Credits

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

- Markus Voelter
- Marco Lombardo
Contents

Front cover i
Credits i
Contents ii

I Start 1

1 Fast Track to MPS 2
  1.1 Ten steps to mastery ................................................................. 2
  1.2 Step 1 - Why MPS ................................................................. 2
  1.3 Step 2 - Install MPS and study the fundamentals ...................... 3
  1.4 Step 3 - MPS from the programmer’s perspective .................... 3
  1.5 Step 4 - MPS from the language designer’s perspective .......... 4
  1.6 Step 5 - Projectional editing tips and tricks ......................... 5
  1.7 Step 6 - Creating your first language ................................. 5
  1.8 Step 7 - Shapes tutorial ..................................................... 6
  1.9 Step 8 - Calculator tutorial ............................................... 7
    1.9.1 The Calculator tutorial in JavaScript ........................... 7
  1.10 Step 9 - Other sample projects ......................................... 8
  1.11 Step 10 - Generator tutorial ............................................ 9
  1.12 Reviewing the progress .................................................... 10
  1.13 Moving up the ladder ....................................................... 10
  1.14 Documentation sources .................................................. 10
    1.14.1 Books ........................................................................ 11
    1.14.2 Academic papers ....................................................... 11
    1.14.3 Cookbooks ................................................................ 11
    1.14.4 User guides ............................................................... 11
    1.14.5 Screencasts ................................................................. 11
    1.14.6 Guided Trainings .......................................................... 11
    1.14.7 General documentation .............................................. 12

II Introduction 13

2 Shapes - an introductory MPS tutorial 14
  2.1 Prerequisites ........................................................................ 14
  2.2 Goal .................................................................................... 14
  2.3 Create a new project .......................................................... 15
  2.4 Languages and programs under the hood .......................... 17
  2.5 Graphical shape ................................................................ 18
  2.6 Circle ................................................................................... 20
  2.7 Square ................................................................................. 23
  2.8 Canvas ................................................................................ 23
  2.9 An early test ride ............................................................... 25
  2.10 Editors ................................................................................. 27
    2.10.1 Shape .......................................................................... 28
    2.10.2 Circle ........................................................................... 28
    2.10.3 Square ........................................................................... 29
    2.10.4 Canvas .......................................................................... 30
  2.11 Second run .......................................................................... 31
## Generator User Guide Demo 1

### New Language

### New Generator

### Root Mapping Rule

### Property-macro

### The main() method

### SWITCH-macro

### Getting Ready for the First Test

### Generating Test Model

## Generator User Guide Demo 2

### New Language

### Conditional Root Rule

### LOOP-macro

### Running First Test

### Abandon Root Rule

## Generator User Guide Demo 3

### New Test Model

## Generator User Guide Demo 4

## Generator User Guide Demo 5

## Generator User Guide Demo 6

## Generator User Guide Demo 7

## Generator

### Generator Demos

#### Generator Tutorial

#### How to Use This Documentation

#### Setting-Up Demo Project

#### New Project

#### New Demo Solution

#### New Test Model

#### Test Generation

#### What's Inside

#### New Generator

#### Conditional Root Rule

#### LOOP-macro

#### Running First Test

#### Abandon Root Rule

### Embedding the editor component

#### Coloring

#### Concept for colors

#### Pre-defined colors

#### First touch on dependencies

#### Concept for color reference

#### Updating Shape

### A third run

### A more robust generation for Squares

### Generating code

### Running the code

### What to do next

### What to do next
## 6 Generator User Guide Demo3
- 6.1 Generator User Guide Demo 3
- 6.2 New Language
- 6.3 addContent(container)
- 6.4 Generating components
- 6.5 Inserting components
- 6.6 New Test Model
- 6.7 Adding an $\text{IF}$ macro
- 6.8 Reusable template
- 6.9 MAP_SRC-macro
- 6.10 INCLUDE-macro
- 6.11 Testing what we have
- 6.12 Reference-macro (Resolving by Name)
- 6.13 The final test

## 7 Generator User Guide Demo4
- 7.1 Generator User Guide Demo 4
- 7.2 New Language
- 7.3 Template for Panel
- 7.4 First Test
- 7.5 Restricting the LOOP macro to root elements
- 7.6 COPY_SRC-macro
- 7.7 Reduction Rule
- 7.8 Final Test

## 8 Generator User Guide Demo5
- 8.1 Generator User Guide Demo 5
- 8.2 New Language
- 8.3 New Test Model
- 8.4 Pre-processing Script
- 8.5 First Test
- 8.6 Utility Classes in Generator
- 8.7 Post-processing Script
- 8.8 Second Test

## 9 Generator User Guide Demo6
- 9.1 Generator User Guide Demo 6
- 9.2 The core idea
- 9.3 New Language
- 9.4 New Test Model
- 9.5 Generator
- 9.6 Reduction Rules
- 9.7 First Test (Failure)
- 9.8 Dividing Generation Process into Steps
- 9.9 Second Test
- 9.10 Saving Transient Models
- 9.11 Root Nodes Copying and Reduction Rules
- 9.12 Using Generation Tracer Tool

## 10 Generator User Guide Demo7
- 10.1 Generator User Guide Demo 7
- 10.2 New Language
- 10.3 Enhance the language
- 10.4 addContent(container)
- 10.5 New Test Model
- 10.6 Root mapping rules
- 10.7 Weaving Rules
- 10.8 Weaving Context
- 10.9 Mapping Label
- 10.10 External Template
- 10.11 IF-macro
- 10.12 Complete the Generator
- 10.13 First Test (Error)
- 10.14 Generating Unique Names
IV  Advanced  148

11 Dive into plugins: the PlantMPS plugin  149
   11.1 Introduction  149
      11.1.1 How the plugin works  149
      11.1.2 Credits  150
   11.2 Getting started  150
      11.2.1 Create the project  150
      11.2.2 Create the tool  150
      11.2.3 Add a tool icon  150
      11.2.4 Complete a dummy tool  151
      11.2.5 Tool life cycle  151
   11.3 Adding actions  151
      11.3.1 Keys binding with Keymap  152
      11.3.2 Use the defined actions programmatically  153
      11.3.3 Tune actions  153
      11.3.4 Dynamic actions  153
      11.3.5 Create actions programmatically  153
      11.3.6 History  153
   11.4 Add PlantUML and SVG support  154
   11.5 Implementing language, generation plan and java_stub diagrams  154
Part I

Start
Chapter 1

Fast Track to MPS

Welcome! This tutorial was designed specifically for developers, 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.

Hi, my name is Václav and I’ve created this tutorial for you. The tutorial is a living creature that I constantly maintain and evolve to make it better serve its purpose - to help you learn how to design languages with MPS.

You can help me on that mission - if you spot a mistake, hit a road-block or become confused - let me know. I’ll be very grateful, if you leave a comment or file a YouTrack issue for me. Thank you.

We admit that there are easier tasks in the world than learning MPS. Language design is a complex domain and projectional editing needs some getting used to. It helps a lot if you have prior experience with DSLs or language workbenches, but it is not necessary for understanding this tutorial. Just expect more bumps on the road. There certainly will be a lot of them - partly also because MPS has its own problems and bugs. So you are in for a tough experience that may take quite a big amount of your time.

Bear in mind that many have walked through here before and succeeded. If you persist you’ll be rewarded - you’ll become a language designer who can prepare custom-tailored DSLs, integrate them into projects and develop high-value software with them.

So, take on good shoes, lace them tight and start going ...

1.1 Ten steps to mastery

1.2 Step 1 - Why MPS

Raw time estimate - 45 minutes

Let’s start with some motivation for learning MPS.

1. First of all, check out a two-minute overview video about MPS - Why JetBrains MPS, so you can assess whether this tool offers something that you need on your projects.

2. Our new VoiceMenu demo application has been created to give people coming from different domains of expertise a quick understanding of the potential of MPS. Watch a short https://youtu.be/pVlywLXDuRo.

3. JetBrains MPS is an efficient and reliable development tool that has been adopted by software vendors on both commercial and academic projects. MPS-based DSLs have been adopted in wide range of domains, such as legislation modelling, electrical engineering, insurance industry, embedded software, medicine systems, bio-informatics and data analysis. The "How it works" page lists several of the successful projects and gives a high-level overview of the technology used.
4. You may also like to slow down a bit and read a few documents describing how people have succeeded with MPS. We’ve collected a few case studies that may inspire you to your own ideas on building something great around MPS.

- You may find a complete list of user stories in the Use cases section of the MPS documentation.
- You may also check out a list of all academic publications on MPS that the MPS users ahem published over time.

1.3 Step 2 - Install MPS and study the fundamentals

Raw time estimate - 20 minutes

1. If you are carefully evaluating MPS there must be many questions hoping up in your head while watching videos or reading stories about MPS usages. I believe that some of these questions have already been answered in the FAQ. Check it out,

2. Stop for a minute now and skim through a short Basic notions like node, Concept, AST in order to understand better the terminology used down the line.

3. Plain reading about MPS will never teach you as much as trying it for real yourself. You might have done this already, but if not, grab MPS from the download site and install it. With MPS installed you will be able to continue on the Fast Track.

Before you proceed, please bookmark the MPS Glossary page and come back to it frequently, until you get a good command of the vocabulary that the MPS community uses.

1.4 Step 3 - MPS from the programmer’s perspective

Raw time estimate - 45 minutes

Before you continue, you should probably get to know the most distinct feature of MPS - the projectional editor. Check out a video that explains all the differences between traditional textual editors and the MPS projectional editor and lists the numerous benefits of this approach.

Since we programmers like trying things, how about taking the MPS editor for a simple test ride first? You may get the first impressions instantly, if you start MPS and open the Robot Kaja sample project. This project defines an artificial language to command a virtual robot. The robot can move within a rectangular area, drop and collect marks and check the surrounding cells for walls and marks. You write Scripts using the built-in commands as well as new commands that you create on the fly. When you run the script a simulator written in Java is started to visualize the execution of your program.

To open the project, start your fresh MPS, click on the Open Sample Project button and pick robot_Kaja. The sample projects have been installed into your HOME/MPSSamples folder.
We also shot an instructional screen-cast to help you start. Check it out.

Do not worry to break things while playing with the MPS samples. You can always delete the HOME/MPSSamples folder and MPS will re-create it the next time you click the Open Sample Project button. Experiment, experiment, experiment!

1.5 Step 4 - MPS from the language designer’s perspective

Raw time estimate - 40 minutes

Now you must be curious about how the robot sample language has been created. Please, open again the Robot Kaja sample project and play our explanatory screen-cast showing you round the language definition fundamentals. As you go through the video, pause frequently and look around the sample project yourself. This way you will understand a typical language structure and see the relationships between its individual parts.

At this point you should understand that language definition consists of several aspects. Some of these are mandatory, some are optional:

- **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.

- **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.
CHAPTER 1. FAST TRACK TO MPS

- **Actions** - Since the MPS editor manipulates the underlying AST directly, some editing actions, such as copy-paste or replacing one node with another one translate into non-trivial AST changes. The **Actions** aspect holds definitions of such customised AST transformations.

- **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.

- **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.

- **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.

- **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.

- **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.

### 1.6 Step 5 - Projectional editing tips and tricks

**Raw time estimate - 30 minutes**

When editing code in MPS you will run into situations when your coding habits interfere with the projectional nature of the MPS editor. I’m actually pretty confident that you have already discovered such surprises during your experiments. Code in MPS is not text, but instead it is a projection of the abstract syntax (AST). This gives several benefits to language designers and, at the same time, makes the editing experience slightly different and more-restrictive than in the commonplace text editors. This may or may not be considered to be an advantage, nevertheless, there is a learning curve to projectional editing. In general, in projectional code you more rely on the assistance of the editor. Code-completion, block selection and intention actions should become your close friends in order to become an efficient projectional coder. There’s a good and a bad side to it:

- **good** - in less than a week of programming in the MPS editor people typically get back to their full speed of coding they experienced before in text-based IDEs

- **bad** - projectional editing is highly addictive and you may find text-based editors less compelling and less helpful than you thought they’d been before

 We summarized the core essential tips on how to leverage the projectional editor. Please, check out the Commanding the editor page and keep it ready when making your first steps in MPS. Although you do not have to memorise all the keyboard shortcuts immediately, they will certainly make your coding more efficient once you start developing your own projects.

### 1.7 Step 6 - Creating your first language

**Raw time estimate - 3 hours**

The next screen-cast to watch, MPS basics - creating your first language, will show you the typical workflow when creating a new language - building the structure, defining the editor and implementing the generator. You will see the creation process of the **Constants** language, which is also included as a sample project with MPS distributions.

Now you’re recommended to watch a series of short high-level videos that will give you a brief introduction into each of the aspects of a language definition:

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.
CHAPTER 1. FAST TRACK TO MPS

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.

1.8 Step 7 - Shapes tutorial

Raw estimate - 2 hours
Here’s an opportunity for you to quickly get your own language. Check out the MPS entry-level Shapes - an introductory MPS tutorial and build a language that enables non-programmers to build scenes of graphical shapes.

![Shapes tutorial example](image-url)
1.9 Step 8 - Calculator tutorial

Raw time estimate - 8 hours
Now you must be curious to find out more details about language design in MPS, right? Having given you all the fundamental information and some level of experience, the introductory tutorials should have left you in a good position to dive deeper and build a sample language all the way from the ground up. The Calculator tutorial will guide you through the process of language creation. You start with an empty project and end up with a language to build Java Swing-based visual applications. Try it out and have fun.

Beware, you are entering a "Suck Zone" here. We’re happy to see you here, but you need to be warned that the road gets quite steep and more bumpy as you progress through the Calculator tutorial and the following exercises. No wonder, you’ll be doing real language design now and you’ll be left intentionally on your own to implement parts of the languages. If things do not work out for you, relax, step back and try again. Sometimes the problem may be as small as a semicolon missing from selection or a method placed in a wrapper class instead of an anonymous inner class. Remember, you can always ask for assistance on the forum and we’ll help you get back on track.

1.9.1 The Calculator tutorial in JavaScript

As an illustration that MPS is not limited to generating Java, we’ve prepared a similar Calculator sample project - CalculatorJS, which generates an HTML page and a JavaScript program to hold the computation. You may check it out on GitHub.
1.10 Step 9 - Other sample projects

MPS comes with several sample projects that you may play with to get a better grasp of language development. I recommend you tried them out, perhaps in the order as they are listed below. Find out what they do and how they are implemented. They may teach you important lessons. Don’t worry if you sometimes cannot figure out how a particular feature has been implemented. Most likely you will find an answer in one of the following exercises and then you can revisit the sample and the thing will just click instantly.

Here’s a list of the samples that we’ve prepared for you:

**Language extensions:**

- **SampleJavaExtensions** - a collection of small handy Java enhancements, such as parallel for loop, unless statement, Money literal, decision tables and Constants definition. The sample will teach you how to extend existing general purpose languages with custom constructs. You may also like to watch a short screen-cast introducing the individual extensions.

- **Complex** - defines a Java extension to work with complex numbers

- **XML Literals** - allows XML literals to be used directly in Java code as first-class language elements. The language builds on top of jetbrains.mp.xml and generates into into org.jdom.* API. The user-inserted XML code may refer back to the surrounding Java context to easily parametrize.

- **math** - provides fancy-looking math language extension to Java, which allows you to enter complex numbers, matrices, intervals, summation, product and other common mathematical operations in a natural way.

**Standalone languages:**

- **LanguagePatterns** - a set of sample languages illustrating implementations of common language patterns in MPS. It is useful for beginners to see in isolation concrete implementations of typical language constructs. This sample project is gradually being updated with new patterns.

- **MultipleProjections** - an example of a simple state machine language that focuses on the mechanism of multiple projections per concept. The programmer, who uses the language, can on-the-fly switch between several pre-defined notations as she progressed with her state-machine definition. All the available notations are defined in the language or its extensions.

- **MultiTarget** - illustrates how to have a single language generated into multiple different target languages.

- **Custom Aspects** - shows the ways to define custom aspects for language definition. Check out the Custom language aspect cookbook for details.

- **Cross Model Generation** *(new)* - illustrates how to use custom generator plans to achieve cross-model generation.

- **Component Dependencies** - a minimalistic example of a language for expressing dependencies between components of a system, which provides three alternative editor notations:
  1. Textual
  2. Tabular
  3. Diagrams

- **Customized Debugger** - an example of hooking a languages with the Java debugger, even when the language neither extends nor generates into BaseLanguage.

- **Mind maps** - an intuitive example of defining a diagramming notation, including a set of customized graphical symbols

- **State machine (Secret Compartment language/DSL book sample)** - an example of a state machine language, including a state machine test language. An overview screen-cast is available to help you get started.

- **The Simplest Language** - a minimalistic example generating a Hello world application printing out a customized message. Shows the very basics of language generation.

- **Migrations** - a simple example of migration scripts to migrate two interconnected languages using both the smodel API and TransformStatement.

- **Lambda Calculus** - a language simulating lambda calculus, with projectional editor and generation into Java

- **Agreement** - an example of a business-oriented DSL
• Expressions - a simplified expression language to allow you to play with the type system and the Type-system Trace tool. The language gives you a bare-bone expressions language with minimal editor support, but with well structured type-system rules, which you can help you learn the core of how MPS calculates types. Read more in the Typesystem Debugging documentation.

• Fixed Length Reader - illustrates how to create a language for describing binary data formats. Comes with a stub for a hypothetical runtime library that would extract elements from binary data using the supplied data format description (called configuration).

• HighLevel Languages - an example of defining a custom set of language definition languages. Illustrates how to define your own way to codify structure and editor for a language.

