JetBrains MPS: Language Eng. Environment

Tutorial on how to specify the abstract syntax, concrete syntax and semantics

Note:
The Metamodel is not Complete

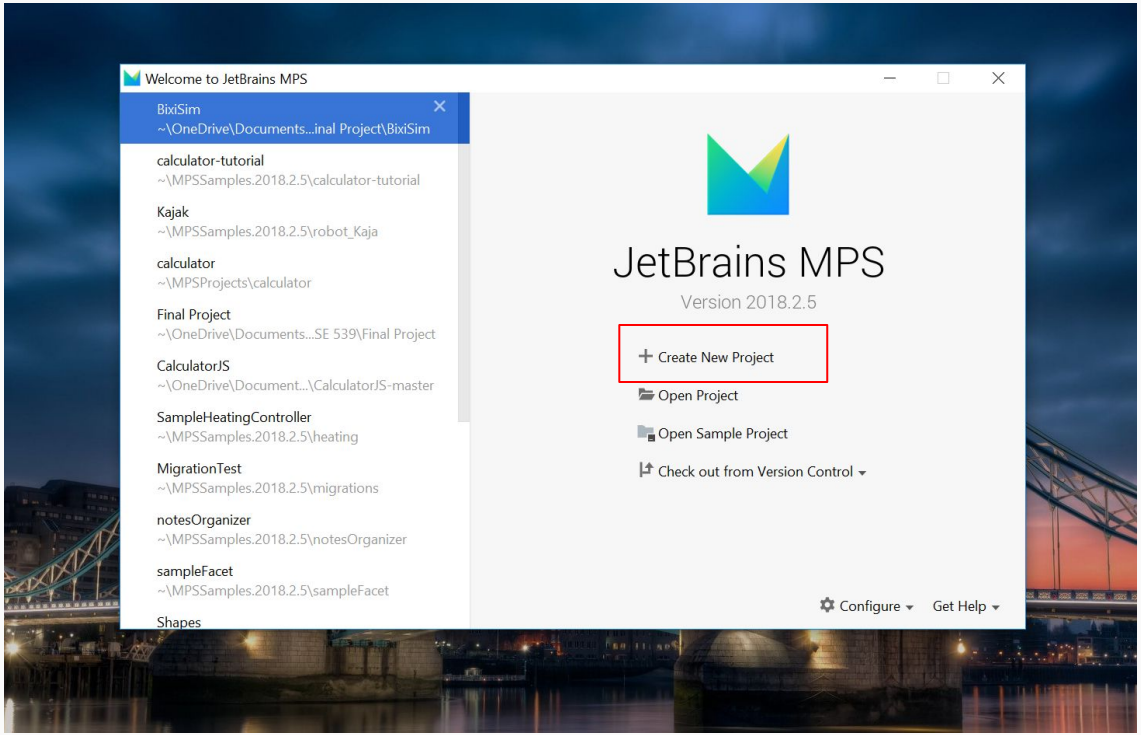
Please note that the metamodel built during this tutorial is not completed. The purpose of the next slides was to instruct the reader on how to use JetBrains MPS in order to build the different parts of its own custom DSL. This tutorial covered the key concepts that were used when building the BIXISim project (see last section).

Note:
This tutorial is incomplete when viewed alone

In order to fully appreciate our JetBrains MPS tutorial, please also read the “Language Engineering and Transformation Environment” section of the accompanying report (pg. 5), as it contains a functional explanation of many of the concepts seen in these slides.

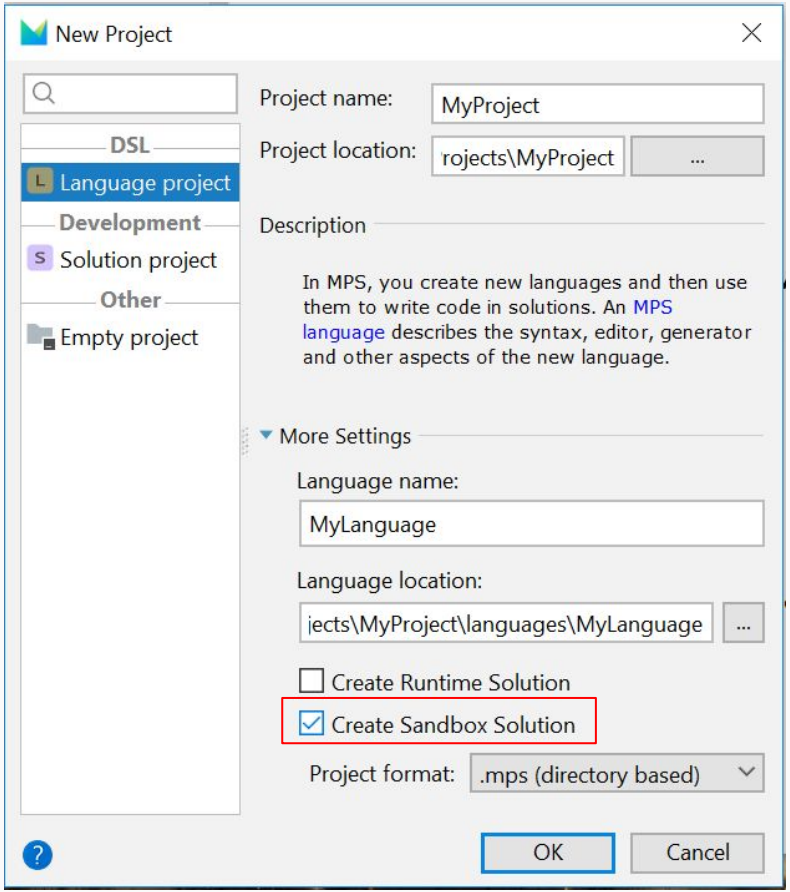
Basics

Create a Project



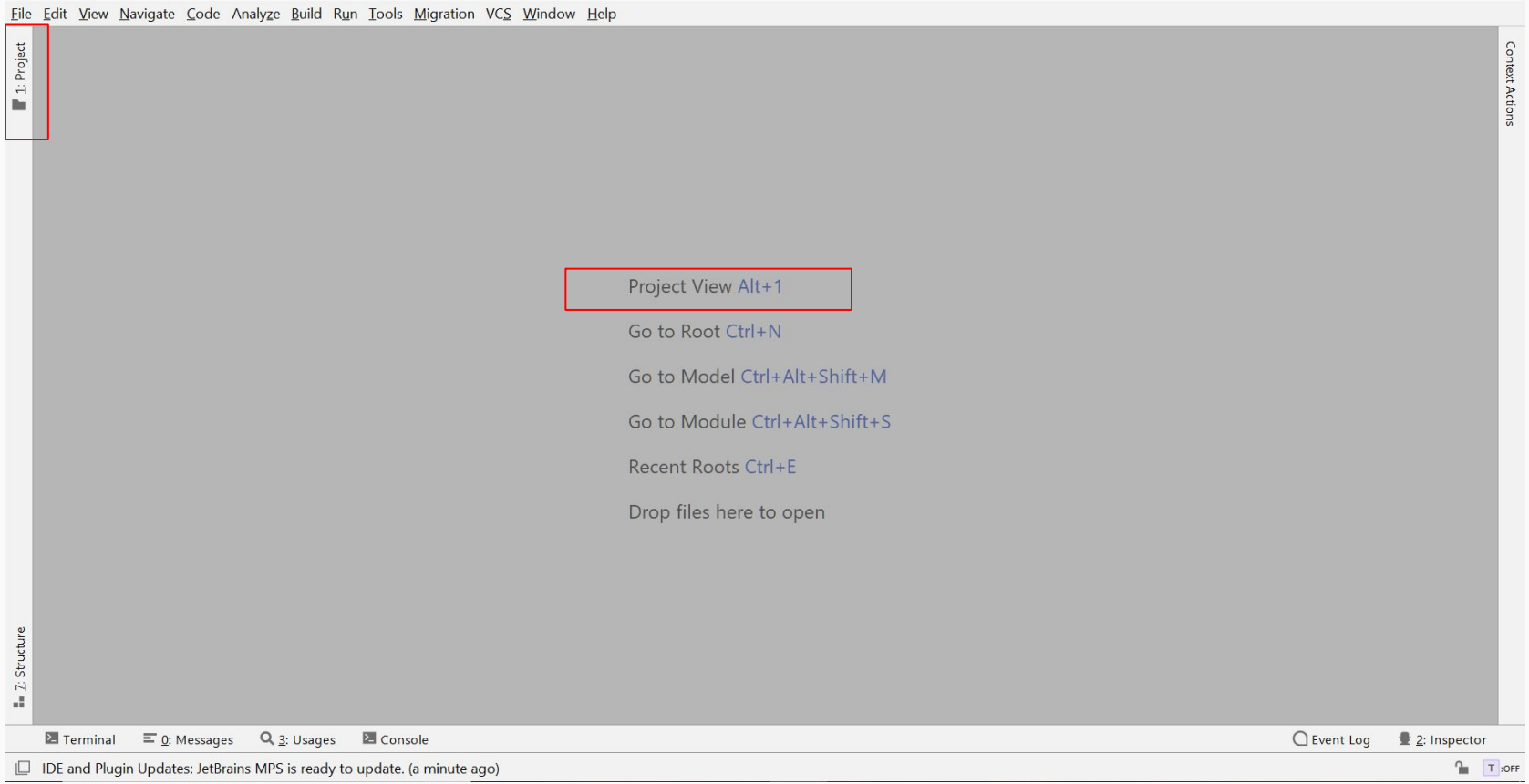
- Select the option “Create New Project”

Create a Project (Cont'd)

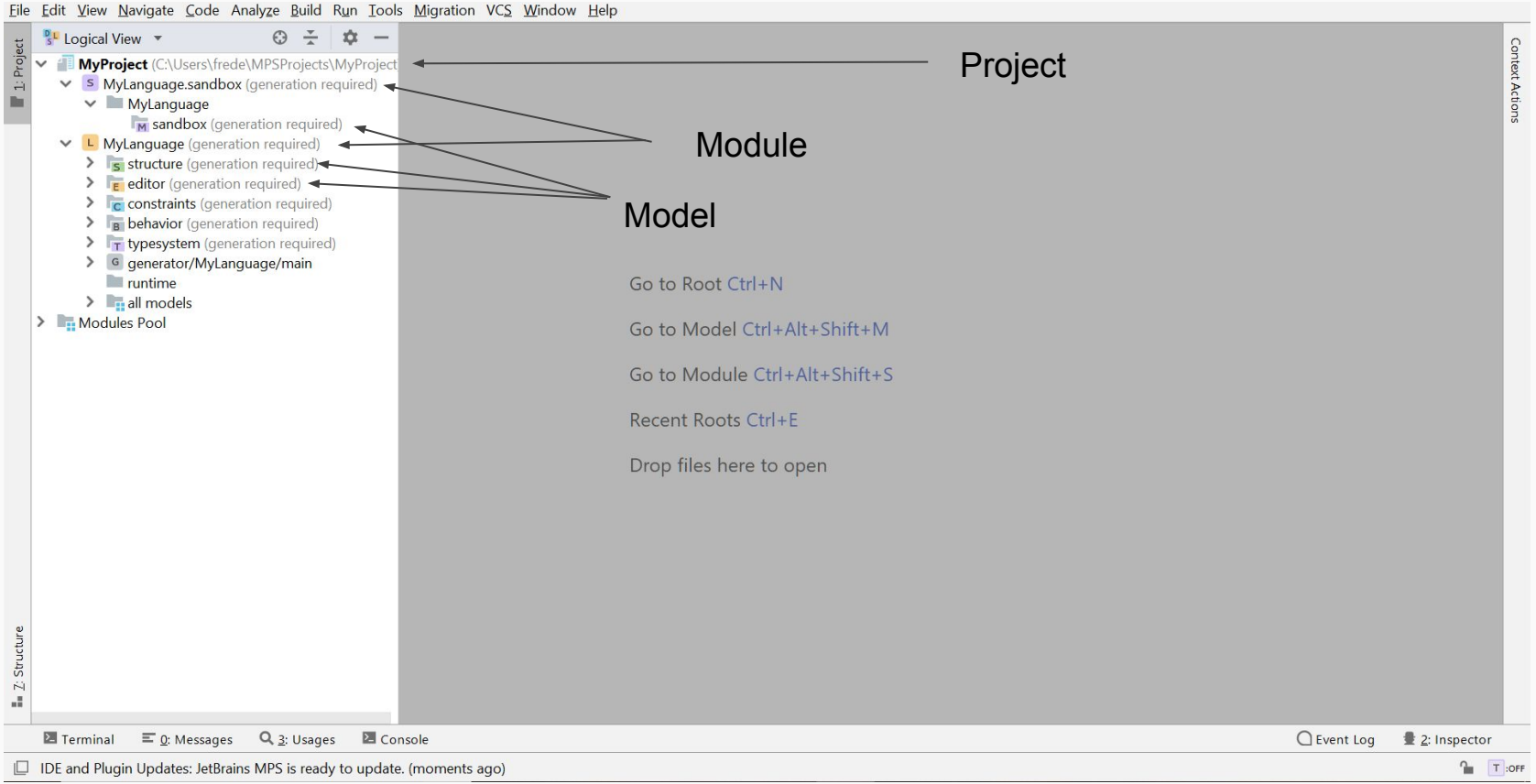


- Specify the project's name,
- Its location
- Specify the language's name,
- Its location
- Check the option "Create a Sandbox Solution"

