The MPS Language Workbench

Fabien Campagne

Volume I

http://books.campagnelab.org
Credits


I am particularly grateful to the executives at JetBrains s.r.o who open-sourced an internal research project and made it possible for a growing community of programmers to become more familiar with a truly different and very promising approach to software development. Special Thanks go to Václav Pech and Alexander Shatalin for providing feedback about earlier drafts of this book and offering clarifications when I had trouble figuring out certain less-traveled aspects of MPS.

I need to thank Philippe for his patience during the winter 2013, when I used all these week-ends to write the first volume of this book series. I apologize to my relatives, colleagues and friends if they felt that I have been less available to them during the time it took to complete this book.

Finally, I want to thank Stanislav Povelikin (http://spdesign.org) for turning some of my illustration ideas into beautiful cover designs for this book series.
# Contents

## 1 Introduction .................................................. 15
  1.1 Motivation .................................................. 15
  1.2 Language Workbenches ..................................... 16
  1.3 When Should you Read this Book? ......................... 16

## 2 Getting Started ............................................... 19
  2.1 The Abstract Syntax Tree .................................. 20
  2.2 The Projectional Editor .................................... 21
  2.3 The MPS Language Workbench .............................. 21
  2.4 The Project Tab ............................................. 21
  2.5 Importing Languages into a Project ....................... 23
  2.6 The Migration Assistant .................................... 25
  2.7 Error Highlighting ........................................... 25
  2.8 Meet the Players ............................................. 26
    2.8.1 The Model Properties ................................. 28
  2.9 The Projectional Editor ..................................... 29
    2.9.1 The Editor Window ................................... 30
    2.9.2 Everything you See is a Projection .................. 30
    2.9.3 Executing the AST Root ............................... 31
    2.9.4 Code Generation ...................................... 32
  2.10 The Intention Menu ........................................ 33
  2.11 The Inspector Tab ......................................... 33
# 3 The Structure Aspect

## 3.1 Overview

## 3.2 Concepts

- **3.2.1** The extends Clause
- **3.2.2** The implements Clause
- **3.2.3** Instance can be root
- **3.2.4** The Concept’s alias
- **3.2.5** The Concept’s short description
- **3.2.6** Properties
- **3.2.7** Children
- **3.2.8** References
- **3.2.9** Inspector Tab: Help and Icon
- **3.2.10** Abstract Concepts
- **3.2.11** Smart Reference

## 3.3 Interface Concepts

- **3.3.1** The extends clause
- **3.3.2** Other attributes
- **3.3.3** Marker Interface

## 3.4 Enumeration Types

- **3.4.1** Member Type
- **3.4.2** No Default/Default
- **3.4.3** Member Identifier
- **3.4.4** Enumeration Elements

## 3.5 Constrained Data Types

- **3.5.1** Matching Regexp

## 3.6 Primitive Data Types

## 3.7 Annotation Link

# 4 Structure In Practice

## 4.1 Overview

## 4.2 A Classical Data Management Problem

## 4.3 Designing the Bibliography Language

# 5 The Editor Aspect

## 5.1 Overview
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2 Concept Editor</td>
<td>52</td>
</tr>
<tr>
<td>5.2.1 Context Hint</td>
<td>52</td>
</tr>
<tr>
<td>5.2.2 Editor For Concept</td>
<td>53</td>
</tr>
<tr>
<td>5.2.3 Node Cell Layout</td>
<td>53</td>
</tr>
<tr>
<td>5.2.4 Inspected Cell Layout</td>
<td>53</td>
</tr>
<tr>
<td>5.3 The Editor Language</td>
<td>53</td>
</tr>
<tr>
<td>5.3.1 Cell Action Map</td>
<td>54</td>
</tr>
<tr>
<td>5.3.2 Cell Key Map</td>
<td>55</td>
</tr>
<tr>
<td>5.3.3 Element Menu</td>
<td>55</td>
</tr>
<tr>
<td>5.3.4 Collection Cells</td>
<td>56</td>
</tr>
<tr>
<td>5.3.5 Constant Cells</td>
<td>56</td>
</tr>
<tr>
<td>5.3.6 Property Cells</td>
<td>56</td>
</tr>
<tr>
<td>5.3.7 Child Cell</td>
<td>57</td>
</tr>
<tr>
<td>5.3.8 Referent Cell</td>
<td>58</td>
</tr>
<tr>
<td>5.3.9 Swing Component Cell</td>
<td>58</td>
</tr>
<tr>
<td>5.3.10 Image Cell</td>
<td>59</td>
</tr>
<tr>
<td>5.3.11 Editor Component Cell</td>
<td>59</td>
</tr>
<tr>
<td>5.3.12 Model Access Cell</td>
<td>59</td>
</tr>
<tr>
<td>5.3.13 Indent Cell</td>
<td>60</td>
</tr>
<tr>
<td>5.3.14 Custom Cell</td>
<td>60</td>
</tr>
<tr>
<td>5.3.15 Next Applicable Cell</td>
<td>61</td>
</tr>
<tr>
<td>5.4 Concept Editor Context Hints</td>
<td>61</td>
</tr>
<tr>
<td>5.5 Editor Component</td>
<td>62</td>
</tr>
<tr>
<td>5.5.1 Overrides</td>
<td>62</td>
</tr>
<tr>
<td>5.5.2 Applicable Concept</td>
<td>63</td>
</tr>
<tr>
<td>5.5.3 Component Cell Layout</td>
<td>63</td>
</tr>
<tr>
<td>5.6 Editor Styles</td>
<td>63</td>
</tr>
<tr>
<td>5.6.1 Base Style</td>
<td>63</td>
</tr>
<tr>
<td>5.6.2 Overriding Style Attributes</td>
<td>64</td>
</tr>
<tr>
<td>5.7 Style Sheets</td>
<td>64</td>
</tr>
<tr>
<td>5.7.1 Style</td>
<td>64</td>
</tr>
<tr>
<td>5.8 Style Key Pack</td>
<td>66</td>
</tr>
<tr>
<td>5.9 Menu Component</td>
<td>66</td>
</tr>
<tr>
<td>5.9.1 Applicable To</td>
<td>66</td>
</tr>
<tr>
<td>5.9.2 Menu Parts</td>
<td>66</td>
</tr>
<tr>
<td>6 The Editor in Practice</td>
<td>71</td>
</tr>
<tr>
<td>6.1 Editor for the Person Concept</td>
<td>71</td>
</tr>
<tr>
<td>6.2 Authors Collection</td>
<td>73</td>
</tr>
</tbody>
</table>
## 13 The TextGen Aspect

