shortDateTime, shortTime, fullDate, longDate, etc (defined in Joda library)

Date format consists of one or more format tokens. The following kinds of tokens are supported:

- literal (referenced with a single quote) - any text, commonly used to insert dash or space
- datetime property (referenced with the name of a property in curly braces) - is replaced with the value of the property when printed
- switch - composite token, which may vary the format depending on the date
- offset (referenced as days ago, months ago, etc.) - calculates the difference between the provided datetime and the current moment
- reference (the name in angle brackets) - a way to include existing format

Additional date formats can be introduced using the `j.m.baseLanguage.dates.DateFormatsTable` root concept. Each format has a name and visibility. Formats with private visibility are local for the table.

```java
formats CustomFormats {
  public time : {hour} : {minute} : {second}
  private daysAgo : {days} ago
  public offset : switch {
    > [now == datetimeToFormat by hours;] -> time
    > [true;] -> daysAgo
  }
}
```

Datetime instance can be printed in the form of existing format by `#` operation.

```java
string text = now in (default timezone) # fullDate;
```

Another possibility is to use `{{#}}` operation, which allows to define format in-place.

```java
string text = now in (default timezone) {{<fullDate> - {dayOfYear}}};
```

Both printing operations accept optional locale argument in parentheses.

```java
string text = now in (default timezone) {{dayOfWeek} } (GERMAN);
```

Parse operation accepts string, date format, timezone and two optional parameters: default value and locale.

```java
datetime t1 = "1/11/70" parse as shortDate in default timezone;
datetime t2 = "1/11/70" parse as shortDate in UCT (US) into (now in (UCT));
```

New print/parse expressions operate on `datetime` instead of `instant`. Use intention to convert deprecated expressions to a new ones.
Chapter 70

Regular expressions

70.1 Regular expressions language overview

70.1.1 Introduction

Regular expressions language – the language that greatly simplifies text manipulation – is one of the most used domain specific languages today. Almost every developer has used it at least once. Some languages, like Perl and Python, have built in support for it; some, like Java, use it through libraries. Java, the language which we use to implement MPS, does not have language level support for regular expressions, so it was natural for us to implement DSL for them, so we would be able to use a DSL instead of a regular expression library. This language is a good example of an MPS language. Having read this introduction, you will be able to understand how to create and use languages in MPS.

Examples

We assume you have MPS already installed.
This document uses many examples. You can find them in a regular expression language project (%MPS_HOME%/platform/regexp) under the jetbrains.mps.regexp.examples solution:

70.1.2 Language overview

Let’s take a look at a simple regular expression application. Suppose we want to get a user name and domain name from an email address. Here is code that prints out a user name and a domain name by analyzing an email address with a regular expression (you can find this example in the EmailExample class):

```java
public static void main ( String emailToAnalyze ) {
    if ( emailToAnalyze matches [ ( user \w+ ) @ ( domain \w+ ( . \w+ ) ) ] ) {
        System . out . println ( "User name = " + user ) ;
        System . out . println ( "Domain = " + domain ) ;
    }
}
```

The regular expression that is used in this match regexp statement does the following. First, it reads one or more word characters (\w+) and saves them in a "user" variable. After that, it reads the "@" character. Then we read a list of words which are separated by a period (\.) character and save it in a domain variable (\w+(\w+)). If a match is found, the program prints out user and domain to System.out.
Here is a syntax tree for this example's regular expression:
70.1.3 Language structure

When we create a language in MPS, we usually start by defining its abstract syntax. Abstract syntax in MPS is called language structure. To do this, we use a structure language. Structure language is an XML Schema counterpart from XML language, or a DDL counterpart from SQL. Let’s take a look at the regular expressions language structure.

Overview

The MPS regular expressions language contains several parts:

- Regular expressions: concepts used to specify regular expressions. They include concepts for string literals, symbol classes, and "or" and "sequence" regular expressions.

- BaseLanguage (BaseLanguage is a Java-like language, used internally by MPS as a target language for generators) integration: this part includes concepts used to embed regular-expressions-related code into BaseLanguage. For example, it includes `MatchStatement`, `ReplaceStatement`, and `SplitStatement`.

- Regular expressions library support. When we work with regular expressions, we want to reuse them, and so we created special concepts for this task.
Regular expressions

All regular expressions concepts in our language are placed into "Regexp" folder in its structure model:

Let’s consider them in detail. We have a single base concept for all of them: Regexp:

```plaintext
concept declaration Regexp

extends: BaseConcept

implements
<< ... >>

is root: false

properties:
<< ... >>

links:
<< ... >>

concept properties:
abstract

concept links:
<< ... >>
```

It is derived from BaseConcept concept. All MPS concepts are derived from it. This concept also has the abstract concept property, which means that it is created to form a concepts hierarchy, not to be used in language to define regular expressions. It is similar to the ‘abstract’ modifier in Java classes.

Let’s consider the concepts that are derived from it. You can see them in a hierarchy view. You can see this view by pressing Ctrl + H on the concept declaration. For the Regexp concept, we will see the following:
StringRegexp represents an arbitrary string which can be matched against text (you can find all examples of regular expression that we consider in this section in the `Regexp` root node):

```java
regexp StringRegexp {
    // string regexp
    some_text
}
```

Let’s take a look at its concept declaration (you can quickly navigate to a concept declaration by pressing Ctrl + Shift + S when an instance of a concept is selected in an editor):
In its declaration, we see a property text with a string type, which is used to store text that will be shown in the editor. Also, this concept declares a concept property "alias." Concept properties differ from simple properties. Simple properties correspond to Java instance fields, and concept properties correspond to Java static fields. The value of a concept property alias will be shown in completion menu, when we press Ctrl + Space:

Binary regular expressions are created to represent regular expressions that combine two different regular expressions into one. BinaryRegexp concept is declared as abstract and has two concrete sub concepts: OrRegexp and SeqRegexp. Here are examples of their instances:

Here is its concept declaration:
It defines two links: one to store the left part and another to store the right part. The word ‘aggregation’ means that the regular expression under this link will be a part of a declared concept instance. i.e. if we look at the syntax tree, we will see a child regular expression under the parent BinaryRegexp:

Dot regexp represents a regexp which matches any character. LineEndRegexp matches only at the end of a line. LineStartRegexp matches only at the start of a line. ParensRegexp are used to group other regular expressions in order to make an enclosing regular expression more readable.

There are a lot of sets of symbols which are often used, but they are quite verbose to enter. So we have character classes that make it possible to enter [A-Z] instead of (A|B|CâĂęZ). We have two kinds of them: negative and positive. Both of them extend abstract SymbolClassRegexp:

Many of these character classes are used in several places, so they can be referenced in a simpler way with PredefinedSymbolClassRegexp. Instead of [A-Z] we can write "\w".
This concept is declared in the following way:

```
concept declaration PredefinedSymbolClassRegexp

    extends: Regexp
    implements
    << ... >>
    is root: false
    properties:
    << ... >>
    links:
    | symbolClass reference | target: PredefinedSymbolClassDeclaration | special
```

Here we have `symbolClass` link declaration, which has a reference stereotype (aggregation, which we mentioned above, is also a link stereotype). Reference stereotype means, that an instance of this concept won’t contain the referenced node as a child. Instead the referenced node can be stored in any place in the model.

Also we have a lot of different `UnaryRegexps` which are derived from an abstract concept `UnaryRegexp`. They include +, * and other regexp operations:

```
regex PredefinedSymbolClass {  // predefined symbol class
    \w
}
```

When we work with a text it is often useful to remember some match, and reference it later. To facilitate this task we have `MatchParensRegexp` that remembers a string which it matches, and `MatchVariableReferenceRegexp` that references a string matched before. The following code matches a pair of the same xml tags with a text inside it:

```
regex MatchParensRegexp {  // plus regexp
    a +
}
regex StarRegexp {  // star regexp
    h *
}
regex Matchreferences {  // match references
    < ( tagName . +? ) > . +? <\ ( ref tagName ) >
}
```

**BaseLanguage integration**

Regular expressions have a little use if they can’t be integrated in the `BaseLanguage` code. So in regular expressions language we have special concepts which make it possible to write regular-expression-related constructs in a program which is written in `BaseLanguage`.

If you want to add new constructs to `BaseLanguage` you usually extend either `Expression` or `Statement` concept from `BaseLanguage`. `Expression` concept represents expressions like "1+2", "a == b". `Statement` concept represents control structures like "if(a{A} { }", "while(a{A} { }"). In the regular expressions language we create both new expressions and statements.

Let’s first take a look at the statements and then at the expressions:

`MatchRegexpStatement` is used when you want to check whether a specified string matches a regular expression (you can find the examples for this section in `BaseLanguageIntegration` class in `jetbrains.mps.regexp.examples` model):
We have an interesting feature here: you can reference named matches in the `MatchRegexpStatement` block. These match variables work in other statements which are defined in the regular expressions language.

`FindMatchStatement` checks whether a specified string contains a match for a specified regular expression. It is similar to `MatchRegexpStatement`.

For `ForEachMatchStatement` allows you to iterate over all matches of a specified regular expression in a specified string:

When we work with a string, we often want to replace all matches of a regular expression with a specified text. In regular expressions language you can do this with the help of `ReplaceWithRegexpExpression`:

It is also often practical to split a string with some regular expression. For example, to extract parts of a string which are separated by one or more whitespace symbols we can write this `SplitExpression`:

When we reference a match in a block, the `MatchVariableReference` concept is used. It is also derived from the `Expression` concept.

**Library support**

When we work with regular expressions, we want to use some of them in many places. To define these reusable regular expressions, we have a special concept – `Regexp`. It contains zero or more named regular expressions:
Accessory models

In many languages we have the following problem: we have a lot of very similar entities, which can be used in any model that is written with this language (like predefined symbol class regular expression). We could create a concept for every such entity. But MPS has a better solution: you can create a special model, called an accessory model, and declare all these entities in it with your language.

We have the `PredefinedSymbolClass` concept which is used to declare a symbol class. Also, we have the `PredefinedSymbolClasses` container concept, which contains these symbol classes. If you look into the accessory model of the regular expressions language, you will see this:

```plaintext
predefined symbol classes {
  symbol class \d {  
    description: a digit [0-9]
  }
  symbol class \D {  
    description: non digit [^0-9]
  }
  symbol class \s {  
    description: a whitespace character [ \t\n\r\x0B\f\v]\n  }
  symbol class \S {  
    description: a non whitespace character [^\s]
  }
  symbol class \w {  
    description: a word character [a-zA-Z_0-9]
  }
  symbol class \W {  
    description: a non word character [^\w]
  }
}
```

70.1.4 Editor

After defining the concept structure, we usually create an editor for it. To accomplish this task, we use the editor language. It is quite straightforward to use, so let’s consider its most common constructs.

All editor-related code is placed in an editor model. You can find it under a language node in a project tree:
Here is an editor of `StringLiteralRegexp`:

```plaintext
editor for concept StringLiteralRegexp

node cell layout:

```
> { text } <
```

inspected cell layout:
```
<choose cell model>
```
```
It contains a horizontal collection, the container which you might use to group other constructs inside it, and \{text\}, which is used to include an editor for an instance property.
```

Here is an editor for `MatchVariableReferenceRegexp`:

```plaintext
editor for concept MatchVariableReferenceRegexp

node cell layout:

```
> {ref ( %match %-> { name } )} <
```

inspected cell layout:
```
<choose cell model>
```
```
It also consists of a horizontal collection, but this time we have a richer set of constructs inside it. \((\text{ref} \text{ and } \text{)}\) are constants, which always contain the same text. \(\%\text{match}\%\text{->}{\text{name}}\) is used to reference the property "name" of match link's target.
```

Here is an editor for `Regexp`:
It contains a vertical collection with nested horizontal collections. Also, it contains a "(> %regexp% <)" construct. It is used to include editors for all the nodes in the role "regexp".

70.1.5 Scopes

After declaring references in structure, we have default substitute menus for them. These default menus include all the nodes of a reference type in the current model and all of its imported models. Sometimes it works, but sometimes we have to narrow down the scope of these menus (For example, if you have a lot of match variables named "name" in different parts of a model, it’s a good idea to follow the Java scoping rules for these variables.) To handle this task, we have constraints language’s scopes.

Scopes are placed in a constraints model under a language node:

Let’s consider a scope for MatchVariableReference:

Scope consists of a referent set handler, a scope condition (labeled "can create"), and a scope constructor. Usually, only a scope constructor is specified. Scope constructor has to return an object that implements the ISearchScope interface. Usually, an instance of the class SimpleSearchScope is returned; it has a constructor which takes a list of nodes, i.e. we return a list of nodes which are visible in a specified place.
70.1.6 Actions

Default editors in MPS aren’t very easy to use. To improve this default behavior, different constructs from the actions language and the editor language can be used.

When we enter code in a text-based language, we usually do it from left to right. We might start from "2", then enter "2+", and finally we might have "2+2". It is also possible to enter code in MPS in this way with the aid of a mechanism called 'right transform'.

To define a right transform action, you have to create a right transform actions root in the actions model and add some right transform actions to it. Let’s consider a right transform action from the regular expressions language which transforms one regular expression to the unary regular expression, i.e. it transforms "a" into "a+", "a*", etc (like constraints, editor and structure, you can find the actions model under the language node in your project tree):

| concept: Regexp tag: DEFAULT |
| description : transform to unary regexp |
| condition : <none> |
| new actions : |
| add ![no concept> |
| concept |
| descendants of UnaryRegexp |
| exclude << ... >> |
| handler |
| (operationContext, sourceNode, pattern, model, result)->snod< > |
| sourceNode . replace with ( result ); |
| result . regexp . set ( sourceNode ); |
| return result ; |

Each right transform has an applicable concept – the type of concept this action can be applied to. Also, it has a condition and the most important part: a right transform menu. There are different types of right transform menus. The menu on the picture above adds one menu item for each non-abstract UnaryRegexp sub concept. The handler of this menu part transforms an expression into a unary expression.