• Sample Language - A very intuitive and minimalistic example showing how to implement a common pattern of a root concept containing definitions as well as references to them. In this sample, we simulate Java classes, method definitions and method calls. However, such a pattern is quite frequent in many languages and it is worthwhile to spend some time investigating. You may also check out a brief screen-cast showing you round this sample.

• Formula Language - a sample definition of an expression language, including editor actions and a generator

• NotesOrganizer - a sample DSL plugin for IntelliJ IDEA including a build script, that simulates a task/todo list. Tasks can have different priorities, states and categories, which by themselves can also be customized. The task lists can be filtered using several kinds of Swing UI components and the user can choose among several alternative visualizations of the task lists. Shows as an example of filtering node collections, alternative editors and direct incorporation of Swing components into form-like notations.

• State Chart (new) - an example of a state-chart definition language that generates state charts in the SCXML standard.

• Attributes - a sample illustrating the transparent handling of attributes in text-gen and the generator. The two CommentAttribute node attribute concepts are passed through the generator unchanged so that their text-gen can convert them to text.

• Sample Facet - illustrates a simple make facet that intercepts the make process.

• Lightweight DSL - examples of usages of the jetbrains.mps.baseLanguage.lightweightDSL language to build internal "lightweight" DSLs embedded into BaseLanguage classes.

• Heating controller - a simple language providing two notations to specify heating plans for a hypothetical home heating controller. The default notation allows for textual representation, while a language extension provides an alternative tabular notation.

• Progress Indicator - a set of actions that illustrate the proper way to handle progress bars in actions, how to make actions cancellable and enable them to run in the background.

• xmlPersistence - a non-trivial example of implementing custom persistence using the new Open API. The plugin contains three solutions (a build script, a persistence implementation and a plugin descriptor). The custom persistence implementation will load/save XML documents. Please check out our Custom Persistence Cookbook for details.

Samples to tutorials:

• Shapes - the language created as part of the introductory Shapes - an introductory MPS tutorial.

• Robot Kaja - the language for commanding a virtual robot used in many of the introductory screencasts. Now includes a sample Context Assistant implementation.

• Calculator - the language created as part of the Calculator Tutorial.

• Generator demo - the set of languages developed as part of the Generator Demos.

1.11 Step 10 - Generator tutorial

Raw time estimate - 12 hours
The Calculator tutorial should have left you with good overall knowledge of language design. The Generator Demos will now teach you more subtle options that the MPS generator offers. You will learn about the different types of macros, generator scripts, utility models and more.
CHAPTER 1. FAST TRACK TO MPS

1.12 Reviewing the progress

Congratulations! By completing the above tutorials and exercises you’ve made a serious dent into the language design world. Your knowledge and command of MPS could now be qualified somewhere above the Advanced beginner stage of the Dreyfus model of skill acquisition. This is definitely a good time to celebrate and perhaps also to let us know about your success.

1.13 Moving up the ladder

The next stage in the Dreyfus model of skill acquisition is Competent. Once you reach that stage you’ll be able to work on your own without concrete guidelines and supervision. But for this to happen, you’ll need to spend more time solving various problems with MPS. If you already have an idea of a language to implement, go ahead and do it. If not, try to come up with a useful little language. You could as well try to re-implement one or more of the MPS samples and then see how closely you got. Or just pick a sample as the starting point and move it forward - add language features, implement a generator, add type-system rules, polish the editing experience - whatever feels like the right challenge for you.

Additionally, you should try these in order to become Competent in MPS and language design:

1. Study the Common language patterns
2. Study the cookbooks to deepen your knowledge of the various aspects of language design
3. Get to know Quotations
4. Familiarise yourself with the languages bundled in MPS, such as Closures, Collections language, Concept Functions, Regexp language, Tuples, SModel language, Build Language and other
5. Learn how to Build Language, how to package languages into Building IntelliJ IDEA language plugins as well as Building MPS language plugins plugins, how to Using MPS inside IntelliJ IDEA and how to Building standalone IDEs for your languages
6. Get a feel for the MPS User’s Guide so you can quickly find the information that you need
7. Learn some of the Commanding the editor by heart
8. Become active in the MPS community and try to answer the forum questions of people that walk behind you on the same path to MPS mastery

After becoming Competent you should definitely continue gaining experience. The more you work with MPS and the more divert your projects are, the faster you’ll be moving up on the Dreyfus model of skill acquisition scale. If you set yourself a goal to become Proficient and ultimately an Expert in MPS, you might consider trying:

1. Studying and practising advanced topics, such as:
   (a) Find usages
   (b) HowTo – Integrating into the MPS Make Framework
   (c) Debugger enhancements and customisation
   (d) Custom Persistence Cookbook
   (e) Dependencies Analyzer - analyze model dependencies and Module Dependencies Tool
   (f) Custom dataflow analyzers
   (g) Extension support
2. Helping MPS expand into there areas by creating a base language for a different target language/platform - Python, Ruby, Dart, Haskell, Go, etc
3. Speaking about MPS at events
4. Helping us evolve the MPS User’s Guide
5. Contributing to JetBrains MPS Project to the MPS project

Although there’s not much detailed documentation to guide you at this experience level and you’ll have to be self-driven most of the time, we’ll always be happy to discuss, advise and cooperate with you.

1.14 Documentation sources

Here’s a list of documentation sources that you may need on your path to MPS mastery
1.14.1 Books

Fabien Campagne wrote The MPS Language Workbench: Volume I and Volume II books, which you can purchase both electronically and as a paper-back. Get them [here](#).

1.14.2 Academic papers

The MPS publications page page collects all relevant papers about MPS and the related field.

1.14.3 Cookbooks

The get slightly simplified practical information as well as some tips and tricks on individual aspects of language design, you may try some of the cookbooks.

- **Common language patterns** - if you need guidelines on how to implement certain functionality
- **Editor cookbook**
- **Generator cookbook**
- **Description comments** - a cookbook showing how to leverage attributes to add support for description comments attached to arbitrary nodes
- **Cookbook - Type System**
- **SModel language** to show you how to query your models
- **Dataflow**
- **Regular expressions**
- **Finding your way out** - a brief collection of guidelines that should help you move forward when you get stuck somewhere.

1.14.4 User guides

Full-blown user guides will provide exhaustive information on their respective subject.

- **MPS user guide for Java developers (IntelliJ IDEA)** (IntelliJ IDEA)
- **MPS user guide for DSL users** (standalone MPS-based IDEs)
- **MPS User’s Guide**

1.14.5 Screencasts

Over time the MPS team has created a large collection of screen-casts covering many aspects of MPS use and language design. You may check them out all nicely sorted at the MPS screen-casts or directly at the MPS channel of JetBrains TV.

1.14.6 Guided Trainings

The MPS team offers hands-on trainings. Check out the details:

- **Introductory MPS course**
- **Advanced MPS course**
1.14.7 General documentation

The MPS documentation page will give you all the remaining pointers to case studies, tutorials, FAQs and other pieces of MPS documentation.
Part II

Introduction
Chapter 2

Shapes - an introductory MPS tutorial

If you’re new to MPS and want to try it out quickly, this is the right tutorial for you. Within two hours you’ll get a new language and functional code that uses that language. In the tutorial you’ll start from scratch and by walking along a safe and a convenient path you’ll design the core elements of a new language. We’ll avoid advanced concepts, complicated constructs and dark corners in order to get to the finish line quickly. At the end you’ll know what MPS is all about and what principles it builds on.

So, please fasten your seat belts. We’re in for a fast ride.

You may also watch this tutorial on-line as a screen-cast. Please check it out.

2.1 Prerequisites

We'll assume you've gone through the initial parts of the Fast Track to MPS and so are familiar with the MPS environment, understand the concept of a language and a solution and can Commanding the editor. If, not, please consider spending the first 30 minutes or so of your time to check it out. Especially these keyboard shortcuts are key to your survival:

- Control + Space - to complete an incomplete word or to turn an invalid (red) identifier into a correct (black) one
- Alt + Enter - to display a pop-up menu with handy options applicable at the current editor position
- Control/Cmd + Up-Arrow - Expand the region of selected text
- Tab - to navigate around editable elements in the editor
- Control/Cmd + Z - Undo

We also assume you’ve installed MPS and you have it running in front of you. So now we can set off.

Cheating allowed

The result of this tutorial has been bundled as a sample project with MPS distributions. If you get stuck at any moment, feel free to open it from your home folder/MPSSamples in MPS and look at the implementation. Try to avoid copy-pasting, though, since you wouldn’t learn much that way. (BTW, the sample also contains additional features explained in the Building an interpreter cookbook, which you can have some fun with later.)

2.2 Goal

You’re going to implement a sample language for specifying graphical shapes. The language will allow its users to lay out visual two-dimensional shapes on a flat canvas. The definition will be then translated into a Java Swing application, which will visualize the layout on the screen.
The language could enable non-programmers to build Java applications without any knowledge of Java, Swing or the 2D Graphics API. You, on the other hand, will play the role of a language designer, who will prepare such an easy-to-use language building on his or her knowledge of Java. You automate the work of a UI programmer by providing a language and a generator that cover some of the cases that a UI programmer currently has to solve manually.

If you are not versed in Java or Swing, do not worry too much too early. We assumed this possibility and made the tutorial to guide you carefully. You’ll be able to pass easily, you’ll see.

2.3 Create a new project

We’ll start with a fresh project. On the welcome screen you click on Create New Project and then follow the wizard.
You’ll get an empty project containing and empty Language Definition and an empty Solution.
Solutions hold your programs. To write these programs, you use languages, either defined in the same project or imported ones. In our tutorial, we'll first define a language and then use it to write code that we can execute.

2.4 Languages and programs under the hood

First, here's a bit of background knowledge that you should know before going on.

The language that we're building must allow for painting definitions, which consist of individual commands, each on a separate line and each defining a single shape to draw. Our language needs to cover each such command with a Concept. Concepts define the abstract syntax of a language, i.e., the set of allowed language logical constructs. A program then consists of Abstract Syntax Trees, which hold instances of these Concepts.
The AST of the short program above shows the abstract syntax - it consists of Nodes, each Node is an instance of a Concept that the language defines. Concepts define properties, children and references and Nodes then give the concrete values. To wrap up the little theory here:

- **Languages** consist of Concepts
- **Concepts** define the logical (abstract) elements with their properties, children and references
- **Programs (solutions)** consist of ASTs, which consist of Nodes
- **Nodes** are instances of Concepts giving concrete values to properties, children and references of their Concepts

### 2.5 Graphical shape

Our language is going to be pretty straightforward. We’ll need only a few concepts:

- **Canvas** - to define the top-most node representing the whole **Painting definition** and holding all Shapes
- **Shape** - representing a command to draw a shape on the Canvas, it will serve as a common super-concept to all concrete shapes, such as a circle or a square
- **Circle** - representing a command to draw a circle
- **Square** - representing a command to draw a square
- **ColorReference** - representing one of several pre-defined Colors defined in `java.awt.Color`

We’ll start our practical exercise with the Shape concept. Just like in object-oriented programming, Concepts can extends one another and thus inherit the capabilities of their super-concept. The Shape concept will serve as such a common super-concept, holding the color property, since all shapes in our language will need the color property, so we can conveniently inherit it.
Right-click on the Structure aspect of the language and create a new Concept. You’ll get a new Concept definition open in the editor.

We should give the Concept a descriptive name, Shape in our case will work fine.
We created the \textit{Shape} concept as a common super-concept for all shapes in our language and by itself \textit{Shape} will not be used directly in ASTs. We'll mark \textit{Shape} abstract to indicate explicitly that no instances (nodes) of \textit{Shape} can be created.

If you misspelled the name of the concept and decide to change it, MPS will prompt you, whether you want to do a proper rename re-factoring, in order to update all possible references and usages of the concept in other parts of your code. Since we've just started and have no references to \textit{Shape}, just hit the "No" button.

It is the time to practice the Alt + Enter keyboard shortcut now.

When positioned on the name of the Concept, the Alt + Enter keyboard shortcut brings up a contextual pop-up menu, which gives you the option to apply the "Make Abstract" intention to the Concept. Once you choose that option, the concept is marked as abstract. Now we could add properties, children and references that should be shared by all sub-concepts of \textit{Shape}, but we will leave that for later to make our learning curve flat.

That's our first concept! Hurray! Time to add another one. How about \textit{Circle}? Sure, by right-clicking on the Structure aspect we create another Concept and give it a name - \textit{Circle}.

\subsection*{2.6 Circle}

Following the same steps, you need to right-click on the Structure aspect of the language and add a new concept. Name it \textit{Circle}. 
Circles should inherit capabilities from `Shape`, so we need to indicate this in the `extends` clause. Position the cursor at the beginning of the cell holding the "BaseConcept" text and hit the most useful keyboard shortcut in MPS - `Control + Space` - to invoke code completion.

MPS will show you a list of options applicable as replacements for the "BaseConcept" text. We need to select `Shape` here to make `Circle` extend `Shape`. 
So this is our first concrete concept that will be used by users of our language. To give the concept a nice textual representation in code-completion dialogs and enable MPS to be smart about creating an instance of `Circle` whenever the user types "circle", you need to give the concept an alias 'circle'.

Notice the **Tab** key can help you navigate around editable elements in the editor. Use the **Tab** key frequently.

Each circle needs to specify its coordinates on the screen and its radius. We'll create properties that will hold these integer values. Navigate to the **properties** section, place the cursor on the "« ... »" symbol that represents an empty collection of values and press **Enter**. This will create an empty property.

You give the property a name "x", then hit **Tab** and provide the type of the property - "integer". While typing "integer" you may hit the Control + Space keyboard shortcut to have the name of the type completed by MPS.
Then add properties for "Y" and "radius" and you’ll be done with your first concrete concept.

2.7 Square

Now you should try yourself to create a concept for square. Simply repeat the steps we did for Circle, just create different properties - upperLeftX and upperLeftY to hold the coordinates of the upper-left corner followed by size to specify the length of the sides of the square. Ultimately you should get to this:

2.8 Canvas

Having created two shapes we can move on defining a concept to hold them all in a painting. We’ll create another concept, called Canvas, that will represent a scene composed of shapes. The user will be able to create multiple scenes (Canvasses), which will be mutually independent and will not share shapes. Each Canvas will hold a name and a list of shapes that it contains.

So yet again right-click on the Structure aspect of the language to create a new concept and give it a name Canvas.
Notice that unlike Circle or Square, Canvas does not extend Shape, but BaseConcept. Canvas is not supposed to be drawn nor placed on other Canvasses, so it should not extend Shape.

Navigate (using Tab) to the "<none>" text in the implements section and specify INamedConcept and the name of the Concept Interface that Canvas will implement. Concept Interfaces can also add new capabilities to Concepts that implements them. INamedConcept in our case enriches Canvas with the name property, so Canvas instances (called Nodes) will have a name property so the user can easily distinguish between them.

Since Canvas will represent the painting scene and will by itself not be part of any other concept, we mark it that instances can be root. This will allow Canvas instances to be the roots of ASTs.

To indicate that Canvasses may hold Shapes, we’ll create a child collection of shapes. Again, hit Enter in the "« ... »" cell of the children section, type in the name of the child and the type of nodes it may hold. Do not forget about the* Control + Space key shortcut to bring up the code-completion dialog.
You should end up with a concepts definition like this:

Now we’ve created enough concepts to have a minimalistic language ready for use. Let’s give it a try.

### 2.9 An early test ride

We need to build the language before we can use it. In the future, after you’ve made a change to your language definition, please remember to repeat the process of "Rebuild" in so that the change could take effect. Right-click on the top-most node (the very most root, representing the whole project, including the Language and the Solution) in the Project View and choose **Rebuild Project**.

Once built, the language will be available for use in the model inside the sandbox Solution. Simply create a new **Canvas** by
right-clicking on the model and choosing the root concept to instantiate. Notice, the root concepts are those that show up in this menu and can be instantiated at the top-most level within a model.

Now give the Canvas instance a name and you’ve got your first painting. You can start using the concepts defined in the language to place visual shapes on the canvas.

The editing experience of our language as well as the layout is at this stage using some defaults, which we will change soon to get more customized look of the code.

Again, Control + Space is the keyboard shortcut to use heavily here. Use it each time you hesitate what to type. Notice that the code-completion dialog offers us the two types of shapes that we have created in the language.
You can insert additional shapes by pressing *Enter* at the end of the last shape in the list of shapes, which in our case is the closing `}` symbol for the "circle".

Cool! I hope you’re celebrating your success properly. Now, how about tuning those editors to make the code look better on the screen?

### 2.10 Editors

MPS uses a projectional editor. You might have noticed that already by feeling that the editor behaves slightly differently from what you would have expected. Unlike text-based languages, MPS never represents code as plain text. Instead, code in MPS means AST - Abstract Syntax Tree. Always. This comes with huge benefits for language design, language composability and non-parseable notations, which you can read more about in the MPS documentation. Here we should focus on the editing aspect of projectional languages. Since plain-text editors cannot represent ASTs reliably and since editing ASTs directly would be highly inconvenient, projectional languages have to provide editors for all Concepts that are part of the language. Sometimes even multiple alternative editors for a single Concept, but that’s not our goal here. MPS can do a good job in many cases to provide a default editor for Concepts that do not have one. This is very nice for
language prototyping. For convenient use by an end-user we, however, should spend some time preparing an explicit editor.

2.10.1 Shape

The Shape concept does not need an editor, since it is an abstract concept. So we will leave this one untouched.