13.1 Overview ................................................................. 147
13.2 Understanding Text Generation ................................. 147
13.3 ConceptTextGenDeclaration ................................. 148
  13.3.1 Use textgen of ancestor .................................... 149
13.4 LanguageTextGenDeclaration ............................... 151
  13.4.1 Operations ...................................................... 151

## 14 TextGen In Practice

14.1 Overview ................................................................. 153
14.2 The TextGen Aspect ................................................. 153
14.3 The Concept TextGen ............................................ 153
14.4 The Language TextGen ............................................ 153
  14.4.1 A Node for Each Language Concept ...................... 154
  14.4.2 Bibliography Operation Implementation .................. 154
14.5 Generating Author Lists ....................................... 157

## 15 The Generator Aspect

15.1 Overview ................................................................. 159
15.2 Understanding Model to Model Transformations .......... 159
15.3 The Generation Context ........................................... 160
15.4 The Template Macro Language ................................... 161
  15.4.1 Property Macros ............................................... 162
  15.4.2 Reference Macros ............................................. 162
  15.4.3 Node Macros ................................................... 163
15.5 The Generator Root Nodes ....................................... 166
15.6 The Generator AST Root Nodes ................................. 166
15.7 Ordering Mapping Configurations ............................. 166
  15.7.1 Mapping Configuration Parameters ......................... 168
  15.7.2 Other Generator AST Root Nodes ............................ 169
  15.7.3 Template Declaration ........................................ 169
  15.7.4 Template Switch .............................................. 170
  15.7.5 Mapping Script ................................................ 172
  15.7.6 Generator Descriptor ......................................... 173
  15.7.7 Root Nodes from Other Languages ........................ 173
15.8 Mapping Configuration

15.8.1 Mapping Labels .......................... 173
15.8.2 Exports .................................. 174
15.8.3 Parameters ............................... 175
15.8.4 Is Applicable ............................. 175
15.8.5 Conditional Root Rules ................. 176
15.8.6 Root Mapping Rules ..................... 176
15.8.7 Weaving Rules ............................ 177
15.8.8 Reduction Rules .......................... 179
15.8.9 Drop Attributes ........................... 179
15.8.10 Pattern Rules ............................ 180
15.8.11 Abandon Rules ........................... 180
15.8.12 Pre-Processing Scripts ................. 180
15.8.13 Post-Processing Scripts ............... 180

15.9 Troubleshooting Generation .......... 181

16 Generator in Practice ....................... 183

16.1 Overview ................................ 183
16.2 Dependencies .............................. 183
16.3 The XML Output Node ..................... 183
16.4 The Mapping Configuration ............. 184
16.4.1 Changing the XML filename .......... 184
16.5 XML elements ............................. 185
16.6 The BiblioElements ........................ 186

17 Other Techniques ........................... 189

17.1 Annotations ............................... 189
17.1.1 Define the Annotation Concept ....... 189
17.1.2 Define an Intention To Toggle the Annotation ........ 191
17.1.3 Define an Annotation Editor ........... 191

17.2 Concept Functions ........................ 191
17.2.1 Function Concept ....................... 191
17.2.2 Function Parameters .................... 192
17.2.3 Function Return Type ................... 193

17.3 Importing JAR Files ...................... 194
17.3.1 First Class MPS Models ................. 194
17.3.2 Libraries ................................ 197
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.4 Path Variables</td>
<td>198</td>
</tr>
<tr>
<td>17.5 Generate Executable Concepts</td>
<td>199</td>
</tr>
<tr>
<td>17.6 Navigate to a Node</td>
<td>199</td>
</tr>
<tr>
<td>17.7 Generic View of Children and References of a Concept</td>
<td>200</td>
</tr>
<tr>
<td>17.8 Undo/Redo Support and Access to Nodes</td>
<td>200</td>
</tr>
<tr>
<td>17.9 Retrieving the Cursor Position</td>
<td>201</td>
</tr>
<tr>
<td>17.10 Migrating to a new version of MPS</td>
<td>201</td>
</tr>
<tr>
<td>17.10.1 Migration from 3.1 to 3.2</td>
<td>202</td>
</tr>
<tr>
<td>17.10.2 Migration from 3.2 to 3.3</td>
<td>203</td>
</tr>
<tr>
<td>18.1 Understanding Diagram Editors</td>
<td>205</td>
</tr>
<tr>
<td>18.1.1 Figure Implementation</td>
<td>205</td>
</tr>
<tr>
<td>18.2 Diagram Cell</td>
<td>206</td>
</tr>
<tr>
<td>18.2.1 Content</td>
<td>207</td>
</tr>
<tr>
<td>18.2.2 Elements Creation</td>
<td>208</td>
</tr>
<tr>
<td>18.2.3 Connector Creation</td>
<td>209</td>
</tr>
<tr>
<td>18.2.4 Palette Declaration</td>
<td>210</td>
</tr>
<tr>
<td>18.3 Diagram Node Cell</td>
<td>211</td>
</tr>
<tr>
<td>18.4 Diagram Connector Cell</td>
<td>212</td>
</tr>
<tr>
<td>18.5 Diagram Port Cell</td>
<td>212</td>
</tr>
<tr>
<td>18.6 Custom Figure Implementations</td>
<td>213</td>
</tr>
<tr>
<td>18.6.1 Creating a Figure Implementation</td>
<td>213</td>
</tr>
<tr>
<td>18.6.2 Creating an ExternalViewFigure</td>
<td>214</td>
</tr>
<tr>
<td>19.1 The Project Diagram Editor</td>
<td>218</td>
</tr>
<tr>
<td>19.2 Rendering Nodes</td>
<td>219</td>
</tr>
<tr>
<td>19.2.1 Import the Figures Language</td>
<td>219</td>
</tr>
<tr>
<td>19.2.2 Define the NodeRendering Editor</td>
<td>219</td>
</tr>
<tr>
<td>19.2.3 Define Create Element</td>
<td>220</td>
</tr>
<tr>
<td>19.2.4 Rebuild the Language</td>
<td>221</td>
</tr>
<tr>
<td>19.3 Defining Input and Output Ports</td>
<td>221</td>
</tr>
<tr>
<td>19.3.1 Anonymous Ports</td>
<td>221</td>
</tr>
<tr>
<td>19.3.2 As Children of NodeRendering</td>
<td>222</td>
</tr>
</tbody>
</table>
19.4 Rendering Edges