70.1.7 Type System

Many languages have a type system. It allows you to check a model against it, and can be used to improve editing experience and simplify the generator. For example, if we know the type of a particular expression, we are able to calculate which methods can be applied to it. MPS has a special language for type systems, called HELGINS. In languages with a very simple structure, it’s possible to live without it, but when we have a complex language or want to integrate with BaseLanguage, we have to create a type system, at least for BaseLanguage integration concepts.

In HELGINS, types are represented as MPS nodes. So, if you have a sublanguage for types, like BaseLanguage does, you can use it for type checking.

Let’s consider a couple of rules from the regular expressions language.

```
RULE MatchVariableReference;

APPLICABLE FOR concept = MatchVariableReference as mvr
overrides: false

DO {
    GIVETYPE ( < String > ) TO mvr ;
}
```

In this code we define a type called String (String here is an instance of ClassifierType from BaseLanguage, which is used in method parameter types, local variables and other places). To do so, we use the GIVETYPE statement.

Let’s take a look at a more complex rule:
In this rule, we require that an expression that we match against a regular expression in `FindMatchStatement` be a subtype of `String` type. We do this by specifying a type equation. The sign `":="` denotes a subtype; expression `TYPEOF` denotes a type of expression in parenthesis.

To calculate types, HELGINS uses a sophisticated algorithm which saves you a lot of time. You don’t have to worry about the order in which types are calculated; all you have to do is to specify type equations in typing rules, and HELGINS will solve them for you.

Of course, the rules in our language are very simple, and if you want to know more about HELGINS, you have to take a look at rules in languages like BaseLanguage or the model language.

### 70.1.8 Generator

Almost any language created with MPS has a generator. Generators in MPS convert the high-level language code into code in a lower level language. The key component of a generator is its mapping configuration. It tells us what to do with a language.

Let’s consider a mapping configuration of the regular expressions language:

It contains one mapping rule and several reduction rules. Each rule has an applicable concept; for each instance of this concept, the rule will be applied. Mapping rules create a new root node on each application. A reduction rule replaces a node to which it is applied with a new node. Each rule has an associated template used to create an output node.

Let’s take a look at an instance of such a template:
Templates contain MPS code with macros and template fragments. The code outside of a template fragment is not used during generation, and is used only to create a context for code inside a template fragment. For example, if we know that our code will be placed inside a method with a parameter named node, we might create a method with such a parameter around the template fragment. During generation, MPS will recognize your intention, and this variable will be automatically resolved.

Macros are used to specify variable parts of code. For example, variable matcher on a picture above has a property macro on it. This property macro generates a unique name for this variable, so we will be able to use nested match blocks. MPS has different kinds of macros: different kinds of node macros, property macros, and reference macros. All of these concepts are declared in the `jetbrains.mps.TLBase` language.

### 70.1.9 Further reading

We have taken a look at the regular expressions language. It uses many MPS language development features, but of course not all of them. The best way to learn how to use MPS is to look at another language, like the base language and bootstrap languages. There are several tools in MPS which can be used to understand how MPS works. One of them is find usages. You can invoke it by choosing **Find Usages** from the editor popup menu, or by pressing Alt + F7 on a node in an editor.

The second one is find concept instances. When you come across a concept, and you don’t know how to use it, the best way to learn it is to find its instances and try to understand what those instances do.
MPS distribution also contains a documentation system in the `%MPS_HOME%/help` folder. Some of it is out of date, some quite incomplete, but it can be used to learn MPS.
Chapter 71

Regexp language

Regular expressions is one of the earliest DSLs in wide use. Nearly all modern programming languages these days support
regular expressions in one way or another. MPS is no exception. Regular expression support in MPS is implemented through
a base language extension.
We also recommend checking out the Regular expressions, to get a more thorough introduction into the language.

71.1 Defining regular expression.

Regexp language allows you to create an instance of java.util.regex.Pattern class using a special pattern expression:
/regexp/. In the generated code, MPS creates for each defined pattern expression a final static field in the outermost
class, so the pattern is compiled only once during application runtime.
Pattern pattern = /\[a-z\]+/

There are three options, you can add after the ending slash of the regexp.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/i</td>
<td>Case-insensitive matching</td>
</tr>
<tr>
<td>/s</td>
<td>Treat string as single line, dot character class will include newline delimiters</td>
</tr>
<tr>
<td>/m</td>
<td>Multiline mode: Â¥nd $ characters matches any line within the string (instead of start or end of the string)</td>
</tr>
</tbody>
</table>

The options can be turned on/off by typing or deleting the character in the editor, or through the Inspector. Generated
regular expression preview is available in the Inspector.

71.1.1 Re-using Definitions

To reuse a regular expression for a frequently used pattern across your project, create a separate root:
model -> New -> jetbrains.mp.baseLanguage.regexp -> Regexps
Each reusable regular expression should have a name and optionally a description.

regexp Identifier {
  // no description
  (identifier: [a-z A-Z _ ] [a-z A-Z _ 0-9]+)
}

71.1.2 Pattern Match Operator

The =~ operator returns true if the string matches against the specified pattern.
"string or variable" =~ /regexp/

71.1.3 Capturing Text

Optional use of parentheses in the expression creates a capture group. To be able to refer to the group later, the best
practice is to give it a name.
/\^ (name: [a-z A-Z _ ] [a-z A-Z _ 0-9]+) /
71.4 Examples

```java
if ("any string" =~ /^ # define (identifier: [a-z A-Z _] [a-z A-Z _ 0-9]+) /) {
    process(identifier);
}
```

If the pattern matches against the string, the matched value is captured into `identifier` and can be accessed in the if-block. Don’t forget to check out the Regular expressions, to get a more thorough introduction into the language.
Chapter 72

Type Extension Methods

The language jetbrains.mps.extensionMethods provides a way to extend any valid MPS type with newly defined or overridden methods, akin to Java static methods.

"This is an ordinary string with a surprising behavior!".myNewMethod()
"Develop with pleasure!".isAPleasureMessage()
int num = ...
num.isPrime()

Whereas static methods never become an internal part of the extended class and one has to always specify the "extended" object to operate on as one of the parameters to the extended method, with an extension method the new method gets added directly to the list of operations available on the target type.

So, provided we wanted to add a reverse method to the string type, instead of the good old "static method" way:

```java
public static string reverse(string target) {
    //reverse the target
}
```

we would create new Extension Methods through New -> j.m.baseLanguage.extensionMethods/type extension, define the new method and tie it to the string class:

```java
public extension methods MySampleMethods for string {
    <<static fields>>
    public string reverse() {
        //reverse the string, refering to it through 'this'
    }
}
```

The very same mechanism can be used to override existing methods. And when in need to call the original method, just call it on this:

```java
public extension methods MySampleMethods for string {
    <<static fields>>
    public string trim() {
        this.trim() + " trimmed" //calls the original trim() implementation
    }
}
```

Since MPS does a good job to visually distinguish the original and overridden methods through the extension methods mechanism, you can’t make a mistake picking the right one from the drop-down list.
Obviously this mechanism can be used to implement orthogonal concepts on your own domain objects as well:

```csharp
public extension methods Sample for my_type {
    <<static fields>>
    public int foo() {
        return this.bar();
    }
}
```

With the declaration as above, one could write an operation on type `my_type`:

```csharp
my_type var = ...;
var.foo();
```

### 72.0.5 Root Nodes

There are two equally good ways to extend types with methods. *Type Extension* allows to you to add methods to a single type in one place, while *Simple Extension Method Container* comes in handy, when you need one place to implement an orthogonal concept for multiple different types.

**Type Extension**

This root contains declarations of extension methods for a single type.

```csharp
[public|protected|private] extension methods containerName for extendedType {
    <<static fields>>
    <<extension methods>>
}
```

Extension method declaration.

```
public ret_Type methodName() {
    <no statements>
}
```

**Simple Extension Method Container**

```csharp
[public|protected|private] extension methods containerName {
    <<static fields>>
    <<extension methods>>
}
```

Extension method declaration. The target type is specified per method.

```
public ret_type methodName() for extendedType {
    <no statements>
}
```

<table>
<thead>
<tr>
<th>declaration part</th>
<th>allowed contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>containerName</td>
<td>any valid identifier</td>
</tr>
<tr>
<td>extendedType</td>
<td>any valid MPS type</td>
</tr>
</tbody>
</table>
Both roots may contain one or more static fields. These are available to all methods in the container.
Chapter 73

Builders

Builders allow users to construct objects and object hierarchies in a more convenient way. Instead of a manual instantiation of each and every object in the hierarchy and setting its properties one-by-one, with a dedicated builder the same data structure can be created in a more concise and intuitive way.

As an example, let’s assume we’re building a house.

```java
House h = new House();
h.houseType = HouseType.detached;
Address address = new Address();
address.street = "Na Lysinach";
address.number = "10";
address.city = "Prague";
address.zipCode = "23459";
h.locate(address);
list<Room> rooms = new arraylist<Room>(0);
rooms.add(new Room("livingroom", 25));
rooms.add(new Room("kitchen", 18));
rooms.add(new Room("bedroom", 20));
rooms.add(new Room("hall", 12));
rooms.add(new Room("bedroom", 15));
h.rooms = rooms;
h.moveIn(6);
println(h);
```

A house needs an address, which itself consists of several items, a bunch of rooms in it, each of which needs a couple of properties, and so on.

Instead of the cumbersome way, builders give you a syntactic shortcut to take:
Looking at the code you can quickly grasp the structure of the created object graph, since the structure of the code itself mirrors the dependencies among created objects. Builders are nested into one another and they can hold properties. Both the property values and mutual nesting of builders is then transformed into the object hierarchy built behind the scenes.

MPS brings a few of handy builders directly to your door as part of some of the languages - JavaBeans, XML, XMLSchema or XMLQuery being the most prominent users.

### 73.1 Building Builders

To build your own builder, you first need to invoke `New -> j.m.baseLanguage.builders.SimpleBuilders`. Now you define builders for each object type that participates in the hierarchy. These builders hold their own properties and children, out of which they build the requested data structure. To stick to our earlier "House building" example, check out the sample below:
We defined a builder for the Room class as well as for the Address class and also a root builder for the House class. Root builders, unlike plain builders, can be used directly in user code after the new keyword. Notice also that we have two builders for the Room class. The first definition allows properties to be nested inside the room block, while the second allows the
two properties to come directly as parameters to the room method call. Both approaches can certainly be combined in a single builder.

The House, Room and Address classes in our case are ordinary classes with methods and properties. The methods as well as setters for the properties manipulated in builders must be visible to the builders. The "package" visibility will do in typical cases. To give you an example, see below the House class definition from our example.

```java
public class House extends <none> implements <none> {

private Address address;
private int numberOfPeople;
public HouseType houseType {get; <no visibility> set;}
public list<Room> rooms {get; <no visibility> set;}

public House() {
    <no statements>
}

public void moveIn(int people) {
    this.numberOfPeople = people;
}

public void locate(Address address) {
    this.address = address;
}

public int numberOfInhabitants() {
    return this.numberOfPeople;
}

public String toString() {
    return "House for " + numberOfPeople + " people at " + address.toString();
}

```

<<static methods>>
<<nested classifiers>>
Chapter 74

Logging

The jetbrains.mps.baselanguage.logging language contains statements for writing arbitrary information into the MPS log as well as to the Messages tool view panel. The LogLowLevelStatement concept (aka "log error...") is used for logging using the system logger, while MsgStatement (aka "message error...") is used for logging into the Messages tool window.

```plaintext
message info "So far so good", <no project>, <no throwble>;
message warn "Oops", <no project>, <no throwable>;
log error "Things happened", <no throwable>;
```

The language supports different severities of log messages:

- trace
- info
- debug
- warn
- error
- fatal

Whenever you want to insert a log statement into code, start by typing either "log" or "message" followed by the desired severity:

```plaintext
log debug prints to the system log
log error prints to the system log
log fatal prints to the system log
log info prints to the system log
log trace prints to the system log
log warn prints to the system log
```

Upon completion the log statement with an empty message will be inserted.

```plaintext
log warn ", <no throwable>;
```

The severity level can always be changed:
The log statement also supports exceptions to be specified:

```
log warn "", <no throwable>;
```

`MsgStatement` contains an additional parameter to specify the project:

```
message error "Commenting out a node", editorContext.getOperationContext().getProject(), <no throwable>;
```

When multiple projects are open in MPS at the same time, each project gets its own MPS windows and thus also its own `Messages` view. The `project` parameter is used to output the log message into the `Messages` window of the right project. If you leave the parameter empty, the message will be logged into `Messages` tool views of all open projects.
Chapter 75

XML language

The `jetbrains.mps.core.xml` language is designed to model closely XML documents in MPS. The language aims at being a 1:1 match to plain XML and is generated into textual XML files.

### 75.0.1 Structure

The `XmlFile` root element should be used to represent an XML file.

It contains a single `XmlDocument` node, which itself holds one or more `prolog` entries and a `root xml element`:

There are several types of prolog elements to choose from and customize:

Use the `Enter` key to separate entries in the prolog, either within the same line or across multiple lines.