2.10.2 Circle

For Circle we could create an editor that nicely places all necessary properties on a single line. We'll open the Circle concept in the editor, click the green '+' button in the lower left corner and pick Editor -> Concept Editor.

You'll get an empty editor definition for the Circle concept. Remember, Control + Space will be needed heavily to edit things here.

The editor for a concept in MPS consists of visual cells, each of which represents some piece of information belonging to the underlying concept. As nodes are composed hierarchically in an AST, so are their editors composed on the screen with sub-nodes’ editors nested inside their ancestors’ ones.

For Circle, we’d like to show value of all the properties \((x, y, radius)\) plus some arbitrary text around them, all on a single line.

So first we’ll choose a layout for these cells - indent layout will work just fine here. So hit Control + Space in the red cell and pick it up from the list. You may speed the search up by typing a square bracket followed by a minus symbol.
Now type "circle" to enter constant text that will be placed at the beginning of the line.

Hit Enter to create a new cell. Type "x:" to mark that the following cell contains the value of the x property.

Now you have to pick the x property from the code-completion menu to bind the cell to the right property.

Now you can continue on your own to insert cells holding constant text and well as values of the y and radius properties. Remember, Enter will insert new cells, Control + Space will bring up the code-completion menu. You should end up with an editor like this.

2.10.3 Square

Square also needs an editor. Open the Square concept in the editor, hit the ‘+’ symbol in the lower left corner and create a new Concept Editor. Following the instructions in the previous section, enter the editor definition as follows:

You can certainly do this on your own.
2.10.4 Canvas

The editor for the Canvas concept will be slightly different, since it spreads across multiple lines and holds a collection of child nodes. You, however, start in the same way as before, open the Canvas concept, hit the '+' symbol to create a new Concept Editor, insert an indent layout and enter some text to get the following:

Now bind the red cell to the name property. Since the property is inherited from the _INamedConcept _concept interface, it will be further down in the list, but it is surely there.

The following cell will hold a collection of shapes that have been added to the Canvas. Note how we compose editors here - Canvas only marks an area, in which individual shapes should be edited, and leaves that up to the shapes how they use that area.

Since the shapes should be organized vertically, each on a single line, you should pick vertical collection layout from the completion menu.

You need to bind the red cell to the shapes child collection of Canvas.

Now, to place the collection below the name of the Canvas, you should use the Alt + Enter shortcut the bring up the intention pop up and pick "Add On New Line".
You’ll get the final editor definition:

```
<default> editor for concept Canvas
node cell layout:
[- Painting { name } [> & shapes ] <]
/empty cell: <default>
]
inspected cell layout:
<choose cell model>
```

2.11 Second run

Now you can rebuild the project again (right-click on the very top node in the Project View) and look at the sandbox code in MyDrawing.

Look how the code layout changed:

```
Painting MyDrawing
circle x: 10 y: 20 radius: 30
square x: 100 y: 200 size: 50
```

It is the same code (AST), but it is organized on the screen differently.

2.12 Coloring

Now we should come back to Shape and add support for colors. Since both Circle and Square extend Shape, they will both inherit the color.

If for any reason you need to jump ahead to the generator, now is a good time to do so. You’ll just need to ignore all the places where we refer to colors in the generator chapter.

Since we would like to allow the users of our language to pick the color for the shape from a list of pre-defined colors, we can’t just use a textual property to hold the color value. Instead we’ll add a reference to Shape and have this reference point to one of the pre-defined color constants. Let’s start with creating such constants that will represent pre-defined colors.
2.12.1 Concept for colors

First we're going to create a concept that will represent a color constant. We'll call it Color and it will be rootable, so that we can place it inside models:

An editor is also needed:

2.12.2 Pre-defined colors

Now we need to provide concrete nodes of the Color concept, which will represent the individual color constants and which the user of our language will be able to refer to from their Canvases. We will utilize Accessory models for this. Accessory models are models inside a language definition that hold arbitrary nodes, which become part of the language and are visible
to the language users.

So first, we need to create an **Accessory model** in the language:

![Accessory model creation](image1)

The model demands a name. Please make sure the **stereotype** box is empty:

![Model name selection](image2)

The **Shapes** language, which declares the **Color** concept, is the only language that we need imported in the accessory model:

![Language import](image3)

Now you should be able to create color constants in the newly created accessory model:
2.12.3 First touch on dependencies

A crucial piece of knowledge in MPS is how to handle dependencies and imported languages. To display the dependencies of a module or a model, you have to navigate to it in the left-hand side **Project View** panel (Alt + 1 to open the panel):
Hit Alt + Enter (or right-click on it and choose Module/Model Properties) and you’ll get a dialog with the properties of the module/model. The Dependencies tab shows the modules/models that the current module/model depends on (aka import in Java, require in Ruby, etc.).

Yours is most likely empty now. You add elements using the ‘+’ button. In the small search dialog type a few characters of the name of the desired dependency to narrow down the search and hit Enter when you locate the correct one.

In the Used Languages section you indicate, which languages (syntaxes) you want to be able to use in your module/model.
We do not need to change anything here, so let's continue.

If you cannot type the code that we propose in the tutorial, most likely you have not properly set the dependencies. So please pay attention to the places, where the tutorial mentions dependencies and suggest that you should add a dependency or import a language.

### 2.12.4 Concept for color reference

Now our language needs a way to indicate the desired color of a shape in our code. Thus *Shape* should be referring to one of the color constants in the accessory model of the *Shapes* language. MPS will automatically populate the completion menu with all available color constants, whenever the user is about to specify a color for a *Shape*. 
ColorReference keeps a reference (a pointer going across the AST hierarchy) pointing to a single color constant (node of the Color concept).
In order to display and edit the colors we also need an editor for ColorReference.

The reference should simply display the name of the color constant that it refers to, so we pick target from the code-completion menu and specify that the name property of the color constant is what we want to show to the user.

You should get an editor definition as follows:
2.12.5 Updating Shape

The Shape concept is a good place to put our new ColorReference, since both Circle and Square will inherit it.

Shape may also define an editor component to define the editor for the color and Circle as well as Square will be able to reuse that editor component in their editors thus avoiding duplication.

Hit the ‘+’ symbol and create a new Editor Component.
The editor component definition a language that you are already familiar with and so you will be easily able to specify an indent layout, define a constant text cell followed by a cell bound to the color property.

### 2.13 Embedding the editor component

The editor component defined for Shape should now be added into the editors for Circle and Square.

The editor component becomes just another cell in the editor's layout.

### 2.14 A third run

Now it is the best time to rebuild the language. Right-click on the Language node in the Project View and select "Rebuild Language".

If you rebuilt the whole project instead of just the Language here, you’d get a couple of errors, since the Solution now misses the newly defined color properties. We’ll add those instantly.
Opening the MyDrawing program should show you the empty cells for colors in red. Try Control + Space in them and you will get a list of colors that you can pick from.

Now our language is fully defined. We can create a Canvas and add Circles and Squares to it, specifying their positions, sizes and colors. That’s quite an achievement for such a short time.

What we’re missing yet is the translation of these programs into Java, so that we could run them and see the shapes nicely drawn on the screen. If you continue you’ll soon realize that we’re almost there.

### 2.15 Generator

Our language now needs a generator so that we could generate code that could be compiled and run. We will choose BaseLanguage as the target for our DSL. BaseLanguage is a copy of Java distributed with MPS and so it can be easily transformed into textual Java sources for the Java compiler to compile into binaries. We could possibly choose any other target platform and language, provided you plug-in a definition of that target language into your project.

The generator will be very straightforward and will only need a few rules and a single mapping configuration.
Your language already contains a skeleton of an empty generator. You can open the mapping configuration, that will specifying what rule to apply when. We will be adding configuration entries here gradually.

Here's the idea behind the generator that we will implement:

- A **Canvas** gets translated into a Java class, which will extend Java's **JFrame** class and hold a **JPanel** that all the shapes will be drawn on
- Each **Shape** gets translated into a method call on the **Graphics** object to draw the shape on the **JPanel**
- A **ColorReference** gets translated into a reference to the appropriate color constant in the **java.awt.Color** class.

Let's start with the class for **Canvas**. You need to add a new entry in the *root mapping rules*, since **Canvas** is a root concept.
This will indicate that Canvas nodes should be replaced with a Java class. You need to use Alt + Enter the be able to select "New Root Template" from the pop-up menu.

The root template should be generating a Java class, so you need to pick "class".

This is the finishes root mapping rule. The map_Canvas is a name of the root template that was created in the generator. You should open it up so we could make changes to it.
First we need to set the dependencies of the **generator module** and the **generator model** as specified below:
Remember, Alt + Enter will bring up the properties of the node selected in the left-hand Project View. With the dependencies you can start typing the Java code that will be part of the generated Java class. We will then parametrize the code with values from the Canvas, to make it reflect the user’s intent. The class needs to extend JFrame.

Now you’ll add the main method to get a runnable Java class. You can use the "psvm" live template to enter the method quickly.
Inside the method, we'll need to instantiate `map_Canvas`.

We'll also need a method to initialize the frame. Type "method" and use Control + Space to complete the method definition:
Call the method `initialize()` and make sure that the **method is called** from `main`.

Notice the `canvas.initialize()` call in the `main()` method. Have you added it?

All the shapes will be drawn on a `JPanel`, so we now need to add one as a field.
Notice, we use an anonymous inner class to be able to customize the JPanel a bit.

**Important:** To create an anonymous inner class in BaseLanguage, position your cursor right after `new JPanel()` and before the ending semicolon. Then hit the "{" (left curly brace) key and MPS will add the ending "}" symbol. Now keep the cursor between these "{" and "}" symbols to add methods to the panel’s anonymous inner class.

We’ll override the `paintComponent` method of JPanel, because this is the method where Java allows us to easily draw the shapes on the JPanel. Hit Control/Cmd + O, while the cursor is inside the JPanel’s anonymous class body between the "{" and "}" symbols, to invoke the **Override method dialog** for the JPanel and select the `paintComponent` method.
Make sure your `paintComponent()` method is correctly nested inside the `JPanel`'s anonymous inner class as displayed in the screen-shots. Also make sure it is called `paintComponent`, not `paintComponents`.

Enter the following code into the method:
Now, please, fill in the `initialize` method and we have a template ready:
2.15.1 Parametrizing the template

The template currently holds no values specified by the user. The properties and children of Canvas should be inserted into the template through macros. MPS gives you three types of macros:

- **property macros** - to insert properties from the input model
- **node macros** - to replace nodes in the template with nodes from the input model
- **reference macros** - to adjust references in the template to point to nodes in the input model

We’ll gradually use all of these.

To start with, we’ll customize the name of the generated class and the **Title** of the frame with the **name** of the Canvas. Place the cursor on the name of the class - `map_Canvas` and hit **Alt + Enter**.

---

Now select the **node.name** property macro from the pop-up menu.
The "map_Canvas" text is now wrapped (annotated) with a property macro, which changes the name property to the name of the Canvas. The Inspector panel (Alt + 2) can be used to enter or modify the property macro, as well. Currently it returns the value of node.name, which is the name of the current Canvas.

Now you can wrap the "Title" text to customize the title of the frame. Using the Control/Cmd + Up Arrow key shortcut select the text "Title" without the surrounding " characters and with Alt + Enter insert the correct property macro:

The code should now look like this:
2.15.2 Drawing shapes

Our template assumes the code that draws shapes should be placed inside the `paintComponent` method of the `JPanel` field. The statement "System.out.println("Draw here");" serves as a placeholder for the real code that will draw all shapes. We will use the COPY_SRC macro to replace the placeholder statement with a statement that draws a single shape and we'll leverage the LOOP macro to repeat that for all shapes defined in the current Canvas.

Now, please select the placeholder statement including its closing semicolon using the Control/Cmd + Up Arrow key shortcut, hit Alt + Enter and choose the appropriate Node macro option to insert a LOOP macro to loop through all the child Shapes of the current Canvas.

Again, the Inspector (Alt + 2) shows the binding code.
The **LOOP macro** will repeat the "System.out.println("Draw here");" statement for each shape listed in `node.shapes`. We, however, need to have the "System.out.println("Draw here");" statements replaced with code that draws each of these shapes. The **COPY_SRC macro** will do just that. Please, select the statement again, hit `Alt + Enter` and choose Node macro.

```java
public class $[MyCanvas] extends JFrame {
    private JPanel panel = new JPanel() {
        @Override
        protected void paintComponent(Graphics graphics) {
            super.paintComponent(graphics);
            $[LOOPS$ System.out.println("Draw here");]
        }
    }
}
```

Type in **COPY_SRC** (Control + Space) and you get a macro that will replace "System.out.println("Draw here");" with the current shape for all shapes that the **LOOP macro** provides.
2.15.3 Generating circles

Now we get Canvas to be translated into a Java class and we also made a place for Shapes to add the code that will draw them. The time is up for us to define the actual translation rules for Shapes themselves, so that we get a "graphics.drawCircle()" method inserted in the generated code as a replacement for the Circle shape. You need to open the main mapping configuration and add a new entry to the "reduction rules" section:
Alt + Enter to create a new template:

The new `reduct_Circle` template will show up in the left-hand Project View.
You may also create a reduction rule for *Square*:

Open the `reduce_Circle` template. We now need to specify the Java code that will replace *Circles*. Remember, that the Java code will be placed in `map_Canvas` inside the `paintComponent` method.
First we’ll enter a `BlockStatement`, that will wrap our template:

```
template reduce_Circle
  input Circle

  parameters
  << ... >>

  content node:
  BloSta
```

We will need a local variable of the `Graphics` type as a place holder for the `paintComponent` parameter of the same name. And yet again a `BlockStatement`. 
To draw a circle in Java, we'll use the `Graphics` object to set the color first and that invoke its `drawOval` method. Please enter the code below:
Then use Control/Cmd / Up Arrow to select the inner BlockStatement, hit Alt + Enter and pick "Create Template Fragment" from the menu. This will mark the selected fragment of the code as the actual template, which will eventually be placed into map_Canvas.
The \textit{graphics} local variable needs our special attention. The current template as we've built is so far relies on \textbf{naming match} between the \texttt{reduce\_Circle} template, which refers to the \texttt{graphics} local variable, and the \texttt{map\_Canvas} template, which defined a \texttt{graphics} method parameter in the \texttt{paintComponent} method. This is a working solution, which, however, is quite fragile. Renaming either of the two \texttt{graphics} variables will result in broken generated code. Will implement a more robust solution for \texttt{Squares} right below the solution for \texttt{Circle}.

### 2.15.4 Parametrization

The code now needs to be parametrized with actual values from the \texttt{Circle} node.

The first value "10" should be replaced with the \textit{x} coordinate of the \texttt{Circle} node. A \texttt{property macro} will do that. Similarly the second value "10" should be replaced with the \textit{y} coordinate.
The third and fourth values "10" should both be replaced with the radius value of the circle node.

The `drawOval()` method’s third and fourth parameters specify the oval’s width and height, respectively. Since we want to draw a circle, we need to provide the same values for both arguments. This is why the property macros for the third and fourth parameters both specify `node.radius`.

Finally, the "Color.red" color placeholder reference should be replaced with the actual target of the color reference of the Circle node. We will use a reference macro to replace the references. Please put the cursor on the "red" word and hit Alt + Enter.
The reference macro will replace the reference to `red` with a reference to a node that we specify in the Inspector window:

The `referent` function returns either a `string` value (a name of the desired declaration to refer to) or node `<StaticFieldDeclaration>` (a node representing a `StaticFieldDeclaration`), because `red` itself is a reference to node `<StaticFieldDeclaration>`. In fact, all color constants in the `java.awt.Color` class are declared as `StaticFieldDeclaration`s.
You may try yourself opening the `java.awt.Color` (Control/Cmd + N, type "Color", maybe also check the "Include stubs and non-project models" checkbox) class and check that, for example, the "white" constant is a `StaticFieldDeclaration`.

So our task inside the reference macro will be to retrieve the `StaticFieldDeclaration` from within the `Color` class that corresponds with the color that the `Circle` has set as its `color` child. We will do this programmatically and so we will need to set dependencies on the right modules and models, which will allow us to write the required code.

First, the generator module has to depend on `BaseLanguage` in order to be able to refer to the `StaticFieldDeclaration` concept, which is declared in that language:

Second, the generator model:
With these languages imported we should be able to enter the code that discovers the correct static field declaration within the `Color` class:

```
The node/.../ construct allows you to obtain a node in the imported models of a specified concept represented with the given name. Since there only exists one `Color` class in JDK, the reference identified as node/Color/ will be unique and will be pointing into the model to the `Color` class.
Make sure you pick the right `Color` concept from the completion menu:
```

Using the `collections` language you can complete a concise query:
Since node is an instance of the Circle concept, node.color is the circle’s reference to the color (an instance of ColorReference) and node.color.target is a Color (an instance of the Shapes.Color concept) from the accessory model. In brief, the query searches the first static field declaration within the static field declarations of the java.awt.Color class with the same name as is the name of the color specified for the Circle.

The :eq: operator is a null-safe equality comparison, equivalent to (it.name != null && it.name.equals(node.color.target.name)). MPS also offers a :ne: operator, which is a negation of :eq:.

### 2.15.5 Reducing Squares

We’ll start mimicking how generator is done for Circles. Identically provide the following code for the reduce_Square template.

The values passed into "drawRect" should be replaced with property macros with the upperLeftX, upperLeftY and size properties of the Square.
The `drawRect()` method’s third and fourth parameters specify the rectangle’s width and height, respectively. Since we want to draw a square, we need to provide the same values for both arguments. This is why the property macros for the third and fourth parameters both specify size.