Open Project View



Terminology

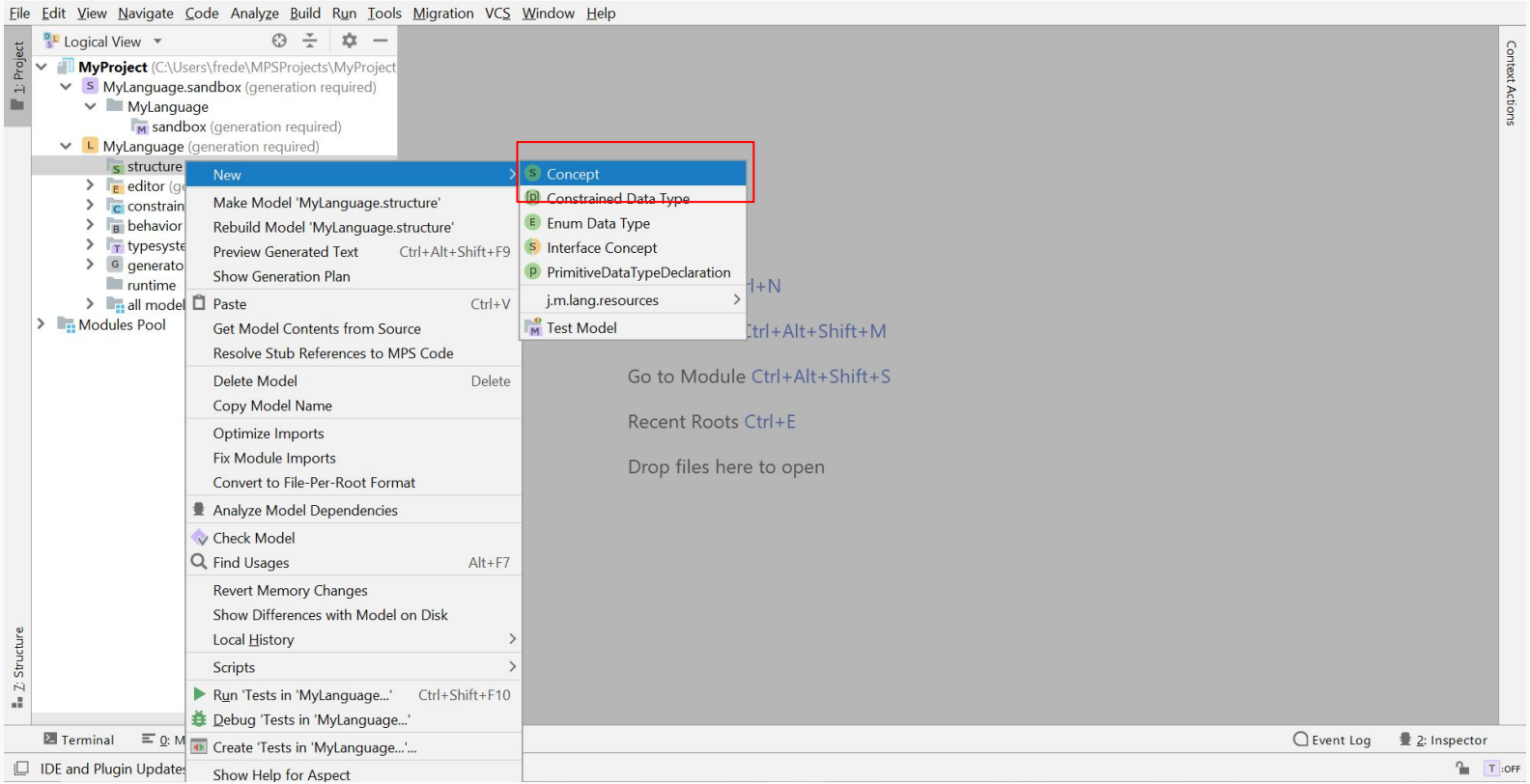


Abstract Syntax

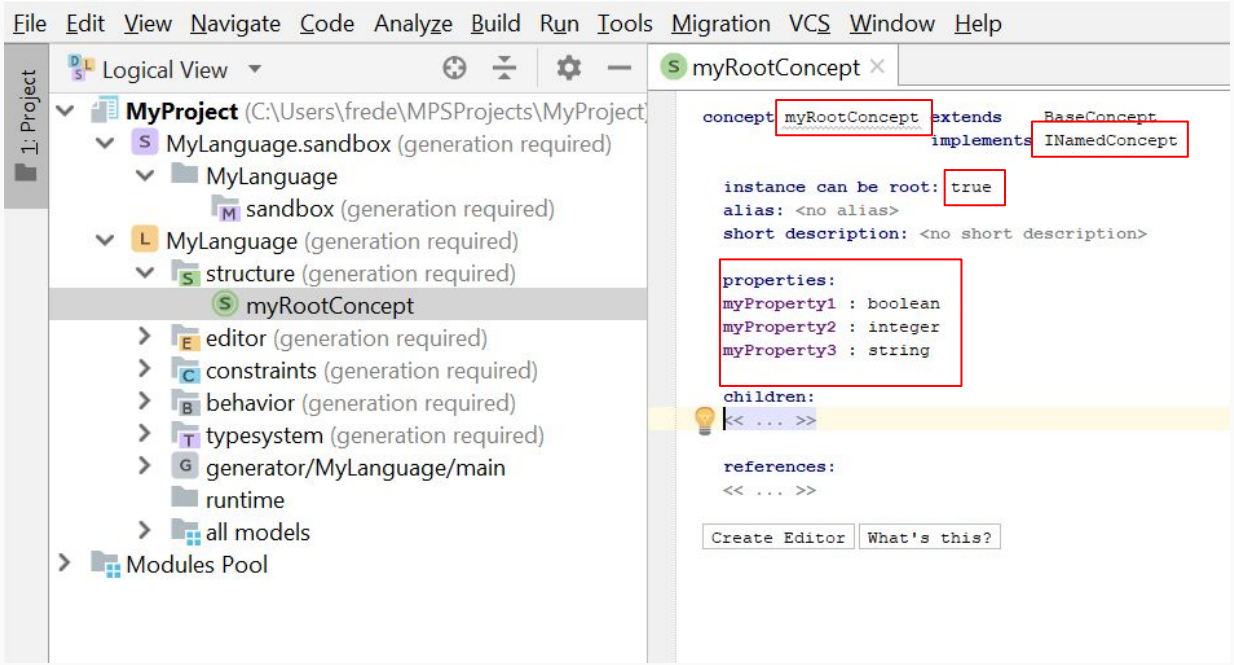
Editing the Structure Model

The structure model defines the domain concepts of our language. These concepts are arranged as a tree structure named the *Abstract Syntax Tree* (AST). It also specifies the concepts' properties, children and references, as well as the different forms of generalization that may apply to the concepts. More specifically, each concept defined in the structure model represents a node in the abstract syntax tree.

Create a Root Concept

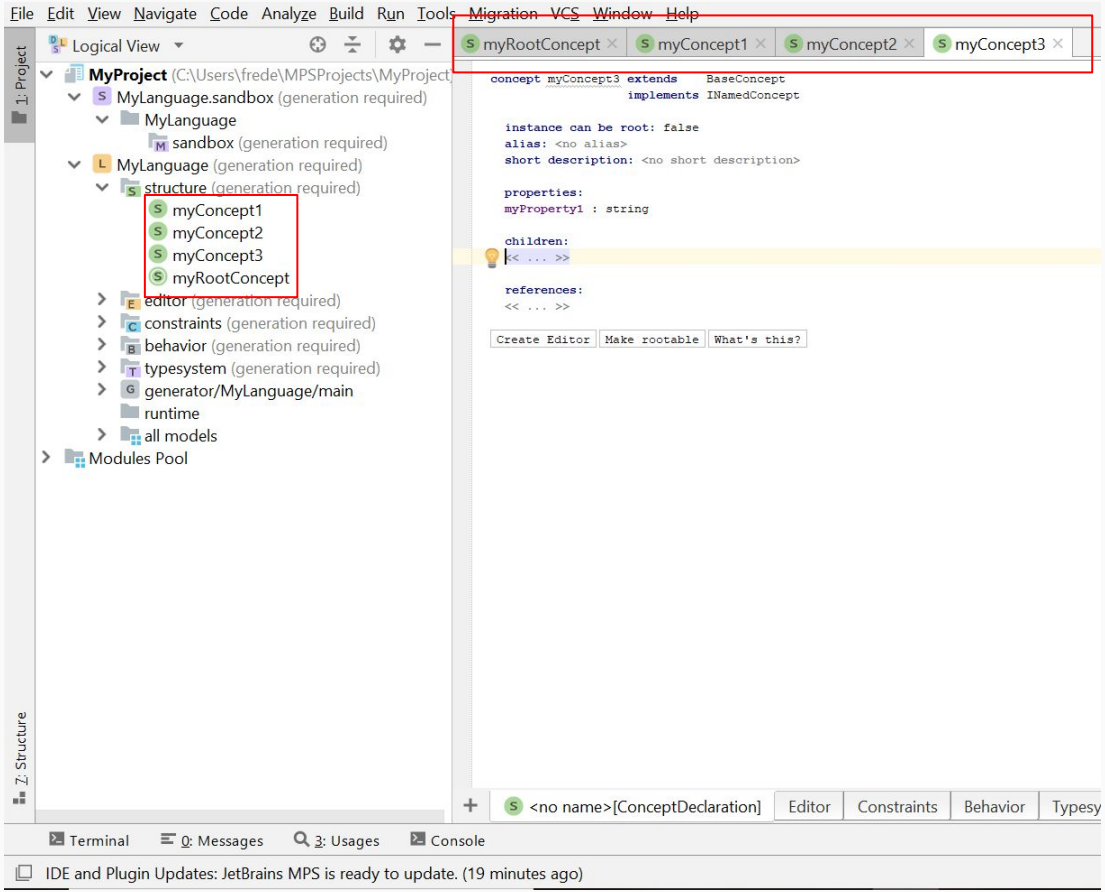


Create a Root Concept (Cont'd)



- Name the concept
- Make it implement INamedConcept which provides the concept with a name property
- Set the option "instance can be root" to *true*
- Add properties by specifying their name and their type. The type is either a Boolean, and Int or a String literal. It could also be a user defined type such as an Enumeration as we will see later.

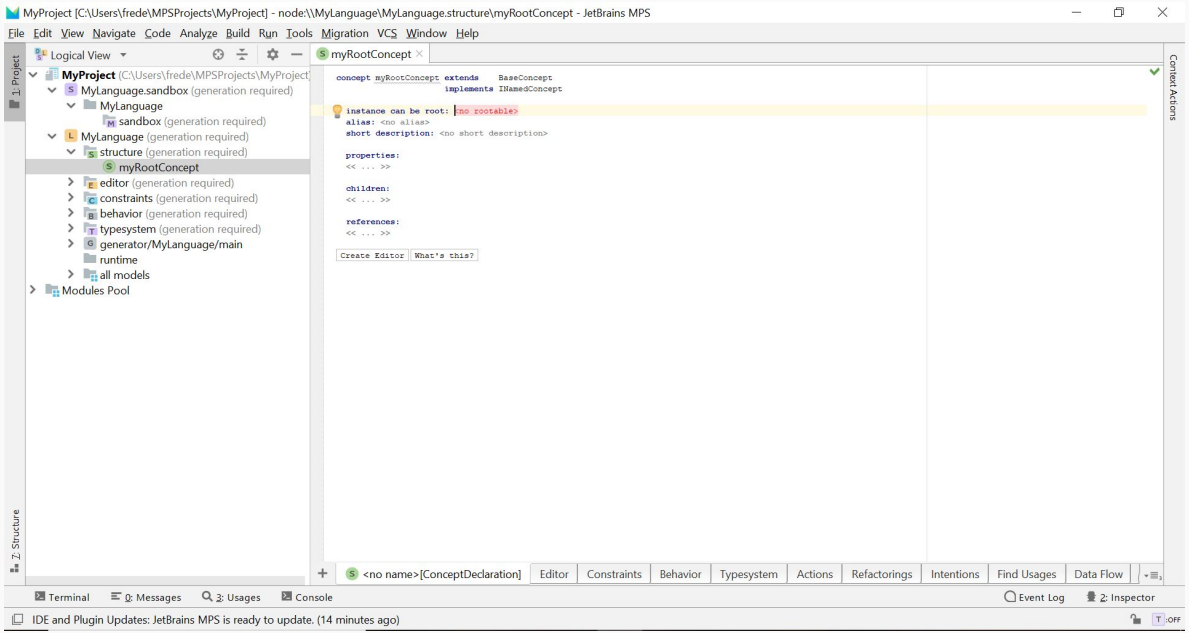
Create Other Concepts



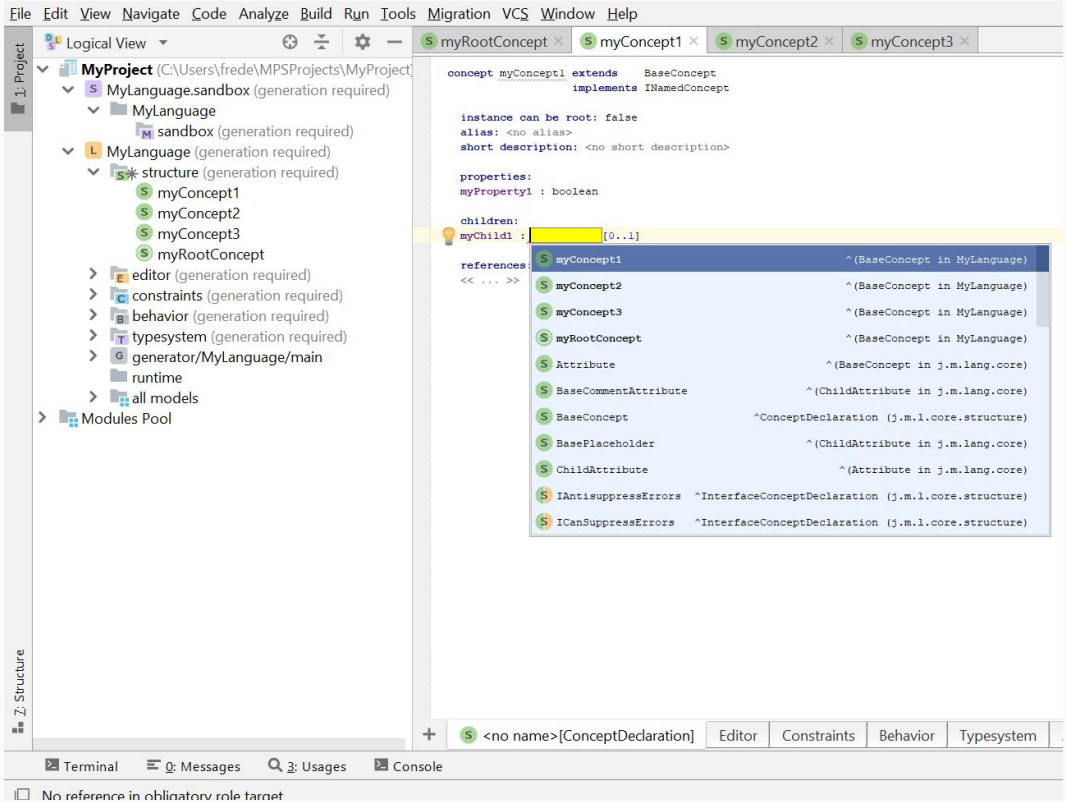
- Create other concepts in similar way. These can be root concepts or not, but you must have at least one root concept in your language.

Sidenote: Projectional Editor

The projectional editor of JetBrains MPS does not let the user write anything as an interpreted language editor would. The editor is expecting the user to write values that make sense. It even provides a list of suggestions to choose from. Simply press *Ctrl+Space* to make it appear.



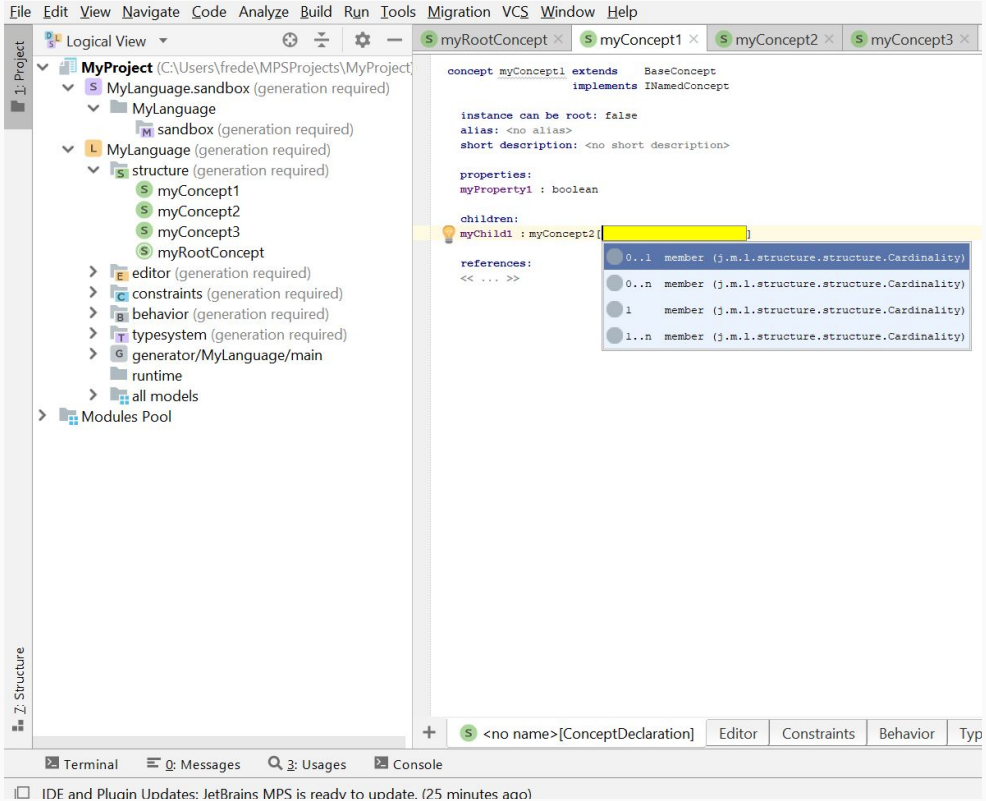
Specify Concepts' Children



- Since the abstract syntax is modeled as a tree, each concept can either have children concepts or references to other concepts.
- Children are used to define a part-whole relationship with their parents. E.g. cars (parent/whole) have wheels (child/part).