### 75.0.2 Editing

The elements, their attributes and values can then be entered naturally. The XML-specific symbols, such as e.g. `'<', '>', '='`, `space`, `&`, are recognized as delimiters and the automatically invoked transformations will correctly insert proper
instances of the desired concepts - XmlElement, XmlAttribute, XmlText, XmlTextValue, XmlEntityRef, XmlEntityRefValue, XmlComment and other. Code-completion should assist you to complete unfinished elements with little effort.

```xml
xml Doc

<?xml version = "1.0" encoding = "default" standalone = "default" ?>
<race>
  <car name="Ford">John</car>
  <car name="BMW">Dave <![CDATA[the master]]></car>
</race>
```

### 75.0.3 Generation

The language is transformed into textual XML using the `TextGen` aspect.
Chapter 76

Other languages

Here we introduce some handy BaseLanguage extensions

76.1 Checked dots

Language: jetbrains.mps.baseLanguage.checkedDots

A Checked Dot Expression is an dot expression extended with null checks on the operand. If the operand is null, the value of the whole checked dot expression becomes null, otherwise it evaluates to the value of corresponding dot expression.

Ways to create a Checked Dot Expression

• The Make dot expression checked intention
• Enter "?" after dot, e.g. customer.?address.?street
• Left transform of operation with "?"

You can transform checked dot expressions to the usual dot expressions using the Make dot expression not checked intention

76.2 Overloaded operators

Language: jetbrains.mps.baseLanguage.overloadedOperators

This language provides a way to overload binary operators. Overloaded operator declarations are stored in an OverloadedOperatorContainer. If there are several overloaded versions for one operator the most relevant is chosen. Note that if an overloaded operators’ usage is in the other model than its declaration, overloadedOperators language should be added to "languages engaged on generation" of usage’s model.

Examples

Overloading plus operator for class Complex:

```java
operator + (Complex, Complex) -> Complex

(left, right)->Complex {
    Complex res = new Complex();
    res.set(left.getRe() + right.getRe(), left.getIm() + right.getIm());
    res;
}
```

Also, you can define your own custom operators. Assume we want to create a binary boolean operator for strings, which tells if one string contains another:
76.3 Custom constructors

Language: jetbrains.mps.baseLanguage.constructors

Custom constructors provide a simple way to create complex objects. They are stored in a special root node - CustomConstructorsContainer.

**Example**  Assume we need a faster way to create rectangle.

```java
class Rectangle
{
  // custom constructor Point
  short description: <no shortDescription>
  (double x, double y) -> Point2D
  {->Point2D {
    return new Point2D.Double(x, y);
  }
  separator: ,

  // custom constructor Rectangle
  short description: <no shortDescription>
  [Point2D topleft, Point2D bottomright] -> Rectangle2D
  {->Rectangle2D {
    double x = topleft.getX();
    double y = topleft.getY();
    return new Rectangle2D.Double(x, y, bottomright.getX() - x, bottomright.getY() - y);
  }
  separator: :
}

Now, let’s create a rectangle:

```java
Rectangle2D r = [(2, 3) > (4, 6)];
```
Part VII

Delivering languages to the users
Chapter 77

Build Language

77.1 What is MPS build language?

Build Language is an extensible build automation DSL for defining builds in a declarative way. Generated into Ant, it leverages Ant execution power while keeping your sources clean and free from clutter and irrelevant details. Organized as a stack of MPS languages with ANT at the bottom, it allows each part of your build procedure to be expressed at a different abstraction level. Building a complex artifact (like an MPS plug-in) could be specified in just one line of code, if you follow the language conventions, but, at the same time, nothing prevents you from diving deeper and customize the details like file management or manifest properties.

As with many build automation tools, project definition is the core of the script. Additionally, and unlike most of the other tools, Build Language gives you full control over the output directory layout. The expected build result is defined separately in the build script and not as a part of some (third-party) plugin.

Every build script is made up of three parts. The first is dependencies, something required that comes already built. Think of libraries or third-party languages, for example. Next is the project structure. It contains declarations of everything you have in your repository and what is going to be built, as well as the required build parameters. Note that declaring an item here does not trigger its build unless it is needed, i.e. referred to from the last part of the script - the output layout. The output could be as straightforward as a set of plain folders and copied files, or much more complex with zipped artifacts such as packaged plug-ins or MPS languages. For example, to build a jar file out of Java sources you need to declare a Java module in the project structure and the respective jar file with a reference to the module in the output layout.

Thanks to MPS, Build Language comes with concise textual notation and an excellent editing experience, including completion and on-the-fly validation. Extension languages (or plugins if we stick to the terminology of the other build tools) add additional abstractions on top of the language. In our experience, it is quite an easy process to create a new one compared to developing Maven or Gradle plugins.

77.2 Build script structure

See an example below of a build script, which builds a plugin for IntelliJ IDEA:
Let's look at it closely. The header of the script consists of general script information: the name of the script (Complex in the screenshot), the file it is generated into (build.xml) and the base directory of the script (in the screenshot one can see it is relative to the script location ../../ as well as full path).

The body of the script consists of the following sections:

- **use plugins** contains a list of plugins used in the script. Plugins in Build Language are similar to those in Gradle: they are extensions to the language that provide a number of tasks to do useful things, like compiling java code, running unit tests, packaging modules, etc. In the screenshot two plugins are used: java and mps, which means that the script can build java and mps code.

- **macros** section defines path macros and variables used in the project (idea_home and plugins_home) together with their default values, which could be overridden during execution of the script.

- **dependencies** defines script dependencies on other build scripts. If a script references to something defined in the other build script it must specify this script in the dependencies section. The example script on the screenshot depends on two other scripts IDEA and mpsPlugin. These are provided by MPS, so in order to use them one has to specify the artifacts location for them, i.e. place where ant can find the result of their work (in the example, idea_home should point to the location of IntelliJ IDEA jars and plugins_home should point to the location of MPS plugins for IDEA). One can as well depend on some build scripts in the same MPS project. In that case, artifacts location is not required and it is assumed that the required script would be built just prior to the current script (there is a target buildDependents to do so).

- **project structure** section contains the description of the project, i.e. which modules does it have, where the source code is located, what the modules classpath is etc. The example project in the screenshot consists of a single idea plugin named Complex and of a group of MPS modules.
• default layout defines how to package the project into the distribution. The example project on the screenshot is packaged into a zip file named Complex.zip.

• additional aspects defines some other things related to the project, for example, various settings, integration tests to run, and so on.

77.3 Macros

There are two types of macros:

- Folder - represents a physical folder in the file-system, if left empty (default), the value is attempted to retrieve from a path variable defined in MPS with the same name
- Var - custom variable representing a value

The Vars can be initialized in one of several ways:

- reference to an earlier macro
- date - a date value, a pattern parameter that follows the Java’s date format rules should be specified, e.g. yyyy-MM-dd, the date value of the time when the build script is run will be inserted into the macro
- load from file - reads a specified property value from a specified property file
- text - a plain text value

77.4 Built-in plugins

Build Language provides several built-in plugins.

77.4.1 Java plugin

The Java plugin adds capability to compile and package java code. Source code is represented as java modules and java libraries. Java module defines its content (source folders locations) and dependencies on other modules, libraries and jars. In content section java module can have:

- folder – a path to source folder on disk;
- resources – a fileset of resources. Consists of a path to resources folder and a list of selectors (include, exclude or includes).
- content root – a root with several content folders.

In dependencies section java module can have:

- classpath – an arbitrary xml with classpath;
- external jar – a jar file from other build script layout;
- external jar in folder – a jar file referenced by name in a folder from some other build script layout;
- jar – a path to local jar;
- library – a reference to a java library;
- module – a reference to a java module.

Each java module is generated into its own ant target that depends on other targets according to the source module dependencies. For compiling cyclic module dependencies, a two-step compilation is performed:

1. A "cycle" target compiles all modules in the cycle together.
2. Each module in the cycle is compiled with the result of compilation of "cycle" target in classpath.

Java library consists of jars (either specified by path or as references to the other project layout) and class folders. The available elements are:

- classes folder – a folder with classes;
• external jar – a jar file from other build script layout;
• external jars from – a collection of jars from a folder from some other build script layout;
• jar – a path to local jar;
• jars – a path to local folder with jars and a list of selectors (include, exclude or includes).

Compilation settings for java modules are specified in java options. There can be several java options in the build script, only one of them can be default. Each module can specify its own java options to be used for compilation.

Java Targets

Java plugin adds the following targets:

• compileJava compiles all java modules in the project.
• processResources extension point for additional resource processing.
• classes does all compilation and resource processing in the project. It depends on targets compileJava, processResources.
• test extension point target for running unit tests.
• check does all testing and checking of project correctness. It depends on target test.

77.4.2 MPS plugin

The MPS plugin enables the build language in scripts to build mps modules. In order to use the MPS plugin one must add jetbrains.mps.build.mps language into used languages.

MPS modules and groups

The MPS plugin enables adding modules into project structure. On the screenshot there is an example of a language, declared in a build script.

Note that there is a lot of information about the module specified in the build script, most of which is displayed in the Inspector tool window: uuid and fully qualified name, full path to descriptor file, dependencies, runtime (for a language) etc. This information is required for packaging the module. So, every time something changes for this module, for example a dependency is added, the build script has to be changed as well. There is of course a number of tools to do it easily. The typical process of writing and managing mps modules in the script looks as following:

1. Adding a module to the script. One specifies, which type of module to add (a solution, a language or a devkit) and the path to the module descriptor file. Then the intention "Load required information from file" can be used to read
that file and fill the rest of the module specification automatically.

2. Reflecting the changes made in the module. One can check a model with build scripts using the **Model Checker** to find whether it is consistent with the module files. Model checker will show all problems in the script and allow you to fix them using "Perform Quick Fixes" button. Instead of Model checker one can use the same "Load required information from file" intention to fix each module individually.

Another thing to remember about MPS module declarations in a build scripts is that they do not rely on modules being loaded in MPS. All the information is taken from a module descriptor file on disk, while module itself could be unavailable from the build script. MPS modules can be added into an **mps group** in order to structure the build script. An MPS Group is just a named set of modules, which can be referenced from the outside, for example one can add a module group into an IDEA plugin as one unit.

### 77.4.3 Module resources

Resources required by a module (images, icons, etc.) should be specified using the **resources** content root:

```
mps group notesOrganizer
language jetbrains.mps.samples.notesOrganizer (4b0f115a-8868-4d72-8d61-97071eaa5f1)
load from ./languages/jetbrains.mps.samples.notesOrganizer/jetbrains.mps.samples.notesOrganizer.mpl
content:
resources files from ./languages/jetbrains.mps.samples.notesOrganizer
includes icons/**, resources/**
dependencies:
(extracted) MPS.Platform
(extracted) jetbrains.mps.kernel
(extracted) MPS.Editor
(extracted) jetbrains.mps.samples.notesOrganizer
(extracted) MPS.OpenAPI
(extracted) JDK
(extracted) MPS.Core
(extracted) jetbrains.mps.baseLanguage
(extracted) jetbrains.mps.lang.editor.forms.runtime
runtime:
<no runtime>
```

**How generating and compiling MPS modules works internally**  
As it was written above, a lot of information about a module is extracted into the build script and stored there. This mandates the user to properly update the script whenever module dependencies change. For a **Solution**, it’s both the reexported and non-reexported dependencies that are extracted to the script. For a **Language**, apart from the dependencies, runtime solutions and extended languages are also extracted.  
"Building" a module with build script consists of two parts: generating this module and compiling module sources. Generating is an optional step for projects that have their source code stored in version control system and keep it in sync with their models. For generating, one target is created that generates all modules in the build script. Modules are separated into “chunks” – groups of modules that can be generated together – and the **generate** task generates the chunks one by one. For example, a language and a solution written with the language cannot be generated together, therefore they go into separate chunks. Apart from the list of chunks to generate, the **generate** task is provided with a list of idea plugins to load and a list of modules from other build scripts that are required for the generation. This lists of plugins and modules is calculated from
the dependencies and therefore their correctness is crucially important for successful generation. This is a major difference between generating a module from MPS and from a build script: while when generating a module from MPS, the generator has all modules in the project loaded and available; when generating a module from a build script, the generator only has whatever was explicitly specified in the module dependencies. So a build script can be used as some kind of a verifier of correctness of modules dependencies.

Compilation of a module is performed a bit differently: for every MPS module a java module is generated, so in the end each MPS module is compiled by an ordinary ant javac task (or similar other task, if it was selected in Java Options). So in order to generate and compile, dependencies of a module are to be collected and embedded into the generated build xml file. Used languages and devkits are collected during generation of a build script from the module descriptor files. The other information is stored inside the build script node. In the picture below a module structure is shown for a project called "myproject", which uses some third-party MPS library called "mylibrary".

The arrows illustrate the dependency system between modules. The purple arrows denote dependencies that are extracted to the build script, the blue arrows indicate, which dependencies are not extracted. It can be easily observed that in order to compile and generate the modules from my project a knowledge of the "blue arrows" inside of mylibrary is not required. Which means that the actual module files from my library may not even be present during myproject build script generation. Every information that the generator needs is contained in the build script. Which is really very convenient: there is no need to download the whole library and specify its full location during build generation and so the generation process saves time and memory by not loading all module descriptors from project dependencies.

Sources and tests When an MPS solution contains test models, i.e models with the stereotype @tests, they are generated into a folder "tests_gen" which is not compiled by default. To compile tests, one needs to specify in the build script that a solution has test models. This is done manually in the inspector. There are three options available for a solution: "with sources" (the default), "with tests" and "with sources and tests".

MPS Settings

mps settings allow to change the MPS-specific parameters for a build script. No more than one instance of mps settings can exist in the build script in the "additional aspects" section. Parameters that can be changed:
• *bootstrap* – setting this flag to "true" indicates that there are some bootstrapping dependencies between modules in the script. Normally the flag is set to false. See Removing bootstrapping dependency problems for details.

• *test generation* – if set to true, the build script tests modules generation and difference between generated files and files on disk. Files can be excluded from diff in *excludes* section.

• *generation max heap size in mb* – maximum heap size for generation and generation testing.

**Testing Modules Generation**

Projects that keep their generated source files in version control can check that these generated files are up-to-date using build script. After setting *test generation* in *mps settings* to true a call of gentest task appears in *test* target of generated build script. Similarly to *generate* task, gentest loads modules in the script, their dependencies from other build scripts and idea plugins that are required. For each module gentest task invokes two tests: "%MODULE_NAME%.Test.Generating" and "%MODULE_NAME%.Test.Diffing". Test.Generating fails when module has errors during generation and Test.Diffing fails when generated files are different from the ones on disk (checked out from version control). Test results and statistic are formatted into an xml file supported by the TeamCity build server.

**IDEA plugins**

*idea plugin* construction defines a plugin for IntelliJ IDEA or MPS with MPS modules in it. In the screenshot you can see an example of such plugin.