### 2.15.6 A more robust generation for Squares

As indicated earlier in the section for the Circle generator, we’ll use the `reduce_Square` template to properly tie the graphics local variable with the graphics parameter that the `map_Canvas` template generates. Relying on name match is not very robust.

We basically need to go through three steps:

1. Define a storage for created graphics parameters
2. Store the graphics parameter in the `map_Canvas` template
3. Retrieve the proper graphics parameter in the `reduce_Square` template

We’ll start by creating a mapping label in the mapping configuration. This will be the storage of `ParameterDeclarations`, each identified by the Canvas from which it was generated. You may also think of this mapping label as of a dictionary that maps Canvases to `ParameterDeclarations`.

The `graphicsParam` mapping label stores `ParameterDeclarations` mapped by Canvases that they belong to. The `map_Canvas` template now needs to store the graphics parameter in the mapping label. The `MAP_SRC` macro can be leveraged for that with great success:
Wrap the parameter declaration (including the type) with the `MAP_SRC` macro (`Alt + Enter, pick node macro, type MAP_SRC_`). Down in the _Inspector_ window you then select the mapping label to use for storing the generated parameter declaration. The current source node, which is an instance of the Canvas concept, will be used as the key to identify the generated graphics parameter declaration in the mapping label.

Finally, we need to retrieve the parameter declaration from the mapping label in the `reduce_Square` template. To indicate clearly that we are no longer relying on naming match, we can use a different name for the variable than `graphics`. We’ll go with `g` in the sample:

```java
public class $map_Canvas$ extends JFrame {
  private JPanel panel = new JPanel();
  @Override
  protected void paintComponent($SNAP_SRC$ graphicParam $COPY_SRC$ Graphics graphics) {
    super.paintComponent(graphics);
    $LOOP$[$COPY_SRC$ System.out.println("Draw here"); ]]
  }
}
```

```java
private void initialize() {
  this.setTitle("$title$").
}
```

![Map Node Macro](image)

If you `Alt + Enter` on the `g` reference and pick `reference macro`, you’ll be able to retrieve the proper `graphics` parameter from the mapping label:

```java
template reduce_Square
input Square

parameters
<< ... >>

content node:
{
    Graphics g = null;
    <TF>
    g.setColor(Color.->[red]);
    g.drawRect($[10], $[10], $[10], $[10]);
    
    
}"
Then repeat the reference macro creation for the second \texttt{g} reference. Both reference macros attached to the \texttt{g} variable reference that you have just created must specify in the \texttt{Inspector} the details on how to obtain the desired target of the reference. The \texttt{genContext} object gives access to helpful methods and properties of the current generation session. Use the \texttt{“get output for label and input”} operation on \texttt{genContext} and provide the \texttt{graphicsParam} mapping label as well as the \texttt{Canvas} holding the currently generated \texttt{Square}.

We could now replicate the retrieval of the \texttt{graphics} parameter for the \texttt{reduce\_Circle} template, as well.

### 2.16 Generating code

Now we're done defining the generator. If you rebuild the language, open \texttt{MyDrawing}, right-click it and choose "Preview Generated Text".
you’ll get a nicely structured Java code that properly initializes a `JFrame` and draws all the shapes:
2.17 Running the code

It is nice to see generated code, but you might actually prefer seeing it running. MPS can compile and run generated Java code easily. We only need to indicate that Canvas is generated into a runnable Java class and thus Canvas itself should be treated as runnable, or as a "main" class. We only need to make Canvas implement the IMainClass interface and MPS will take care of the rest. The IMainClass interface comes from the jetbrains.mps.execution.util language and so we need to add it to the list of dependencies of our language:
Use the Alt + Enter to get the properties dialogs. Notice that the language needs to be marked as Extends. The Canvas concept can now have the IMainClass interface added to the implements section.

Rebuild the language and then right-click on MyDrawing in the Project View and click "Run".

After creating a new drawing you may actually need to compile it first. If the Run 'Node MyDrawing' is not shown in the context menu, it is an indication that MyDrawing has not yet been compiled. Please right-click on the sandbox solution or on the model holding MyDrawing and choose Make. As soon as it finishes you should be able to run your drawing.
You will get a running Java application with your drawing on it as a reward for your efforts.
2.18 What to do next

Congratulations! You’ve just completed your introductory tutorial into MPS.
Now you can continue on your own adding more shapes to the language. Point, Line, Triangle, Rectangle or shapes with colorful fills might be nice additions to our little language.
If you want to understand MPS more thoroughly, it might be a good time to try the in-depth Calculator Tutorial, which explores many of the advanced concepts and will teach you much more about code generation, type-system and scoping.
Part III

Generator
Chapter 3

Generator Demos

3.1 Generator Tutorial

Welcome to the updated Generator Tutorial, which will guide you through the process of defining and extending language generators in MPS.

The tutorial has seven parts with gradually increasing complexity. All the demos share the same underlying story - a transformation of XML code into Java Swing components. Each demo is based on the previous one and demonstrates more advanced (or sometimes just different) features, practices and approaches.

All the tutorial demos are bundled with MPS distributions as a sample project, named tutorial-demos. This will help you especially, if you want to review the existing code or skip some of the demos.

Although all the demos are logically related, for educational purposes, we will create a fresh generator in each of the demos. Since generators don’t exist in vacuum and always belong to a concrete language, we will create a new MPS project structure for each of the demos to wrap the generator. Those languages (except for demo 6 and 7) do not introduce their own new Basic notions (aka syntax). Instead, they give new semantic to the existing concepts. For instance, the xml element with name ‘button’ is not just an xml element any more, but a ‘virtual concept’ representing a GUI component button.

Note that this is not a typical way to do Language Oriented Programming. Languages are rarely purely ‘virtual’, defining the semantics without adding or changing any syntax elements. Choosing this approach for our demos lets us focus solely on our main topic - the generators.

3.2 How to Use This Documentation

The best approach is to Download MPS on your computer, set-up a new empty demo project and carry out all the steps described in demos. This way you will gain great experience in generators development and beyond.

The second option is to read the documentation and, perhaps, browse generator demo project included in the MPS distribution. The generator demo project includes all the demo languages and generators, as well as the test models, which we are going to create in these demos.

The generator demo project is located at:
{mps}/samples/generator_demo

3.3 Setting-Up Demo Project

If you go the more educative way and plan to build the code from scratch, you need to do some little setup work before starting the individual exercises.

3.3.1 New Project

- choose File->New Project
- create a new empty project
- enter project name: ’generator_demo’
In this demos we are going to define new semantics for xml elements, create test models with some xml in them and generate java code from that xml. The syntax for the XML language has already been defined for you in the SampleXML sample language, which is also bundled with MPS distributions as a sample project. The SampleXML project is located at:

```
$HOME/MPSSamples3.0/samples/sampleXML
```

You can obviously try the SampleXML project separately by opening it in MPS. In our demos, however, we will reuse the jetbrains.mps.sampleXML language only. To start with, let’s add the jetbrains.mps.sampleXML language to the 'generator_demo' project.

- Right-click on the generator_demo node in the left-hand side Project View panel and choose Project Paths. The Alt + Enter key shortcut will have the same effect.
- Hit the + button to add a new module
- Choose the $HOME/MPSSamples.3.0/sampleXML/languages/sampleXML/jetbrains.mps.sampleXML.mpl file
Now the `jetbrains.mps.sampleXML` language is easily accessible.

Right-click the sampleXML language node and choose "Rebuild language ..." so that the language is ready to be used.

### 3.3.2 New Demo Solution

- choose New Solution in project’s popup menu
- enter the solution name: ‘test_models’
3.3.3 New Test Model

- choose New Model solution’s popup menu
- enter the model name: ‘generator_demo.test1’

- add the ‘jetbrains.mps.sampleXML’ language to the Used Language section in the model properties dialog

Now you should have a solution with a single model in it and be ready for writing some xml using the just imported simpleXML language.

In the ‘text1’ model create new Document node and give it name 'Button'.

- create a Document ‘Button’ in the model ‘test1’
- add root element ‘button’ to this document
- in the same manner create Document ‘Label’ containing element ‘label’
3.4 Test Generation

- choose Generate Text in popup menu of the model 'test1'

The result of the text generation will be shown as new tabs in the MPS editor and it will be more or less identical to the input.

Now we are all set to go on and define semantics for those otherwise meaningless xml concepts!

3.5 What’s Inside

3.5.1 Generator User Guide Demo1

This is an easy start. We will transform two types of XML elements into labels and buttons and practice how to use root mapping rules, root template, property-macro, SWITCH-macro and a template switch along the way.

3.5.2 Generator User Guide Demo2

This demo will teach you the usage of conditional root rules, LOOP-macros and abandon root rules.

3.5.3 Generator User Guide Demo3

Demonstrates usage of template declarations and template fragments; mapping labels; the 'unique name' service; reference-macros and IF/INCLUDE/MAP_SRC macros.

3.5.4 Generator User Guide Demo4

Demonstrates usage of reduction rules, COPY_SRC-macros and more advanced reference-macros with reference re-solving by mapping labels.

3.5.5 Generator User Guide Demo5

Explains how to use generation scripts and how to create utility classes in the generator.
3.5.6 Generator User Guide Demo6

In this demo we will create a 'real' language defining its own higher-level concepts (button and label) and see how languages are integrated together.
This demo explains in greater details the generation process in MPS and demonstrates the usage of transient models and the generation tracer tool.

3.5.7 Generator User Guide Demo7

Demonstrates usage of weaving rule; template declaration and template fragment; mapping label; 'unique name' service; reference-macro and IF/INCLUDE/MAP_SRC macros.
Chapter 4

Generator User Guide Demo1

4.1 Generator User Guide Demo 1

In this demo we will create a generator that will generate a java application for each input XML Document. We’ll re-use the ‘generator_demo.test1’ model in solution ‘test_models’ that we have created Generator Demos. This model uses the ‘jetbrains.mps.sampleXML’ language and contains two XML Documents - a ‘Button’ and a ‘Label’. We will apply a root mapping rule to each of these documents in order to generate the output java application. In the application’s main method the generated code will instantiate a swing frame and add a requested swing component to its content pane. The component is going to be either a JButton or a JLabel, depending on the name of the root element in the input document.
So, our ‘Button’ XML Document will be translated into a Java application that will show a button inside a frame.

4.2 New Language

The first thing to do is to create a language that will enhance simpleXML with a new generator. Instead of generating XML, we now need to generate Java code.

- choose New -> Language in project’s popup menu
- enter language name: ‘generator_demo.demoLang1’

- add the ‘jetbrains.mps.sampleXML’ as well as ‘jetbrains.mps.baseLanguage’ languages as extended languages into the Dependencies section in the demoLang1 language properties dialog
4.3 New Generator

MPS has created an empty generator module inside the `generator_demo.demoLang1` language as well as a generator model `main@generator` containing an empty mapping configuration node `main`. If not, you can easily have them created through the context menu for the `generator_demo.demoLang1` language.

Mapping configurations serve as generator’s ‘entry points’ where all generator rules are declared and from where templates can be referenced. Templates contain snippets of target code with declared placeholders for values to fill in from the input model. The templates are the core of MPS model-to-model transformation process.

The model stereotype `generator` (shown after symbol `@` in the model name) allows MPS to tell a generator model from a regular model. Any node in generator model is interpreted as a template unless that node is a part of the generator language itself.

For instance, a mapping configuration node is a part of the generator language (the ‘MappingConfiguration’ concept is declared in language `jetbrains.mps.lang.generator`). The class node, which we will create in the next step, is not a part of the generator language and so will be used as a template in our generator.

4.4 Root Mapping Rule

Let’s create our first rule which will generate a Java application class for each input XML Document.

- open the mapping configuration named ‘main’ in the editor
- add a new rule in the root mapping rules section (press Insert or Enter with cursor is in the section)
- choose the rule’s applicable concept: ‘Document’
As we want to generate a class for each input Document, the template for this rule that we have to specify as part of the mapping rule is going to be a Class - i.e. an instance of the 'ClassConcept' concept defined in BaseLanguage. The easiest way to create that template is to apply intention (see the on the left edge of the editor?):

- press Alt-Enter while cursor is in the red field `<no template>`
- choose New Root Template
- choose class.
- open the newly created class in the editor (use Ctrl+left-click or Ctrl+B on the reference)

4.5 Property-macro

A template needs to be parametrized with values from the input model to be of any good. MPS gives you several types of macros to hook values from the input model into your templates. Property-macros, which we are going to start with, are used to compute values of output model nodes’ properties at the time of generation.

Let’s start with something very trivial - we want to generate an output class with the same name as was the name of the input Document. Thus we will create a property-macro for the output class’s name:

- put the cursor somewhere inside the name of the class
- press Ctrl-Shift+M and choose property macro in menu
- alternatively you could use Intentions through the Alt + Enter command
- select the macro node in editor (rendered as ‘$’ symbol) and open the inspector window for it (Alt-Shift+)
• the function can refer to the original Document node through the node parameter
• this value function will in our case return the name of original Document

Type of input node

Type of the node in this case is ‘node<Document>’ (to find out type of node select it in code and press Ctrl-Shift-T). MPS knows the type of the input node, because this template’s header explicitly specifies Document as the template input.

4.6 The main() method

Since we plan to use Java Swing components in the generated code, we need to import the MPS java_stub models for the ‘java.awt’ and ‘javax.swing’ packages into the generator model.

• press Ctrl-M (import model)
• choose ‘java.awt@java_stub’ model
• press Ctrl-M again
• choose ‘javax.swing@java_stub’ model

With the imports out of the way, we can now create the main() method. Typing ‘psvm’ will trigger an automated action that inserts an empty main method for you. Alternatively you can just type the method signature yourself.
4.7 SWITCH-macro

You might have wondered why we pass null into the container.add() method, while instead we need to pass in either a new Button or a Label, depending on the actual value in the input XML Document. That is the last thing we have to do in this template - to replace the 'null' argument in the 'container.add(null)' expression with expression creating a swing component.

We will generate 'new JButton()' expression if the root element in the input document is named 'button', and 'new JLabel()' if the root element is named 'label'.

To perform this replacement we will 'wrap' the 'null' argument into a SWITCH-macro.

A SWITCH-macro replaces the 'wrapped' template node with a new node. The replacing node is created depending on which case-condition is satisfied in the associated template switch.

To begin with, let's create new template switch named 'switch_JComponentByElementName'.

```java
public class map_Document { 
    public static void main(String[] args) {
        JFrame frame = new JFrame("Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container container = frame.getContentPane();
        container.setLayout(new FlowLayout());
        container.add(null);
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }
}
```

Tip: To start entering a condition code press Insert.

The '<...>' things on next screenshot are called in-line template. See also the note below the screenshot.
That’s it. We are testing element’s name. If it is ‘button’ we generate ‘new JButton()’. Identically, if it is ‘label’ we generate ‘new JLabel()’. If it is neither ‘button’ nor ‘label’ we are raising an error.

In-line template

Creation of in-line template like `<T new JButton( ) T>` is not a trivial task. If you stumbled trying to do that consider the following diagram.

Now we are ready to attach the SWITCH-macro to the ‘null’ argument node in the in the ‘container.add(null)’ expression.

- return to the ‘main()’ method in the ‘Document’ template
- select the ‘null’ node in the ‘container.add(null)’ expression
- press Ctrl-Shift-M to add an abstract node-macro
- press Ctrl+Space to invoke completion menu and choose $SWITCH$
- go to inspector of the $SWITCH$ macro
- enter the code (see below) in the mapped node function (press Insert to start entering) to set the node on which to make a decision regarding the switch
- make reference to the ‘switch__JComponentByElementName’ template switch node that we created earlier to refer the switching process to
What does the mapped node function do and why do we need it?
Up to this point our input node inside the map_Document template has been a Document. But the template switch 'switch_JComponentByElementName' expects an Element as its input, not a Document. Thus we have to replace the input node - a Document, with a new input node - an Element. The expression 'node.rootElement' returns the root element of the current input document. The root element will become the new input node applicable inside the template switch.

4.8 Getting Ready for the First Test
It is very important to remember that whatever changes you made in generator models that only become visible to other modules after the generator has been made. Your last action after making changes should always be the same: Make the generator.
There is just one model in our generator so hit Shift + Control + F9 or Shift + Cmd + F9 (make model) when you finish editing the template.

4.9 Generating Test Model
Let’s return to the ‘test1’ model in the ‘test_models’ solution. The solution contains several XML Documents and we have just defined the semantics for XML nodes in demoLang1. Unfortunately, MPS cannot combine these two pieces without us telling it first. MPS doesn’t understand that model ‘test1’ is now written in two languages - sampleXML and demoLang1, because we’re not using demoLang1 explicitly anywhere in ‘test1’.
Since demoLang1 doesn’t define any concepts of its own and so we cannot really use demoLang1 model ‘test1’, therefore we will explicitly tell MPS to include the demoLang1 generator in the generation process for model ‘test1’.

- open properties of model ‘test1’
- add language ‘generator_demo.L1’ to the Languages Engaged On Generation section
Run the **Preview Generated Files** command in the 'test1' model’s popup menu. MPS will generate two java files and show them in the editor:

```bash
$PROJECT_HOME\MPS_generator_demo\solutions\test_models\source_gen\generator_demo\test1\Button.java
$PROJECT_HOME\MPS_generator_demo\solutions\test_models\source_gen\generator_demo\test1\Label.java
```

MPS can also compile your generated Java sources into Java byte code as part of **make** or **rebuild** of the 'test1' model. To enable Java compilation of generated sources you need to add a dependency on **JDK** to the 'test_models' solution:
Chapter 5

Generator User Guide Demo2

5.1 Generator User Guide Demo 2

In this demo we will again generate a Java Swing application, but unlike in previous Demo 1, this time it is going to be a single Java application per input model (in Demo 1 we generated a separate Java application for each input XML Document). The single Java application that Demo 2 creates, will contain all components mentioned in all XML Documents of the input model.

5.2 New Language

We need to do some little technical setup first:

- create a new language: ‘generator_demo.demoLang2’
- in the language properties dialog add an extends dependency on ‘jetbrains.mps.sampleXML’ and ‘jetbrains.mps.baseLanguage’
- create a new generator for this language, if it has not been created automatically (see Demo 1 for details).

As the code that we are going to generate is almost identical to that in Demo 1, we will copy the application class template from the demoLang1 generator:

- go to model ‘main@generator’ in demoLang1
- select the ‘map_Document’ template in project tree and copy it to clipboard
- return to model ‘main@generator’ in demoLang2 (select this model in tree) and paste from clipboard
importing model on copy-paste

The 'Imports and Languages' dialog will offer to import some additional models into our new demoLang2 generator model. Accept all but model 'main@generator' in demoLang1 (our new generator is not going to depend on the demoLang1 generator):

- uncheck the demoLang1 generator model
- press OK

• in the same manner copy the template switch 'switch_JComponentByElementName' from demoLang1 to the demoLang2 generator model

5.3 Conditional Root Rule

The process of code generation in MPS can be viewed as a series of model-to-model transformations. An AST in one language is gradually transformed into an AST expressed in concepts from another language. In each step individual concepts from one language get translated (reduced) into concepts from another language. Conditional Root Rule is somewhat special in that it creates new nodes out of nothing. It generates a brand new root node in the output model using a template referred to in this rule without having any original node in the input model.

• go to demoLang2 generator and open the mapping configuration 'main' in editor
• add new Conditional Root Rule (press Insert or Enter while cursor is in conditional root rules section)
• make reference to the 'map_Document' template so that this template is used to generate the resulting class
• open the `map_Document` template (Control + <left-click> or Cmd + <left-click> on the reference in the editor) so we can tweak it a bit for it to work in the changed context

• in the template’s header remove the mention of `Document` - press Del while cursor is on the word ‘Document’ (we are using this template in Conditional Root Rule, which no longer passed a `Document` as the template’s input)

• remove the property-macro from the class name, as now we only get a single class, we can hard-code its name

• rename the generated class to ‘DemoApp’

• open the Inspector for the `SWITCH` macro and resolve the reference to the template_switch by hitting Control + Space
5.4 LOOP-macro

The SWITCH-macro in 'DemoApp' template is expecting an XML Document on the input, since this is how we implemented it in Demo 1. It then extracts the Document's root element and applies a template switch to it. However, there may be multiple Documents in the input model (in fact in our demo app will have two - a button and a label), so we need to gradually process them all in sequence and generate an appropriate "container.add(null);" statement for each of these Documents. To provide the SWITCH-macro with a single Document node, we will wrap the 'container.add(..);' statement inside a LOOP-macro:

- in the template code select the statement 'container.add($SWITCH$[null]);' (with your cursor placed inside the statement, use Ctrl/Cmd+W to expand block selection)
  
  ! make sure that +whole statement+ is 'wrapped', i.e. including the ';' symbol
  
- press Ctrl-Shift+M to add node-macro
- choose $LOOP$ from the auto-completion menu (Ctrl+Space)
- enter code in the LOOP's mapped nodes function as below

The mapped nodes function will find all Documents in the input model and return a sequence of these Documents to the generator (the return statement can be omitted in BaseLanguage). Then, the generator will iterate through this sequence, create 'container.add(..);' statement for each Document and gradually pass each Document as an input node to the SWITCH-macro.

That's it. The last step, as usual, is hitting Shift-F9 to make the generator.

5.5 Running First Test

To test the demoLang2 generator we need the exactly same input as in Demo 1 except that we are going to 'engage' different language on its generation.

- go to the 'test1' model in 'test_models' solution (select it in project tree)
- choose Clone Model in the popup menu
- enter the new model name: 'generator_demo.test2'
- in properties dialog of the 'test2' model, in the Languages Engaged on Generation section:
  - remove 'generator_demo.L1' language
  - add 'generator_demo.L2' language

Generate the 'test2' model and preview the generated sources.
5.6 Abandon Root Rule

Note that the generated output also contains two ‘generated’ XML files: Button.xml and Label.xml. MPS did not generate those XML files in previous Demo 1. Why does it do so in this demo?
Because in Demo 1 we have defined explicit mapping of input Documents to output Classes using Root Mapping Rule, the input was consumed and the xml files were not generated.
In Demo 2 we do not explicitly map input Documents to any output roots and so they remain in the model until the text-generation phase, which turns them into xml files.

If an input root node is not explicitly mapped into an output root node, then MPS will copy this input root node to output model.

To prevent copying Documents to the output model we will use the Abandon Root Rule.

- return to demoLang2 generator and open mapping configuration ‘main’ in editor
- add a new Abandon Root Rule - press Insert while cursor is in the abandon roots section
- choose ‘Document’ as the concept for application of the rule

After rebuilding the project you will no longer get xml files on the output.
Chapter 6

Generator User Guide Demo3

6.1 Generator User Guide Demo 3

In Demo 2 we were generating java statements like:

```java
container.add(new JButton());
```

to create a Swing component and add it to the application’s content pane.

In Demo 3 we are going to add support for component properties, which will require generation of more complex initialization
code - not just a constructor invocation. Moreover, the generated property initialization code is going to be different for
different types of components.

Therefore we will choose a generation strategy that is capable of handling such requirements.

6.2 New Language

Again, we need to setup a new language and copy some of the Demo 2 generator artifacts to it.

- create a new language: ‘generator_demo.demoLang3’
- in the language properties dialog add an extended dependency on ‘jetbrains.mps.sampleXML’ as well as on
  ‘jetbrains.mps.baseLanguage’
- create a new generator for this language, if it does not exist (see Demo 1 for details)
- delete the mapping configuration ‘main’ from the demoLang3 generator (as in Demo 2, we will copy all needed parts
  from the demoLang2 generator to the demoLang3 generator)
- copy-paste the mapping configuration ‘main’ from the demoLang2 generator to the demoLang3 generator

```
when pasting nodes - don’t forget to exclude the demoLang2 generator model from imports

See Demo 2 for details.
```

- copy-paste the ‘DemoApp’ template from the demoLang2 generator to the demoLang3 generator

6.3 addContent(container)

The generator will resemble in many aspects the one we created in Demo 2. So we’ll be using the conditional root rule
to insert the DemoApp class into the output model. Initially, we need to make some tweaks to the DemoApp class.

- open the ‘DemoApp’ template
- add a static void method ‘addContent(Container)’
- in the ‘main()’ method find the statement:

```java
$LOOP$[container.add($SWITCH$[null]);
```
• replace the statement above with the statement below:

```java
addContent(container);
```

Now, the `addContent()` method is supposed to add all components to the `JFrame`. So it will call `container.add()` and pass in initialized visual components in turn - one for each input XML Document. So we'll add a dummy method that we can use in the code and have the generator replace it with real component initialization code.

```java
public class DemoApp {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container container = frame.getContentPane();
        container.setLayout(new FlowLayout());
        addContent(container);
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }

    private static void addContent(Container container) {
    }
}
```

### 6.4 Generating components

In order to have a component initialization method generated for each input `Element`, we will wrap the `component()` method declaration with a `LOOP` macro. The `LOOP` macro will iterate over all `Elements` in the input model and generate a method for each of them.

```java
private static void addContent(Container container) {
    container.add(component());
}
```

```java
 LOOP private static Component component() {
   return null;
 }
```

Similarly, we need to call these generated methods for each input `Element` and add the returned components into the container, so we need to use another `LOOP` macro around the call to the `container.add(component());` method inside the `addContent()` method.
Last but not the least, we have to replace the call to the dummy `component()` method with a call to the real generated method corresponding to the same input `Element`. So select `component` (without the ending parentheses) and hit `Control + Shift + M` to insert a reference macro.

This is where we can’t move forward just yet. We have no clue how to resolve the method declaration corresponding to the current node so that we could refer to it from the reference macro. We will employ mapping labels at this stage. They will serve as registries for method declarations, accessible by the `Element` that they were created for. So our reference macro will be able to retrieve the corresponding method declaration from there.

To define a mapping label, go back to the ‘main’ mapping configuration:

```
mapping label
```

Mapping label’s `input concept` and `output concept` are optional parameters which are used to provide static type checking in expressions which are making use of mapping label to find output node.

In our case we are going to attach the mapping label to rules, which generate `StaticMethodDeclaration`. That’s why we have chosen `StaticMethodDeclaration` as the label’s output concept.

The expression (we will use it soon):

```
genContext.get output "method"
```

will then have type: ‘node<StaticMethodDeclaration>’.

The fact that the ‘get output’ expression has type, provides no benefits for us in this particular case, but in general it is a very helpful feature.
Obviously now we need to complete the generation of method declarations and properly store them in the method mapping label.

### 6.5 Inserting components

Just like in Demo 2, we’ll use a \texttt{SWITCH} macro to accommodate for different types of \texttt{Elements} when generating the methods for them. So right-click on the generator model and choose \textit{New -> template switch}. Depending on the \textit{name} of the \texttt{Element}, we either generate code for a button or for a label.

This time, since the generated code is more complex, in-line templates would not be enough to capture the logic, so we need standalone templates for both generating a button and a label - named \texttt{insert\_Button} and \texttt{insert\_Label} respectively. Create them by right-clicking on the generator model and choosing \textit{New -> template declaration}.

Templates are snippets of the code to generate, parametrized with values calculated from the input model. In our case, we need to generate a new static method that will be added to the \texttt{DemoApp} class and will have a signature compatible with the \texttt{component()} method declared in the \texttt{DemoApp} class template. So what we’ll do now for a button is to create a dummy class with a static method in it. This static method will be made a \textit{template fragment}, so that only the method and not the whole class will be inserted into the target model.

First, choose \texttt{ClassConcept} as the content node:
Enter the code below declaring the static method with the signature that we want.

```java
template insertButton
input <any node>
parameters
<< ... >>

content node:
```public static Component createComponent() {
    JButton component = new JButton();
    return component;
}
```}

Now select the whole method declaration, hit Alt + Enter and pick Create Template Fragment from the menu. This will indicate that it is only our method declaration that should be used as a template. The surrounding class serves merely as context for the method.

This is the time when we register all methods generated with this template into the method mapping label, so that we can retrieve these later when generating calls to these methods. Simply enter the desired mapping label in the Inspector window for the template fragment (the `<TF visual element`).
At the moment we are generating methods for all input Elements with identical name - `createComponent`, which would cause Java compile-time troubles, if we have more than one Element in the input model. We need to give each generated method a unique name and we’ll do it through a property macro.

We’re using the genContext capabilities to generate unique identifiers. The `templateValue` parameter is going to be "createComponent" (i.e. the method’s name as written in template).

Now, please, create template declaration for creating a JLabel.

Once done, we should be able to complete the SWITCH template declaration.

After fixing the SWITCH macro in the DempApp class template and making the language, we should be able to give our new generator a first try.
6.6 New Test Model

Let's create a new input test model:

- go to the 'test_models' solution
- clone the model 'test2' to model 'test3'
- in the model properties dialog replace 'engaged on generation' language demoLang2 -> demoLang3 (see Demo 2 for details)
- open the 'Button' document (from model 'test3') in the editor
- add an attribute 'text="Hello"' to the 'button' element
- add an attribute 'enabled="false"' to the 'button' element
- add an attribute 'text="world!"' to the 'label' element in the 'Label' document
- add an attribute 'background="orange"' to the 'label' element in the 'Label' document

When you hit Preview generated code, you should get a valid Java application.
Notice the uniqueness of the `createComponent()` methods.
Now, let’s define the semantics for the extra XML attributes.

## 6.7 Adding an $\text{IF}\$ macro

Let’s focus on the `insert_Button` and `insert_Label` templates. These define the code that will initialize the Swing components based on the values specified in the input XML `Element`. So far we ignore all the attributes in the input model, but this will change now. Focusing on `insert_Label` for now, if the input `Element` has an attribute called ‘text’, we will generate a statement:

```java
component.setText( _text_ )
```

where `text` is the string specified in the ‘text’ attribute of the input `Element`.

- at the top specify `Element` as the input of the template so that MPS can type-check code involving the template’s current node
- enter the following code in the body of the ‘createComponent()’ method
- create IF-macro around the ‘component.setText("text");’ statement (select whole ! statement, press Ctrl-Shift+M)
- enter the code into the condition function of IF-macro, which will check the presence of the ‘text’ attribute
• create property-macro inside the string literal "text", because we need to parametrize the value with the actual attribute value from the input model

• in the value function of the property-macro enter code that returns the value of the 'text' attribute of the input Element.

The same steps should now be repeated to the insert_Button template.

6.8 Reusable template

We can go on adding in the same manner support for more and more component properties, but this way we are going to end up with a lot of duplicated code in our templates.

A better idea is to create one template containing some common code and re-use it in template of each component. For example, let's add support for the property 'enabled'. We'll create a "shared" template and re-use it in other templates using an INCLUDE-macro:

• go to the 'main@generator' model in demoLang3 generator (select in project tree)

• create new template declaration node using Create Root Node in model popup menu

• name it 'include_ComponentProperties'

• choose input - Element

• choose StatementList as the template’s content node:
• in the statement list create a variable declaration:

```java
JComponent component = null;
```

The name of variable is important - generator will resolve references by this name.

• add a block-statement (press <Enter> after the variable declaration, type '{', press Ctrl+<space> to auto-complete)

• create a statement:

```java
component.setEnabled(false);
```

inside the block-statement

• mark the block-statement (i.e. excluding the declaration of the component variable) as a template fragment:

```java
template include_ComponentProperties
  input Element
  parameters
    << ... >>
  content node:
    if (JComponent component = null;
    { component.setEnabled(false); }
```

why use the _block-statement_?

We don’t really have to use a block statement, but choosing so will make it easy for us to add more statements to this template fragment in the future.

6.9 MAP_SRC-macro

As in case of the ‘text’ property, the ‘setEnabled()’ method call generation should be conditional - this statement should only be generated if the input element posses the attribute ‘enabled’.

This time, to achieve the conditional generation, instead of an IF-macro we will use a MAP_SRC-macro. As we will see, MAP_SRC-macros have several advantages over IF-macros in this case.

• create a MAP_SRC-macro (it should wrap the whole statement)

• enter code in the macro’s mapped node function as shown:
The **mapped node** function returns the node for the 'enabled' attribute, if present, and *null* otherwise. If the **mapped node** function in a MAP_SRC-macro returns *null*, the generator then ignores the node wrapped by this macro (just like an **If macro** would). However, if the mapped node is not *null*, then it becomes the current **input node**, while processing the wrapped template code. This helps us get hold of the attribute making creation of the property-macro a bit easier compared to using an IF-macro - we don’t have to find the ‘enabled’ attribute again, this attribute is already our **input node**. Now, to set the correct value to the template:

- attach a property-macro to the boolean constant 'false'
- enter the code into its **value** function (note that this time the **value** function expects a boolean return value):

```java
content node:
JComponent component = null;
<TF {
  $MAP_SRC$component.setEnabled(false);}
}
```

### 6.10 INCLUDE-macro

We will use INCLUDE-macro to specify places in code, into which we want to insert the 'component.setEnabled(..);' statement during generation.

- open the 'insert_Button' template in the editor
- insert a new empty line just after the statement

```
component.setText("text");
```

- insert an INCLUDE-macro on the empty line (Control + Shift + M)
- make a reference to the 'include_ComponentProperties' template in the macro’s inspector:
create a similar macro in the 'insert_Label' template

re-generate generator model (Shift+F9)

6.11 Testing what we have

Generating code for the 'test3' model will render a valid Java application:

```
package test_models.test3;

//{{Generated by MPS with/
import javax.swing.JFrame;
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.Component;
import java.awt.Button;
import java.awt.Label;

public class DemoApp {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container container = frame.getContentPane();
        container.setLayout(new FlowLayout());
        addContainer(container);
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }

    private static void addContainer(Container container) {
        container.add((createComponent_a()));
        container.add((createComponent_a_0()));
    }

    public static Component createComponent_a() {
        JButton component = new JButton();
        component.setText("Hello");
        component.setEnabled(false);
        return component;
    }

    public static Component createComponent_a_0() {
        JLabel component = new JLabel();
        component.setText("World!");
        return component;
    }
}
```

We correctly get the `setEnabled()` method generated only for the button, but not for the label.

6.12 Reference-macro (Resolving by Name)

**what references?**

A method-call node or variable-usage node contains a reference to a method or a variable declaration.