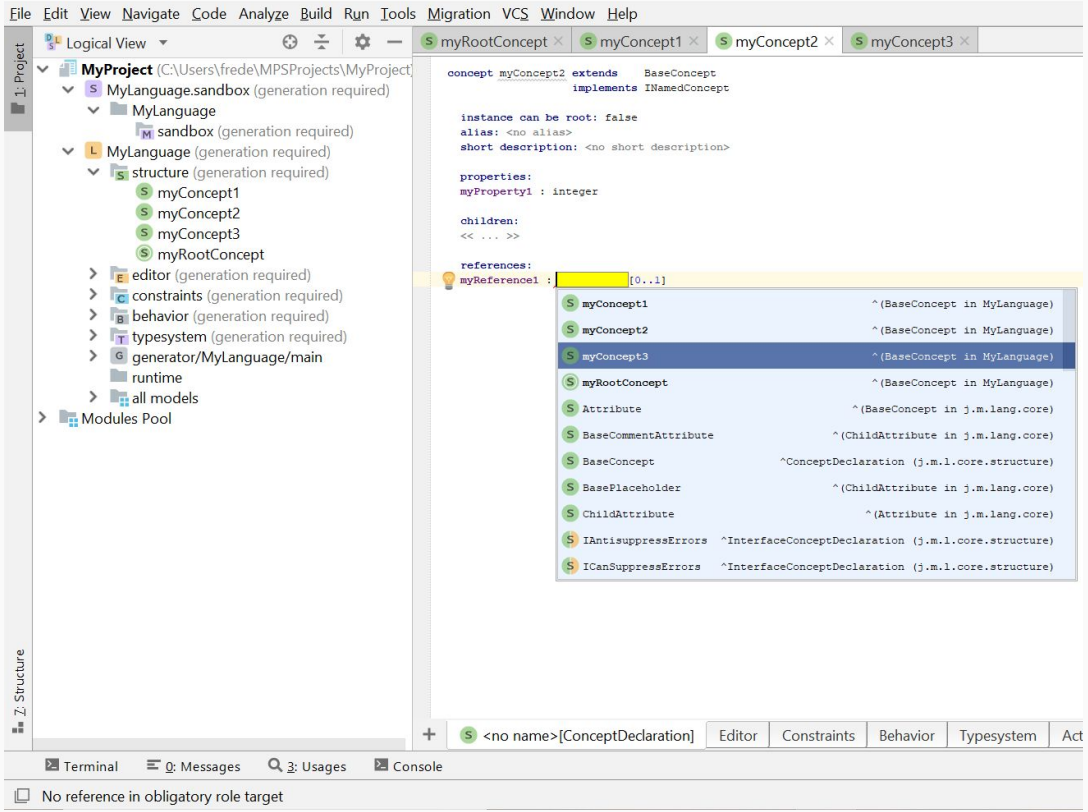
Suggestions made by the projectional editor.

Specify Concepts' Children (Cont'd)



- Specify the cardinality of the children (i.e. the number of children expected)

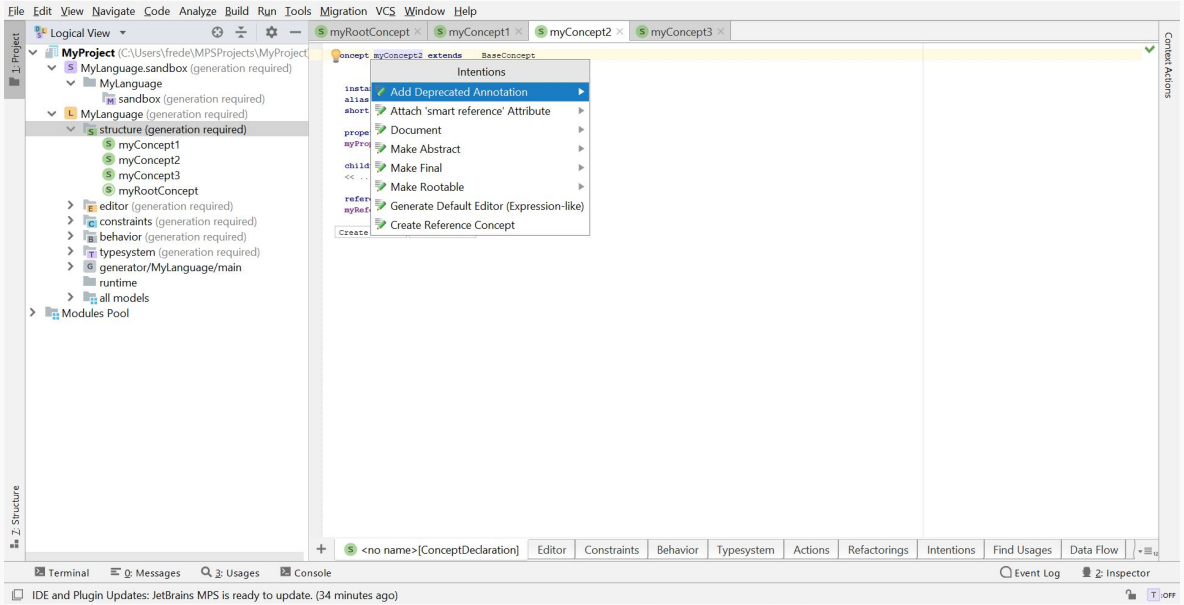
Specify Concepts' References



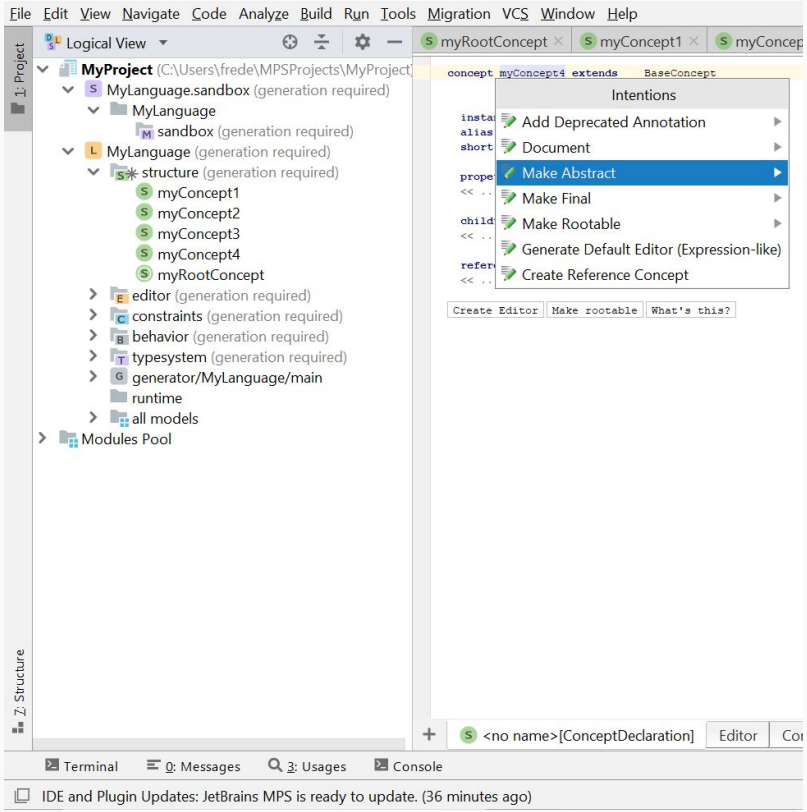
- References are not the same as children. References are used to define an association between 2 logically distinct concepts.

Sidenote: Intentions

Intentions provide fast access to the most used operations with syntactical constructions of a language. While editing its language, the user can access the most used operations by using the intentions menu. Press *Alt+Enter* to make it appear when editing.



Create an Abstract Concept



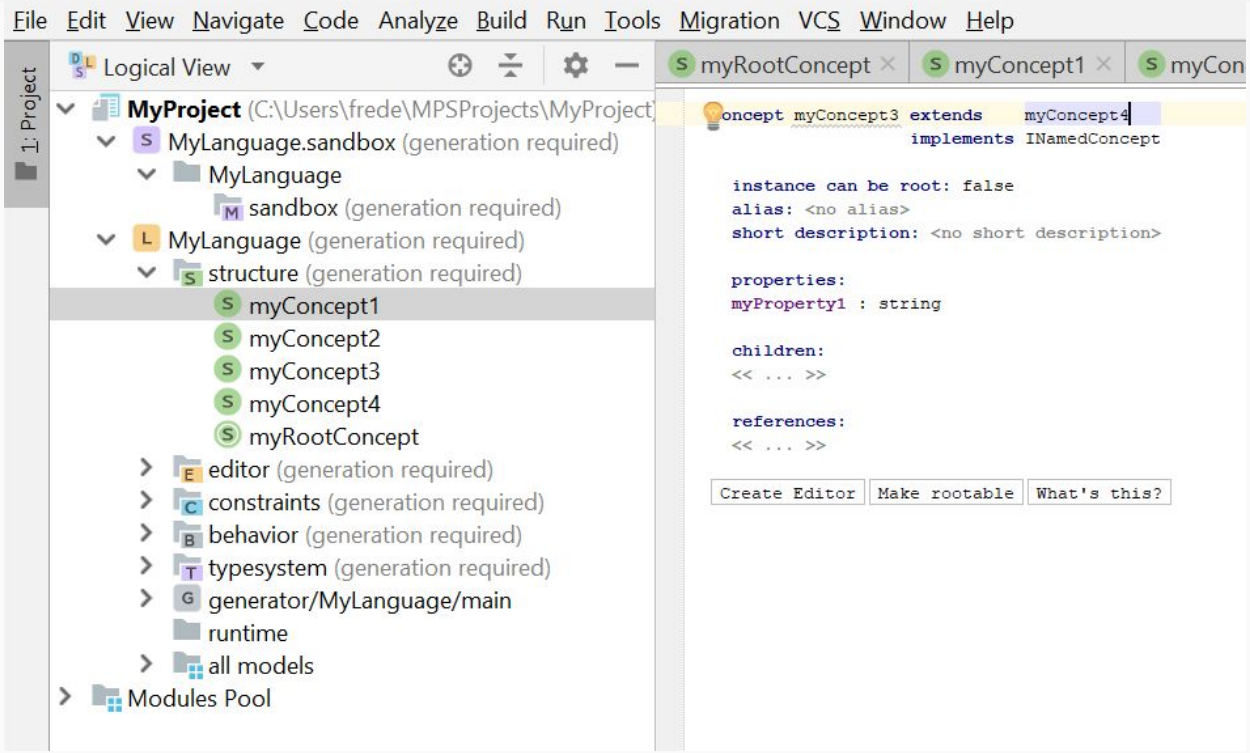
- Abstract concepts cannot be instantiated

Sidenote: Extends and Implements

Concepts can extend from other concepts and can implement conceptInterfaces. These 2 forms of generalization lead to the concept that extends/implements inheriting from the references, properties and children of the generalized component. A concept extends just once, but may implement multiple multiple times.

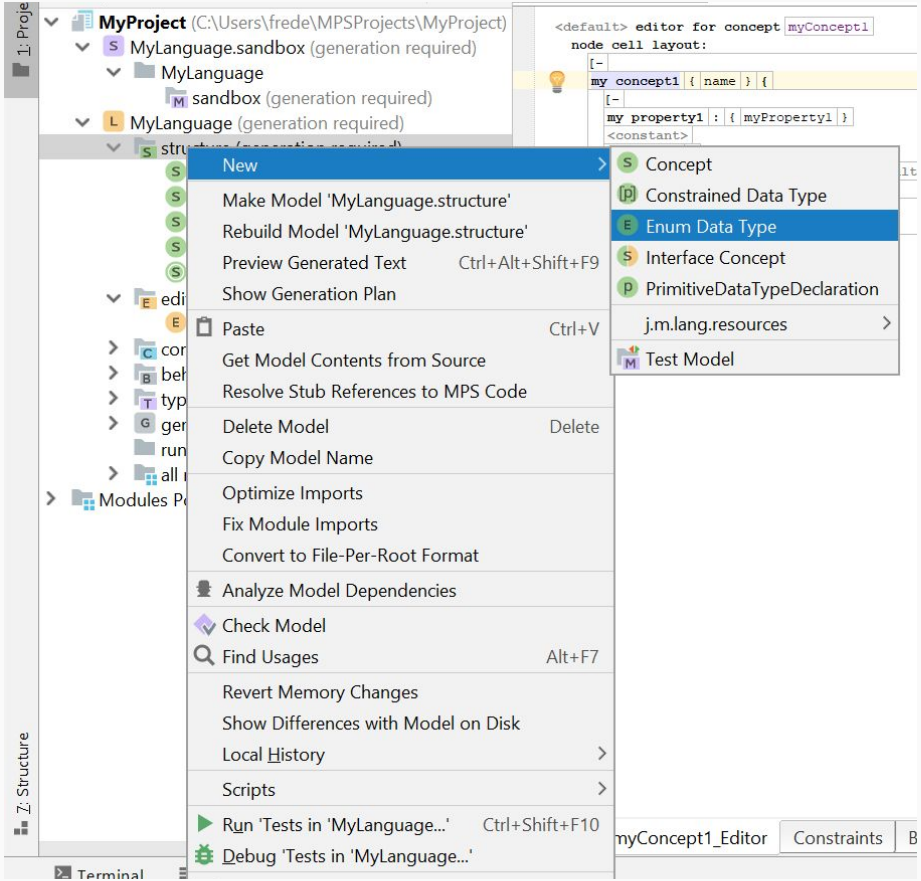
```
concept myConcept1 extends    BaseConcept
                   implements INamedConcept
```

Extend/Implement a Concept from Another Concept

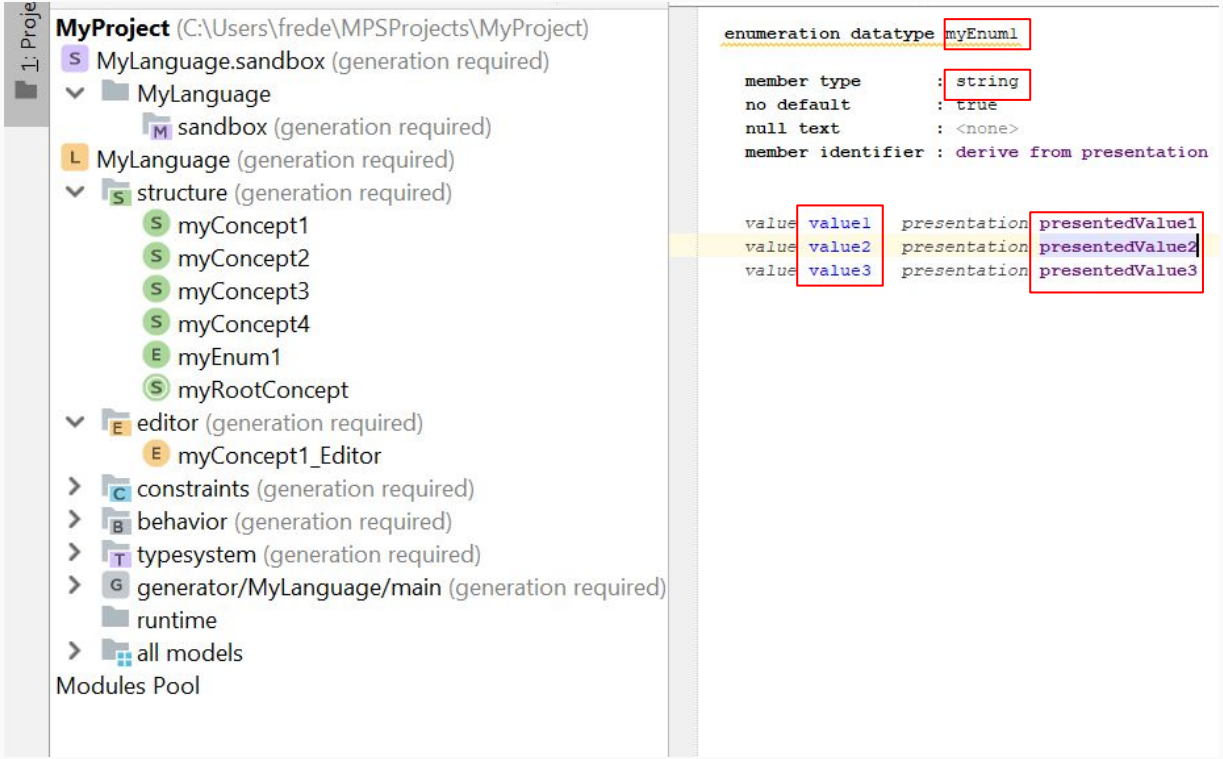


- myConcept3 extends from the abstract concept myConcept4 and implements INamedConcept.