```
idea plugin jetbrains.mps.samples.complex
  name MPS Complex Language
  short (folder) name mps-samples-complex
  description Enables working with complex numbers in base language
  version 1.0
  vendor JetBrains
  url http://www.jetbrains.com/mps/
  icon16 <no icon16>
  content: Complex (custom packaging for jetbrains.mps.samples.complex.library)
  dependencies:
    jetbrains.mps.core
```

The first section of the plugin declaration consists of various information describing the plugin: its name and description, the name of the folder, the plugin vendor, etc. The important string here is *plugin id*, which goes after the keywords *idea plugin*. This is the unique identifier of the plugin among all the others (in the example the plugin id is jetbrains.mps.samples.complex).

The next section is the actual plugin content – a set of modules or module groups included into the plugin. If some module included in the plugin needs to be packaged in some special way other than the default, this should also be specified here (see the line "custom packaging for jetbrains.mps.samples.complex.library").

The last section is dedicated to the plugin dependencies on other plugins. The rule is that if we have a "moduleA" located in plugin "pluginA", which depends on "moduleB" located in "pluginB", then there should be a dependency of "pluginA" on "pluginB". A typesystem check exists that will identify and report such problems.

The layout of the plugin is specified last:

```
plugin jetbrains.mps.samples.complex
  folder languages
    module jetbrains.mps.samples.complex.library
```

In the screenshot, module jetbrains.mps.samples.complex.library is packaged into the plugin manually as it is specified in *idea plugin* construction not to package it automatically.

**MPS Targets**

The MPS plugin provides the following targets:

• *generate* - generates the mps modules that are included in the project structure.

• *cleanSources* - cleans the generated code (only for modules without bootstrapping dependencies). See more about bootstrapping dependencies in article Removing bootstrapping dependency problems.
• declare-mps-tasks - a utility target that declares mps tasks such as generate or copyModels.

• makeDependents - invokes the generate target for a transient closure of this script’s dependencies (if there is one) and then invokes assemble to put them together. It is guaranteed that each script is executed only after all its dependencies have been built.

77.4.4 Module Testing plugin

The Module testing plugin, provided by jetbrains.mps.build.mps.tests language, adds to build scripts the capability to execute NodeTestCases and EditorTestCases in the MPS solutions. Tests are executed after all modules are compiled and packaged into a distribution, i.e. against the packaged code, so they are invoked in an environment that closely mimics the real use of the code.

Test modules configurations

Solutions/module groups with tests are grouped into test modules configurations, which is a group of solutions with tests to be executed together in the same environment. All required dependencies (i.e. modules and plugins) are loaded into that environment.

In the screenshot, you can see a test modules configuration, named execution, which contains a solution jetbrains.mps.execution.impl.tests and a module group debugger-tests.

There is a precondition for solutions to be included into a test modules configuration. A solution should be specified as containing tests (by selecting "with tests" or "with sources and tests" in inspector). A module group should contain at least one module with tests.

Test results and statistic are formatted into an xml file (which is supported by TeamCity).

77.4.5 MPS-runner plugin

The MPS-runner plugin, provided by jetbrains.mps.build.mps.runner, enables a new build script entry - run code from solution. By pointing it to a solution that holds your Java code the build script will be able to run it as part of the build process.

A minimalistic build script that invokes a Java class located in a "sandbox" solution of a project could lose somewhat like this:
77.4.6 Control over the repository

The MPS ant task provides full control over the repository contents with several new tags - *module*, *modules* and *allmpsmodules*.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project name="${project}" default="run" basedir=".">
  <migrate project="${project}">
    <repository>
      <allmpsmodules/>
      <modules dir="${moduleLibFolder}"
      <module file="${uniqueModule.msd}"/>
    </repository>
  </migrate>
</project>
```

77.5 How-to's

The following articles explain how to build a language plugin:

- Building IntelliJ IDEA language plugins
- Building MPS language plugins
- Building standalone IDEs for your languages
Articles on the topic of building with MPS:

- Removing bootstrapping dependency problems
Chapter 78

Building IntelliJ IDEA language plugins

So you have created a set of languages and would like to make them available to Java developers inside IntelliJ IDEA. In this document we are going to look at ways to package a set of languages, perhaps together with the runtimes they depend on, into a valid IntelliJ IDEA plugin.

Do you prefer video? Then you may also like to check out our screen-cast covering the topic of IntelliJ IDEA language plugin creation.

Note: The JavaExtensionsSample sample project that comes with MPS contains a fully functional build script to build and package the sample Java extensions into a plugin. You can take inspiration from there.

78.0.1 Starting point

I assume you have built your languages and now it is the time share them with the world. There are a couple of steps that you need to follow in order to get a nice zip file, which you can upload to the server for others to grab.

In brief:

- Create a build script (manually or through a wizard)
- Generate an Ant build xml file
- Run Ant
- Pick the generated files and share them

Now we’ll continue in more detail. Alternatively you may like to try our new screen-casts that covers the topics of building as well as using IntelliJ IDEA language plugins.

Create a build script

First of all, we need to create a new build script in the newly created build solution. We have two options - using a wizard or creating the build description manually.

Using the Wizard  We can use the Build Solution wizard to generate a solution for us

The wizard will ask whether the new build script should become a part of an existing solution or whether a new one should be created.
A model has to be created inside the new solution:

You can also specify, whether you intend to package the outcome as an MPS or IntelliJ IDEA plugin:
Finally, you select the languages and solutions to include in the build script:

The generated build description script will look something like this:
The manual approach  To get more control over the process, we can alternatively create the build script ourselves. So first we will have to pick an existing solution or create a new one. In the root of your project’s logical view right-click and pick "New Solution". Once the solution exists, create a new model in it. The model should have jetbrains.mps.build and jetbrains.mps.build.mps listed as its Used languages and jetbrains.mps.ide.build should be mentioned as an Dependency.
CHAPTER 78. BUILDING INTELLIJ IDEA LANGUAGE PLUGINS

With the solution and the model prepared you can now create a new build project through the pop-up menu:
CHAPTER 78. BUILDING IntelliJ IDEA LANGUAGE PLUGINS

Editing the build script

Either way you created the build script, now it is the time to edit your build description. In order to be able to package your languages and solutions as IntelliJ IDEA plugins, you should ultimately get a script similar to this one:

It is important to define artifact locations for both dependencies. The IDEA dependency needs to know where to find an IntelliJ IDEA installation. The mpsPlugin dependency requires to know where to find the MPS core IDEA plugin. After you enabled MPS inside IntelliJ IDEA by adding the plugin, it is typically located in the IntelliJ IDEA installation folder/plugins folder. Alternatively the plugin may also have been installed into the user home folder, under IntelliJ IDEA setting folder/config/plugins.

Please check out the build language documentation for details on the build language capabilities.

78.0.2 Generate an Ant build xml file

With the build script prepared you can now generate an Ant build.xml file. Just rebuild your build solution and the file will be generated into the location that you specified in the properties of your build script as indicated in the following picture:
The generated `build.xml` file should start something like this:

```xml
<project name="calculator-tutorial" default="build">
  <!-- Generated by MPS -->
  <property name="build.dir" location="build" />
  <property name="build.tmp" location="${build.dir}/tmp/calculator-tutorial" />
  <property name="build.layout" location="${build.dir}/artifacts/calculator-tutorial" />
  <property file="${artifacts.mpsPlugin.build.properties}">
    <import prefix="import.mpsPlugin" />
    <property name="mpsPlugin.plugin.version" value="${import.mpsPlugin.plugin.version}" />
    <property name="mpsPlugin.plugin.artifact" value="${import.mpsPlugin.plugin.artifact}" />
  </property>
  <path id="path.mps.ant.pach">
    <path element="${artifacts.mpsPlugin}/mps-core/lib/antlr/lib/antlr-pars.jar" />
    <path element="${artifacts.mpsPlugin}/log4j.jar" />
  </path>
</project>
```

Notice the `artifacts.IDEA` and `artifacts.mpsPlugin` properties in the generated script. When running the script from the command line, we have to make them point to the IntelliJ IDEA installation and the location of the MPS core plugin respectively.

Since in the script above we set the locations of both artifacts through macros (`$idea_home` and `$plugins_home`), we may set these properties instead (necessary only if the locations on the build machine differ from the paths set in the build script).

### 78.0.3 Run Ant

Time to run the build and get our plugin. Again, we have two choices - either stay inside MPS or use the command line.

**From within MPS**

Just right-click on the build node in the Project View and choose run:
The generated artifacts will be located in the build folder inside the folder you specified as the base.

From the command line

You need to navigate the command line to the folder where the build.xml file is located and run ant. If the locations of the MPS core plugin and IntelliJ IDEA on the build machine are different from the path set in the build script, we should specify the actual location on the command line ant -Dartifacts.mpsPlugin="..." -Dartifacts.IDEA="...". Since in our example above we use $idea_home to point to the IntelliJ IDEA installation folder and $plugins_home to point to the MPS core plugin, we can instead change the values of these properties on the command line: ant -Didea_home="path to idea" -Dplugins_home="path to the MPS core plugin".

This step, no matter whether you run the script from MPS or the command line, will generate a zip file containing all the necessary jar files organized as we specified in the layout section of our build script or (as in our displayed case) using the default layout of IntelliJ IDEA plugins.

78.0.4 Pick the generated files and share them

Now we need to distribute the generated files to the IntelliJ IDEA users so that they can add them as plugins. Check out Using MPS inside IntelliJ IDEA for detailed instructions on how to install the plugin for use in IntelliJ IDEA.
Chapter 79

Using MPS inside IntelliJ IDEA

We are going to show how to use MPS languages inside IntelliJ IDEA in this document. You may also like to check out the screen-cast covering this topic.

Note: Please make sure you have IntelliJ IDEA installed. Both Community and Ultimate Edition would work.

79.0.5 Get the MPS core IntelliJ IDEA plugin

Download the MPS IntelliJ IDEA plugin from the MPS download page or install it directly using IntelliJ IDEA’s plugin manager. If installing manually, the downloaded plugin zip file must be unzipped into the IDEA’s plugin folder.

- **Windows and Linux**: typically `USER_HOME/IDEA_version/config/plugins` or `IDEA_HOME/plugins`
- **MacOS**: `/Library/Application Support/Idea_version/` or `IDEA_HOME/plugins`

Do not use the plugin manager to install the plugin from disk using the downloaded zip, since the zip file is not a valid plugin zip and it contains several plugins packaged together.

The zip file holds several MPS plugins, three of which are essential:

- mps-core - holds the core MPS functionality
- mps-java - enables BaseLanguage and its extensions
- mps-vcs - allows VCS for MPS models in IntelliJ IDEA

Please, bear in mind that each MPS plugin version is compatible only with a particular version of IntelliJ IDEA. Typically, the latest released MPS plugin is compatible with the IntelliJ IDEA version that was latest at the time of the MPS release and the following IDEA minor versions. The EAP versions of the MPS plugin are usually compatible with the current IntelliJ IDEA version.

The plugin manager in IntelliJ IDEA will only give you access to a compatible version of the MPS plugin, so this option is always a safe bet.

If you manually install an incompatible version of the plugin, IDEA will refuse to load it during start up, will log an error and mark the plugin in red in the plugin manager.

After installation, the plugin directory should look something like this:

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>mps-build</td>
<td>16.7.2013 20:53</td>
<td>File folder</td>
</tr>
<tr>
<td>mps-core</td>
<td>16.7.2013 20:53</td>
<td>File folder</td>
</tr>
<tr>
<td>mps-java</td>
<td>16.7.2013 20:53</td>
<td>File folder</td>
</tr>
<tr>
<td>mps-testing</td>
<td>16.7.2013 20:53</td>
<td>File folder</td>
</tr>
<tr>
<td>mps-vcs</td>
<td>16.7.2013 20:52</td>
<td>File folder</td>
</tr>
<tr>
<td>availables.xml</td>
<td>15.7.2013 17:47</td>
<td>XML Document</td>
</tr>
</tbody>
</table>

This will enable MPS and the core set of languages to be available inside IntelliJ IDEA once you restart it.
79.0.6 Turn MPS on and off

The Plugin Manager in IntelliJ IDEA should now list several MPS plugins, which you can turn on and off as needed:

79.0.7 Try BaseLanguage inside IntelliJ IDEA

You can now enable the MPS facet for your IntelliJ IDEA projects:

The facet offers several configuration points, but you most likely want to stick to the default settings for your first experiments.