In many cases reference cannot be resolved automatically and this is where reference-macros come in handy. For example, let's add support for the 'background' property into our generator:
• open the 'include_ComponentProperties' template
• add a block-statement just after the 'component.setEnabled(false);' statement
• enter the following code inside that block-statement:

```java
component.setOpaque(true);
component.setBackground(Color.black);
```

• wrap the block-statement in a MAP_SRC-macro
• enter the code into the macro’s mapped node function as shown:

```java
template include_ComponentProperties
input Element
parameters
<< ... >>

candidate node: JComponent component = null;
<T>
\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n```

	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```
	```

In this example, the referent function has combined return type:

```
JOIN(node<StaticFieldDeclaration> | String)
```

having the either-or semantics. That means that we have two alternatives for what to return from the function:

1. we can find a suitable static field declaration in class ‘Color’ and return that node
2. or we can return a reference info string - a name of static field declaration (name of color) and let it up to MPS to find the static field for us

The 2nd option, of course, is much more attractive. Thus we are returning a value of the ‘background’ attribute - the name of the desired color.
6.13 The final test

Now, after a language rebuild the 'test3' model will be generated into a Java application that now takes into account all the input element's attributes.

```java
package test_models.test3;

import javax.swing.JFrame;
import javax.swing.JPanel;
import java.awt.FlowLayout;
import java.awt.Component;
import javax.swing.JButton;
import javax.swing.J2Label;
import java.awt.Color;

public class DemoApp {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container container = frame.getContentPane();
        container.setLayout(new FlowLayout());
        addComponent(container);
        frame.pack();
        frame.setVisibleRelativeTo(null);
    }

    private static void addComponent(Container container) {
        container.add(createComponent_a1());
        container.add(createComponent_a_0());
    }

    public static Component createComponent_a1() {
        JButton component = new JButton();
        component.setText("Hello");
        component.setEnabled(false);
        return component;
    }

    public static Component createComponent_a_0() {
        J2Label component = new J2Label();
        component.setText("world!");
        component.setOpaque(true);
        component.setBackground(Color.orange);
        return component;
    }
}
```
Chapter 7

Generator User Guide Demo4

7.1 Generator User Guide Demo 4

In this demo, we will further evolve the demoLang3 generator (see Generator User Guide Demo3) and add support for a *panel* component. Unlike *buttons* or *labels*, *panels* can contain other components, including other panels. So we get recursion in the generator. We will see how *reduction* can help to solve problems of this kind.

7.2 New Language

We’ll reuse a great part the demoLang3 generator in demoLang4.

- create new language ‘generator_demo.demoLang4’
- in language properties dialog add extended languages : 'jetbrains.mps.sampleXML' and 'jetbrains.mps.baseLanguage'
- create a new generator for this language if it does not exist (see Generator User Guide Demo1 for details)
- delete the (empty) mapping configuration ‘main’ from the demoLang4 generator (we will copy all needed parts from the demoLang3 generator into the demoLang4 generator)
- copy-paste all nodes from the demoLang3 generator into demoLang4 generator

<table>
<thead>
<tr>
<th>In-line with how buttons and labels are generated, we’ll create a new template to generate code for panels - <em>insert_Panel</em>. As this template is going to be very similar to, say, <em>insert_Label</em>, we will create the new template as a copy of one of the existing ones.</th>
</tr>
</thead>
<tbody>
<tr>
<td>• select (in the tree) the <em>insert_Label</em> template in the demoLang4 generator and invoke Clone Root command in the popup-menu</td>
</tr>
<tr>
<td>• in the editor - rename the new template: <em>insert_Label</em> -&gt; <em>insert_Panel</em></td>
</tr>
<tr>
<td>• in the template code - replace JLabel with JPanel</td>
</tr>
<tr>
<td>• remove the ’setText(…)’ statement - panels don’t support the ‘text’ property</td>
</tr>
</tbody>
</table>

See Generator User Guide Demo2 for details.

Doing so you won’t be even offered to import the generator model from demoLang3 generator.
7.4 First Test

Let’s create a new test model:

- go to the ‘test_models’ solution
- clone model ‘test3’ to model ‘test4’
- in the model properties dialog replace ‘engaged on generation’ language demoLang3 -> demoLang4 (See Generator User Guide Demo2 for details)
- in the model ‘test4’ add a new document ‘Panel’ with a root element - ‘panel’
- add an attribute: ‘background=”white”’ to the ‘panel’ element
- add a couple of ‘label’ elements to the ‘panel’ element.
• generate files for the 'test4' model
• preview the generated files

The generated UI clearly ignores that the 'Hello MPS!' labels should be nested inside the new panel. Instead, they are added to the frame directly. This is because our generator currently does not make any difference between root and non-root elements and simply turns them all into a top-level visual components. For each element in the input model the generator inserts a static method declaration 'createComponent()' into the 'DemoApp' class (which is perfectly okay) and generates a corresponding method call inside the 'addContent()' method in the 'DemoApp' class (which we need to change, since this should only be done for root elements).

The current implementation flattens our component hierarchy (all components are added directly to the application's content pane).

7.5 Restricting the LOOP macro to root elements

The first problem to attack is the LOOP macro in the DemoApp template, which should only iterate through root elements to generate container.add() calls for them. The elements nested inside panels, must not be included in the loop since it should be the containing panels that add them to the UI at some point later.

Let's alter the first LOOP macro. Notice also, that the second LOOP macro, which generated static method declarations for each Element, remains untouched.
Since we're now looping over Documents, not Elements, we need to alter the reference macro inside the LOOP.

If you rebuild the language and see the generated files, you'll notice that we're no longer adding the nested components into the top-most application frame.
However, we’re not adding them into the panel, either. So this is what we’ll fix now. We will use the COPY_SRC-macro for that.

### 7.6 COPY_SRC-macro

COPY_SRC macros replace the wrapped dummy piece of template code with a node from the input model. If this node has a reduction rule defined, it will be reduced before being added into the output model. In our case, we need panels to have their child Elements correctly added as visual components to the panel’s visual representation. If, for example, our concrete Panel get converted into a JPanel, we need its two child Labels to be converted into JLabels and these JLabels need to be added to the JPanel. The conversion part is already done for all Elements correctly irrespective of their nesting thanks to the second LOOP macro in the DemoApp template. The addition part is only done partially, thought. For the nested Elements, it must be the Panel that adds them.

- open `insert_Panel`
- add code to add `null` to the component
- wrap the whole statement (including the; character) with a LOOP macro that iterates over the content of the XML Panel

```java
private static void addContent(Container container) {
    container.add(createComponent_a_1());
    container.add(createComponent_a_2());
    container.add(createComponent_a_3());
}

public static Component createComponent_a_1() {
    JButton component = new JButton();
    component.setText("Hello");
    component.setEnabled(false);
    return component;
}

public static Component createComponent_a_2() {
    JLabel component = new JLabel();
    component.setText("world");
    component.setOpaque(true);
    component.setBackground(Color.orange);
    return component;
}

public static Component createComponent_a_3() {
    JPanel component = new JPanel();
    component.setOpaque(true);
    component.setBackground(Color.white);
    return component;
}

public static Component createComponent_a_4() {
    JLabel component = new JLabel();
    component.setText("Hello");
    return component;
}

public static Component createComponent_a_5() {
    JLabel component = new JLabel();
    component.setText("WPS!");
    return component;
}
```
We obviously do not want to add `nulls` into the panel, do we? Now, the trick with the `COPY_SRC` macro - we’ll add the current `node`, which refers to the `Panel`’s child `Element` that we currently iterate over with the `LOOP` macro. The `COPY_SRC` macro will take care of converting `Element` into a swing component before being added to the `component`. 
While processing COPY_SRC-macros, MPS creates a copy of the input node (aka mapped node) in output model. The wrapped node in the template code is ignored.

Well, it may sound really weird - are we going to generate java code where XML element is passed as a parameter in the method call? Of course, we are not, and the trick is that the ‘copying’ during a generation is not that simple.

Whenever the MPS generator performs copying of an input node to output model, it tries to reduce this node (i.e. find and apply a reduction rule).

Thus, our input xml element can easily become something else after being ‘copied’ by the MPS generator. Now we will define what that ‘something else’ is going to be.

7.7 Reduction Rule

Reduction rules define transformations of source-language elements into target-language elements. For example, we want to transform XML Elements, which represent a Panel’s children, into Java swing components that have been created by already generated static factory methods. In our particular case, we want to replace the Panel’s children with calls to static methods. Luckily, the static methods to look for have been preserved in the method mapping label, so we can easily get hold of the correct one.

- go to the mapping configuration in the demoLang4 generator and create a Reduction Rule, which is applicable to Elements
- apply the intention (Alt + Enter) to create a new template for this rule
• open the 'reduce_Element' template in editor (Ctrl-click on reference)

• select ClassConcept as the rule content node

• add a static method to this class (name doesn’t matter)

• create this method’s call expression and make it a template fragment (Ctrl-Shift+F):

```
public class _class_ {
  public static void _method_() {
  }
}
```

Make sure that only the method call expression is marked as a template fragment, not the whole statement. The rightmost ';' symbol should be outside the template fragment, otherwise, you’d be wrapping a Statement not a MethodCall.

• add reference-macro for method name in method call expression - we’ll retrieve the method declaration from the method mapping label:

```
reference_macro
comment : <p noe>
referent : (outputNode, genContext, operationContext, node)->node<MethodDeclaration> {
  genContext.get output method for (node);
}
```
7.8 Final Test

Rebuild the generator and preview generated text for the 'test4' model.

```java
package test_models.test4;

import java.awt.Frame;
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.Component;
import java.awt.Button;
import java.awt.Label;
import java.awt.Color;
import java.awt.Panel;

public class DemoApp {
    public static void main(String[] args) {
        Frame frame = new JFrame("Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container container = frame.getContentPane();
        container.setLayout(new FlowLayout());
        addContent(container);
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }

    private static void addContent(Container container) {
        container.add(createComponent_a_0());
        container.add(createComponent_a_1());
    }

    public static Component createComponent_a_0() {
        JButton component = new JButton();
        component.setText("Hello");
        component.setEnabled(false);
        return component;
    }

    public static Component createComponent_a_1() {
        JLabel component = new JLabel();
        component.setText("world!");
        component.setOpaque(true);
        component.setForeground(Color.black);
        return component;
    }

    public static Component createComponent_a_2() {
        JPanel component = new JPanel();
        component.setOpaque(true);
        component.setBackground(Color.white);
        return component;
    }

    public static Component createComponent_a_3() {
        JLabel component = new JLabel();
        component.setText("Hello");
        return component;
    }

    public static Component createComponent_b_0() {
        JLabel component = new JLabel();
        component.setText("MPS!");
        return component;
    }
}
```

This time both labels "Hello" and "MPS!" are correctly added to the content pane. We're done!
Chapter 8

Generator User Guide Demo5

8.1 Generator User Guide Demo 5

In this demo we will learn how to use generation scripts and utility classes. There are two kinds of generation scripts: pre-processing and post-processing scripts. Pre-processing scripts are invoked before applying the generator rules and they usually alter the input model in a way that makes it easier for further processing by the rules. Post-processing scripts are invoked after all generator rules have finished and they are applied to the output model. Additionally, in case of having multiple scripts in a generator you can influence the order, in which the scripts will run.

8.2 New Language

We'll start with an exact copy of the demoLang4 language.

- create new language ‘generator_demo.demoLang5’
- in the language properties dialog add extended languages: ‘jetbrains.mps.sampleXML’ and ‘jetbrains.mps.baseLanguage’
- create a new generator for this language if it has not been created (see Generator User Guide Demo1 for details)
- delete the (empty) mapping configuration ‘main’ from demoLang5 generator
- copy-paste all nodes from the demoLang4 generator to the demoLang5 generator

when pasting nodes - don’t forget to exclude demoLang4 generator model from imports

See Generator User Guide Demo2 for details.

Doing so you won’t be even offered to import the generator model from demoLang4.

8.3 New Test Model

- in the ‘test_models’ solution clone the model ‘test4’ into ‘test5’
- in the model properties dialog replace ‘engaged on generation’ language demoLang4 -> demoLang5 (see Generator User Guide Demo2 for details)
- in the model ‘test5’ open the ‘Panel’ document
- add a Text node to the ‘panel’ element:
8.4 Pre-processing Script

Our demoLang5 generator doesn’t support text nodes, but text node in this context has the same meaning as a ‘label’ element with a ‘text’ attribute set. Therefore, instead of adding more rules to our generator, we will add a pre-processing script, which will convert all text nodes in the input model into ‘label’ elements and let existing rules do the rest of the job.

- in the ‘main@generator’ model if the demoLang5 generator create a root node of type mapping script
- give it a name ‘fix_text’
- set script kind = pre-process input model
- set modifies model = true, since we’re going to alter the input model inside the script
- enter the code, which will perform a search and a replace of all text nodes in the input model:

```plaintext
mapping script fix_text

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

gencxt@script fix_text(void)

{nlist<Text> texts = model.nodes(Text);

foreach text in texts {

// replace with a 'label' element

node<Element> label = text.replace.with.new<Element>();
label.name = "label";

// add the text attribute to the 'label' element

node<Attribute> attribute = label.attribute.add.new("text");
attribute.value = text.text;
}
}
```

- add the ‘fix_text’ script to mapping configuration ‘main’ to the pre-processing scripts section:
8.5 First Test

Re-build the language and check out the generated text from the 'test5' model:

```java
public static Component createComponent_a0() {
    JPanel component = new JPanel();
    component.setBackground(Color.white);
    component.add(createComponent_a0());
    component.add(createComponent_b0());
    component.add(createComponent_c0());
    return component;
}

public static Component createComponent_a0() {
    JLabel component = new JLabel();
    component.setText("Hello");
    return component;
}

public static Component createComponent_b0() {
    JLabel component = new JLabel();
    component.setText("MPS");
    return component;
}

public static Component createComponent_c0() {
    JLabel component = new JLabel();
    component.setText("Hello everybody!");
    return component;
}
```

You can see that a label was correctly created for the 'Hello everybody' text and that the label is being added into the panel.

8.6 Utility Classes in Generator

Now, let’s suppose we got an idea to fix possible syntax and stylistic errors in the generated texts. We’ll create a post-processing script to do that. These NLP algorithms tend to be quite complex, so let’s develop them in a separate utility class, instead of embedding them into the script directly.

The issue here is that a utility class cannot be created in the generator model 'main@generator' directly, because then it would be treated as a root template, just like any other root node in any ‘generator’-stereotyped model would. We’ll have to create another model in the generator module:

- select the generator node (the demoLang5 generator) in the tree and choose New->Model in the popup menu:

- give the new model name: 'util':

![Model creation](image)
• Make sure you clear our the Stereotype text field
• in model ‘util’ create class ‘TextUtil’
• add static method ‘fixText()’ to class ‘TextUtil’:

We are going to call the ‘fixText()’ method in a post-processing script and pass the output model as a parameter. The output model contains classes, methods, expressions and so on. Thus we are manipulating a java-like syntax tree here. The ‘fixText()’ method will replace all strings starting with "MPS" with strings that start with "JetBrains MPS".

8.7 Post-processing Script

Now we can use this utility method ‘fixText()’ in a post-processing script:
• in model ‘main@generator’ (in the demoLang5 generator) create a new mapping script
• give it a name ‘refine_text’
• import the ‘generator protect_demo.demoLang5.generator.util’ model to model ‘main@generator’ (see Generator User Guide Demo for details on model importing)

! You can also use Ctrl+R command and type ‘TextUtil’ to import the ‘generator protect_demo.demoLang5.generator.util’ model
• in the ‘refine_text’ mapping script enter the code as shown:

```
{genContext, model, operationContext}>>void {
    TextUtil.fixText(model);
}
```

• add the ‘refine_text’ script to the post-processing scripts section in the mapping configuration ‘main’:

Re-build the generator.
You can use Ctrl/Cmd-F9 (generate all changed models in module). Note that Shift-F9 (generate current model) is not enough any more because now we have two models requiring generation.

8.8 Second Test
Generate model ‘test5’ and preview the generated text:
Notice the label now holds a "JetBrains MPS" text.
Chapter 9

Generator User Guide Demo6

9.1 Generator User Guide Demo 6

This demo is probably the most exciting one of all demos, because here we are going to create a ‘real’ language in the sense that this language will actually define a couple of higher-level concepts. We will also see how easily this DSL can be integrated with existing languages - ‘jetbrains.mps.sampleXML’ and ‘generator_demo.demoLang5’ (created earlier in the Generator User Guide Demo5).

9.2 The core idea

We’ll allow for a more convenient syntax for entering XML Elements in the test models. Developers will use new syntax for button and label, which will be more compact and intuitive. At the same time, we’ll be able to reuse the generator defined in the demoLang5 language that we built in Demo 5. The new demoLang6 language will translate the new concepts into XML Elements, which then can be accepted by the demoLang5 generator.

```
document Panel
  < panel background = " white " >
    button ( Hello )
    label ( MPS! )
    Hello everybody!
  </ panel >
```

9.3 New Language

- create new language `generator_demo.demoLang6`
- in language properties dialog add extended languages: ‘jetbrains.mps.sampleXML’ and ‘jetbrains.mps.baseLanguage’
- create a new generator for this language, if it does not exist (see Generator User Guide Demo1 for details)
- in demoLang6 structure model create new concept named ‘Button’ extending the concept ‘ElementPart’ (from ‘jetbrains.mps.sampleXML’)
- in ‘Button’ concept declare a property named ‘text’ and set an alias (we need an alias to make auto-completion menu look nice):
• create an editor for the 'Button' concept:

The editor consists of three cells: two constant-cells (shown in bold) and one property-cell (shown as {text}) which will render the actual value of the 'text' property.

• create a similar concept named 'Label'

• re-generate the language (Ctrl-F9)

9.4 New Test Model

Now let’s try out our new DSL:

• go to the 'test_models' solution

• clone the 'test5' model into 'test6' (this time DO NOT replace engage on generation language generator_demo.demoLang5 -> generator_demo.demoLang6, since we still need the demoLang5 generator to run with test6)

• add the 'generator_demo.demoLang6' language to the used languages section (new!), so we can use the demoLang6 language for editing our model.
• in the ‘test6’ model open the ‘Panel’ document and replace the two ‘label’ elements with our new button and label concepts:

With the new DSL our ‘code’ became clearer, shorter and less error prone. For instance, users now no longer can add elements inside labels or buttons.