Create an Enumeration

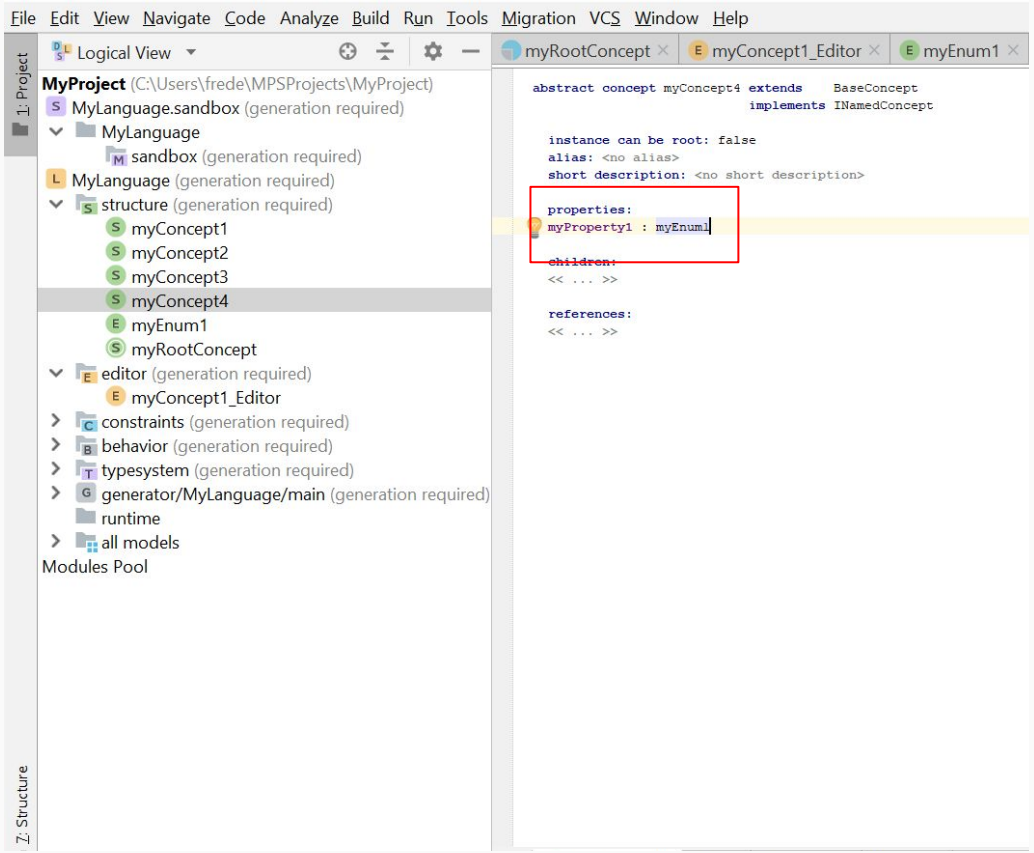


Create an Enumeration (Cont'd)



- Give the Enumeration a name
- Choose the member type. In that case the enumeration provides a list of string values.
- For each item in the enumeration, specify the value (its type = member type defined previously), and a presented value (to be presented/chosen by the user of the language).

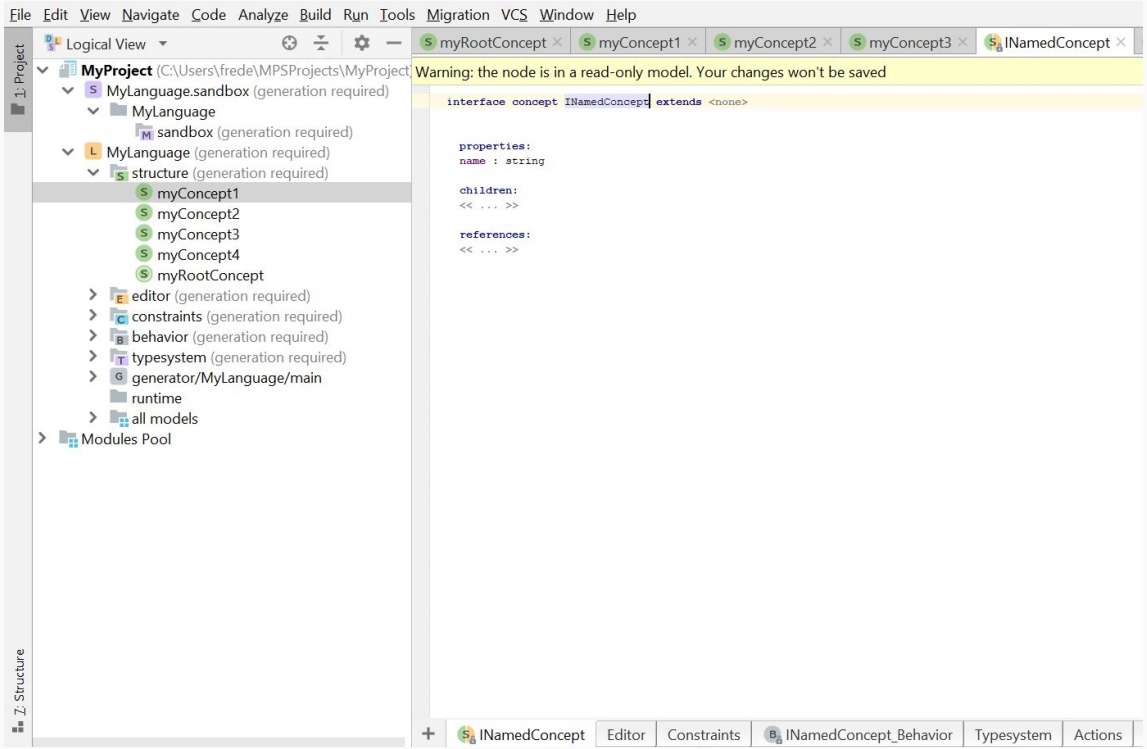
Use Enumerations as Property Type



- We define a new property for a concept and we choose its type to be *myEnum1* (Enumeration defined earlier)

Sidenote: Access Concept Definition

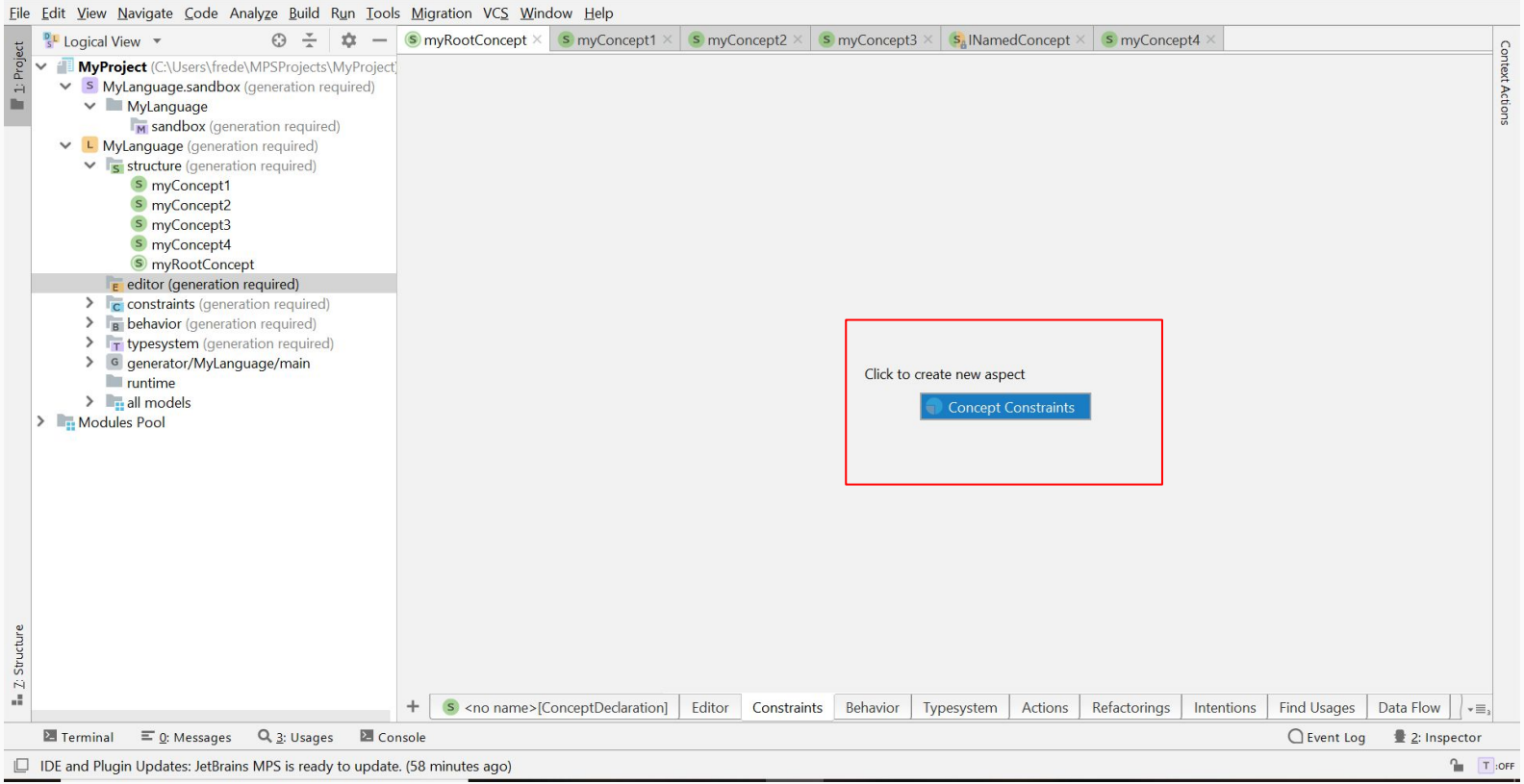
When the caret is positioned on a concept's name, the user can access its concept definition by pressing *Ctrl+B*. The image on the right presents the results of pressing *Ctrl+B* when standing over the concept interface called *INamedConcept*.



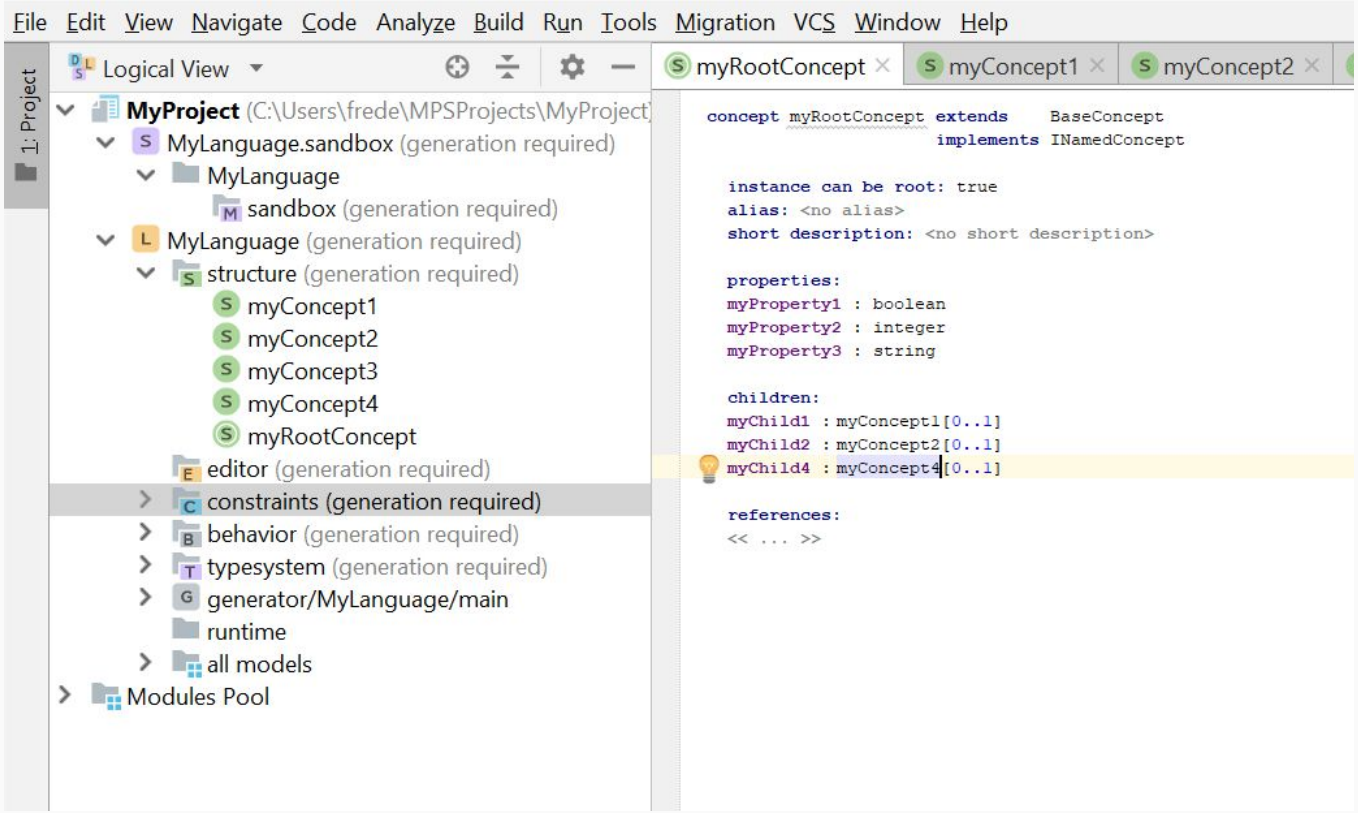
Editing the Constraints Model

The constraints model defines the constraints concerning the AST's concepts as well as their properties, children and references. The definition of the AST constraints is done using the SModel language. The user guide for this language can be found here: https://confluence.jetbrains.com/display/MPSD20182/SModel+language