Adding the MPS facet to a project’s module will create a models node in the Project View and allow you to create MPS models. If not, just create one on the Sources tab:
You need to mark it as a **Model Folder** (the icon next to the `models` folder should turn blue):

The final step is to mark the `models` folder as a **Sources** directory of the module:
79.0.8 Playing with MPS models

Once the MPS facet is configured and has one or more Model Roots marked Model as well as Sources Folders, you can create models in them:

When creating a new model, you have to specify its name and choose one of the available model Kinds through the Create New Model dialog:

Each model Kind in this case represents a preconfigured set of used languages for the model. This is a convenient shorthand that still allows you to tune the list of used languages manually at any time through the Model Properties dialog by pressing F4 or Alt+Enter on the model node inside the Project View: The list of Kinds is currently set to contain only these two options (Empty and Java), but will be configurable in the future releases.

Right-clicking on a model will display the imported models and used languages:
The MPS models let you create new nodes from the imported languages the same way you would do it in MPS itself: Model Nodes in the Project View represent the content of MPS models and just like in MPS they are similar to Java packages. You can create model elements (Roots) in them:

Similarly to when we were creating new MPS models, you give each root element a name and a kind, which now means its MPS Concept:

The root nodes are displayed in the Project View and look very similar to how other resources and files look in IntelliJ IDEA. The projectional editor for a root node can be opened by double-clicking on the root node or by pressing F4.
79.0.9  Code generation

The code process of MPS models is tied to the IntelliJ IDEA’s make/rebuild project actions and so will be triggered automatically.

It’s possible to configure where to store the generated source code. This is configured through the MPS facet settings dialog. There are three options you can choose from:

- The generated code stays in memory only. It will be available during the compilation process only and never saved to disk at all
- It can be saved into a configured source folder of this module
- It can be saved into any other directory wherever on the disk

79.0.10  Install the language plugin

Now you probably can’t wait to try out your own language inside IntelliJ IDEA. All you need to do is to unpack the plugin zip file into the same IntelliJ IDEA’s plugins folder that we unzipped the core plugin into earlier:
Additionally, many of the plugins have been shared through the IntelliJ IDEA plugin repository and so can be comfortably downloaded through the IDEA’s plugin manager.

After you restart IntelliJ IDEA, you will be able to add your language(s) that came through the plugin to the list of used languages:

This should be enough to enable your custom language(s) for use inside IntelliJ IDEA.

79.0.11 Adding runtime libraries

Your languages will typically come with runtime dependencies on other languages or solutions. Although they typically come bundled as jar files inside the plugin zip file, they need to be imported as module dependencies into IntelliJ IDEA. The MPS facet does that automatically once you add a language with a runtime dependency.
In case you needed to add run-time libraries manually, the real location of these dependencies (jar files) is under the particular language plugin folder in the IntelliJ IDEA’s plugins location.

### 79.0.12 Interoperability between Java and BaseLanguage
The MPS IDEA plugin integrates your MPS code tightly into the rest of the Java project. MPS code will seamlessly participate in IDEA’s build and make, you can cross-navigate to usages or definitions between Java and BaseLanguage as well as refactorings will correctly include all the sources.

### 79.0.13 Converting Java code to BaseLanguage
MPS gives you a convenient option to quickly convert Java code to BaseLanguage and migrate it into an MPS model. Just right-click on the desired Java package and select **Convert Java to MPS**.

### 79.0.14 MPS configuration options
Just like MPS itself, the MPS plugin can be configured and customized using the **Settings** dialog, which you can invoke through the menu, a toolbar or using a keyboard shortcut (**Control + Alt + S / Cmd + ,**). MPS has added a couple of configuration screens to the IntelliJ IDEA’s **Settings** dialog, which allow you to configure several aspects of how MPS behaves.
79.0.15 Configure the generator

Additionally, you need to make sure the Use external build flag in the IDEA Compiler settings is turned off, otherwise you won’t be able to rebuild your project.
CHAPTER 79. USING MPS INSIDE INTELLIJ IDEA

![Settings](attachment:image.png)

- **Resource patterns:** 
  - Use `:` to separate patterns and `*` to negate a pattern. Accepted wildcards: `*` — exactly one symbol; `*` — zero or more symbols; `/` — path separator; `<dir>name>pattern>` — restrict to source roots with the specified name.

- **Use external JRE:** 
  - Check if your project automatically compiles.
  - If disabled, set your project to use an external JRE.
  - Compile independent modules in parallel. 
    - May require larger heap size.
  - Compiler heap size (MB): 700
  - Additional compiler VM options: 

---

**Project Settings (Dmca)**

- Code Style
- Compiler
  - Excludes
    - Java Compiler
    - Annotation Processors
    - JRE Compiler
  - File Encodings
  - File Locations
  - Schematics and UMLs
  - Scope
  - Template
  - Data Languages
  - Version Control

**IDE Settings**

- Appearance
- Console Folding
- Editor
Chapter 80

Building MPS language plugins

Distributing languages as plugins for MPS is very similar to packaging languages for IntelliJ IDEA. You follow the same instructions described in Building IntelliJ IDEA language plugins with two important differences:

1. Your build script should depend on mps as you can see in the attached screen-shot.

   ![Calculator Tutorial Build XML](image)

   Additionally, by depending on mps your languages can leverage the whole MPS infrastructure and the whole set of languages, while mpsPlugin restricts IDEA plugins to only the subset of functionality that is available in IntelliJ IDEA.

2. When running your Ant build script from the command line, you set an artifacts.mps property (instead of artifacts.mpsPlugin) and make it point to an MPS installation home directory.
Note: Since in the example build script above we set the artifacts location for mps to $mps_home, we may set 
-Dmps_home instead (necessary only if the MPS location on the build machine differs from the path set in the build 
script).

The generated plugin can then be imported into MPS. You can install the plugin either manually or through the MPS plugin 
manager. If installing manually, the downloaded plugin zip file must be unzipped into the MPS plugin folder.

- **Windows and Linux:** typically USER_HOME/MPS_version/config/plugins or MPS_HOME/plugins
- **MacOS:** ~/Library/Application Support/MPS_version/ or MPS_HOME/plugins

After restart the imported languages will become available in MPS.
Chapter 81

Building standalone IDEs for your languages

81.1 Introduction

The term standalone IDE refers to a stripped down version of the IDE that only contains those artifacts that are necessary for a given business purpose. Standalone IDEs provide a convenient way to distribute DSLs to the end users, who will be able to use the languages including all the IDE support, refactorings and code analysis prepared by language designers in the comfort of a dedicated IDE. All the distracting language-design-related functionality and unnecessary languages will have been removed.

Once you have designed your languages, MPS can help you build a stripped down version of MPS that only contains those artifacts that are necessary to use these domain-specific languages. Various aspects of MPS can be removed or customized to deliver a custom user experience for a given group of users. This article describes how.

In particular, the following aspects of MPS can be customized:

- various icons, slogans, splash screens and images
- the help URL
- the set of languages available to the users
- the set of plugins available in MPS

81.2 Process Overview

To build a custom RCP version of MPS, you have to create a solution that contains a so-called build script. A build script is written in MPS’ build language, which is optimized for building RCP applications (as well as IntelliJ and MPS plugins). When running the generator for this build script, MPS generates an ant file that creates the actual RCP distribution.

81.3 Building an example RCP build

In this document we describe the development of an RCP build scripts for the robot Kaja sample that is bundled with MPS distributions. You can open the project in MPS and follow the instructions described here.

81.3.1 Creating the Solution and the Build Script

The wizard way

The preferred way to create the necessary build scripts is to run the Build Solution Wizard on your current project. You start it from the Project View tool window:
The wizard will create a new solution and a model to hold the scripts and set the necessary imports as well as used languages. You can instruct the wizard to reuse an existing solution or a model, instead of creating new ones. You’ll then need to select the **Standalone IDE** option:
On the last page of the wizard, you de-select the modules that should not be made part of the IDE. Typically you skip the sandbox modules and keep the languages as well as plugin and runtime solutions.

The new solution should now be listed in the **Project View** and should hold two new build scripts:

The dependencies have been also set properly:
The manual way

Alternatively you could create the scripts manually. You’d have to create a new solution with a new model inside. In the model, configure the used languages

- `jetbrains.mps.build`
- `jetbrains.mps.build.mps`

Also import the `jetbrains.mps.ide.build` model. You can now create a new build project in the model (the name is confusing: it is not actually a new build project, just a build script).

81.3.2 Investigating the build scripts

A build script contains several sections, which we can observe on an empty build script:
Let us look at the various sections of a build script:

- **base directory** The base directory defines an absolute path relative to which all other paths are specified. By default, this is the directory in which the resulting `ant` file will be generated, and in which it will be executed.

- **use plugins** The build language itself can be extended via plugins (Note that these are not the plugins that make up MPS itself; those will be configured later). These plugins contribute additional build language syntax. Typically, the `java` and `mps` plugins are required. We will use syntax contributed by these plugins below.

- **macros** Macros are essentially name-value pairs. In the Macros section, these names are defined and values are assigned. In the remainder of the build script these macro variables will be used. MPS supports two kinds of Macros: `var` macros are strings and can be assigned any value. `folder` represents paths, relative to the base directory defined above and/or based on other Macros. Note that MPS provides code completion for the path components in `folder` macros.

- **dependencies** This section defines dependencies to other build scripts. MPS bundles a set of build scripts (e.g. `buildStandalone`, `buildWorkbench` or `buildMPS`). By establishing a dependency to any one of them, the structures defined in that references build script can be used in the referencing build script. For example, the macros defined in the referenced build scripts can be used.

- **project structure** This section defines the actual contents of the to-be-built RCP application. Such contents may be MPS modules (languages, solutions, devkits), Java libraries, IDEA branding and plugins.

- **default layout** This section defines the files that are output, it creates the directory, file and JAR structure of the new RCP distribution. It references and includes the artifacts defined in the project structure section and in other build scripts this one depends on.

The wizard created two builds scripts, each of which serves a separate purpose:

1. **Kajak** - generates the modules, compiles the generated sources and packages them into an IDEA plugin (a plugin for the IntelliJ IDEA platform)

2. **KajakDistribution** - creates platform-specific distributions out of the artefacts created by **Kajak**

### 81.3.3 Editing the build scripts

Let’s focus on the **Kajak** build script first.

**Name of the generated build script**

The default name of the ant `xml` file to generate from each MPS build script is `build.xml`. It is advisable to change the name so that you avoid clashes between multiple build scripts:
Base directory

Specifies the directory, into which the build script will be generated.

Use plugins

The build language has packaged its capabilities into plugins. The plugins used in the build script are listed in its use plugins section.

- **java** - contains the capability to compile Java sources
- **mps** - contains the capability to generate MPS models
- **module-tests** - adds capability to run tests in the build script (you need to import the `jetbrains.mps.build.mps.tests` language)

The wizard has already pre-set **java** and **mps** for us.

Macros

We should define a couple of macros to reused across the script. The first two represent string variables we will use in the build, the set of folders represent the directory for MPS itself as well as the two root directories where the MPS languages are stored. These languages will become part of the RCP distribution. Note how all of these folders are relative to the base directory.

```text
macros:
  var date = date 20150422
  var build.number = Kajak-139.SNAPSHOT
  folder mps_home = ./../../../../Applications/MPS 3.2.app/Contents
```

Note that you can leave a macro undefined (i.e. just specify the name) and then define it either in the MPS Path Variables section of the Project Settings dialog, or when you call `ant` by using a property definition:

```text
ant -Dsome.variable=a.value
```

Dependencies

Next, we define the dependencies. Dependencies indicate, on which other build scripts and platforms this script depends. We need dependencies to **mpsStandalone** (it represents the minimal set of a standalone MPS installation) and optionally also **mpsMakePlugin** that enables creating build scripts in the standalone IDE, **mpsDebuggerPlugin** (because the robot Kaja
sample requires debugger and Java execution integration) and mpsExecutionPlugin (because one the robot Kaja sample languages uses the execution framework to run an external executable). Most RCPs will likely just use mpsStandalone.

dependencies:
mpsStandalone (artifacts location $mps_home)
mpsMakePlugin (artifacts location $mps_home/plugins)
mpsDebuggerPlugin (artifacts location $mps_home/plugins)
mpsExecutionPlugin (artifacts location $mps_home/plugins)

The artifacts location specifies from where the artifacts will be copied. The build scripts work by copying code from an existing MPS installation, so you don’t have to check out the MPS sources. The artifacts location should hence be pointing to the MPS home directory.

Project Structure

The plugins specified in the use plugins section influence the options available here. With java, mps and module-tests plugins enabled we get the following ones:

- **generator options** - configure the generator
- **idea branding** - configure the visual aspect of standalone IDEs, such as icons, splash-screens, urls, images and other
- **idea plugin** - details the information to hook the generated artefacts into the underlying IntelliJ IDEA platform
- **Java library** - groups jars and classes so they could be referred to as a unit
- **Java module** - groups Java sources so they could be referred to as a unit
- **Java options** - configures the java compiler process
- **mps group** - groups MPS modules (solutions, languages, devkits) so that they could be referred to as a unit
- **solution** - represents a solution
- **language** - represents a language
- **devkit** - represents a devkit

In the project structure we start with the branding. The idea branding section supports the definition of all kinds of icons, splash screens and the like. It is a good idea to take those from the MPS distribution and modify the pictures to suit your needs, while retaining the sizes and transparency values. You can also specify a help file and an update website.