9.5 Generator

We could generate methods and method calls out of button and label mimicking what has been done earlier for XML Elements. But this isn’t a good idea, because this way we would introduce indirect dependency between the demoLang6 generator and the demoLang5 generator (the demoLang6 generator would have to know about the implementation details of the demoLang5 generator).

Luckily, we have a far better option. As we are on a higher level of abstraction now (comparing to XML), we can simply reduce our semantically rich concepts into lower-level concepts (XML Elements). We don’t care anymore about who and how will transform those lower-level (XML) concepts further into even lower-level concepts.

9.6 Reduction Rules

We’ll employ reduction rules to convert our new concepts into XML Elements. from the sampleXML language.

• open the (empty) mapping configuration ‘main’ in the demoLang6 generator
• add a new reduction rule applicable to the Button concept
• choose an in-line template as the rule consequence (Control + Space):

The template is going to be pretty straightforward and it will not require any surrounding ‘context’. That’s why it is not necessary to create a full-blown external template as we did in other demos.

Since we want to use XML Elements from the ‘jetbrains.mps.sampleXML’ language in generator templates, we first have to declare that language among Used Languages by the main@generator model:
back in the main mapping configuration choose Element as the template’s content node:

- create the ‘button’ XML element with a ‘text’ attribute
- attach a property macro to the attribute value
- enter the code in the macro’s value function so that the value of the text property gets propagated into the XML button:

That’s it. Our high-level button is being reduced into a ‘button’ XML element with a ‘text’ attribute. The XML element will presumably be further transformed down to Java by somebody else.

- add a similar reduction rule applicable to the Label concept
9.7 First Test (Failure)

Re-build the generator and generate the ‘test6’ model. There will be a bunch of error and warning messages in the MPS messages view. Click on the ‘was input node’ error message. MPS will show you the node to, which the generator was trying to apply the rule that failed:

This is a reduction rule in the demoLang5 generator (we are generating the ‘test6’ model with two generators: demoLang5 and demoLang6, remember?) and it has failed to find a static method declaration through the mapping label method. The method label is being assigned the generated static methods inside the insert_Button and insert_Label templates, so you may wonder what went wrong. Why couldn’t we find the methods created inside the DempApp class and stored in the mapping label?

To answer that question let’s take a closer look at the generation process as a whole. As the first step, the generator creates a transient model, which will serve as an output model for the current generation phase. This model name is ‘test6@0’.

You can actually inspect the transient models yourself, if you enable ‘Save transient models’ through the button in the bottom right corner of the MPS frame. The transient models will show up in the Project View panel.

Then generator applies our conditional root rule and creates the ‘DemoApp’ class in output model. Then, at some point, the insert_Panel template is invoked and its LOOP-macro is executed (we can see the panel factory method containing the panel initialization code that we are familiar with from the insert_Panel template). The LOOP-macro creates a statement ‘component.add()’ for each child of ‘panel’ element and reduces the child to generate an actual argument to the ‘component.add()’ method call. Our ‘panel’ element contains three children - a button, a label and a text, which has been turned into a label by a pre-processing generation script. Both the button and the first label are high-level abstractions introduced by demoLang6 and their reduction yields corresponding XML Elements. (We can see it clearly in the first screenshot, where the XML element representing a button is passed as an actual argument to the call to the ‘component.add()’ method. This is reported as a warning by the generator earlier up in the trace output in yellow color).

This ends the generation of ’test6@1_1’ and the generator now creates a new transient model, named ’test6@1_2’. This model becomes a new output model and the former output model becomes a new input model. The generator starts to generate a new model ’test6@2_2’ pretending that there was nothing before. The generator has no memory about previous activities with only one exception - conditional root rules are never applied twice. Thus, this time, the output ‘DemoApp’ class is created by copying of ‘DemoApp’ class from the input model, not by applying a conditional root rule.

At this stage, the generator attempts to reduce further the XML Elements that represent the button and the label. The only rule available is the reduce_Element reduction rule from demoLang5. It will attempt to replace the XML Elements with references to static factory method declarations, however, since none have been created, none can be found in the method mapping label. The method definitions were not created simply because the XML Elements were still represented as the Button and Label demoLang6 concepts at the time when the LOOP macro inside DempApp was run.

So this is a timing issue between the demoLang5 and demoLang6 generators. The latter should be made run first. Now that we understand what is happening, the next question is how to fix it.
We are relying on the demoLang5 generator and we know that the demoLang5 generator works well providing that the input model is a 'valid XML' (we have tested it in Demo 5). In Demo 6 our input model is not exactly a 'valid XML'. Moreover, we witnessed that the transient model 'test6@1_1' is not a valid model at all - a method call can't accept an XML element as argument (you can even find a warning message "child 'Expression' is expected for role 'actualArgument' but was 'Element'" in the MPS message view). Now we could probably return to the demoLang5 generator and make some 'improvements' to allow it to handle 'invalid' input models. Fortunately, there is a better option - we can divide the generation process in two steps so that the original input model gets transformed into a 'valid XML' model first, and then this 'valid XML' model will be transformed to Java by demoLang5 generator in a second step.

9.8 Dividing Generation Process into Steps

We will certainly get 'valid XML' model from the 'test6' input model if we reduce button and label (in the 'Panel' document) into XML Elements. This transformation must happen before demoLang5 starts transforming XML Elements into Java. In other words, the reduction rules specified in the demoLang6 generator must be applied before any rules in the demoLang5 generator.

Let’s specify this constraint:

- go to the generator in generator_demo.demoLang6 and open the generator properties dialog

- add an extension dependency on the demoLang5 generator

- in Generators priorities tab add a desired priority rule
This will ensure the demoLang6 generator runs before the demoLang5 generator.

9.9 Second Test

Re-build the language and preview generated text for the 'test6' model. You should now run into no issues during generation and the generated code will correctly include the new button and label.

```
public static Component createComponent_a_1() {
    Panel component = new JPanel();
    component.setOpaque(true);
    component.setBackground(Color.white);
    component.add(createComponent_b());
    component.add(createComponent_c());
    return component;
}

public static Component createComponent_b() {
    JButton component = new JButton();
    component.setText("Hello");
    return component;
}

public static Component createComponent_c() {
    JLabel component = new JLabel();
    component.setText("JetBrains MPS!");
    return component;
}

public static Component createComponent_c0() {
    JLabel component = new JLabel();
    component.setText("Hello everybody!");
    return component;
}
```

9.10 Saving Transient Models

Our demoLang5 and demoLang6 generators are working together well, because we have divided the generation process into two distinct steps.

In the first step the demoLang6 generator reduces our high-level concepts (button and label) into corresponding XML elements and produces a 'valid XML' model as output.

In the second step the demoLang5 generator generates Java from that 'valid XML' model. Thus there should be at least one transient model between step 1 and step 2.

Let’s take a look at what models have actually been created in the course of generation:

- in Project Settings enable the **Save transient models on generation** option:
• generate the 'test6' model

• in the project tree find and expand the node named 'transient models' (this node is at the bottom)

There are (surprisingly) five transient models.

We can browse the transient models and we will find sometimes subtle and sometimes dramatic differences between them. This will give us a clue about what kind of transformation has taken place on each step.

For instance, open the 'Panel' node in model 'test@1_0' (this model has been generated in very first step).

We can see that high-level button and label concepts have been replaced with their XML counterparts, but the text 'Hello everybody!' is still there.

In the next model 'test@2_0' the text 'Hello everybody!' has been replaced with a 'label' element (as a result of running the pre-processing script 'fix_text' in the _demoLang5 generator).

Model 'test@2_1' contains the generated DemoApp class.

And model 'test@2_3' contains the same class, but the string "MPS!" is replaced with "JetBrains MPS!" (as a result of the post-processing script 'refine_text' in the _demoLang5 generator).

9.11 Root Nodes Copying and Reduction Rules

As we can see, the model 'test@1_0' is identical to the original input model 'test6', except for high-level button and label concepts that have been reduced to XML elements.

This is what we wanted, but it is probably still not clear, how roots in 'test@1_0' have been created (we don’t have any 'root rules' that would create a Document) and why our reduction rules have been applied.

From our previous experience with reduction rules we remember that we had to create a COPY_SRC _macro for reduction to take place. But in the _demoLang6 generator we don’t have any _COPY_SRC _macros, do we?

Whenever the generator meets an input root node, for which none of root mapping rules or abandon root rules is applicable, the generator creates a deep copy of that root in the output model.

While copying the node, the generator is always trying to find a reduction rule applicable to the currently copied node.

If a suitable reduction rule is found, it is applied to the node. Otherwise, the generator creates an output node (instantiates the same concept), duplicates all properties and references of the input node (local references are re-resolved) and repeats the same copying procedure for all the child nodes recursively.
This way the input Documents (Button, Label and Panel) from 'test6' have been copied to model 'test6@1_0' and reduction has been applied to the button and label concepts that were inside the 'Panel' Document.

Returning to the _COPY_SRC _macro - it doesn’t ‘invoke’ a reduction (as it might seem). Instead it merely applies the same copying procedure to the mapped node and, if any reduction rules happen to be applicable, then the transformation occurs.

### 9.12 Using Generation Tracer Tool

We already know Generator User Guide Demo6 save and browse transient models. Actually, with the option to Save transient models on generation activated, MPS not only saves transient models, but also collects a great deal of data regarding the process of transformation.

This data can be viewed using the Generation Tracer Tool.

For instance, open the 'Panel' Document in the transient model 'test6@1_0', select 'label' and choose Show Generation Traceback in the popup menu.

The Generation Tracer View will be opened.
In the root of this tree there is the 'label' node, for which we have requested the traceback info (the blue arrow denotes an output node) and the rest of the tree shows the sequence of events, in reversed order, that led to this output. Looking at this tree and clicking on its nodes (in order to open them in the editor) we can reproduce the process of transformation to a great level of detail.

For instance, we can see that the output 'label' is created by the template in the reduce Label rule.

... and the reduce Label rule has been applied to the input node 'Label' while the 'Panel' Document (the root input node) was being copied.
document Panel

    < panel background = " white " >
    button ( Hello )
        Hello everybody!
    </ panel >
Chapter 10

Generator User Guide Demo7

10.1 Generator User Guide Demo 7

In this final demo we will go back to Demo 3 and implement the desired capabilities in a slightly different way, so that we explore another area of the MPS generator - the weaving rules and root mapping rules.

In Demo 2, which we will build on here, we were generating java statements like:

```
container.add(new JButton());
```

to create a Swing component and add it to the application’s content pane.

In Demo 7, just like in Demo 3, we are going to add support for component properties, which will require a generation of more complex initialization code - not just a constructor invocation. Moreover, the generated property initialization code is going to be different for different types of components. Therefore we will choose a generation strategy that is capable of handling such requirements - instead of making use of a SWITCH macro we will use Weaving Rules, which will 'inject' component’s initialization code into the DemoApp class.

10.2 New Language

Again, we need to setup a new language and copy some of the Demo 2 generator artifacts to it.

- create a new language: ‘generator_demo.demoLang7’
- in the language properties dialog add extended dependencies on 'jetbrains.mps.sampleXML' as well as on 'jetbrains.mps.baseLanguage'
- create a new generator for this language, if it does not exist (see Generator User Guide Demo1 for details)
- delete the (empty) mapping configuration 'main' from the demoLang7 generator (as in Demo 2, we will copy all needed parts from the demoLang2 generator to the demoLang7 generator)
- copy-paste the mapping configuration 'main' from the demoLang2 generator to the demoLang7 generator

when pasting nodes - don’t forget to exclude the demoLang2 generator model from imports

See Generator User Guide Demo2 for details.

- copy-paste the 'DemoApp' template from the demoLang2 generator to the demoLang7 generator

10.3 Enhance the language

This time, instead of having all input XML Elements represented as standalone roots in the input model, we’re going to wrap them into a single root concept, called XMLDocument. The demoLang7 language will introduce this new concept as well as its editor:
The concept may hold a collection of XML Elements and it will present them on the screen as a vertical collection.

10.4 **addContent(container)**

- open the *DemoApp* template
- add a static void method `addContent(Container)`
- in the `main()` method find the statement:
  ```java
  $LOOP$[container.add($SWITCH$[null]);
  ```
- replace the statement above with statement below:
  ```java
  addContent(container);
  ```
10.5 New Test Model

Before going on with the generator let’s create a new input test model:

- go to the ‘test_models’ solution
- create a new model ‘test7’
- in the Used Languages tab add ‘jetbrains.mps.sampleXML’ and ‘jetbrains.mps.samples.generator_demo.demoLang7’
- add a new XMLDocument root node and type the code:

```
<button text = " Hello " enabled = " false ">
  ... 
</button>

< label text = " world! " background = " orange ">
  ... 
</label>
```

Now, let’s define the semantics for these documents.

10.6 Root mapping rules

We used to utilize conditional root rules for instantiating the DemoApp class template in the previous demos. In Demo 7 we’ll use a different mechanism - root mapping rules. In the ‘main’ mapping configuration delete the entry in the conditional root rules section as well as the one in the abandon roots section and instead add a root mapping rule:

```
is applicable:
  <always>
  conditional root rules:
    << ... >>
  root mapping rules:
    concept XMLDocument --- DemoApp
    in input root default
```

This rule will replace an XMLDocument with our desired DemoApp class template.
10.7 Weaving Rules

Weaving rules are a vehicle that can add extra nodes into the output model. In our case we'll utilize them to insert code for adding swing components to their containers.

- return to the demoLang7 generator and open the mapping configuration 'main' in the editor
- create a new Weaving Rule (press Insert or _Enter _while cursor is in the weav

10.8 Weaving Context

Weaving Rules inject additional generated code into other generated code. The Weaving Context is the exact place to which the Weaving Rule will inject the code. (the injected nodes will be the children of the context node).

In our case, we will inject the component's set-up code into our DemoApp class. Thus, we have to find the generated DemoApp class in the output model and pass this class to our weaving rule as its weaving context.

10.9 Mapping Label

We will use a mapping label to find the generated 'DemoApp' node in the output model. As already discussed in Demo 3, they serve as registries for generated nodes so they could be retrieved later by other parts of the generator. This loosens the coupling between independent generator parts.

Declaring Mapping Label

Before a mapping label can be used it must be declared in the mapping configuration:

- go to the mapping labels section of the main mapping configuration
- add a new mapping label (press Insert or Enter)
- give it a name: 'main_class'
- select 'ClassConcept' as the label's output concept, since we're going to store ClassConcepts in the mapping label

In our case, we are going to attach the mapping label to a rule, which generates the 'Demo App' class (a Root mapping rule).

That's why we have chosen 'ClassConcept' as the label's output concept.

The expression (we will use it soon):

```
    genContext.get output "main_class"
```

will then have type: 'node<ClassConcept>'.

The fact that the 'get output' expression has a type, provides no benefits for us in this particular case, but in general, it is a very helpful feature.
CHAPTER 10. GENERATOR USER GUIDE DEMO7

Attaching the Label to a Rule

Attach the mapping label ‘main_class’ to the Root mapping rule using the rule’s inspector:

![Image of mapping label 'main_class' attached to rule]

Using the Mapping Label for Finding the desired output node

Now, return to Weaving Rule and enter the following code into the context function:

```plaintext
context : {genContext, operationContext, node} -> node { 
  genContext.get output main_class for (node.ancestor<concept = XMLDocument>); 
}
```

This will now correctly resolve the DemoApp class corresponding to the processed XMLDocument and pass it into the weaving rule, so that it can add new methods to it. Notice the use of the jetbrains.mps.smodel query node.ancestor, which retrieves the closest node’s ancestor of the specified concept in the AST.

10.10 External Template

Now let’s create a template for this weaving rule, which will be used to generate the code to add to the DemoApp class:

- type ‘weave_Button’ in the red ‘choose consequence’ cell
- press Alt-Enter and choose ‘New Template’ (apply intention)
- open the template in the editor (use Ctrl-left-click> or Ctrl+B on reference cell)

The template should already have a Class set as the template’s content node, because that is the type of elements stored in the main_class mapping label. If not - choose ‘ClassConcept’ as the template’s content node.

- give the class a name

```
note
```

You don’t really have to give the class a name. This class is not going to be generated. This class will serve as a context or a place, into which we can put the newly generated code.
We will mark this ‘real code’ with the template fragment tags.

- add a static method ‘createComponent()’
- create a template fragment: select the whole method declaration and press Ctrl-Shift+F:
10.11 IF-macro

The following steps are very similar to what we’ve done in Demo 3, since we’re effectively generating the same code just using a different technique - weaving rules instead of reduction rules.

Inside the ‘createComponent()’ method we are going to create and initialize a JButton component. Optionally, if the input Element has an attribute ‘text’, then we will generate a statement:

```java
component.setText( _text_ )
```

where the text is the string specified in the ’text’ attribute in input Element.

- enter the following code in the body of the ‘createComponent()’ method
- create an IF-macro around the ‘component.setText("text");’ statement (select the whole statement, press Ctrl-Shift+M)
- enter the code for the condition function of the IF-macro, which will check the presence of the ‘text’ attribute

```java
public class _class_ { 
  <TF public static Component createComponent() { TF>
    JButton component = new JButton();
    $IFScomponent.setText("text");
    return component;
  }
}
```

- create a property-macro inside the string literal "text"
- in the value function of the property-macro enter code that returns the value of the ‘text’ attribute of the input Element
10.12 Complete the Generator

To complete this step we have to create another Weaving Rule, which would be applicable to the 'label' Element.