Create Constraints for a Concept

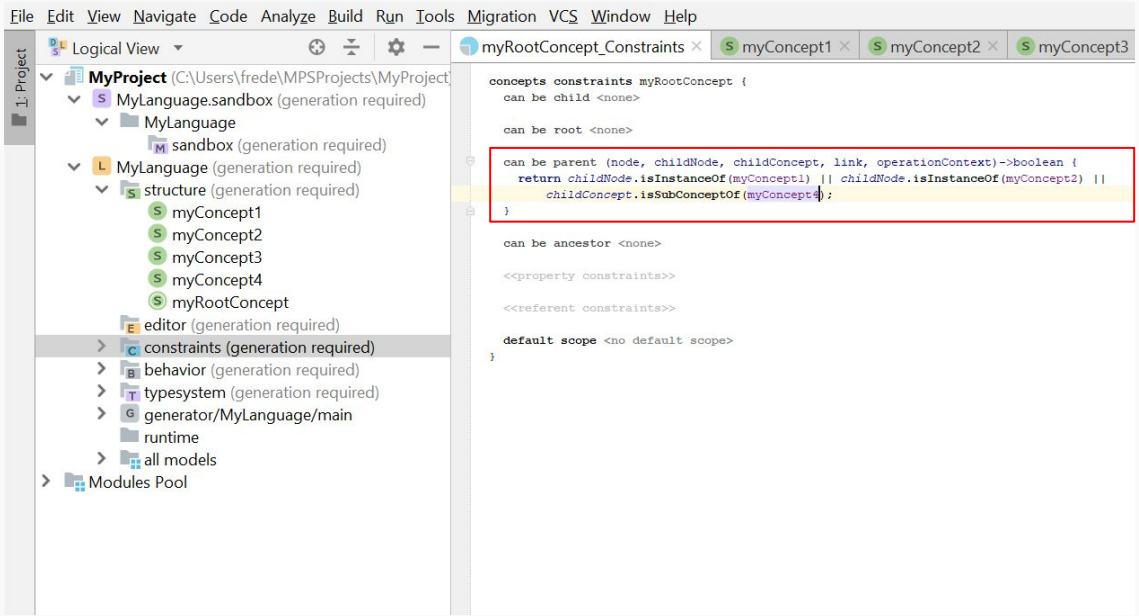


Define Child Constraints



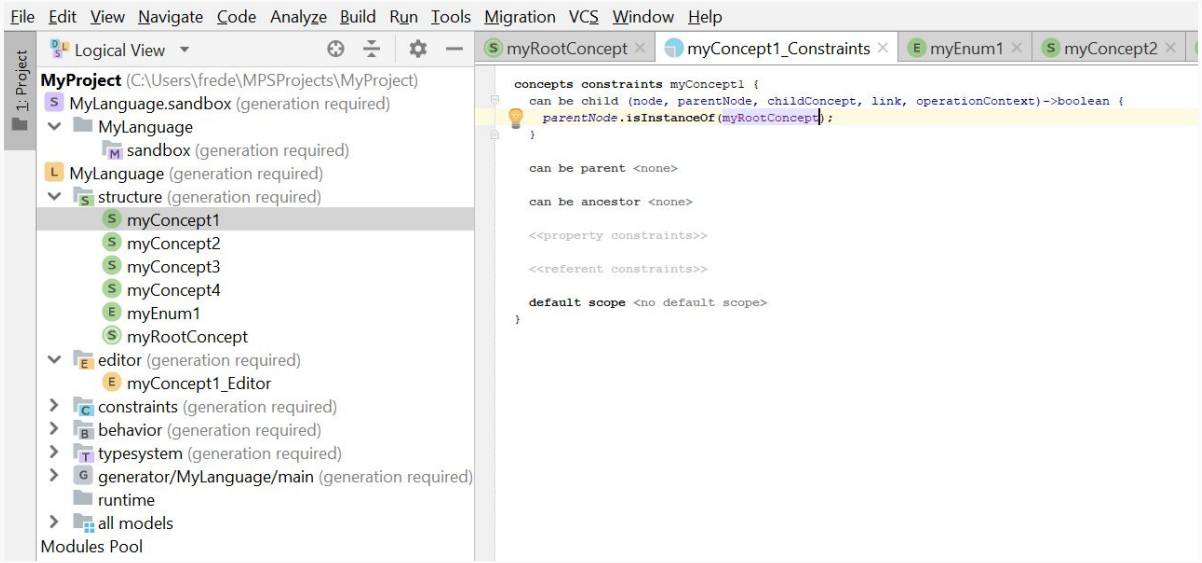
- Notice how this concept has 3 child concepts

Define Child Constraints (Cont'd)



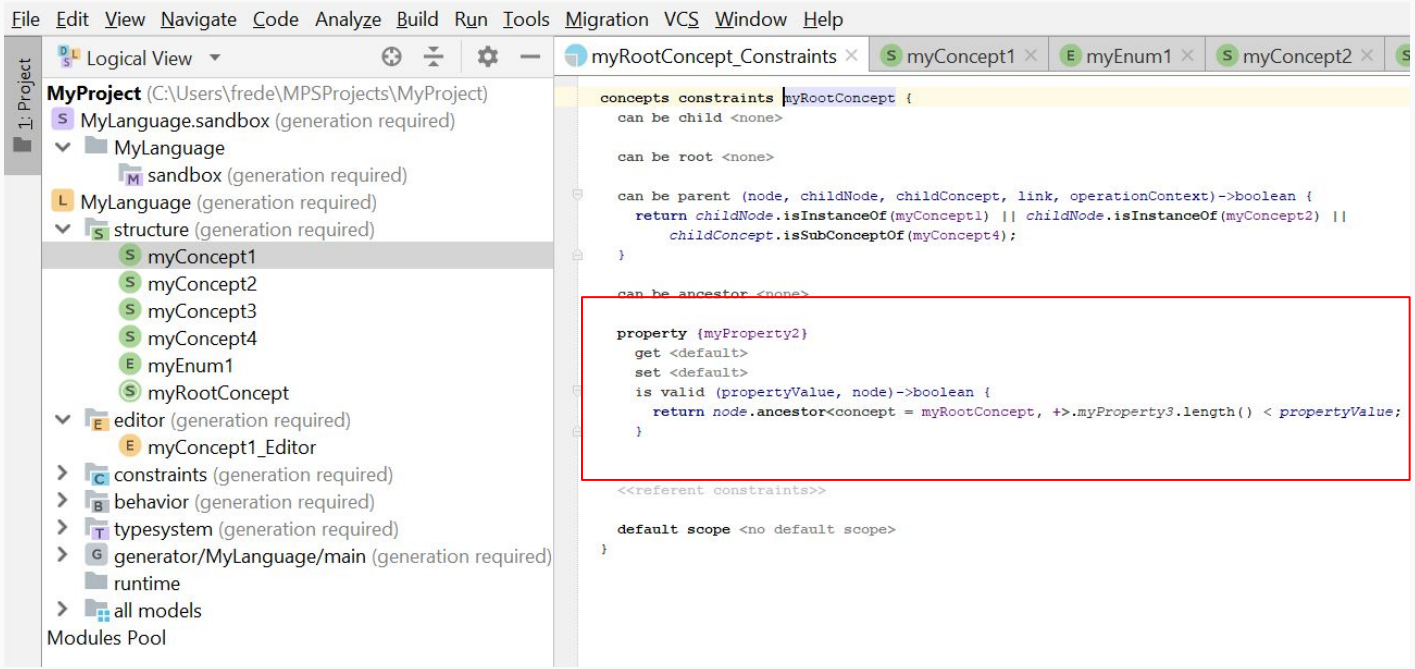
- Use SModel language to specify the child constraint.
- Here, we specify what concepts the child nodes should be instances of.

Define Parent Constraints



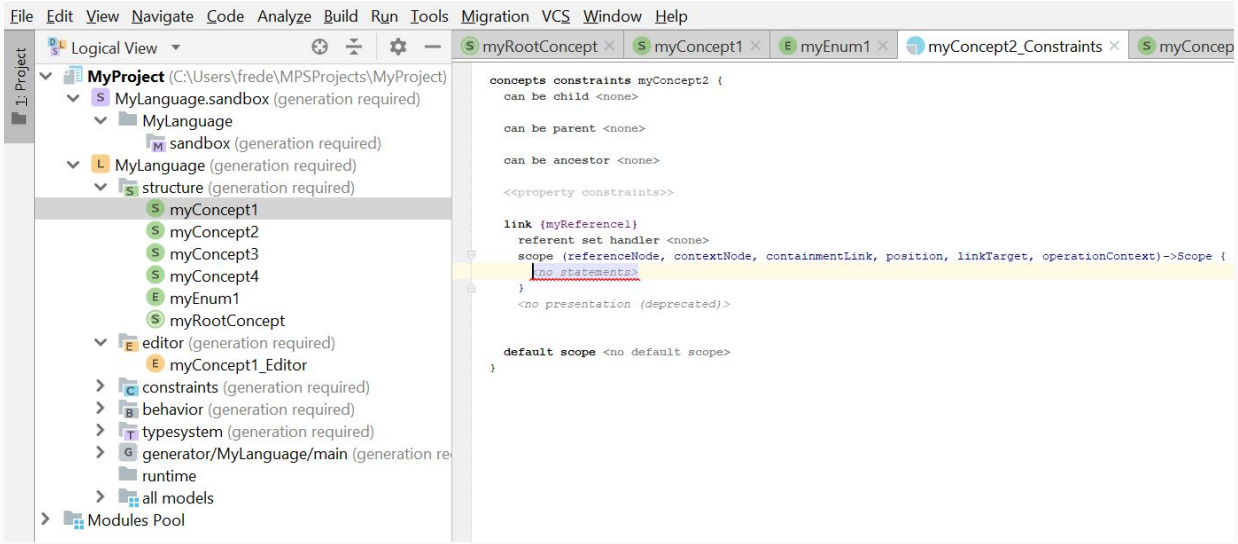
- Proceed in a similar manner to define the parent constraints for each child
- Here, we specify what concept the parent node should be an instance of.

Define the Property Constraints



- The property constraints are used to specify what values are valid for the concept properties
- Here, we sepcify that the length of the string value of myProperty3 should be shorter than the Integer value of myProperty2

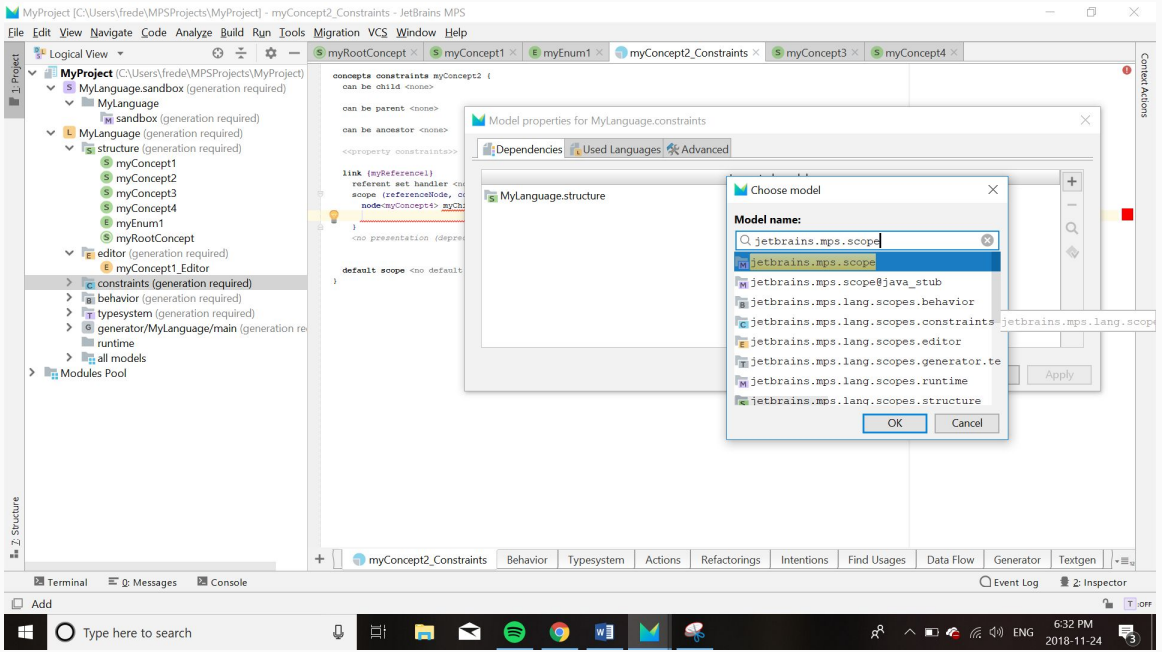
Define a Reference Constraint



- Reference constraints work by defining the *scope* of permitted values for the referenced nodes.

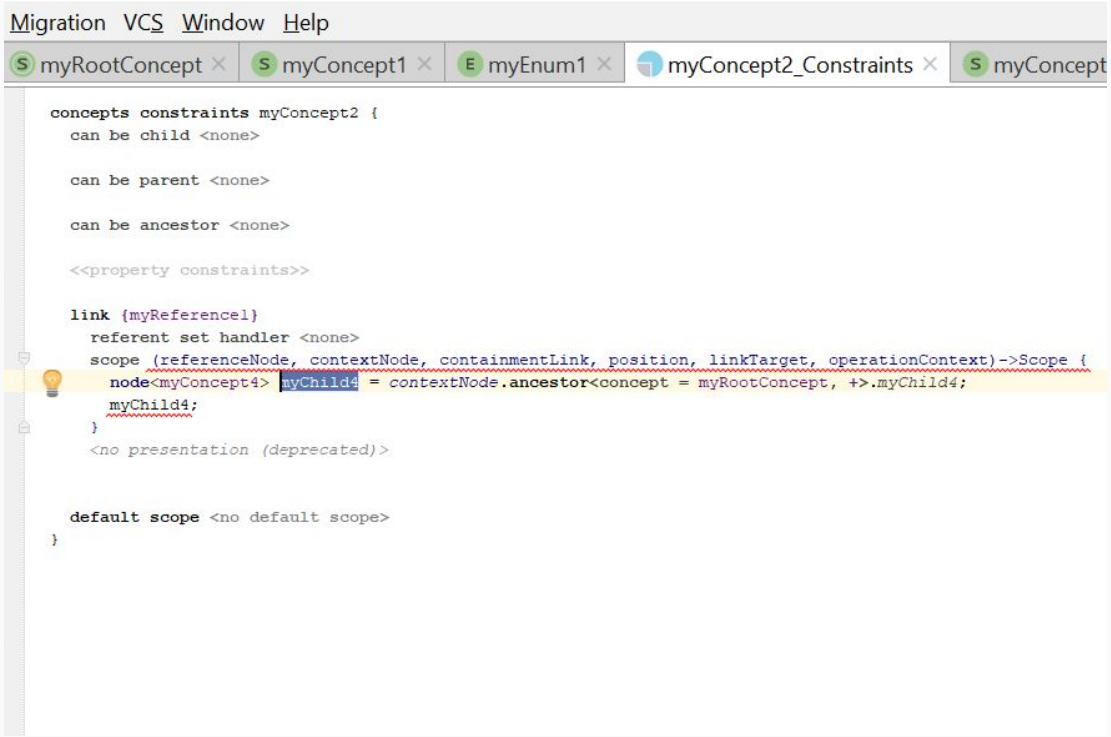
SideNote: Add a Dependency to the Model

It is possible to add a dependency to the model being edited. Import the dependency by right clicking on the Constraints model and selecting “model properties”. Then press Alt+Insert, write the name of the desired dependency, select it, press “ok” and press “ok”.