In the sample we have copied the MPS icons into the ./icons folder and changed them accordingly without changing the names — this is why most of them start with mps.
Next we define an idea plugin that contains the languages and solutions to bundle. An idea plugin is a plugin to the IDEA platform on which MPS is built.

```plaintext
idea plugin Kajak
  name Kajak
  short (folder) name Kajak
  description <no description>
  version 1.0
  << no vendor >>
  content:
    Kajak
dependencies:
    jetbrains.mps.core
  << ... >>
```

You can define a name and a version, dependencies to other plugins as well as the actual contents. Whenever your RCP contains languages, you will need jetbrains.mps.core because it provides things like BaseConcept or the INamedConcept interface. The Kajak entry in the contents is a reference to an mps group defined below. An mps group is a group of MPS artifacts. Since we have included the mps plugin (at the very top of the build script) we can use MPS-specific language constructs. mps groups are an example.

Let us look at the Kajak group. It references a set of languages and solutions. A language reference points to a language defined in an MPS project. A language reference in a group consists of the name (which must be the same as the name of the actual language) and a pointer to the respective mpl file. The simplest way to enter it is to start with the path to the mpl file and then use the load required information from file intention (Alt + Enter) to adjust the name of the reference.

```plaintext
mps group Kajak
  solution jetbrains.mps.samples.JavaKaja
    load from ./solutions/JavaKaja/JavaKaja.msd
  language jetbrains.mps.samples.KajaAndOr
    load from ./languages/KajaAndOr/KajaAndOr.mpl
  language jetbrains.mps.samples.Kaja
    load from ./languages/Kaja/Kaja.mpl
  language jetbrains.mps.samples.KajaSceneConstruction
    load from ./languages/KajaSceneConstruction/KajaSceneConstruction.mpl
```

Note that a group has to contain the transitive closure of all languages. For example, if you have references a language A which references another language B, then B must also be included in the group. This makes sense, because otherwise the resulting RCP application would not run because dependencies could not be resolved. If a language is missing, then an error will be reported in the build script (red squiggly line). After adding the required languages you may have to rerun the load required information from file intention (Alt + Enter) on the language with the error to make it “notice” that something has changed.

In addition to languages, you can also reference solutions with the solution construct and devkits with the devkit construct.

The Inspector tool window contains further details on the dependencies of the selected language, solution or devkit.

**Default Layout**

The layout constructs the directory and file structure of the RCP distribution. It copies in files from various sources, and in particular from the project structure discussed in the previous section. It also compiles, builds and packages these files if necessary. It can also refer to artifacts defined in other build scripts, on which this script depends. We start out with the following code:

```plaintext
default layout:
  import mpsStandalone::languages
  import mpsStandalone::license
```

Those two imported are folder elements in the referenced mpsStandalone build script. By importing them, the contents of these folders are imported (i.e. copied into) our RCP distribution.

The wizard has added code to create a bin folder and copy common configuration and properties files into it. You may further customise it or leave it untouched.
Next up, we create a new folder `lib` into which we import all the stuff from `mpsStandalone::lib` except the MPS sources and the existing `branding.jar` (which contains the default MPS splash screen etc.). We then create a new jar file `branding.jar` (it has to have this name to work; the IDEA platform expects it this way) into which we put all the files defined in the `branding` section of the project structure defined above.

We then create a folder `plugins`. The `mps-core` plugin is required in any RCP app, so it needs to be imported in any case. We then import various version control plugins (CVS, SVN and git) defined in the `mpsStandalone` build script. For obvious reasons, a standalone Kaja IDE needs version control support, so it makes sense to import those plugins. Finally, we include the debugger plugin, since Kaja also integrates with the MPS debugger.

We then integrate the Kajak plugin defined in the project structure earlier. Using the `plugin` construct we can import the complete definition of the plugin defined earlier.

Inside the plugin we define further subdirectories that contain various library jars. These jar files are automatically made visible to the plugin classpath. Using the `file` construct we copy various files into the plugin directory, using the macros defined earlier.

Note how this plugin does not just contain the jars we copied in, but also all the language artifacts necessary. That has been accomplished by defining these `mps groups` and referencing them from the plugin. The fact that we can use these `mps groups` is thanks to the `mps` build language plugin. It provides direct support for MPS artifacts, and all the necessary files and metadata are handled automatically.

We may also import `mps-make`:

```groovy
# optional plugins
import mpsMakePlugin::mps-make
```

Property Files

Finally we create a property file called `build.number` that contains a bunch of meta data used in the running RCP application. It is required by the underlying IntelliJ platform.

```groovy
properties file build.number
    build.number = ${build.number}
    date = ${date}
    version = 1.0
```
81.3.4 Creating the Platform-Specific Distributions

At this point, the generated directory structure constitutes the platform-independent core of a stripped-down MPS distribution that contains all the artifacts you've configured into it, including your custom branding. However, you cannot run it, since the platform-specific aspects are still missing.

It is time to look at the second generated build script - *KajakDistribution*.

```plaintext
build KajakDistribution generates buildDistribution.xml

base directory: ../../

use plugins:
<< ... >>

macros:
 folder mps_home = ../../work/MPS 3.2
     var version = 1.0

dependencies:
    Kajak

project structure:
<< ... >>

default layout:
tar ${build.number}-linux.tar.gz (compression gzip)
    folder Kajak ${version}
        import files from Kajak::/
            <any>
        folder bin
            file $mps_home/bin/linux/fsnotifier (755)
            file $mps_home/bin/linux/fsnotifier64 (755)
            file $mps_home/bin/mps.vmoptions (644)
                fix eol: convert to a single LF, remove eof (Ctrl-Z): true
            file $mps_home/bin/mps64.vmoptions (644)
                fix eol: convert to a single LF, remove eof (Ctrl-Z): true
            files from $mps_home/bin/linux
                exclude **/fsnotifier
                exclude **/fsnotifier64

            file $mps_home/mps.sh (755)
                fix eol: convert to a single LF, remove eof (Ctrl-Z): true

        zip ${build.number}.zip
 ...
```

The build script depends on *Kajak* so that it could properly package the platform-neutral artefacts into platform-specific distributions. The `mps_home` macro points to a *generic* MPS installation, since the distributions targeting a single platform (MacOS, Linux or Windows) would miss artefacts needed by the other platforms. The generic distribution is also available for download from the MPS website.

81.3.5 Creating a Windows distribution

There is no windows-specific distribution generated by the builds scripts. To run the generated standalone IDE on windows, you need to use the generated *generic* zip file, copy the contents of the `win` directory into the `bin` directory and use the `mps.bat` file to start the IDE.
81.4 The final Robot Kaja IDE build scripts

81.4.1 Kajak

build Kajak generates myBuild.xml

base directory: ../..

use plugins:
java
mps

macros:
var date = date 20150422
var build.number = Kajak-139.SNAPSHOT
folder mps_home = ./../../../../Applications/MPS 3.2.app/Contents

dependencies:
mpsStandalone (artifacts location $mps_home)
mpsMakePlugin (artifacts location $mps_home/plugins)

project structure:
idea branding Kajak
codename Kajak
company <no company>
version 1.0, eap false
full name Kajak
build number ${build.number}, date ${date}
icons
16x16 ./icons/MPS16.png
32x32 ./icons/MPS32.png
32x32 opaque ./icons/MPS32.png
128x128 <no icon128>
splash screen ./icons/splash.png textcolor 002387
about screen ./icons/about.png
dialog image ./icons/dialogImage.png
welcome screen
  caption ./icons/caption.png
  slogan ./icons/slogan.png
  logo ./icons/logo.png
<no updateWebsite>
plugins <no plugins>
whats new <no whatsnew>
<no stats>
<no help>
feedback url <no feedbackUrl>

idea plugin Kajak
name Kajak
short (folder) name Kajak
description <no description>
version 1.0
<< no vendor >>
content:
Kajak
dependencies:
jetbrains.mps.core
<< ... >>

mps group Kajak
solution jetbrains.mps.samples.JavaKaja
  load from ./solutions/JavaKaja/JavaKaja.msd

language jetbrains.mps.samples.KajaAndOr
  load from ./languages/KajaAndOr/KajaAndOr.mpl

language jetbrains.mps.samples.Kaja...
81.4.2 KajakDistribution

build KajakDistribution generates buildDistribution.xml

base directory: ../../

use plugins:
  << ... >>

macros:
  folder mps_home = ../../work/MPS 3.2
  var version = 1.0

dependencies:
  Kajak

project structure:
  << ... >>

default layout:
  tar ${build.number}-linux.tar.gz (compression gzip)
  folder Kajak ${version}
    import files from Kajak::/
    <any>
    folder bin
      file $mps_home/bin/linux/fsnotifier (755)
      file $mps_home/bin/linux/fsnotifier64 (755)
      file $mps_home/bin/mps.vmoptions (644)
        fix eol: convert to a single LF, remove eof (Ctrl-Z): true
      file $mps_home/bin/mps64.vmoptions (644)
        fix eol: convert to a single LF, remove eof (Ctrl-Z): true
    files from $mps_home/bin/linux
      exclude **/fsnotifier
      exclude **/fsnotifier64
    file $mps_home/bin/mps.sh (755)
    fix eol: convert to a single LF, remove eof (Ctrl-Z): true

zip ${build.number}-macos.zip
  folder Kajak ${version}
    import files from Kajak::/
    <any>
    folder bin
      file $mps_home/bin/mps.vmoptions (644)
      file $mps_home/bin/mps64.vmoptions (644)
      file $mps_home/bin/mps.exe.vmoptions (644)
      file $mps_home/bin/mps64.exe.vmoptions (644)
    folder win
      files from $mps_home/bin/win
        exclude **/*.exe
    filemode folders: <default (755)>, files: 755
      files from $mps_home/bin/win
        include **/*.exe

    folder linux
      files from $mps_home/bin/linux
        exclude **/fsnotifier
        exclude **/fsnotifier64
      file $mps_home/bin/linux/fsnotifier (755)
      file $mps_home/bin/linux/fsnotifier64 (755)

    folder nix
      filemode folders: <default (755)>, files: 755
      files from $mps_home/bin/nix

    folder mac
      file $mps_home/bin/mac/libbreakgen.jnilib (644)
      file $mps_home/bin/mac/libbreakgen64.jnilib (644)
      file $mps_home/bin/mac/restarter (755)
      file $mps_home/bin/mac/fsnotifier (755)
      folder Contents
        folder Resources
          file $mps_home/bin/mac/Contents/Resources/mps.icns (644)
          file $mps_home/bin/mac/Contents/MacOS/mps (755)
          file $mps_home/bin/mac/Contents/Info.plist (644)
          file $mps_home/mps.sh (755)
          file $mps_home/mps.bat (755)
        filemode folders: <default (755)>, files: 755
          file $mps_home/bin/mac/Contents/Resources/mps.icns (644)
          file $mps_home/bin/mac/Contents/MacOS/mps (755)
          file $mps_home/bin/mac/Contents/Info.plist (644)
          file $mps_home/bin/mac/libbreakgen.jnilib (644)
          file $mps_home/bin/mac/libbreakgen64.jnilib (644)
          file $mps_home/bin/mac/restarter (755)
          file $mps_home/bin/mac/fsnotifier (755)
          file $mps_home/bin/mps.vmoptions (644)
            fix eol: convert to a single LF, remove eof (Ctrl-Z): true
          file $mps_home/bin/mps64.vmoptions (644)
            fix eol: convert to a single LF, remove eof (Ctrl-Z): true
          file $mps_home/mps.sh (755)
          fix eol: convert to a single LF, remove eof (Ctrl-Z): true
          file $mps_home/mps64.vmoptions (644)
          fix eol: convert to a single LF, remove eof (Ctrl-Z): true
          <<additional aspects>>
81.5 Generating and running the Build Script

Assuming your build scripts are free of errors, you can generate them (Rebuild Model or Ctrl-F9). This creates an ant file that has the name that is mentioned at the top of the build script:

```
build Kajak generates myBuild.xml
```

This myBuild.xml (in this case) resides directly in your project directory. You can either run the file from the command line (using `ant -buildfile myBuild.xml`) or directly from within MPS (via the context menu on the build script in the project explorer).
Note that the build script compiles and packages all your languages, but it does **not** regenerate them! So you have to make sure by other means that your language definitions are fully generated and up to date.
81.6 Inspecting the Generated Directory Structure

The figure below shows the project structure after running the build scripts. In particular, inside the `build/artifacts` directory, the structure defined in the `default layout` section of the build script has been created.

We suggest you browse through the directory and file structure to connect what you see to the definitions in the build script.
Chapter 82

Extending the user interface

82.1 Creating wizards

Sometimes, you may have the need to provide wizards that would automate some tasks of the users of your languages or IDEs. MPS allows you to create those and so let the users customize specific settings (e.g. in the model) for the actions that will be performed by the wizard, possibly in several steps. This wizard will have similar look and feel to the MPS migration wizard and thus seamlessly fit into the system.

You need to extend the AbstractWizard class to create your own wizards. The StepAdapter class lets you create additional steps for the wizard, which you then register with the wizard using its addStep() method.

We recommend you to look into the sources and check out some of the existing wizards, e.g. GenerateBuildWizard. Please check out the Contributing to MPS Project page for a few useful tips on getting and opening the MPS sources.
Chapter 83

Removing sources from generated code

MPS bundles sources with generated models by default. Users of the models are thus able to navigate to definitions of the concepts that they are using in their code. It is a matter of a single Control + click for them to see the implementation of a method that they are calling or a class they are instantiating, for example. This is very convenient for the users, since they can grasp many of the ideas of the language/library authors simply by peeking at the implementation side.