19.4.1 The EdgeRendering Concept ........................................ 223
19.4.2 Configuring the Connector Cell ........................................ 223
19.4.3 Second Build ............................................................ 223
19.4.4 Connector Creation Behavior ........................................... 224
19.4.5 Final Diagram ............................................................ 224

20 MPS Key Map .............................................................. 227

List of Figures ............................................................... 235

Bibliography ................................................................. 243

Index ................................................................. 247
1.1 Motivation

After contributing three text-book chapters and knowing the effort involved in drafting and finishing each of them, I thought long and hard before deciding to embark on a book project. My key motivation for writing this book was that I needed a reference text that would make it easier to teach others how to use the Meta Programming System (MPS).

MPS represents a new paradigm to programming and software design. Intentional programming [Sim95] and meta programming [Dmi04], were conceived to overcome some of the challenges of traditional language design and development approaches.

I first came across MPS in the fall of 2006 when I downloaded one of the first Early Access Program (EAP) releases¹. The description of the project suggested that the tool could help solve a number of problems I was encountering in the field of bioinformatics and this potential sparked my interest². I quickly realized that the version of MPS I experimented with was not for the faint of heart. There was very limited documentation at the time and the only tutorial available with the EAP appeared incomplete. While I spent some time trying to understand this version of MPS, I had to give up when difficulties arose that I could not resolve³ and when other priorities surfaced. I would have liked then, to find a resource such as I hope this book will be for newcomers to MPS. Of course, the MPS documentation has greatly improved since the version I tested in 2006. I am writing these lines in the winter of 2013. Despite these improvements, I think that there is still a need for a resource that will help beginners become familiar with MPS in a short amount of time. I hope this book will fill this gap and provide both a reference and a gentle introduction to a very unique and promising tool.

---
¹See http://icb.med.cornell.edu/wiki/index.php/Language_Oriented_Programming
²More information about my research in bioinformatics can be found on the laboratory web site: http://campagnelab.org
³See http://icb.med.cornell.edu/wiki/index.php/MPS/Examples/TextLanguage
1.2 Language Workbenches

Language Workbenches are software engineering tools developed in the last ten years that help their users create new languages and the tools to write programs in these languages.

**Definition 1.2.1 — Language Workbench (LW).** A Language Workbench is a software tool designed to help its users develop new computer languages.

Some LWs focus on non-technical users and provide languages to help organize knowledge. Intentional Software (http://www.intentsoft.com/) is a commercial Language Workbench that falls in this category [Sim95; SCC06]. Another prominent language workbench aimed at software programmers is the JetBrains Meta Programming System (MPS) [Dmi04; Voe13; VS10]. MPS is developed as an open-source project (source code is available at https://github.com/JetBrains/MPS). This workbench was developed to provide the tools necessary to design new programming languages.

1.3 When Should you Read this Book?

If you are reading this book, you are either taking one of my courses or more likely are trying to use MPS to help with software engineering or programming problems that you encounter in your hobbies or professional life. MPS offers a different paradigm from the traditional programming tools that you may be used to and this section aims to explain what type of problems MPS can help you with.

You will find that MPS is a useful tool if your project requires that you do one or more of the following:

1. Develop a Domain Specific Language (DSL),

   **Definition 1.3.1 — Domain Specific Language (DSL).** A Domain Specific Language is a language designed to meet the needs of a specific application domain [For+04]. Examples of widely used DSLs include HTML and SQL.

2. Develop an Application Specific Language (ASL),

   **Definition 1.3.2 — Application Specific Language (ASL).** An Application Specific Language is a language designed to meet the needs of a specific application program. An example is the PlantUML language used to describe UML diagrams so that the PlantUML program can render them. Some well-designed ASL which are reused by several programs have evolved towards DSLs. An example is the DOT language, initially used for input to GraphViz, which has evolved towards a DSL for representing graphs and is supported by several programs.

3. Need to automate the generation of source code,
4. Need to automate the generation of configuration files,
5. Need to develop a software framework for a specific application domain. Instead of developing a framework, consider modeling the domain as a set of languages in MPS that generate to the code that you want the framework for.
1.3 When Should you Read this Book?

**Definition 1.3.3 — Software Framework.** A Software framework is an abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software. [Gac+04]

6. Need to write long boiler-plate code sections as part of usual development that are very repetitive. If you find that the interesting part of your programs are buried among a lot of implementation code, you should consider modeling the interesting part of the program with MPS and generate the implementation part automatically from the short language description.

7. For data analysis. Since the publication of the first edition of this book, my laboratory has shown that the MPS LW can be applied to facilitate data analysis by non-specialists. If data analysis is an area of interest, you can learn more about this application in these publications and pre-prints: [BC15; CDS15; KSC16]

You can find detailed discussions of the activities and the motivation for using a Language Workbench in the book “DSL Engineering” by Markus Voelter and colleagues [Voe+13]. The book that you are reading now is concerned with teaching you how to perform these activities and helping you quickly become productive with the MPS Language Workbench.
Traditional compiler technology relies on lexers and parsers to read programs expressed as text files [Lam+06] and transform them into data structures called Abstract Syntax Trees (AST). This process is illustrated on Figure 2.1, top panel.

Figure 2.1: **Two Paradigms.** This figure compares the MPS programming paradigm to a more traditional programming approach. The top panel shows how programmers develop programs with a traditional language environment consisting of a text editor and compiler. The bottom panel shows how a programmer interacts directly with the MPS system.