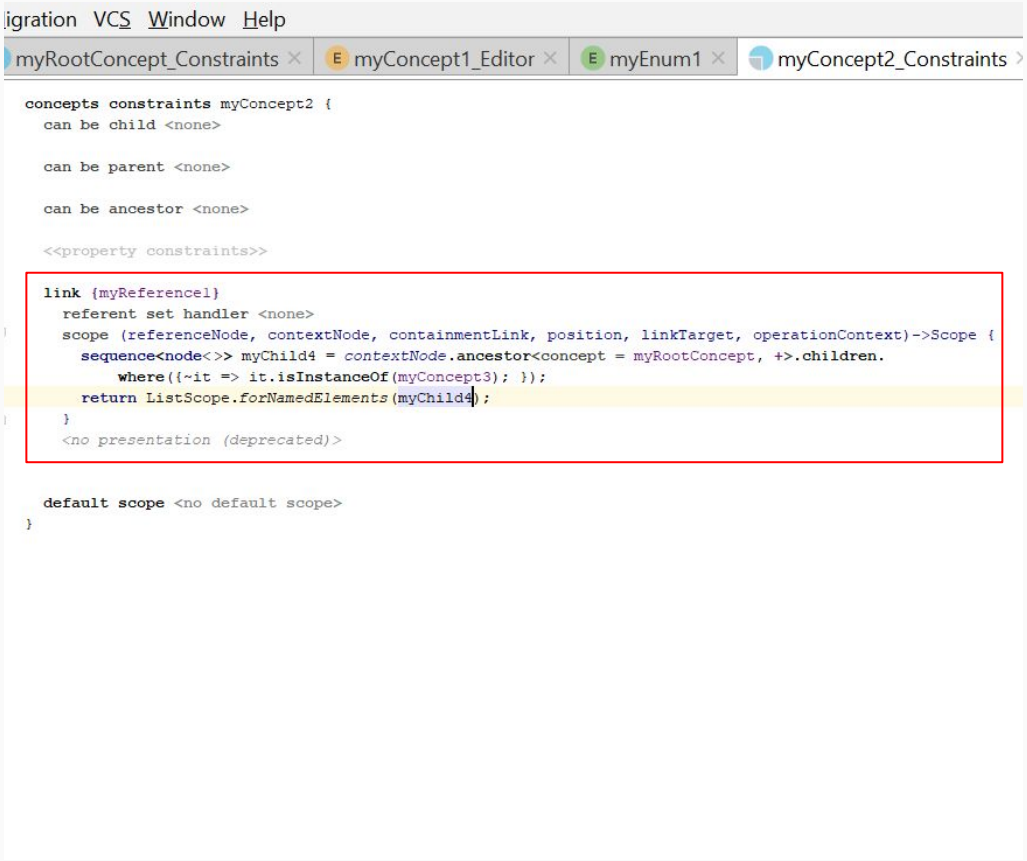


SideNote: Create a Variable

The SModel Language allows a user to define variable which hold the value of what they are associated to during their initialisation. Highlight the expression that is desired to be stored in a variable and press *Ctrl+Alt+V*.



Define a Reference Constraint (Cont'd)

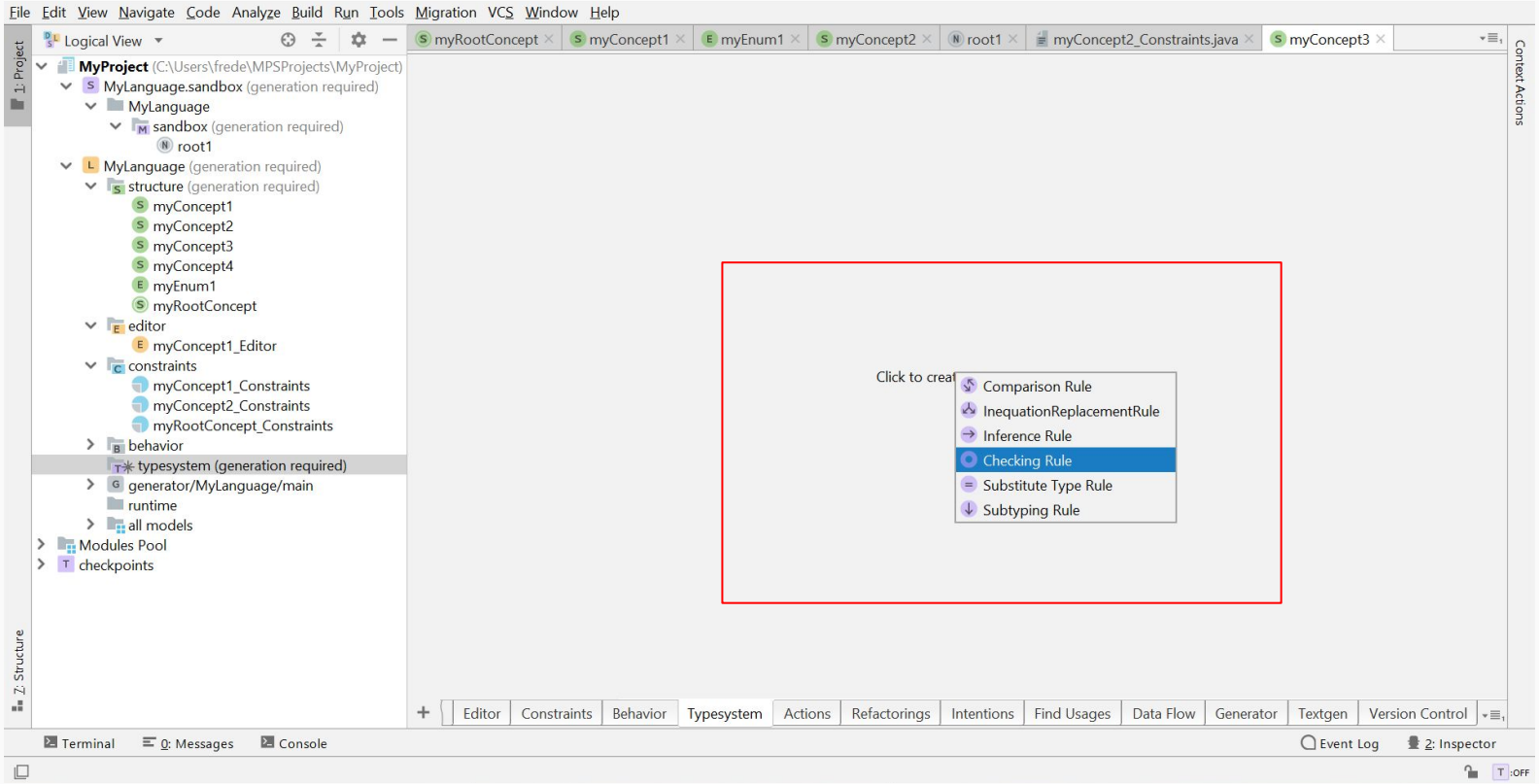


- Here, we define the scope of permitted values for the referenced nodes.
- This constraint specifies that the scope is composed of all the instances of *myConcept3* that are children of the *myRootConcept* parent node.

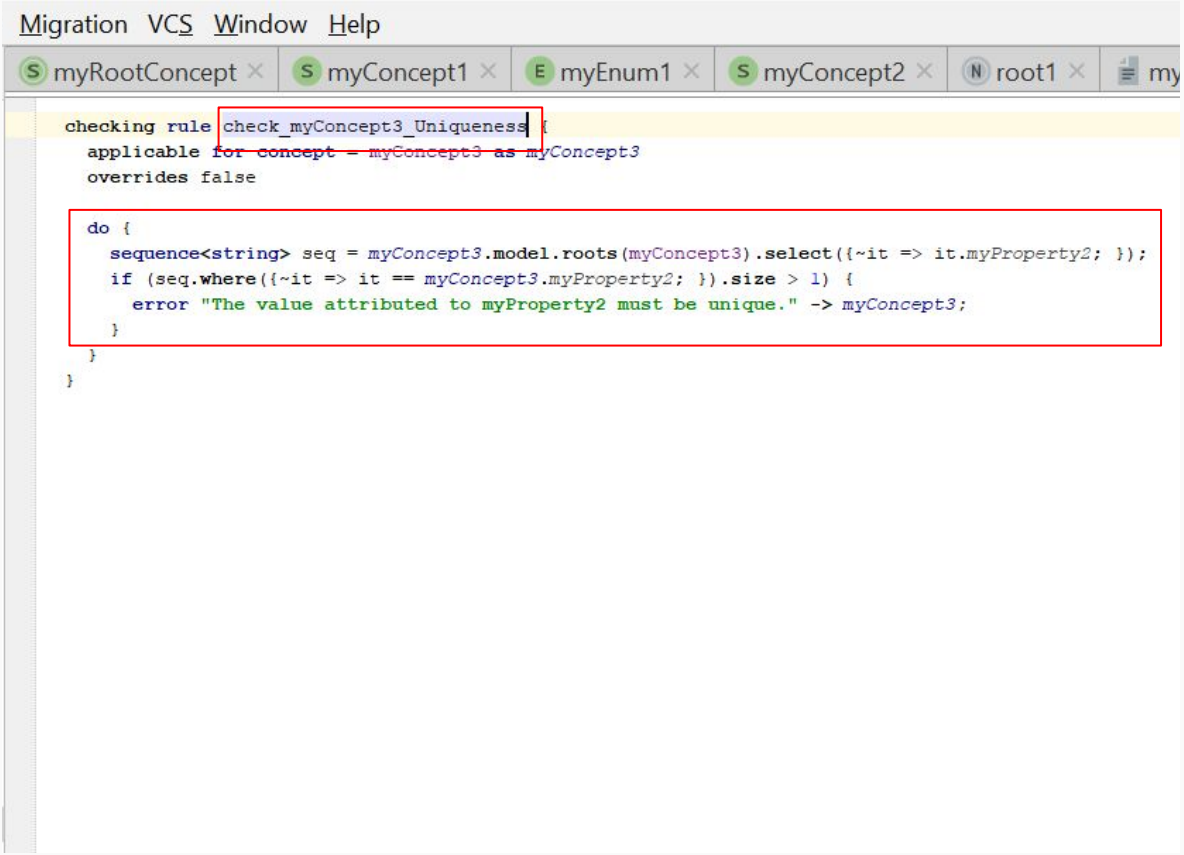
Editing the TypeSystem Model

This is part of the language definition. It is responsible for assigning types to the nodes instantiated in the models written using the language. It is also used to define constraints on the nodes and their type.

Creating a TypeSystem Checking Rule



Creating a TypeSystem Checking Rule (Cont'd)

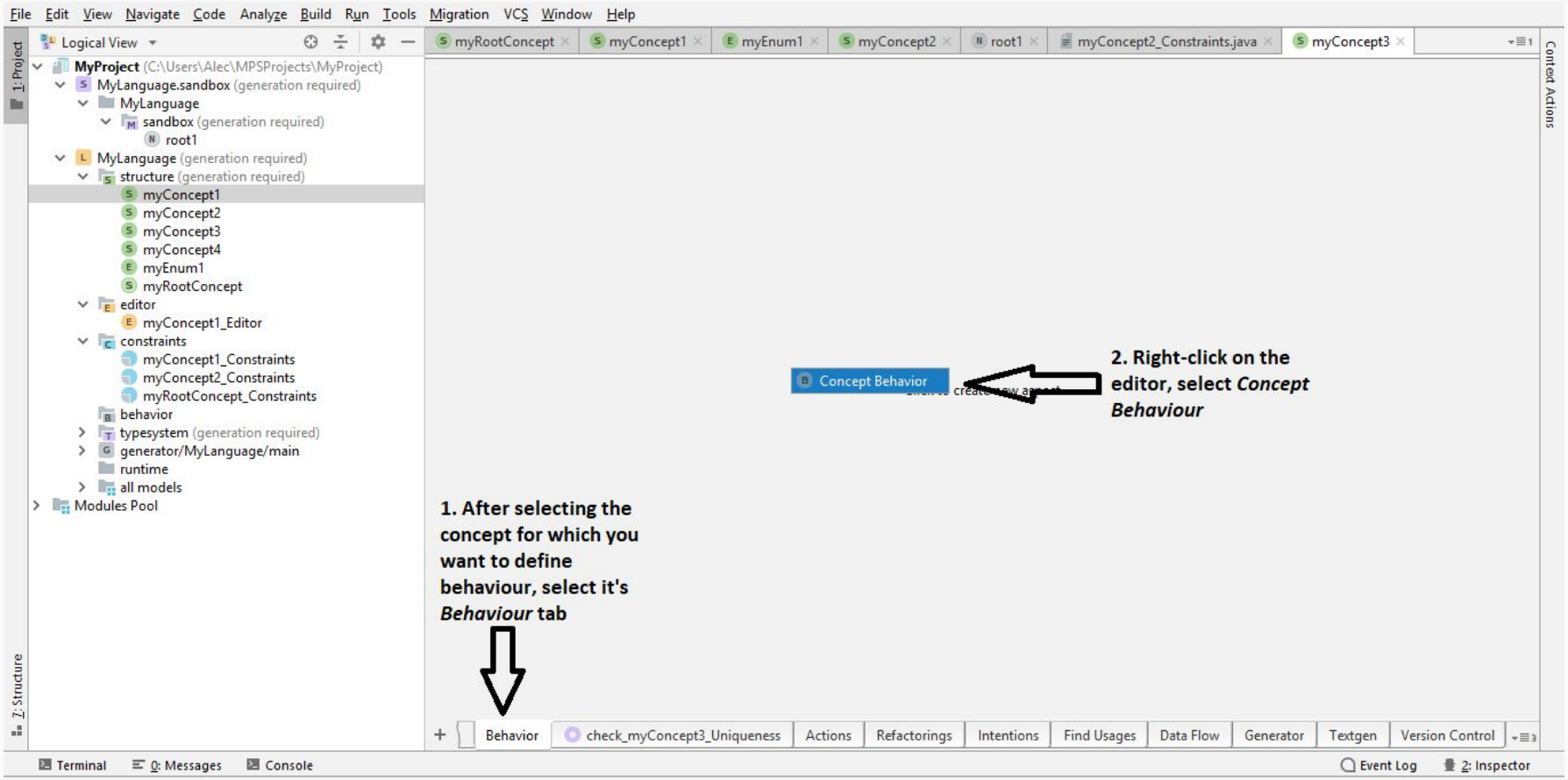


- Here, we specify a Checking rule for the concept *myConcept3.*
- Give this rule a name.
- This particular checking rule makes sure that the property *myProperty2* of a node of *myConcept3* has a unique value amongst the instances of this same concept.

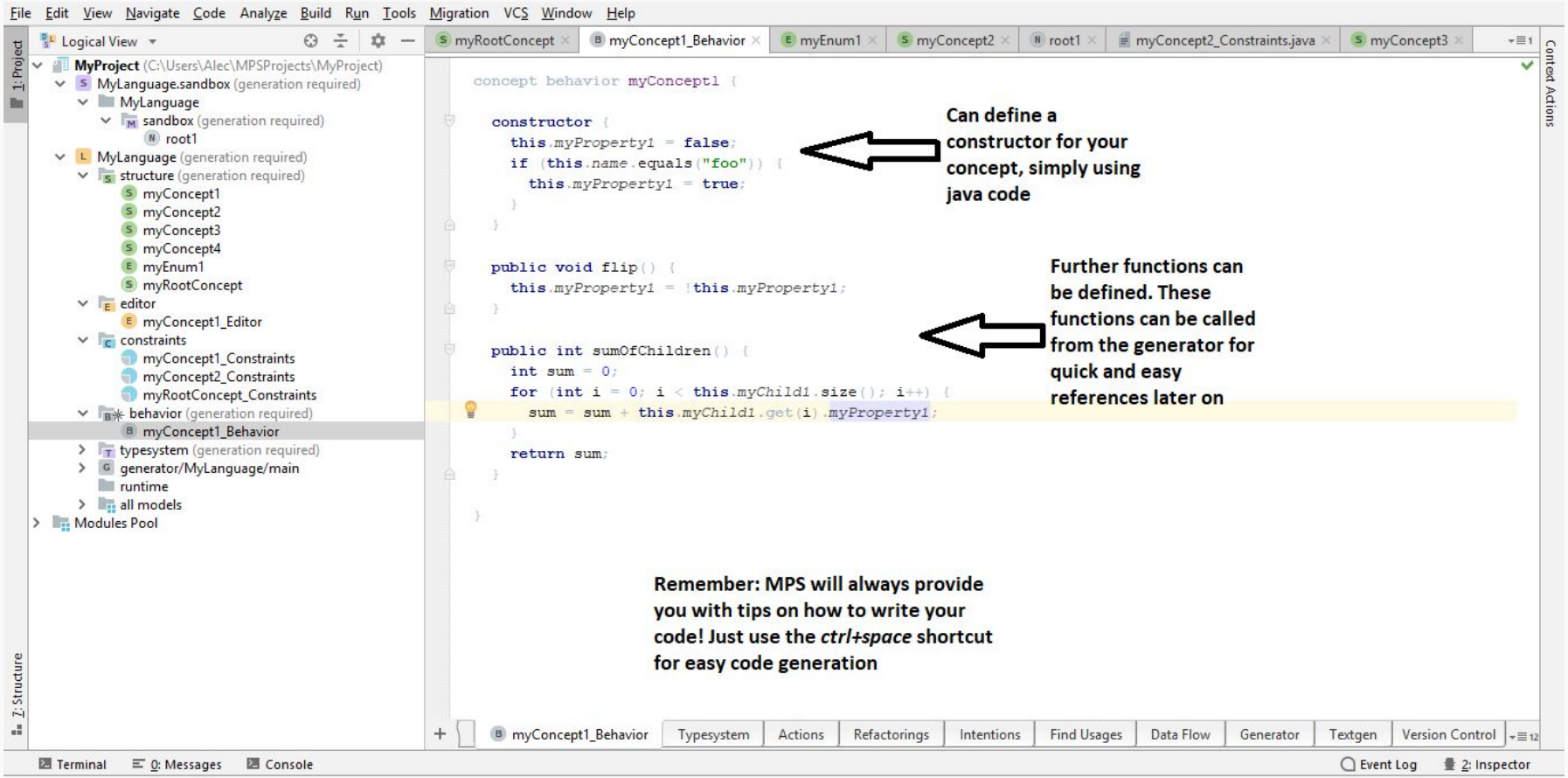
Editing the Behaviour Model

The Behaviour model defines methods that can be assigned to concept instances.