There may be situations in life when hiding the implementation could be desired, though. Especially closed-source projects need to protect carefully their intellectual property contained in the implementation. Their users should still be able to call the code, but after pressing Control + B they only get to see the class and method signatures:

```
public class Memoizator {
    public Memoizator() {
        /* compiled code */
    }
    public static map<Integer, Object> buildMemeizeCache(int sizeLimit) {
        /* compiled code */
    }
    private static ConcurrentHashMap<Integer, Object> createMap() {
        /* compiled code */
    }
    private static map<Integer, Object> buildBoundedMemoizeCache(int fixedSize) {
        /* compiled code */
    }
    public static Object retrieveFromCache(map<Integer, Object> cache, KeyCalculator key) {
        /* compiled code */
    }
    public static void storeInCache(map<Integer, Object> cache, KeyCalculator key, Object value) {
        /* compiled code */
    }
}
```

Combine that with obfuscated class files and chances for someone reverse-engineering the fruits of your hard work are pretty low. Please read on to find out how to do this.

83.1 Remove sources from BaseLanguage code

The BuildLanguage offers a strip implementation flag to indicate that a particular artifact should have the sources removed.
Setting this flag to `true` on one of the build layout commands (`module`, `sources of`, `plugin`) will instruct the build process to remove the sources of implementation from the generated artifacts. This flag will ensure that `BaseLanguage` methods have their bodies replaced with an empty `StatementList` and classes have static and instance initializers removed. May you wish to hide implementation of a `Language`, the flag will ensure the behavior methods have their bodies also replaced with an empty `StatementList`.

When building a language and the `strip implementation` flag is set, the generator module must have the `Generate Templates` flag set on. Otherwise the stripped generator will not work for the users of your language since the templates would be missing.

To summarize, MPS supports out of the box:

- Hiding your implementations written in `BaseLanguage`
- Hiding aspects of your language definitions

You only need to set the `strip implementation` flag in your build scripts for the desired solution or a language.

If you strip implementation from a language definition, the users of your language will be able to use your language fully. They, however, will not see the implementation and they will not be able to extend many aspects of your language.
83.1.1 A few handy notes

1. The ability to *Invalidate caches* in MPS may come handy when switching between projects containing stripped and not-stripped versions of your language.

2. Sometimes it may be necessary to manually introduce changes in your language so that the build script reflects the changes in the generated artifacts.

83.2 Customizing implementation stripping for your own language

Just as *BaseLanguage* code allows for hiding implementation, your languages can allow for implementation hiding, as well.

Please, do not confuse "Customizing implementation stripping" with "Hiding implementation of your language". While the latter is concerned with hiding the way you implemented your language and works automatically, as described above, this paragraph is concerned about creating a language that allows its usages to hide their implementation.

MPS gives you three marker interfaces to demarcate the intended behavior of the concepts of your language with respect to implementation stripping:

- **InterfacePart** - concepts that are fully visible in the generated models. Users will be able to navigate to them and keep references to them in their code.
- **ImplementationPart** - concepts that get removed from the generated models. Users will not be able to navigate to them or keep references to them in their code.
- **ImplementationWithStubPart** - concepts that get replaced with empty stubs. It behaves like *ImplementationPart* except that a stub replacement will be used to represent the nodes in the code. Think of /* compiled code */ marks for empty method bodies, for example.

83.2.1 Robot Kaja sample

If, for example, we wanted to the Robot Kaja sample language (bundled with MPS) to allow its usages to hide their implementation, we can achieve that in a few steps. Let’s assume the following scenario:

1. An end user is writing scripts as part of a *SampleRobotScripts* project in the *Robot Kaja* language. He wants to download and reuse a library called *RobotRoutines* (MPS solution) of robot routines.

2. The author of *RobotRoutines* wants to hide the implementation of her library.

3. The *Robot Kaja* language needs to be modified to support hiding implementation of *RobotRoutines*. It needs to declare, which of its concepts form the public interface (contract) of applications/libraries written in that language and which hold the implementation. We picked a simple language, so it is pretty straightforward to identify that only three concepts need really to be made part of the interface. The others can have their sources removed during packaging.

(a) Script

(b) Library

(c) RoutineDefinition

83.2.2 Initial situation

The Robot Kaja language allows *Libraries* of routines to be created. *Library* is a root concept and holds a collection or *RoutineDefinitions*. The *RobotRoutines* library that uses the *Robot Kaja* language may create several *Library* root nodes and implement a few handy routines in them.
When used in the SampleRobotScripts code, the developer can always navigate to definition and see the full implementation of the RobotRoutines library.

```plaintext
Library Common defines
  routine turn around means
    repeat 2 times
      turnLeft
    end
  end
end
routine run means
  repeat 8 times
    if not wall ahead do
      step
    end
  end
routine pirouette means
  repeat 2 times
    turn around Library call
  end
end
```

```plaintext
Script Sample runs as
  step
    require Common
  end
```
83.2.3 Goal

We want to offer the authors of libraries such as RobotRoutines the ability to hide the implementation. Once implemented properly, the developers of SampleRobotScripts would only see the routines signatures and their empty bodies:

```
Library Common defines
routine turn around means
  repeat 2 times
    turnLeft
  end
end
routine run means
  repeat 80 times
    if not wall ahead do
      step
    end
end
routine pirouette means
  repeat 2 times
    turn around Library call
  end
end
```

83.2.4 Changes to the Robot Kaja language

The Robot Kaja language needs to have its Script, Library and RoutineDefinition concepts marked with the InterfacePart interface, because user code can refer to them.
To hide the implementation of routines we need to hide their bodies. The routine body is a `CommandList` concept. To hide it we have to mark it with either the `ImplementationPart` interface or the `ImplementationWithStubPart` interface. They both result in hiding the implementation, however, the latter lets you provide a replacement "stub" concept, that will be inserted instead of the removed implementation. The stub gives you the chance to provide a nicer look of the code with removed implementation and should always be considered for hidden implementation that gets occasionally seen by the end user.
83.2.5 Stub creation

The overhead of `ImplementationWithStubPart` over `ImplementationPart` is in creating the stub concept. For our `CommandList` we’ll need a `StubCommandList` concept and mark it with `IDontSubstituteByDefault` and `IStubForAnotherConcept`. The `IStubForAnotherConcept` is needed since `StubCommandList` inherits the `ImplementationWithStubPart` interface from `CommandList`, and thus MPS needs to be told explicitly that `StubCommandList` is a stub itself and does not need to be stubbed by yet another stub.

If `StubCommandList` was extending `AbstractCommand` instead of `CommandList`, it would not have to be marked as `IStubForAnotherConcept` as `AbstractCommand` is not an `ImplementationWithStubPart`.

The stub editor should make it politely obvious to the reader that the implementation has been removed and is not to be seen here.
Upon rebuild and packaging the language is ready to help the author of RobotRoutines to hide the implementation of her library.

83.2.6 Build script for the library authors

Once the Robot Kaja language supports implementation stripping, the author of the RobotRoutines library can set the strip implementation flag in her build script and thus have the sources of the implementation of her library removed from the generated plugin.
83.2.7 General guidelines and additional notes

1. The marker interfaces are inherited from super-concepts and super-concept-interfaces in a traditional way. In case multiple marker interfaces are applicable to a concept (directly or through inheritance), \texttt{InterfacePart} wins over others and \texttt{ImplementationWithStubPart} wins over \texttt{ImplementationPart}.

2. If none of the marker interface is specified a concept behaves as if the \texttt{InterfacePart} was set on it. Marking a concept with \texttt{InterfacePart} serves two purposes:
   - Documenting the fact that a concept is a necessary public element of your language
   - Preventing the concept from being accidentally marked as part of implementation, if, for example, it inherits any of the other flags.

3. Use the \texttt{ImplementationPart} interface to mark the concepts of your language that need not to be referred from client code nor are directly accessible from \texttt{InterfacePart} concepts through mandatory (cardinality 1 and 1..n) links. Sources of these concepts will be removed from the solution during build and so the user will not be able to see their definitions, for example using \texttt{Go To Concept Declaration}.

4. Use \texttt{ImplementationWithStubPart} interface to mark the concepts that should be removed just like with \texttt{ImplementationPart}, but which need to be replaced with a place-holder instead of being simply removed from the sources, because the user may get to see them as part of definition of an \texttt{InterfacePart} concept.

5. \texttt{ImplementationWithStubPart} is typically needed for concepts that represent children or target of references of cardinality 1 and 1..n pointing from \texttt{InterfacePart} concepts, because their containing links cannot remain empty and would report a validation error.

6. Children and targets of references with cardinality of 0..1 and 0..n can safely be marked as \texttt{ImplementationPart}. The containing links will remain empty.

7. Stubs should follow the naming convention \texttt{Stub + name of the concept being replaced} and must be located in the same package with the replaced concept.

8. Stubs concepts could implement \texttt{ISuppressErrors} to avoid type-system errors being reported from their child nodes.

9. Stubs should also implement \texttt{IDontSubstituteByDefault} so that they are not offered in code-completion menu.
10. MPS will report a warning if a concept is both `InterfacePart` and `Implementation(WithStub)Part`.

11. MPS will report an error if a concept declares or inherits the `ImplementationWithStubPart` interface and no suitable stub concept can be found in the same virtual package.

12. Stubs that inherit the `ImplementationWithStubPart` interface, perhaps by extending the stubbed concept, need to implement `IStubForAnotherConcept` to indicate that they are stubs and thus do not need to be stubbed themselves. It is a good strategy to have all stubs implement the `IStubForAnotherConcept` interface.

13. Stubs cannot be defined to replace abstract super-concepts. They must always replace all concrete concept individually, one stub for one concrete concept implementing `ImplementationWithStubPart`.

14. Redefine the editors of the stub concepts so that they show some clear messages, aka "compiled code" to clearly indicate to the reader that the implementation has been removed.

15. To avoid model validation errors that the users of stripped languages would see in stubbed concepts, consider "specializing" of all children and references with "at least 1" cardinality inherited into the stub concepts from their super-concepts.

This is needed for concepts, such as the one below - the `modelAccessor` child has cardinality 1 and so is mandatory in the stubbed concept.

```
$StubCellModel_ReadOnlyModelAccessor | CellModel_ReadOnlyModelAccessor

concept CellModel_ReadOnlyModelAccessor extends CellModel_AbstractLabel
    implements ImplementationWithStubPart

    instance can be root: false
    alias: read only model access
    short description: reference to read only accessor

    properties:
        << ... >>

    children:
        modelAccessor: ReadOnlyModelAccessor[1]

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

The stub concept has to follow the naming convention of prepending `Stub` to the stubbed concept’s name. It may or may not need to extend the stubbed concept - this depends on how the original concept is being referred to from the rest of the language. If the stub concept has to extend the stubbed concept, it also needs to somehow treat the mandatory child so that the child reference does not stay empty.

The suggested way is to specialize the child or reference relationship and change the target concept to `BaseConcept`.

```
$StubCellModel_ReadOnlyModelAccessor | CellModel_ReadOnlyModelAccessor

concept StubCellModel_ReadOnlyModelAccessor extends CellModel_ReadOnlyModelAccessor
    implements IDontSubstituteByDefault
        ISuppressErrors
        IStubForAnotherConcept

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

    properties:
        << ... >>

    children:
        modelAccessor: BaseConcept[1] specializes: modelAccessor

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

In the behavior aspect in the constructor we then set the reference to point to some dummy node, such as an `IntegerConstant`. This will ensure the link does not stay empty, yet we avoid the need of creating nodes fully satisfying the original `modelAccessor` link in the stubbed concept.
concept behavior StubCellModel_ReadOnlyModelAccessor {

    constructor {
        this.modelAccessor.set(IntegerConstant(value: 0));
    }

    <<concept methods>>
}
Part VIII

Miscellaneous
Chapter 84

FAQ

84.1 DSLs, LOP and Programming

Here you can find answers to the most frequent questions regarding MPS.

84.1.1 What are DSLs? How are they different from "real" programming languages?

A DSL is a language optimized for a specific class of problems. It is usually less complex than a general-purpose language, such as Java, C or Ruby. A DSL may not even be Turing-complete and only state facts about the domain of interest. Usually DSLs are developed in close coordination with the people, who are experts in the field for which the DSL is designed. In many cases, DSLs are intended to be used not by software people, but instead by non-programmers, who are fluent in the domain the DSL addresses. This requires the language's notation and tool support to be optimized for non-programmers, for example, by using mathematical symbols, a mix of textual and graphical notations or a simplified IDE that does not expose DSL users to the full complexity of an IDEA or Eclipse.

84.1.2 What are the benefits of DSLs? Why should I care?

Using DSLs can reap a multitude of benefits. The most obvious benefit of using DSLs is that - once you've got a language and a transformation engine - your work in the particular aspect of software development covered by the DSL becomes much more efficient, simply because you don't have to do the grunt work manually. This is most obvious if you generate a whole truck load of code from a relatively small DSL program. If you are generating source code from your DSL program (as opposed to interpreting it) you can use nice, domain-specific abstractions without paying any runtime overhead, because the generator, just like a compiler, can remove the abstractions and generate efficient code. If you have a way of expressing domain concerns in a language that is closely aligned with the domain, your thinking becomes clearer because the code you write is not cluttered by implementation details. In other words: using DSLs allows you to separate essential from incidental complexity. DSLs, whose domain, abstractions and notations are closely aligned with how domain experts (i.e., non-programmers) express themselves, allow for very good integration between the techies and the domain people. Using DSLs and an execution engine makes the application logic expressed in the DSL code independent of the target platform. Using DSLs can increase the quality of the created product: fewer bugs, better architectural conformance, increased maintainability. This is the result of the removal of (unnecessary) degrees of freedom, the avoidance of duplication in code and the automation of repetitive work. In contrast to libraries and frameworks, DSLs can come with tools, i.e. IDEs, that are aware of the language. This can result in a much improved user experience. Code completion, visualizations, debuggers, simulators and all kinds of other niceties can be provided.