ASTs are the data structures that a compiler processes to generate executable code. In contrast, in the paradigm used by the Meta Programming System, the user interacts directly with one or several ASTs using a projectional editor. Figure 2.2 presents the relationships between the text representation of an arithmetic expression and a corresponding AST. The MPS approach works directly with the AST, without the need to express programs as text. This brings several advantages:
1. Language extensions are simple to develop. Extending a lexer and parser for a complicated language requires specialized skills, while extending a language in MPS consists in defining new AST concepts, with their associated editor(s) and semantic. Extending a language consists in describing its structure and creating an editor for the new language concepts. This process is much simpler than that needed to extend a compiler for an existing language because, the larger the language, the more difficult it becomes to develop a text syntax that avoids ambiguities.

2. Different languages can be composed effectively without the risk of introducing ambiguities in the concrete syntax.

2.1 The Abstract Syntax Tree

**Definition 2.1.1 — Abstract Syntax Tree (AST).** An abstract syntax tree (AST) is a data structure traditionally used by compilers to represent and manipulate programs. An AST starts with a root node. The root node can contain child nodes. Nodes in the tree can be of different types, often arranged in one or more concept hierarchies. Figure 2.2 presents an example of an AST.

![Abstract Syntax Tree](image)

**Figure 2.2: Understanding the AST.** This figure presents the relationship between a program text view for an arithmetic expression (left), the corresponding Abstract Syntax Tree (middle) and the associated fragment of the node concept hierarchy (right). The expression shown adds three integer constants \((1 + 2 + 3)\). The root of the AST is a BinaryOperator node (operator: +). Lines connecting nodes of the AST indicate aggregation relationships. For instance, the root note has a left child of type BinaryOperator, and a right child of type IntegerConstant, with value 3.

**Definition 2.1.2 — AST Concept Hierarchy.** The AST Concept Hierarchy describes types of the nodes that make up possible ASTs, and their relationships to other AST nodes concepts. The AST concept hierarchy is analogous to a data schema since its structure defines which ASTs can be constructed.

In the MPS tool, the AST Concept Hierarchy can be defined in the Structure Aspect, which...
2.2 The Projectional Editor

Definition 2.2.1 — Projectional Editor (PE). A projectional editor is a user interface that makes it possible to create, edit and interact with one or more ASTs.

Good user interfaces are usually difficult to develop, but MPS makes it easy to create a robust projectional editor for new languages. This book describes how to develop a projectional editor in Chapter 5.

2.3 The MPS Language Workbench

Figure 2.3 presents a snapshot of the MPS workbench. This snapshot should help you orient yourself around the MPS user interface. In the next section, we will look at the Project Tab in detail.

2.4 The Project Tab

A key user interface element of MPS is the Project Tab. This tab gives you access to a logical organization of the languages included or imported in a project. When working with MPS,
Figure 2.4: The Project Tab Explained.

**Solutions and Models.** Orange icons with a S inside them denote Solutions. The name of the Solution follows the orange square. Solutions contain models, which themselves contain ASTs.

**Language and Language Aspects.** The yellow square icons with an L in them represent languages. The name of the language is shown to the right of the icon. Solution and language icons are prefixed with a little triangle. Click on the triangle to open the solution or language. Click again to collapse the solution or language. Opening a solution presents the models in the solution (ExecuteCommand is a model in the `org.campagnelab.NYoSh.sample` solution). In contrast to solutions, opening a language exposes a list of language aspects. Several aspects will be described in detail in the following chapters. See Table 2.1 for a brief description of the function of these aspects.

**Modules Pool.** Shown in green at the bottom is the Modules Pool. Note the other Languages and Solutions shown under Modules Pool are not part of the project, but provided by the platform or by some active MPS plugin. It is a great place to look for languages that you might wish to use in your own languages or solutions.
you will often use the Project Tab to modify properties of languages or solutions that make up an MPS project. Let’s take a moment to become familiar with this MPS user interface element. A blown-up Project Tab is shown in Figure 2.4 to show details and explain the meaning of the icons included in the tab.

Developing a new language in MPS consists in adding and configuring nodes in these various language aspects. Note that some aspects are not created by default when creating a new language. You can add new aspects to a language by right clicking on a language name and selecting `New * Aspect` (see Figure 2.5).

![Figure 2.5: Create New Aspect. Some language aspects are not created by default for new languages and can be created as shown in this snapshot.](image)

The function of the various MPS languages aspects shown in the Project Tab are explained in Table 2.1.

It is useful to realize that MPS aspects are implemented as MPS models, which contain ASTs and are created and edited with the same mechanisms that are available for languages developed with MPS. For instance, the structure aspect for the language `org.campagnelab.Swift` is implemented as a model stored under `languages/org.campagnelab.Swift/languageModels/structure.mps` and contains ASTs expressed in the `jetbrains.mps.lang.structure` language.

### 2.5 Importing Languages into a Project

When you create an empty MPS project, you might find it useful to import other languages into the project alongside the languages that you are developing. For instance, you could open the sample projects shipped with MPS as standalone projects, but it is often more convenient to import some languages from these projects into your own project. You then become able to switch back and forth between languages in the same workench window and are able to copy and paste across languages.

Let’s assume you created an empty MPS project (or a project with one solution and/or language). You can import languages into this project by right clicking on the name of the project, at the very top of the Project pane and selecting ‘Project Paths’. This opens the
### Language Aspect