Creating Behaviour for a Concept



Defining Behaviour for a Concept

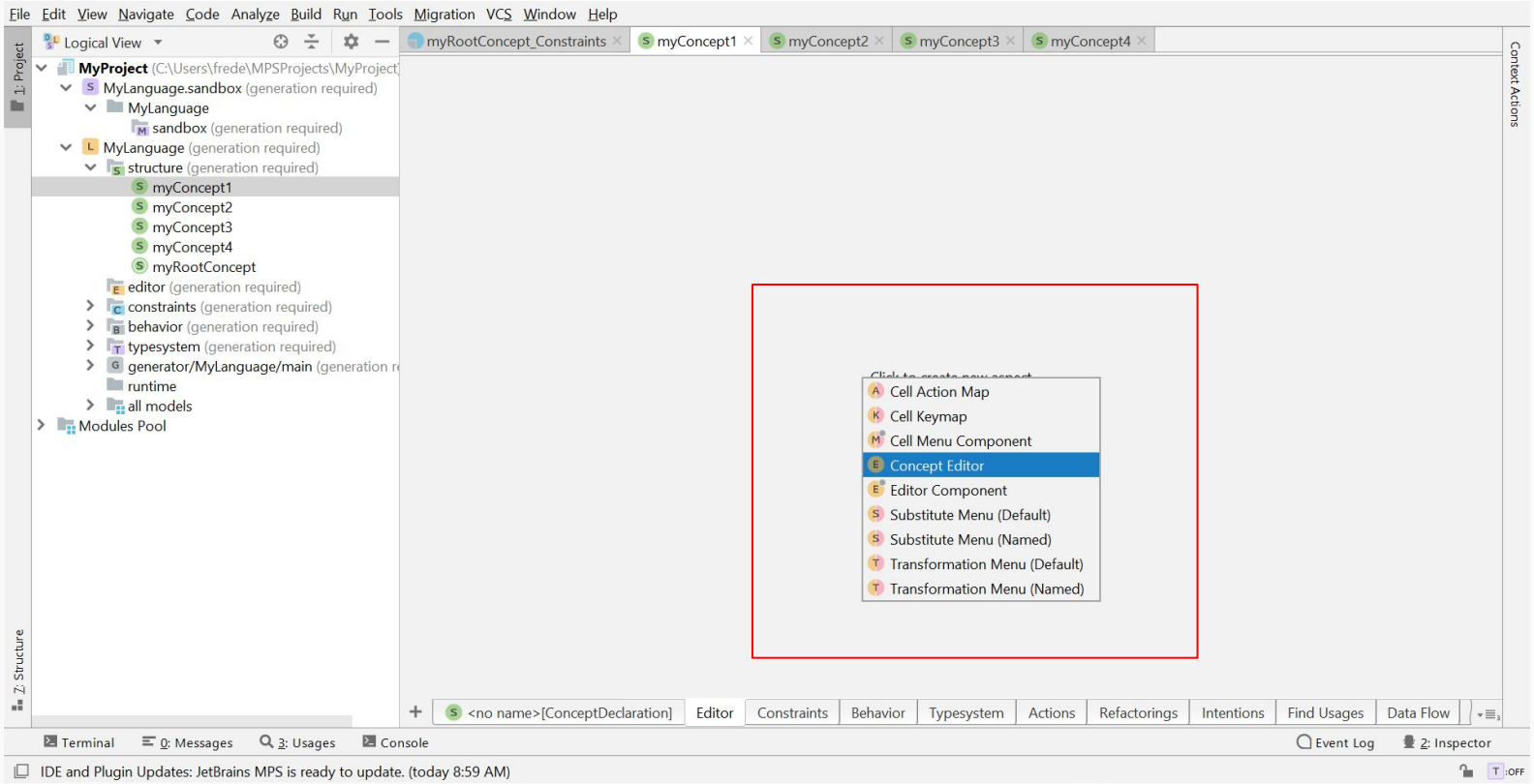


Concrete Syntax

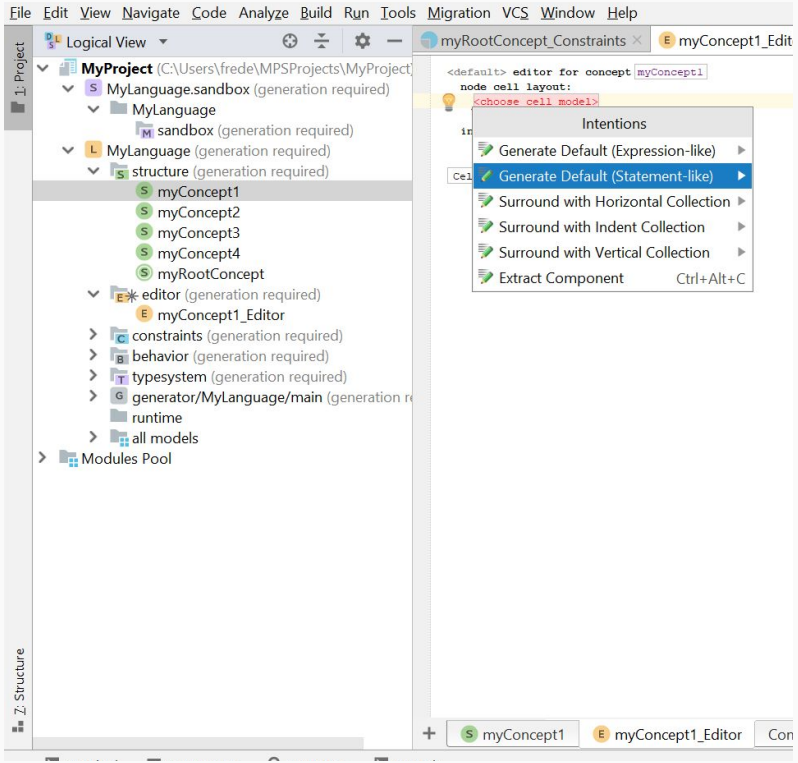
Editing the Editor Model

The editor model defines the concrete syntax associated with the concepts of the AST. The user can either decide to create its own custom editor or to generate a default editor. The generated default editor can be modified and used as a basis.

Generate the Default Editor Model

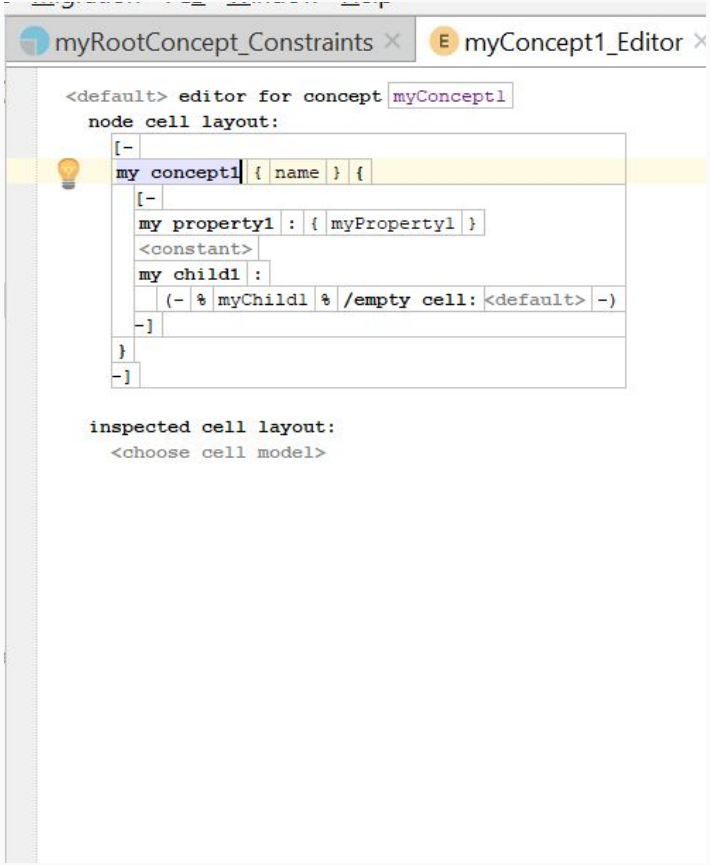


Generate the Default Editor Model (Cont'd)



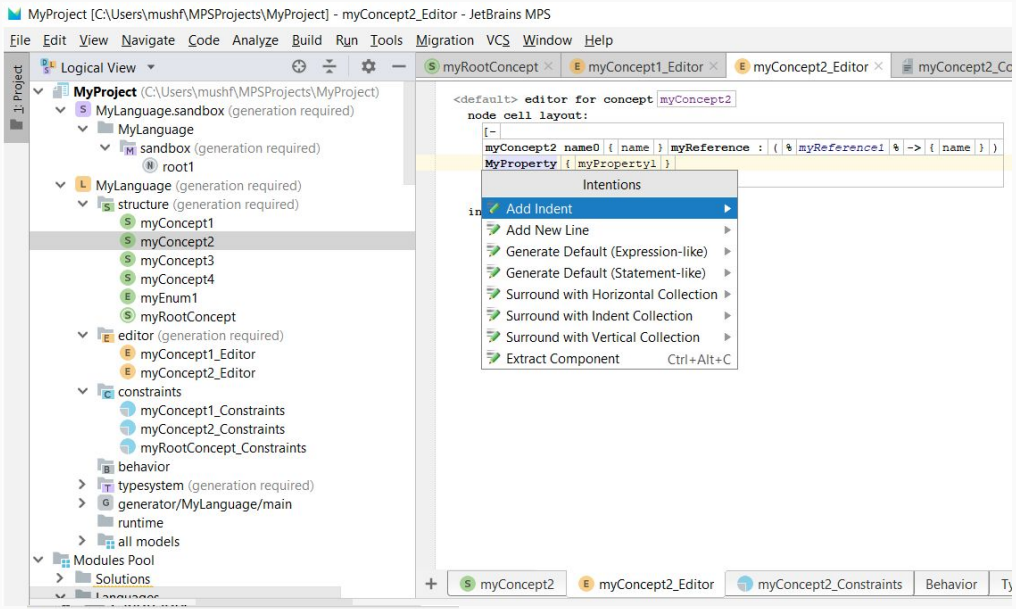
- This auto-generation of the editor provides us with an editor that allows the user to specify every aspects of a concept (i.e. properties, children, references, etc.)

Generate Editor Manually



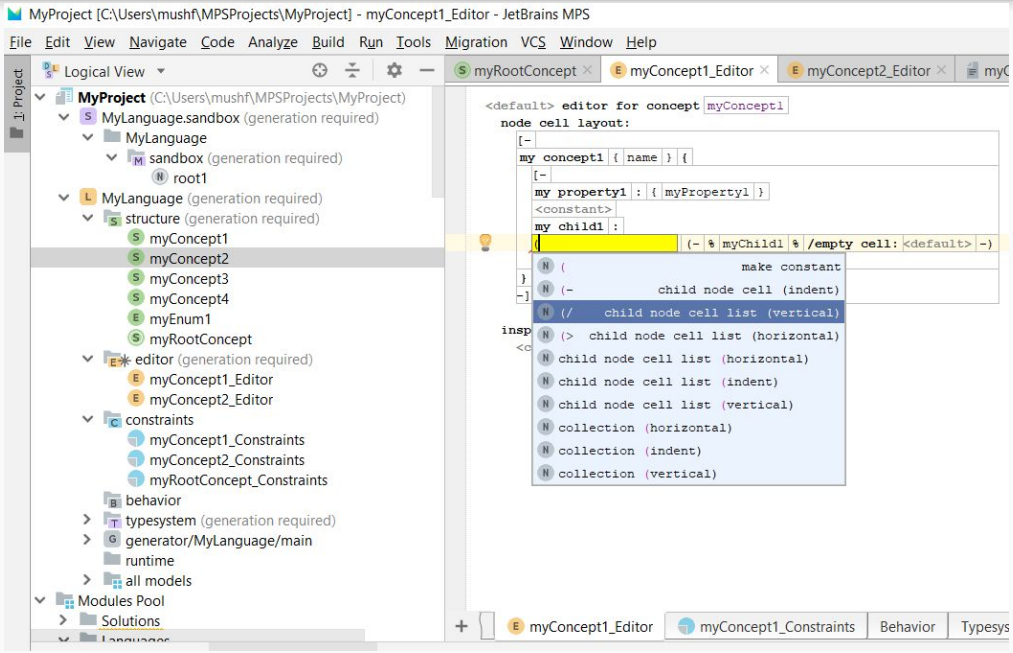
- Here is an example of an editor.
- The “[-....-]” represents a block of collection that encapsulates the layout. We can have multiple nested collections based on our needs
- The first entry is just a string literal that is not editable during the model generation.
- Entries in “{...}” are references to the properties, such as name.
- Pressing Ctrl + Space opens up all the properties, references and child nodes of the current node.

Generate Editor Manually (cont'd)



- Pressing Alt + Enter gives gives operations such as Add Indent, Add New Line, etc.
- Pressing Alt + 2 brings out the inspector for the selected component
- We can change the style manually by adding a base, for example "Keyword"
- We could also change the font size, style colour inside.