- open the 'main' mapping configuration in editor
- select the weaving rule, which we have just created for 'button' (select the whole node: put cursor inside the rule, use Ctrl+W to expand selection)
- press Ctrl+D (duplicate) to create an identical rule right next to the original one
- in the condition function, in the statement
  ```java
  node.name.equals("button");
  ```
  replace "button" with "label"
- in the project tree select the template node 'weave_Button'
- duplicate this template node using the Clone Root command in the popup menu
- in the editor rename this new template to 'weave_Label'
- in the template code replace the statement
  ```java
  JButton component = new JButton();
  ```
  with
  ```java
  JLabel component = new JLabel();
  ```
- re-auto-complete 'setText' in the statement if the method call is not resolved automatically
  ```java
  $IF$[component.setText("$[text]");]
  ```
• attach the ‘weave_Label’ template to the second weaving rule in the ‘main’ mapping configuration

**weave rules:**

```java
concept Element

condition (genContext, node, operationContext)->boolean {
    node.name.equals("button");
} 

context : (genContext, operationContext, node)->node<> {
    genContext.get output main_class;
}

concept Element

condition (genContext, node, operationContext)->boolean {
    node.name.equals("label");
} 

context : (genContext, operationContext, node)->node<> {
    genContext.get output main_class;
}
```

• re-generate generator model

### 10.13 First Test (Error)

Try to rebuild the language and generate files from model ‘test7’.

Generation should run with no problems but compilation will fail with an error:

`generator_demo\test7\DemoApp.java : Duplicate method createComponent() in type DemoApp (line: 37)`

Click on the error message to view the error in generated code:

```java
public static Component createComponent() { // <-- error
    JButton component = new JButton();
    component.setText("Hello");
    return component;
}

public static Component createComponent() { // <-- error
    JLabel component = new JLabel();
    component.setText("world!");
    return component;
}
```

The problem is that our weaving rules are always injecting method declaration with the same name: 'createComponent()'

### 10.14 Generating Unique Names

To make the names of each generated 'createComponent()' method unique we will create another property-macro:

• open the ‘weave_Button’ template in editor

• add a property-macro to the name of 'createComponent()' method

• enter the code in its value function as shown
The templateValue is going to be "createComponent" (i.e. the method's name as written in the template).

- make similar changes in the 'weave_Label' template
- re-generate generator model

10.15 Second Test

Generate files form the 'test7' model - this time we should get no error.

Preview the generated text for 'test7':

```java
public class DemoApp {

    public static void main(String[] args) {
        JFrame frame = new JFrame("Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container container = frame.getContentPane();
        container.setLayout(new FlowLayout());
        addContent(container);
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }

    public static void addContent(Container container) {
    }

    public static Component createComponent0() {
        JButton component = new JButton();
        component.setText("Hello");
        return component;
    }

    public static Component createComponent1() {
        JLabel component = new JLabel();
        component.setText("world!");
        return component;
    }
}
```

The code has no compilation problems, but it is still not working, because the body of the 'addContent()' method is empty. We're not calling the creational methods that we've just weaved in. Let’s fix that by weaving the following code:

```java
container.add( createComponent0() );
container.add( createComponent1() );
```

into the body of the 'addContent()' method.
10.16 Second Template Fragment

To generate a method call of the 'createComponent()' method, we will add another weaving rule and a template. The calls to all the generated 'createComponent()' methods must be weaved into the 'addContent()' method of the main generated class, thus we first have to store the 'addContent()' method in a mapping label in order to refer to it from weaving rules.

We can use the LABEL node macro to store the generated method in the mapping label:

```java
root template
input XmlDocument

public class DemoApp {
  public static void main(String[] args) {
    JFrame frame = new JFrame("Demo");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container container = frame.getContentPane();
    container.setLayout(new FlowLayout());
    addContent(container);
    frame.pack();
    frame.setLocationRelativeTo(null);
    frame.setVisible(true);
  }
  $LABEL$ addContentMethod public static void addContent(Container container) {
    <no statements>
  }
}
```

Now a new weaving rule needs to be added to the weaving rules section of the main mapping configuration that will insert the calls to the individual 'createComponent()' methods into the 'addContent()' method:

```
<table>
<thead>
<tr>
<th>concept</th>
<th>Element</th>
<th>condition</th>
<th>--&gt;</th>
<th>template</th>
</tr>
</thead>
</table>
| Element | ElementInitialization | <always> | get output addContentMethod for (node.ancestor<concept = XmlDocument>); | node<StaticMethodDeclaration> method = genContext.method.
```

Notice that the context retrieves the 'addContent()' method from the mapping label and returns the method's body as the node that will have the template fragments weaved in.

The `weave_ElementInitialization` template will insert the call to the individual 'createComponent()' methods into its context node (a method body, i.e. a `StatementList`):
Be sure you tagged the whole statement as a template fragment, including the trailing semicolon.

Add a parameter 'Container container' to the method declaration (1) This parameter must have the same name as parameter in the actual 'addContent()' method in the 'DemoApp' template).

If you decide to name the Container parameter differently, so that there's no longer a name match between the 'container' in the template fragment and the container parameter in the generated 'DemoApp.addContent()' method, you will need to resolve the actual parameter of 'DemoApp.addContent()' with a reference macro:

The actual 'createComponent()' method must be obtained from somewhere - a mapping label would work well, so we need to first create a new mapping label to hold these methods and insert the methods into the mapping label:

The LABEL node macro will help us have the 'createComponent()' methods stored in the createComponentMethods mapping label for both Buttons and Labels in their respective weaving rules/templates:
This should be done for both `weave_Button` and `weave_Label`.
Now we can return to the `weave_ElementInitialization` template and specify a reference macro on the call to 'createComponent()' to obtain the appropriate 'createComponent()' method from the `createComponentMethods` mapping label:

```java
template weave_ElementInitialization
input Element

parameters << ... >>

content node:
public class _class_ {  
    public static Component createComponent() { 
        null;
    }
    public static void addContent(Container container) { 
        container.add($createComponent());
    }
}
```

```
<reference target>
    comment: <none>
    referent: (outputNode, genContext, operationContext, node) -> join(node, genContext.get output.createComponentMethods for (node));
</reference target>
```

### 10.17 Third Test

Now you can make the generator model a generate files from the 'test7' model in the 'test_models' solution.
Now you get correct and complete code for our simple application.
Part IV

Advanced
Chapter 11

Dive into plugins: the PlantMPS plugin

11.1 Introduction
The PlantUML plugin developed by the mbeddr.com project seemed to me very interesting, I already knew PlantUML so I
decided to study the plugin to learn how an external tool can be plugged into MPS.
As a bonus I’ve created a self contained plugin called PlantMPS that let one uses PlantUML without importing into MPS
the whole mbeddr platform.
I’ve also ported the LanguageVisualization plugin to PlantMPS so one can use it without mbeddr.
The tutorial describes almost all the steps I’ve followed to write the plugin but it’s not a step-by-step guide. There are
other resources available about plugins so here I have not documented generic plugin features, rather I’ve focused more on
specific technics and API used to implement the plugin by the mbeddr guys.

11.1.1 How the plugin works
First of all the plugin define a language with an interface called IVisualizable, concepts that implement that interface
can be visualized in one or more PlantUML diagram types. To implement this the IVisualizable interface defines two
methods:

- the first returns an array of strings that are the categories the concept supports, for instance Class diagram and
  Activity diagram.

- the UI let the user to choose one of the supported categories, so the second method receives the select category and
  fill the diagram, also recevied as parameter, with PlantUML statements required to display the selected category for
  that concept.

An example of these methods can be:

```java
public string[] getCategories() {
    return new string[]{"Class diagram", "Activity diagram"};
}
```

and

```java
public void getVisualization(string category, VisGraph graph)
overrides IVisualizable.getVisualization {
    if (category.equals("Class Category")) {
        graph.add("\n        class " + this.name);
    } else if (category.equals("Activity Category")) {
        //...
    }
}
```

To use the plugin your concepts should only implement this interface, the rest is handled by the plugin and include:

- history to handle next and previous diagram visualization.
- save the generated SVG image.
- copy the PlantUML source to the clipboard.
- zoom
- go-to-node feature when the user click on the SVG image.
PlantMPS also differs from the original mbeddr plugin in:

- mbeddr starts an HTTP server that exposes several services that can be requested by internal or even external processes. In the original plugin the go-to-node feature was implemented transforming the user click over the SVG into an HTTP request. PlantMPS directly selects the requested node. In case you are interested in this topic take a look at application plugin and extension point concepts in the c.mbeddr.mpsutil.httpserver module.

### 11.1.2 Credits

- mbeddr.com project.
- LanguageVisualization plugin.

### 11.2 Getting started

#### 11.2.1 Create the project

Usually I place the MPS stuff under code, so in the project root I can place documentation, README, etc, not related with MPS. So the project one can open from MPS is in our case under code/plantmps. The language created with the project will hold the IVisualisable interface and the VisGraph class used to hold the diagram source.

Then we need a plugin solution as the container for our plugin:

- right click over the project node and choose New -> Plugin Solution.
- the chosen name is org.mar9000.plantmps.plugin.

The MPS IDE should now look this way:

#### 11.2.2 Create the tool

Now that we have a plugin solution we can create an MPS Tool:

- right click over the plugin model and select New -> jm.lang.plugin -> Tool.
- set the name to SVGViewer and caption to Visualization.

#### 11.2.3 Add a tool icon

This is an interesting part. First the icon size used by MPS is 13x13, I copied the icon from the mbeddr project after investigating the source code of their Idea project where the class AllIcons comes from. The icons dir has been placed at the same level of the solution, indeed code/plantmps/solutions/org.mar9000.plantmps.plugin/icons.

Second to specify icons you need the jm.lang.resources language, so include it as used language, you can do it from the Module Properties dialog:

- place the cursor over the icon: field and hit ctrl+space
- you should see the IconResource option.
- select it, a button should appear on the UI that let you select the icon file from the filesystem.
- if you look at the the inspector you should see how paths can be specified relative to module directory, that is ${module}/icons/...
11.2.4 Complete a dummy tool

The only thing to complete to open our first tool is the getComponent() method, for the moment just return a JLabel(). To do this you need to import JDK/javax.swing@java_stub. The fastest way to import this model is:

- hit ctrl+R.
- write JLabel.
- one of the suggested option should be the model indicated above, select it.
- now you can use new JLabel() as java statement.

The tool should look this way:

```java
tool SVGViewer {
    caption: Visualization
    number: <no>
    icon: <here you have the added icon>
    position: right
    [snip]
    getComponent()->JComponent {
        new JLabel("Your first Tool.");
    }
}
```

11.2.5 Tool life cycle

Remember that a Tool is instantiated only once when you open it or in case of modifications while you are developing it, indeed the getComponent() method is called once. Actions when invoked alter the state of the Tool, in our case set the node to be displayed, and call openTool() method that has the effect to set the Tool visible on screen.

11.3 Adding actions

An action is required to open our tool and an action group is required in order to display the action menu somewhere in one of the MPS menus. First create the action:

- right click on the plugin model and select New -> j. m. lang. plugin -> Action.
- set the name and other trivial parameters.
- action context parameters are parameters passed to the action at runtime. They are specified from fields of classes of the MPS or Idea API.
  Here we need the key PROJECT contained into CommonDataKeys, include its model with ctrl+R.
  Another important container of the keys is MPSCommonDataKeys, we are going to use it in this tutorial.
- now you should be able to select PROJECT into the drop down menu that appears with ctrl+space.
- set the mnemonic: parameter to "V", it does not bind any key to our menu only display the first letter of Visualize underscored.

To complete the execute() method as below your should add MPS.Platform as dependency of the plugin solution. The tool should look now this way:
CHAPTER 11. DIVE INTO PLUGINS: THE PLANTMPS PLUGIN

```java
action VisualizeAction {
    mnemonic: V
    execute outside command: false
    also available in: << ... >>

caption: Visualize
description: <no description>
icon:<our icon>

construction parameters
<< ... >>

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

<update block>
execute(event)->void {
    tool<SVGViewer> svgViewer = this.project.tool<SVGViewer>;
    svgViewer.openTool(true);
}
}
```

Last thing to do to open the tool is to add our action somewhere into the MPS menus:

- right click on the plugin model and select New -> j. m. lang. plugin -> ActionGroup.
- call it VisualizeActionGroup.
- you should see a small red/mandatory field, here select element list, another field where you can select actions should appear.
- select our VisualizeAction, should be the first option.

The modifications section is used to specify where the action menu should appear, as in the case of CommonDataKeys we need to import a model in order to specify interesting places:

- EditorPopup contained in j.m.ide.editor.actions is the menu that appears when you right click over the editor area.
- NodeActions contained in j.m.ide.actions is the menu that appears when you right click over a node in the tree area.
- search ActionGroupDeclaration to find more, another one is ModelActions.

Your action group should be:

```java
group VisualizeActionGroup
is popup: false
contents
    VisualizeAction
modifications
    add to EditorPopup at position <default>
```

It’s time to open our tool:

- rebuild the solution, ctrl+F9.
- either from a node on the left tree or from the editor you should the menu Visualize.

11.3.1 Keys binding with Keymap

As stated above the mnemonic: parameter does not realize any binding rather it’s just to visually underscore one of the characters of the menu.
Keys binding is declared with instance of KeymapChangesDeclaration from the j. m. lang. plugin language:

- after creation add a simple item.
then select VisualizeAction into the action field.

• indicate ctrl+alt+VK_V as binding.

Rebuild with ctrl+F9 and you should be able to open the tool using ctrl+alt+V.

11.3.2 Use the defined actions programmatically

As we saw actions can be defined that displayed somewhere in the MPS menus. But actions and action groups can also be instantiated programmatically, actions will be displayed like buttons and action groups like toolbars.

See SVGViewer.getComponent() where the Tool toolbar is built, take a look at the actionGroup<> and toolbar<> creators. For this you need to import j.m.workbench.action@java_stub. All actions into the toolbar package are very simple they get the Tool from the project field and call one or more methods to update its state. The Save action has also the FRAME context parameter.

11.3.3 Tune actions

The action we’ve created is always available but we want to show it only when we are on a node IVisualizable. To do it add a context parameter to the VisualizeAction:

```
action context parameter (always visible = false)
Project project key: PROJECT required
node<IVisualizable> elementToVisualize key: NODE required
```

Also note the event.getPresentation().setText() to dynamically set the text displayed as menu. If you clone and open the PlantMPS plugin you can verify that the Visualize action appears only for IVisualizable nodes, try the DummyVis1 and DummyVis2 nodes.

11.3.4 Dynamic actions

As we have just seen the DummyVis2 nodes can be visualized in several categories. This is implemented with dynamic actions. In particular when a node has only one category it’s handled by the VisualizeAction action, see VisualizeAction.isApplicable(event). When a node as more than one category this action become disabled and VisualizeActionParametrized become enabled. This ladder action is almost equal to the first but it has a construction parameter, in this case a string named cat. If we have the category we want to display the execute() method can display this category through VisGraph instead of the first available category as done by VisualizeAction.

The parametrized actions are created by VisualizeActionGroupDynamic that differs from VisualizeActionGroup in several places. First right after its creation you should select (right below it’s invisible when disable) update instead of element list. The content of the update(event) method is simple, however note:

• the add statement used to add the parametrized action.

• the disable(boolean) method.

• there is also a method setPopup(boolean) to dynamically visualize the added actions inline with all the other menus or as a separate menu (popup).

Remember to open the Inspector when you specify the cat parameter because you when to enter the toString() method. Here the PlantMPS plugin differs from the original mbeddr plugin that always enable both parametrized and non parametrized actions. Actually the current implementation of PlantMPS could have been done only with dynamic actions enabling the popup in case of more than one action and disabling the popup in case of just one action. But this would have given a smaller tutorial. The concept that result to be handled by the dynamic group is DummyVisualizableMoreCategory and its instance DummyVis2.

11.3.5 Create actions programmatically

Actions can also be created, and used programmatically, see for instance ChangeCategoryAction that implements the combobox that let the user switch from one category to another category of the same node. In the AbstractChangeCategoryAction class note how the enabled property is created with a specific concept (Property from baseLanguage).

11.3.6 History

The class VisualizationHistory is almost pure java but:

• to implement the history i necessary to save a reference to the node to visualize at that point of the history and which category to show. However the node is not assigned directly to a field of an history item, as you can see from the use of the SNodePointer class. This is to prevent memory leaks as pointed out in HowTo – Adding additional Tools (aka Views).
11.4 Add PlantUML and SVG support

At this point we are going to use the IVisualizable interface from the o.m. plantmps language, it’s not described in details because it really simple and it has nothing related with plugins. Just remember that to create the field Project project one needs to import j.m. project @java_stub. I’ve pointed out this because dependencies are often the dark side of MPS.

Note that the PlantMPS plugin differs from mbeddr plugin in the way it responds to user click over the SVG image. As stated above mbeddr executes an HTTP request toward its HTTP server and it execute the node selection on the editor. Whereas PlantMPS directly execute the code that select the node, see MbeddrUserAgent.openLink().

11.4.1 Adding jars

After you have placed your jars into a directory, in our case the lib dir. under the plugin solution dir, from the Module Properties window (alt+enter) you have to:

- Common tab: click Add Model Root and choose javaclasses.
- then on the right select the jars for your dir and click the Models button.
- into the Java tab add the jars into the Libraries section.

If everything went well you should see a new item into the navigation tree of your language/solution named stubs. Remember that in case you add other jars the Add Model Root -> javaclasses is required only when you add the firsts jars.

11.5 Implementing language, generation plan and java_stub diagrams

to be continued