<table>
<thead>
<tr>
<th>Language Aspect</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Structure</strong></td>
<td>Defines the AST Node Concept Hierarchies.</td>
</tr>
<tr>
<td><strong>Editor</strong></td>
<td>Define mini-editors for each concept of the structure aspect.</td>
</tr>
<tr>
<td><strong>Actions</strong></td>
<td>Provide means to customize the editing behavior of the editor.</td>
</tr>
<tr>
<td><strong>Constraints</strong></td>
<td>Provides means to restrict which AST nodes can be child or parents or other nodes and help define scopes.</td>
</tr>
<tr>
<td><strong>Behavior</strong></td>
<td>Provides method declarations and definitions for concepts of the structure aspect.</td>
</tr>
<tr>
<td><strong>Typesystem</strong></td>
<td>Makes it possible to implement a typesystem for the language.</td>
</tr>
<tr>
<td><strong>Migration</strong></td>
<td>Holds Migration classes to evolve the language during its life-time (MPS 3.2+, see Volume II [Cam15]).</td>
</tr>
<tr>
<td><strong>Scripts</strong></td>
<td>Holds Language Migration Scripts to evolve the language during its life-time (older mechanism: MPS 3.1 and earlier, see Volume II).</td>
</tr>
<tr>
<td><strong>Intentions</strong></td>
<td>Defines language intentions: context-dependent AST transformations that the user can activate on demand.</td>
</tr>
<tr>
<td><strong>DataFlow</strong></td>
<td>Defines the language data flow rules which govern which statements are reachable, which variables are read from or written to.</td>
</tr>
<tr>
<td><strong>Test</strong></td>
<td>Provides the means to test languages by developing unit tests to check type calculations and error detection.</td>
</tr>
<tr>
<td><strong>Refactorings</strong></td>
<td>Provides the means to offer refactoring tools for the language to end-users [Ele+01].</td>
</tr>
<tr>
<td><strong>TextGen</strong></td>
<td>Provides means to transform AST nodes to text.</td>
</tr>
<tr>
<td><strong>Generator</strong></td>
<td>Provides means to transform AST nodes to other languages.</td>
</tr>
<tr>
<td><strong>Plugin</strong></td>
<td>Holds code that when compiled integrates with MPS to extend its user interface and functionality (see Volume II).</td>
</tr>
</tbody>
</table>

Table 2.1: Function of the most common MPS Language Aspects.

---

**Figure 2.6:** Adjusting Project Paths. You can import new languages and solutions into a project by following these instructions. Notice the names of the user interface dialogs as we will refer to them by name in the future.
dialog shown in Figure 2.6. Try it now by creating a new project, and adding the Complex language to your new project\(^1\).

This chapter is organized in such a way that the tutorial triggers a number of situations that you may encounter with MPS, and give you the tools to resolve the problems that arise in these situations. For instance, I will show you how to create an empty project and add languages to this project. It would be much simpler to tell you to open the sample project directly in MPS, but I think you can figure this out on your own. The tutorial instead shows how to import languages into an empty project to help you understand different concepts that you will need to work productively with MPS. Equipped with this understanding, you will be free to use the easier path, and will know what to do when you run into problems.

### 2.6 The Migration Assistant

When opening a project with MPS, you may be prompted to run the Migration Assistant and see a dialog like the following:

![Migration Assistant Dialog]

This will happen for instance if the version of MPS that you are using is more recent that the version used to save the project.

The migration assistant will fail to run if the project has any missing dependencies, so it is best to postpone running the assistant until you are sure the project includes all dependencies. If the migration assistant is triggered at this stage of the tutorial, click the Postpone button.

When you have added all dependencies to the project, you can restart the migration assistant manually with the menu [Migration] > Run Migration Assistant. The migration assistant will run as needed any time you open a project.

### 2.7 Error Highlighting

After adding the language following the instructions of Figure 2.6, you should notice that the jetbrains.mps.samples.complex language is highlighted in red. MPS uses this cue to indicate an error in the language. See Figure 2.7 to learn how to find the cause of this error.

\(^1\)The complex language should be located under ~/.MPSSamples.3.0/complexLanguage/languages/complex/jetbrains.mps.samples.complex.mpl, where ~/. indicates your home directory.
Figure 2.7: An Example of Error Message. Browsing over the name of the language highlighted in red reveals the error message. In this case, MPS indicates that the runtime solution associated with the complex sample language cannot be found in the project. To fix this error, you can add the complex runtime solution to the project following the approach you used to add the complex language.

The complex solution should be located under `~/MPSSamples.3.3/complexLanguage/solutions/jetbrains.mps.complex.runtime/jetbrains.mps.samples.complex.runtime.msd`, where `~/` indicates your home directory. Note that the directory includes the version number for MPS, which may be different in your installation.

Once you have followed similar steps to those shown in Figure 2.6 and added the runtime solution, you should notice that the error has been resolved. Dependencies encoded in languages and solutions must be resolved explicitly within a project for solutions and languages that are not available under Modules Pool.

Modules pool contain languages shipped with the MPS platform or bundled with plugins that are active in the MPS workbench. You can inspect the plugins active in your session by opening `MPS Preferences` and locating the Plugins section. MPS can download plugins through a remote repository, or you can install plugins manually by installing plugin distributions under the plugins directory of the MPS distribution. Follow the installation instructions given with each plugin.

### 2.8 Meet the Players

An MPS project can contain a number of parts, organized in a hierarchical fashion. Typical parts are shown below, organized starting with Languages and Solutions:

- Languages
  - Structure Aspect
    - Concepts
    - Concept Interfaces
    - Enumeration Types
    - Constrained Data Types
    - Primitive Data Types
  - Other Aspects [See Aspect Chapters for description of parts]
Solutions

- Models
  - AST Roots
  - AST Nodes

In this hierarchical structure, parts are organized from parent to descendants, where descendants are contained in a parent. For instance, Solutions contain Models that contain AST Roots, which contain AST Nodes and their descendants.

MPS Project parts can also be connected across this hierarchy. For instance, Solutions contain models described in specific Languages.

**Definition 2.8.1 — Language.** An MPS Language describes what types of ASTs can be created with the language. A language includes a set of AST Node Concepts, semantic and behavior, represented in one or more MPS Aspects. A language is a kind of MPS Module.

**Definition 2.8.2 — Solution.** An MPS Solution holds a set of models. Solutions are convenient packaging units that make it possible to reference a set of models as a unit from other Solutions or Languages. A solution is a kind of MPS Module.

**Definition 2.8.3 — Model.** In MPS, a Model is a container for a set of AST Roots. Models are serialized to disk in XML (.mps extension) or binary format (extension .mpb).

**Definition 2.8.4 — AST Root.** An AST root is an AST node that can be used as an immediate descendant of a Model. AST Roots are analogous to source files in a traditional programming paradigm, but different because AST Roots are serialized to disk when saved and deserialized when loaded.