84.1.3 How do DSLs and regular Code fit together?

There are two fundamentally different ways of how traditional code and DSL code can be integrated. The first one keeps DSL code and regular code in separate files. The DSL code is then transformed into programming language code by a code generator, or alternatively the program loads the domain-specific code and executes it. This first approach, with seperated General Purpose Language (GPL) and DSL code is termed external DSLs. Think of SQL as an example of an external DSL. An alternative approach mixes DSL code and general-purpose code in the same program file, leading to a much tighter integration between DSL code and programming language code. The DSL reused the grammar and the parser of the GPL and exploits available extension options of the host language. It is worth to mention that some GPL are more suitable for extension than others.

Both approaches can make sense, depending on the circumstances, and MPS supports both. Traditionally, DSLs have been embedded in programming language code by using the meta programming facilities of the host language. The DSL’s sturcture and syntax was defined by writing code in the language into which the DSL code were to be embedded. Usually the IDE didn't know about the DSL and hence did not provide support (code completion, custom error checking, etc.). With MPS, you use the MPS framework with its specialized DSLs for language development to
define language extensions. The IDE knows about them, so the system can provide full IDE support for the domain-specific embedded languages.

84.1.4 What is Language Oriented Programming?

The term language oriented programming has been coined by Sergey Dmitriev, the CEO of JetBrains and "father" of MPS in a 2004 article called Language Oriented Programming: The Next Programming Paradigm. Other people have come up with related approaches, usually under different names; a primary example is Charles Simonyi and his Intentional Programming approach, and Martin Fowler has described the approach in his 2005 article Language Workbenches: The Killer-App for Domain Specific Languages?

The core idea is that we don’t just use one language when developing software, but rather use those languages that fit each of the tasks best. In contrast to polyglot programming, which on the surface advocates a similar approach, language oriented programming explicitly encourages developers to build their own DSLs, or to extend existing languages with domain-specific concepts as part of the approach. Developing a new language should become an integral part of software development and not left to the ÄIbERGEeks. To make this feasible, languages workbenches such as MPS are an important ingredient of the language oriented approach.

84.1.5 Why do I want to extend a language? Aren’t libraries good enough?

There are a couple of differences between a library and a language extension. A language extension can come with its own syntax. With MPS' projectional editor, this syntax can be arbitrary and is not at all limited by the syntax of the extended language. A language extension also comes with its own constraints and type system, so your IDE can report errors statically. More generally, language extensions, in the way MPS supports them, are fully integrated into the IDE: you get code completion, syntax highlighting and refactoring support for your new language constructs. Finally, a language extension is executed by compile-time transformation and translated into the target programming language code, so there is no runtime overhead that reflection or stacks of indirections would impose. This may be important in particular when targeting resource-constrained systems.

84.2 Projectional Editing

84.2.1 What is Projectional Editing?

In parser-based approaches, users use text editors to enter character sequences that represent programs. A parser then checks the program for syntactic correctness and constructs an abstract syntax tree (AST) from the character sequence. The AST contains all the semantic information expressed by the program i.e. keywords and purely syntactic aspects are omitted.

In projectional editors, the process happens the other way round: as a user edits the program, the AST is modified directly. This is similar to the MVC pattern where every editing action triggers a change in the AST. A projection engine then creates some representation of the AST for the user to interact with. This approach is well-known from various graphical editors. When editing a UML diagram, for example, users don’t draw pixels onto a canvas for an "image parser" to read the drawing, parse it and then create the AST. That would be way too limiting on what you can draw so as the engine would understand. Rather, the editor creates and instance of Class as you drag a class from the palette to the canvas. A projection engine renders the diagram, in this case drawing a rectangle for the class. You can then re-arrange visual elements on the screen without changing the meaning of your diagram.

This approach can be generalized to also work with text editors. Every program element is stored as a node with a unique ID (UID) in the AST. References are based on actual pointers (references to UIDs). The AST is actually an ASG, an abstract syntax graph, from the start because cross-references are first-class rather than being resolved after parsing. The program is then persisted to disk as XML, but this process is transparent to the user.

84.2.2 Why use Projectional Editing? Aren’t text editors good enough?

Since no parsing is used in projectional editors, and the mechanism works basically like a graphical editor, notations other than text can be used in the editor. For example, MPS supports tables as well as simple diagrams. Since these non-textual notations are handled the same way as the textual ones (possibly with other input gestures), they can be mixed easily: tables can be embedded into textual source, and textual languages can be used within table cells. Textual notations can also be used inside boxes or as connection labels in diagrams.

After composing separately developed languages, the resulting language may become ambiguous in parser-based systems since most grammar formalisms are not closed under composition. In projectional systems, this cannot happen. Any combination of languages will be syntactically valid (semantics is a different issue). If a composed language would be ambiguous, the user has to make a disambiguating decision as the program is built. For example, in MPS, if in a given location two language concepts are available under the same alias, just typing the alias won’t bind, and the user has to manually decide by picking one alternative from the code completion menu.
84.2.3 What are the key benefits of using projectional editors instead of parsed languages

Projectional editor gives the **Abstract Syntax Tree (AST)** the prominent role in code manipulation. Code is persisted, versioned, refactored and edited in the AST form, which eliminates the need for repetitive text-to-model-and-back transformations. AST is a much richer representation of code, which avoids ambiguities. The AST can be projected on the screen for editing in any way the language designer desires. With projectional editors your languages can:

- allow for non-parseable notations - tables, images, GUI components may become part of the language syntax. The AST can easily accommodate for such language elements and the projectional editor can draw any graphical shapes on the screen, if needed.

- combine multiple languages, potentially developed by different vendors, in a single piece of code. Ambiguities that parsers would have problems to recover from are not an issue for projectional editors.

- handle context-sensitive and positional grammars

- provide form-like notations - typically when targeting non-professional developers and domain experts the preference could be to use languages that offer strict notation with clear indication of the "moving" parts where the user is supposed to provide a value

- offer multiple notations for a single language - users can view and edit the same piece of code using several different editors, each suitable for a different task. Some editors may use graphical or tabular notations, some may show extra information from other sources, some may reduce the amount of visual clutter yet all are tied to the same piece of code (AST). The developer can thus choose the best visualization of the cdd for the given task.

- use diagramming notations in combination with text - think of electrical circuits, for example, modeled as graphical schemes with embedded pieces of code attributed to individual elements of the scheme

84.2.4 If it isn’t text, how does the integration with version control work? Diff, Merge?

Special support is needed for infrastructure integration. Since the concrete syntax is not pure text, a generic persistence format needs to be used. Therefore, special tools need to be provided for diff and merge. MPS provides integration with the usual VCS systems, such as SVN or Git, and handles diff and merge in the tool, using the concrete, projected syntax. Note that since every program element has a UUID, a move can be distinguished from delete/create providing more useable semantics for diff and merge. The picture below shows an example.
84.2.5 How long does it take to get used to the editor?

In early 2010 we performed an experiment as part of a training. Ten people, who had never used MPS before, were taught the principals of MPS development. After two days all of them agreed that, while the editing experience is a bit unusual at the beginning, the editor is in fact not worse at all than text editors, it is just different. After two days, you get used to it and feel no discomfort. In fact, and not surprisingly to us, some of the people claimed they prefer the MPS way of editing things, since in many context you’re selecting/changing/removing elements based on the syntax tree anyway and projectional editors have an edge over text-base IDEs in this regard.

84.3 MPS Specifics

84.3.1 How is MPS licensed?

MPS is licensed under the Apache 2.0 License. Apache 2.0 is very liberal and allows you to use the system in basically any context you like. Companies like e.g. Realaxy have used MPS to build commercial applications.

84.3.2 Which languages are already in MPS so I can extend them?

MPS comes with a language called BaseLanguages, which is essentially Java 6 (with optional extensions for syntax of Java 7 and 8) plus a pile of custom extensions such as closures, collection API, regular expressions, extension methods, tuples and builder support. This language is also used for many aspects of language development in MPS, in which case it uses a couple of additional extensions for working with program trees. MPS also comes with an implementation of XML that can be used and extended. There is also an implementation of C available as an additional base language, see mbeddr.com. Other languages are being gradually developed by third-parties and as open-source projects (e.g. JavaScript). Check out the MPS Languages Repository page for more details.

84.3.3 Is there a language repository somewhere?

Since languages can be packaged and distributed as jar files, nothing prevents languages from being shared the same way libraries are. A handful of language plugins (both by JetBrains and third parties) for MPS as well as for IntelliJ IDEA have been made publicly available at the JetBrains plugin repository:

- MPS Languages Repository page in the documentation
- MPS language plugins site
- IntelliJ IDEA language plugins developed in MPS site

84.3.4 Are there some books on MPS?

Yes, there are:

- Fabien Campagne wrote a reference guide - The MPS Language Workbench Volume I
- Markus Voelter wrote a book on DSL engineering featuring MPS - DSL Engineering
- Numerous academic papers have been published over the past couple of years - check out the MPS publications page

84.3.5 How much effort does it take to add new base languages to MPS?

The time it takes to add new languages obviously depends a lot on the complexity of the language. Small DSLs can be added in a matter of minutes or hours, if you are familiar with MPS. Any language added to MPS can be used as a base language and can be easily extended by you or others. But that’s probably not what you wanted to know :) Adding *real* programming languages as new base languages is a little bit more work, because general-purpose languages are typically rather large (with the exception of languages like Lisp, which would be trivial to add :-)). A good benchmark is the C base language recently implemented by the mbeddr.com folks. They reported 2-3 person month of work to implement all of C in MPS.

84.3.6 If projection is so cool, when can we do notations other than text?

MPS already supports notations that are not possible with parser-based systems. For example, MPS ships with an extension to BaseLanguage that can do fraction bars and big math symbols (such as the big symbol sign). In addition, MPS supports tables, as demonstrated by the decision tables example in the mbeddr project. It is also possible to embed any swing widget into an editor. Support for graphical notations out of the box has been added in the MPS 3.1 release.
84.3.7 Are there any successful projects developed with MPS?
MPS has originally been developed as an in-house project at JetBrains, so there are several use cases within JetBrains. A well-known example is the mbeddr.com project, which provides an extensible C-based IDE for embedded software development. MPS is gradually being adopted by different industries - electrical engineering, insurance, government organizations, to implement powerful DSLs for their respective domains.

84.3.8 How does MPS scale?
The mbeddr folks have done a little load test. They have found that single root elements (the equivalent of files in a classical IDE) can easily go up to 4,000 lines without serious performance issues. They have also measures that a C system with up to 100,000 lines of C code should work fine.

84.3.9 What are the future plans for the product?
For details, take a look at the MPS public roadmap.

84.3.10 MPS is Open Source, but still "owned" by JetBrains. Huh?
Yes, JetBrains is the main contributor to the project providing more than 10 full-time developers. External contributions are welcomed, though, either as ideas shred through the forum, requests and bug reports in the tracker or patches to the code. Regular contributors may become project committers and participate on the project planning an implementation process.

84.3.11 Can I import Java code into MPS?
MPS works hard on making Java interoperability easy. For example, you can use any Java libraries in MPS without problems. You can mix-and-match Java and MPS code on the same project so that your MPS code can see and use your Java code and vice versa - in Java you can use the code generated from MPS. Also, MPS can parse snippets of Java code that you decide to paste into MPS and transform them into valid BaseLanguage code..

84.3.12 Which Java versions are supported?
MPS can run on JDK 1.8 and later. Check out the MPS Java compatibility page for more details.

84.3.13 How about Java IDE Integration?
Currently, MPS is a standalone application, implemented in Java so it runs on all major platforms. In the 3.0 version MPS added support for deploying languages into IntelliJ IDEA. Language *development* is still done with the MPS standalone application, but language *use* is now possible in IntelliJ IDEA. This gives users much better options when integrating MPS-based DSLs into your Java and other applications.

84.3.14 How about Eclipse Integration?
While technically feasible, integration with Eclipse is currently not actively pursued. Depending on the interest among MPS users and their ability to contribute to or sponsor such activity, JetBrains is ready to consider implementing Eclipse integration.

84.3.15 How can I get support?
There is a MPS community forum, where you can get support. An issue tracker is available for submitting and tracking bugs. Last but not the least, you can also get professional support from people, who have real hands-on experience with MPS.

84.3.16 How does MPS compare to other language workbenches?
This is of course hard to say in an objective way from where we stand, but let’s try :-) Unlike most other language workbenches, MPS uses a projectional editor. We’ve already discussed the benefits of this approach in the FAQ above. Also, MPS is the most comprehensive system in that it provides support for language structure, syntax, type system, constraints, refactoring, transformation and code generation all in a single integrated package. As of now, other alternatives such as MetaEdit+ provide better support for graphical syntax, and Xtext provides better Eclipse integration and has a bigger community surrounding it. For more detailed comparison you have to make up your own mind, and consider all facts in the context of your own project’s needs. Take a look at the language workbench competition to learn more. You may also consider checking out the MPS case studies, reading some of the MPS publications page and getting in touch with experienced consultants.
84.3.17 How can I learn about using MPS?

You should take a look at the freshly renovated documentation page. It holds logically organized references to overview articles, tutorials on MPS basic and advanced concepts as well as screencasts and pointers to interesting case studies.
Chapter 85

Copyrights

(C) Copyright JetBrains, s.r.o. 2003, 2008-2017. All Rights Reserved
JetBrains MPS includes software developed by third parties. You can find their licenses in the license directory of the MPS installation.