The specific set of languages associated with a given Solution is recorded in a dimension orthogonal to the part hierarchy. Each solution includes references to the set of languages needed to write ASTs in the Solution.

**R** Remember that MPS Projects are organized as a large AST (whose root node is the project node). Links between Solutions and Languages are an example of reference across an AST. AST References will be described in Chapter 3.

**Create a New Project**

Follow the instructions on Figure 2.8 to create a new project with one solution.

Now, import the MPS Sample `math` language into the new project. We described how to import languages in a project in Section 2.5. You will need to import the complete set of dependencies shown below to resolve errors:

1. jetbrains.mps.baseLanguage.math
2. jetbrains.mps.baseLanguage.math.pluginSolution
3. jetbrains.mps.baseLanguage.math.runtime

After configuration, the project properties on my machine look like this:
In MPS, all code exists inside a model. You can create a model inside the solution by selecting the solution name in the Project Tab. Right-click on the solution name and select New Model in the popup menu. The New Model Dialog appears. Type LetsStartHere.Math in the Model Name field and press OK. You are then presented with a Model Properties Dialog for the LetsStartHere.Math model. The next section describes how to edit model properties with this dialog.

2.8.1 The Model Properties

The Model Properties dialog has three tabs: Dependencies, Used Languages and Advanced. Enter math and baseLanguage in the Used Languages tab in a similar manner to that used for the Solution. See Figure 2.9 for directions.

The Dependencies Tab makes it possible to define dependencies on other languages that must be present before this language can be built and used. The Used Languages Tab defines what languages are used directly inside the model. Changing the set of languages inside this tab will directly control which root nodes you can create inside a model. Finally, the

Figure 2.8: Create A New Project with a Solution. This snapshot provides directions to create a new project, starting at the window displayed when you first start MPS. If you already have one or more projects opened, close these projects to see this dialog. Alternatively, you can select File>New Project from the menu in any project to bring up the New Project Dialog (steps (2-4)). Create the project as instructed to continue learning how to use the editor.
Advanced Tab offers the option to not generate the model (useful if the model stores code that you know will yield errors, and you need to skip it in an automated build), and other options to customize what languages participate to generation. It also shows the path where the model is serialized on disk inside the project.

Figure 2.9: **Model Properties.** Follow these directions to specify model properties for the MeetTheEditor.Math model. The set of used languages should look as shown when you are done. The language names are greyed out because they are currently not used anywhere in the model. When a language is used in a model, the language name will appear solid black in this dialog.

| Definition 2.8.5 — Used Languages. | Each MPS module has a set of Used Languages. Only languages that are declared under Used Languages can be used to write ASTs in this module. |

## 2.9 The Projectional Editor

Let's try the *math* and *baseLanguage* languages by creating a root node under the MeetTheEditor.Math model. See Figure 2.10 for step by step instructions.

Figure 2.10: **Create an AST Root.** This snapshot provides directions to create an AST Root node. Select a class root node from *j.mps.baseLanguage*. If you open the Model Properties dialog again, you may notice that *baseLanguage* is no longer greyed out (the language is now used in an AST node of the model). If you also open the Module properties dialog, you will see Used Languages for the module includes *baseLanguage*. Module Used Languages are calculated dynamically since MPS 3.3 (the set is the union of all used languages in the models of the solution/language).
2.9.1 The Editor Window

After you have created the class AST Root node, you will see in the editor tab the content shown in the top left corner of Figure 2.11. Follow the directions in this figure to enter two complex variable declarations.

```
public class HelloWorld {
    public static void main(String[] args) {
        complex a = new complex(2+3*i); // (1) Press return here
        complex b = new complex(2+3*i); // (1) Start typing complex
    }
}
```

Figure 2.11: Walkthrough the math language Editor Example. Immediately after creating a class AST Root node, follow these directions to learn how to edit the AST in the editor. In step 11, notice how the editor presents two possible completion choices after you have typed the ‘*’ character.

If you are familiar with the Java programming language, you will recognize the class language construct, but may be puzzled by the complex variable type. This type appears in the editor as if it was a primitive type of the language, but the Java language does not offer such a type [Gos+05]. The explanation is that we are not editing Java code in a text editor, but are editing an AST in MPS by composing two languages. The baseLanguage (language) provides most of the Java language syntax and behavior, while the jetbrains.mps.baseLanguage.math (math) extends baseLanguage with features to make it easier to work with mathematical abstractions. The complex primitive type is offered by the math language. Similarly, typing the I letter in the editor enters the imaginary number i and makes it possible to use this complex number in expressions.

Well-designed MPS languages present a smooth editor experience and include notations that feel natural to the intended end-users of the languages. In the previous walk-through the complex data type is well-integrated with baseLanguage: (i) it is possible to declare new variable declarations of type complex, as if the complex type was a primitive type of the language, (ii) the complex instance i can be used inside expressions of the language. This book aims to describe how to develop languages with MPS that feel natural to use for end users of the language.

2.9.2 Everything you See is a Projection

The content of the editor window is a projection. It is a data structure that is projected for display according to some choices made by the designer(s) of the language editors.
In contrast with textual editors, a projectional editor like MPS lets you change how data structures are projected onto the editor display. To demonstrate this, we will use a feature introduced in MPS 3.3, which offers a reflective editor for any part of an AST. When it is active, the reflective editor displays every attribute of the AST node. To change projection, try selecting different parts shown in the editor, and activate the Reflective Editor. You can do this by right-clicking on the node and selecting Show Reflective Editor. The display will change and show all the attributes of the node you have selected. You can revert to the previous editor by selecting Show Regular Editor. The reflective editor is useful when you need to inspect or edit part of a node that are not shown by the editor provided by the language.

Exercise 2.1 Try selecting the reflective editor when the cursor is positioned over main (as in the root node shown in Figure 2.11). This will display the reflective editor and show all properties, children and references of the main method.

2.9.3 Executing the AST Root

Some MPS languages are designed to generate code that can be executed. This is the case of the math language because it generates to baseLanguage and baseLanguage generates to Java source code. MPS make it possible to generate and run Java code directly within the workbench (the second volume of this book explains how to implement this functionality for other languages [Cam15]). Figure 2.12 shows how to execute the program that we created in Figure 2.11.

Figure 2.12: Running the Complex Math Example. Follow these steps to generate, compile and execute the complex example. (1) Right click on the main method. Select Run ‘Class HelloWorld’ in the popup menu. Execution occurs in the Run tab (3), at the bottom of the MPS workbench. Comparing the output to the program shows a seamless integration of the complex type in baseLanguage.

---

2 This editor is called reflective because it uses a reflection mechanism to be able to access and render node content irrespective of node type.
2.9.4 Code Generation

What kind of code does MPS execute when you select Run ‘Class HelloWorld’? The answer is code generation. The math language includes a Generator Aspect that transforms the concepts of the math language into concepts of baseLanguage (model to model transformation is described in Chapter 15). Because MPS ships with a Build System capable of compiling Java code seamlessly, and because baseLanguage classes with a public static void main method can be executed by the workbench you can seamlessly run the generated code in the workbench. Let’s have a look at the generated code. You can do this with the solution name > AST Root > right-click > Preview Text Output. See Figure 2.13 for detailed steps.

![Figure 2.13: The Complex Math Example Generated to Java Source Code. To generate this output from the editor window, locate the HelloWorld AST Root, right click and select ‘Preview Generated Text’. An alternative way to preview the generated code is to right click on the Model or AST Root node in the Project Tab. Select ‘Preview Generated Text’. Note that models may generate multiple output text files if they contain several AST roots. Compare the generated output to the content of Figure 2.12.](image)

The Generator Aspect is explained in Chapter 15 and describes how to generate text output or to transform an AST expressed in one language into another one. While many approaches have been developed over the years to generate code, including StringTemplate[Par] and Velocity Template[Con], MPS provides a principled and very effective platform to generate code from concise language notations. MPS Generators scale easily as projects grow and benefit from full editor support in the templates. This is a strong advantage that has no equivalent in existing code generation tools based on templates.
2.10 The Intention Menu

Intentions are a convenient way to interact with the AST when there is no simple way to map an action to a word or set of keys. An intention menu is available on nodes that provide intentions. In MPS, the intention menu is shown as a yellow light bulb in the left margin of the editor when the cursor is immobile on a node for a few seconds. You can also trigger the display of the intention menu with the keyboard. Use option + on mac, or ctrl + on Windows or Linux. When the intention menu appears, select the intention that you want to run, or navigate to the code that runs the intention by using the intention submenu.

2.11 The Inspector Tab

The Inspector Tab is located at the bottom right of the main MPS window. It can be opened by clicking on the icon. The Inspector Tab shows the Inspected Cell Layout for a node. The Inspected Cell Layout can be defined when developing a custom AST node editor (see Chapter 5 Section 5.2.4). For existing MPS Concepts, the Inspector Tab will often provide a means to edit properties associated with a node which are not shown in the main editor window. This can be done to avoid clutter or to provide a complementary editing dimension when working with some kinds of AST nodes. This is particularly useful when working with the Editor and Generator languages.

2.12 Book Organization

This book is organized in a succession of reference chapters interleaved with chapters that present applications of the concepts covered in the reference chapters. Reference chapters focus on clear definitions and exhaustive explanations of MPS capabilities. Application chapters highlight some of the most important points through a worked example.

If you have read an earlier version of this book and only need to lookup new material introduced in a new edition, I suggest to use the index and look for the entry “New in MPS X.Y”. For instance, the third edition of this book contains an entry called “New in MPS 3.3” which points to the pages introduced to discuss features new in MPS 3.3.

Conventions

This book uses the following typographical conventions:

1. *italic* identifiers refer to MPS languages (e.g., `baseLanguage` or `jetbrains.mps.baseLanguage`). Note that I also use italic to refer to concept interface names, but there should be no ambiguity with language names.

---

3 Section 5.3.1 describes how to run actions when some text is typed.
4 Section 5.3.2 describes how to run code when a key combination is pressed.
5 In MPS 3.1+, you may need to enable the inspector tab before you can open it. You can enable/disable tabs by clicking or browsing over the icon in the lower left corner of MPS 3.1 window. While you are adjusting the 3.1 user interface, you should also enable the Toolbar (View Toolbar) because you will need it often (for some reason, it is disabled by default in MPS 3.1). The Toolbar is enabled if you see a checkbox next to it in the View menu.
2. typewriter text is used either to show a concept name, or to make it easier to see fragments of code when they are embedded in the text.

3. Keystrokes are shown as follows: \texttt{option+SPACE}. This means that you should press the option key together with the SPACE key on the keyboard. Note that the return key is shown as \texttt{\textbackslash{}}.

4. Menus that you should select to perform some action are shown as \texttt{Solution right-click Model Properties}. Note that MPS has a menu bar, but attaches menus over several types of user interface elements.

5. The symbol represents the icon that appears in the left margin of the editor and makes it possible to execute an intention or navigate to the code of the intention.

6. The icon is used to suggest that your should look at the content of the Inspector Tab.
$CALL$, 237
$COPY_SRCL$, 237
$COPY_SRC$, 236
$EXPORT$, 237
$IFS$, 238
$INCLUDES$, 238
$INSERTS$, 238
$LABELS$, 238
$LOOP$, 239
$MAP_SRCLS$, 239
$MAP_SRC$, 239
$SWITCH$, 240
$TRACE$, 240
$VAR$, 240
$WEAVES$, 240

Abandon rules, 261
Access language, 289
Anchor, generator weaving rule, 258
Anonymous ports, 306, 320
AntiQuotations, 124
Applicable for, 189
Application Specific Language, 19
ASL, 19
AST Concept Hierarchy, 23
AST root, 33
Auto-completion Menu, 48, 66, 93
baseLanguage, 113
Bootrapping dependencies, 124

Braces, collections, 78
Build system, 40

CALL, 237
Caret, 290
Cell Action Map, 75
Cell Key Map, 76
Cell Menu Component, 93
Cell Menu Parts, 95
collapse by default, 78
Color, 92
come from, 184
CompositeScope, 181
Computable, 142
Concept Editor Context Hints, 87
Concept functions, 87
concept// expression, 118
concept<Concept>, 118
concepts menu, 149
ConceptTextGenDeclaration, 214, 215
Connector Creation, 301
container, 300
Content, 299
Content Node, 246
Context Hint, 73, 87–89
Context Hint, add, 81
Context Hint, remove, 81
COPY_SRC, 236
COPY_SRCL, 237
Create Template Fragment, 246
Cursor position, retrieving, 290
Custom cell, 86
Custom Figure Implementations, 307
DelegatingScope, 181
Dependencies, 35
Deprecated in MPS 3.2, 118
Diagram Cell, 298
Diagram Connector, 301
Diagram Connector Cell, 306
Diagram editors, 295
Diagram Element, 299
Diagram Node Cell, 304
Diagram Port Cell, 307
Diagrams, 295
Domain Specific Language, 19
Downcast operator, 83
Drop attributes, 260
DSL, 19

EAP, 17
Editor Cell, 74
Editor Component, 89
Editor Styles, 90
editorContext, 141
Element Menu, 78
Elements Creation, 299
EmptyScope, 180
ensure, 202
enummember<Enum Data Type>, 121
executeCommand(), 142
EXPORT, 237
Extends Clause, 46, 53
ExternalViewFigure, 310

Figure Implementation, 297
Figure parameters, 305
FilteringByNamedScope, 182
FilteringScope, 181
Find usage, 109
Folding, collections, 78
genContext, 232
Generate Executable Concepts, 287
Generator Descriptor, 250
generic query, 152
Generic View of Children and References of a Concept, 288
get original copied input, 232
GlobalScope, 176
Greying out, 36, 37
Help, 52
Icon, 52
IF, 238
Image cell, 83
IMainClass, 287
Implements, 46
INamedConcept, 66, 101
Indent cell, 85
indent-layout-new-line-children, 108
Inline Template, 234
inputModel, 233
inputPorts, 305
INSERT, 238
Inspected Cell Layout, 43, 74
Inspector Tab, 42
Instance can be root, 48
Intention, 132
Intention menu, 52
Internal Concept Declaration Holder, 114
isEditable(), 143
isInspector, 142
isNotNull(), 116
isNull(), 116
isSubConceptOf method, 148
jetbrains.mps.lang.access language, 289
jetbrains.mps.smodel, 113
JetPad, 309

LABEL, 238
Language, 33
Language Workbench, 18
language// expression, 119
LanguageTextGenDeclaration, 219
Layout, TextGen, 215
INDEX

Light Quotation Language, 124
LinkDeclaration, 145
ListScope, 180
LOOP, 239
LW, 18

MAP_SRC, 239
MAP_SRCL, 239
Mapping Script, 250
Marker interface, 54
Meta Programming System, 17
Migration assistant, 30
Model, 33
model, 119
Model to model transformation, 40
ModelAccess, 83
ModelPlusImportedScope, 183
ModelsScope, 182
Module, 33
Motivation, 17
MPS, 17, 18, 23
MPS documentation is not clear, 120, 147, 152, 163, 240, 249

Naming conventions, 59
Navigation to a node, 287
New in MPS 3.2, 43, 118, 119, 239, 243, 253, 254, 287, 290, 292, 293
New in MPS 3.3, 37, 78, 86, 136, 214, 215, 258, 260, 294
New Model Dialog, 35
Next applicable cell, 86
nlist<Concept>, 118
Node Cell Layout, 74
Node Macros, 236
node<>, 117
node<Concept>, 117
Not used, 36
null and SModel, 115, 116

on create, 301
on create, (diagram connectors), 303
One operand type, 208
openInspector(), 142

Operation Concepts, 208
Operation context, 146, 178
operation type, 202
Operator :==:, 201
Operator :~:, 202
Operator :~~:, 202
Operator :>=:, 201
Operator :>>=:, 201
Operator :<<=:, 201
Overloaded Operation Rules, 207
OverloadedOperatorTypeRule, 209
OverloadedOpTypeRule_ OneTypeSpecified, 207
Override behavior method, 277
Override type with annotations, 205

Palette, 304
Parameterized intention, 135
parameterized item, 150
parent scope, 184
Pattern Language, 189
Pattern rules, 261
Plugins, 32
Pointing the editor to a node, 287
PProtected behavior method, 113
Preferences, 32
Preview Text Output, 40
Private behavior method, 113
Projectional editor, 39
Property Fields, diagrams, 308
Property Macros, 234
Property Value Mappers, 308
Protected behavior method, 113
Public behavior method, 113
punctuation-left style item, 104
punctuation-right style item, 104
Quotation language, 122
Quotations, 122
reference, 120
Reference Macros, 235
Reference visibility, 174
Reference, how to set value generically, 121
Reflective editor, 39
Runnable, 142
runReadAction method, 83
runWithContextCell(), 143
runWriteAction method, 83
Scope, 179
Scope helper classes, 180
Scopes, 174
search scope, 120
select(), 142
selectRange(), 142
selectWRTFocusPolicy(), 141
Semantic error detection, 188
Side transformations, 154
side-transform-anchor-tag, 156
simple, 150
SimpleRoleScope, 181
Smart reference, 69, 141, 179
Smart Reference Concept, 53
SModel Language, 115
SModel language, 113
Solution, 33
Source node, 139
Specialization, 50
specializes, 50, 51
Strong sub-typing, 196
Style, 81
Style items, 92
Style Key Pack, 93
Style Sheets, 91
Substitute Type Rule, 212
Substitute type rule, 204
Substitution menu, 140
Super-concept, 46
Surround-With Intention, 134
SWITCH, 240

Target node, 140
Template Declaration, 245
Template Macro Language, 233, 247
Template Switch, 247

TextGen layout, 215
The Constraints Aspect, 169
The Editor Language, 74
The Generation Context, 232
The Meta Programming System, 17
TRACE, 240
Transform Menu Actions, 154
Traversing the AST, 115
typeof, 202

Undo/Redo Support, 289
Unfolding, collections, 78
Universal intention, 136
unordered, 50
URL, 52
Used Languages, 36
Used languages, 35

VAR, 240

Weak sub-typing, 196
WEAVE, 240
when concrete, 202
wrapper, 151