Generate Editor Manually (cont'd)



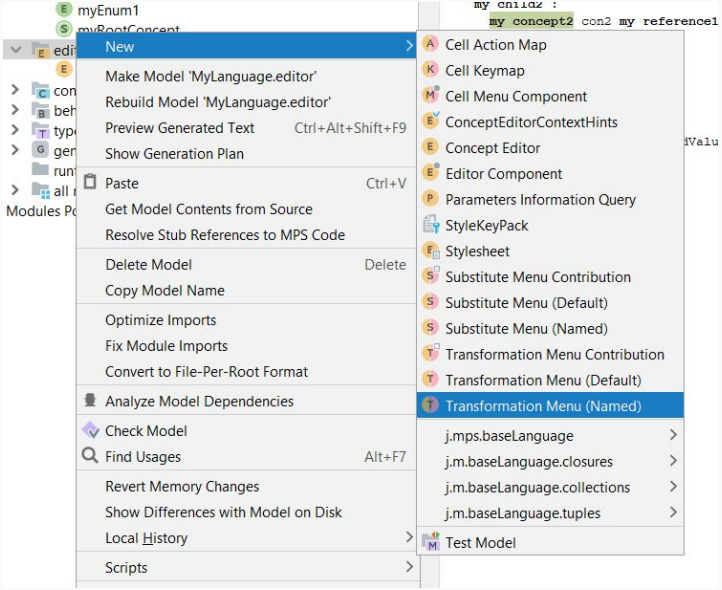
- We can find more options as we press Ctrl + Space.
- One of the options is a list of the child nodes
- We could have a Vertical or Horizontal list or we could simply have indented child node

Editing the Action Model (Editor Actions)

The editor model allows the users to work on the projectional editor in a static way, i.e. once you have created a concept you cannot change it dynamically, but to delete it completely and create another one. To support the dynamic editability the Action Model (or Editor Actions) have been introduced.

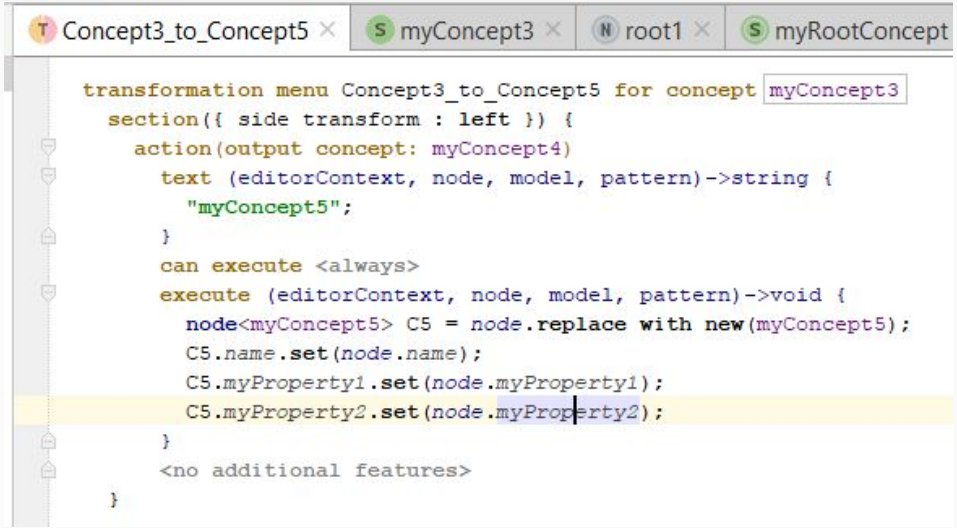
For our DSL, we have used one type of Action Model (Transformation Menu) which is described below. Also, in the MPS tutorial the Action Model is defined for an older version, whereas in the newer version of MPS the Action Model is ***merged*** with the Editor Model.

Creating a Dynamic Editor (Transformation Menu)



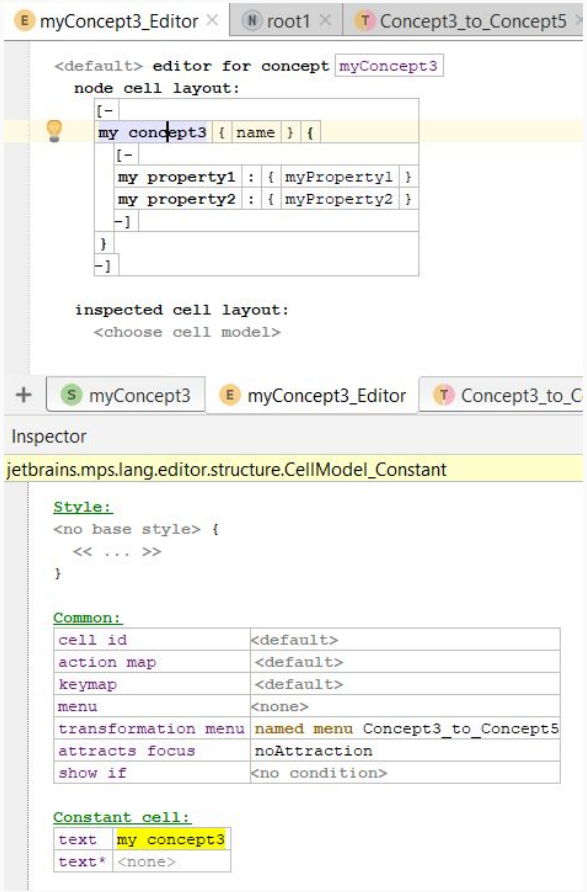
- We create a Transformation Menu (Named) in the editor model.
- This will allow us to Transform a specific concept to another specific concept, along with specific properties, references and child types.

Creating a Dynamic Editor (Transformation Menu) (Cont'd)



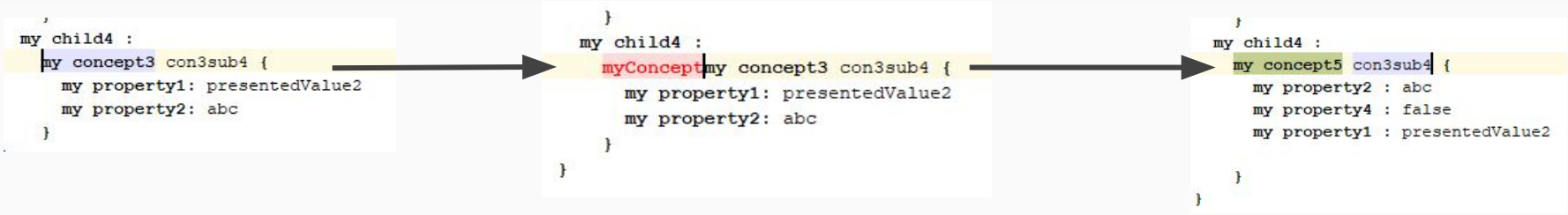
- The name is set to be *Concept3_to_Concept5*
- This is implemented on *myConcept3*
- The transformation type is "side transformation" from the "left" side.
- In the action block, we need to press Alt + Enter to bring out the option of adding the output concept, which, in this case is *myConcept5*.
- The text block specifies what the user will need to type in order to transform the concept. Here we specify "myConcept5".
- The execute block implements the actual transformation.
- A new instance of myConcept5 is created (C5) which will replace the current node concept (node).
- Finally we set the properties of the current node to those of C5

Creating a Dynamic Editor (Transformation Menu) (Cont'd)



- The next step is to connect the concept to the transformation menu.
- We do that my clicking on the desired cell in the desired concept editor, here its "my concept3" cell in "myConcept" editor.
- Then we press Alt + 2 to bring up the Inspector
- Inside the Inspector we add the transformation menu to the one we have defined. We can use Ctrl + Space to show us the list of the available transformation menus

Runnning the Dynamic Editor (Transformation Menu)



- Here we can see that unlike previously the user can type something before the concrete syntax of “my concept3”, but he/she can ONLY type what has been defined in the transformation menu, in this case “myConcept5”
- As soon as the user finishes typing the keyword, the concept is converted dynamically while preserving the properties as specified in the transformation menu.
- We could notice that “my property4”, which was not specified in myConcept3 has been initiated to default value as myConcept5 is initialized.

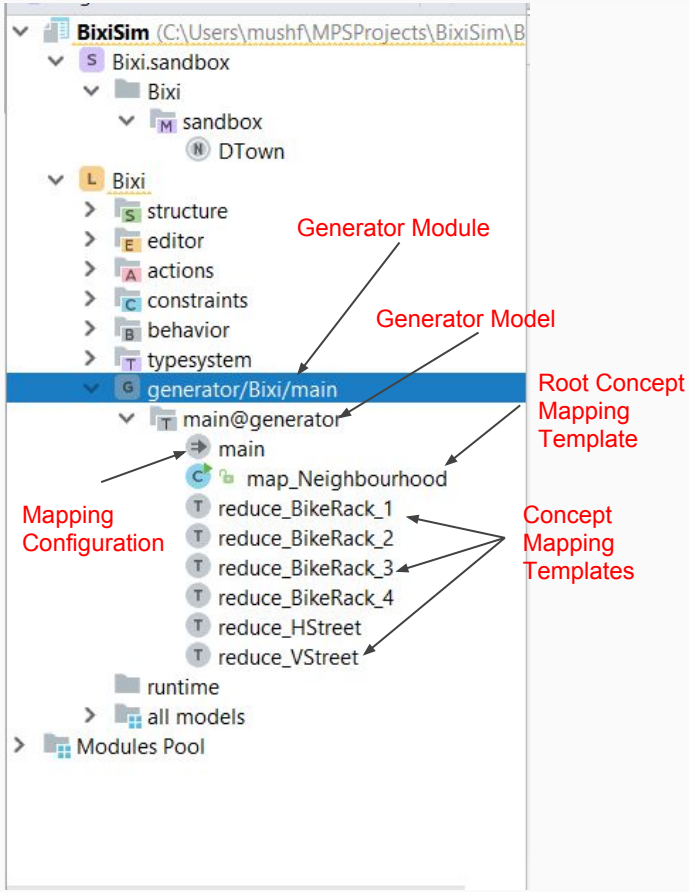
Semantics and Transformation

Editing the Generator Module

The semantics of our DSL is defined using the Generator Module. MPS follows the model-to-model transformation approach. The MPS generator specifies translation of constructions encoded in the input language into constructions encoded in the output language. For our DSL we have translated our DSL to java swing application to run our simulation for the Bixi Domain.

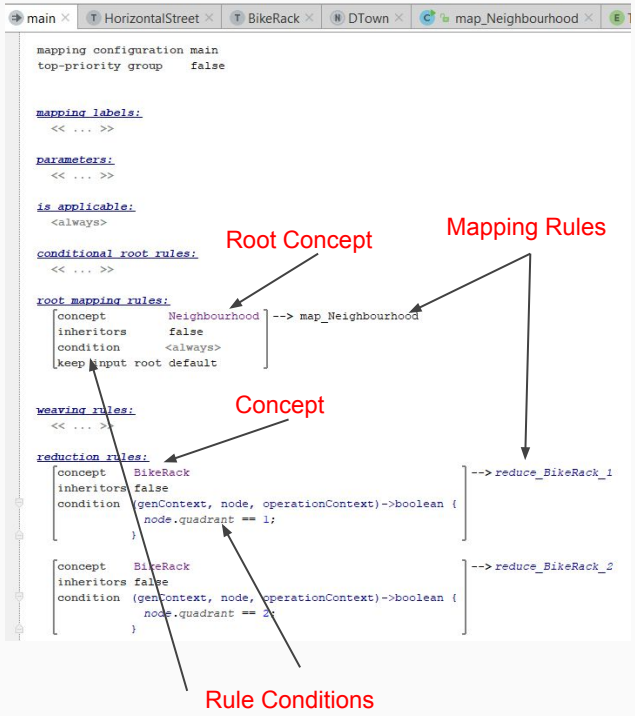
To illustrate the Generator Module, we have used our actual DSL instead of a generic version.

Generator Module Overview



- The Generator Module is found inside the Language Module, distinguished by the term “Generator”
- The Module contains the generator model with sterotype - ‘<name>@generator’, in this case it’s main@generator.
- This model is created by default when one opens up a new project.
- The model by default contains an empty mapping configuration node, called ‘main”.
- We can then add more mapping templates for the different concept nodes in our DSL.

Generator Module Overview (Cont'd)



- In the "main" configuration node, the projectional editor allows us to define the mapping configurations for different aspects of the language.
- For our DSL, we needed to use 2 of the aspects - "root mapping rules" and "reduction rules"
- The root mapping rules allows to set the rules of mapping the root concept, which is defined in a template "map_Neighbourhood"
- The reduction rules allows to set the mapping rules for the other node concepts.
- For all the rules to be implemented the condition defined in the rule must be met.

Generator Module Overview (Cont'd)

map_Neighbourhood | main | HorizontalStreet | reduce_BikeRack_1 | DT

Text: exec Match Case Regex Words

```
root template
input Neighbourhood

public class $[map_Neighbourhood] extends JFrame {
  public ArrayList<Street> streets = new ArrayList<Street>();
  public ArrayList<Movement> movements = new ArrayList<Movement>();
  public ArrayList<BikeRack> racks = new ArrayList<BikeRack>();
  public ArrayList<Member> members = new ArrayList<Member>();
  public int day = 8;
  public int time = 100;
  public boolean initialized = false;
  public boolean iterate = false;
  public static void main(string[] args) {
    map_Neighbourhood n = new map_Neighbourhood();
    n.initialize();
  }

  public void initialize() {
    $LOOP$[streets.add(new Street("$[name]", $[1]));]
    $LOOP$[members.add(new Member($[1], "$[name]"));]
    Street hStreet = null;
    Street vStreet = null;
    $LOOP$[for (Street s : streets) {
      if ("$[name]".equals(s.name)) {
        hStreet = s;
      }
      if ("$[name]".equals(s.name)) {
        vStreet = s;
      }
      try {
        if (!hStreet.equals(null) && !vStreet.equals(null)) {
          racks.add(new BikeRack($[1], $[1], $[1], hStreet, vStreet));
          hStreet = null;
          vStreet = null;
```

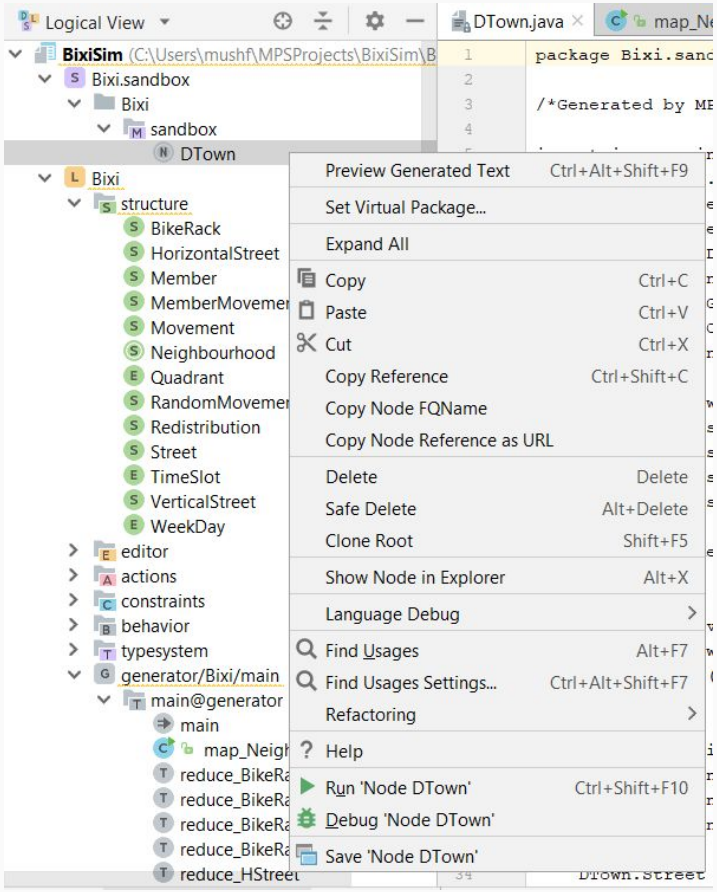
- Inside the mapping rule template for the root node (Neighbourhood) we have implemented our java.swing code.
- Inside the template we have used macros to access the different nodes and their properties.
- There are 2 types of macros used - node macro and property macro.
- The node macro, example *$LOOP$[...]*, allows to access the different concept nodes.
- The property macro, ("*$[...]*") allows to access the different properties of the concept node inside the parenthesis.
- The macros can be edited inside the "Inspector", which is accessed by the pressing *Alt + 2*.

Inspector

jetbrains.mps.lang.generator.structure.PropertyMacro

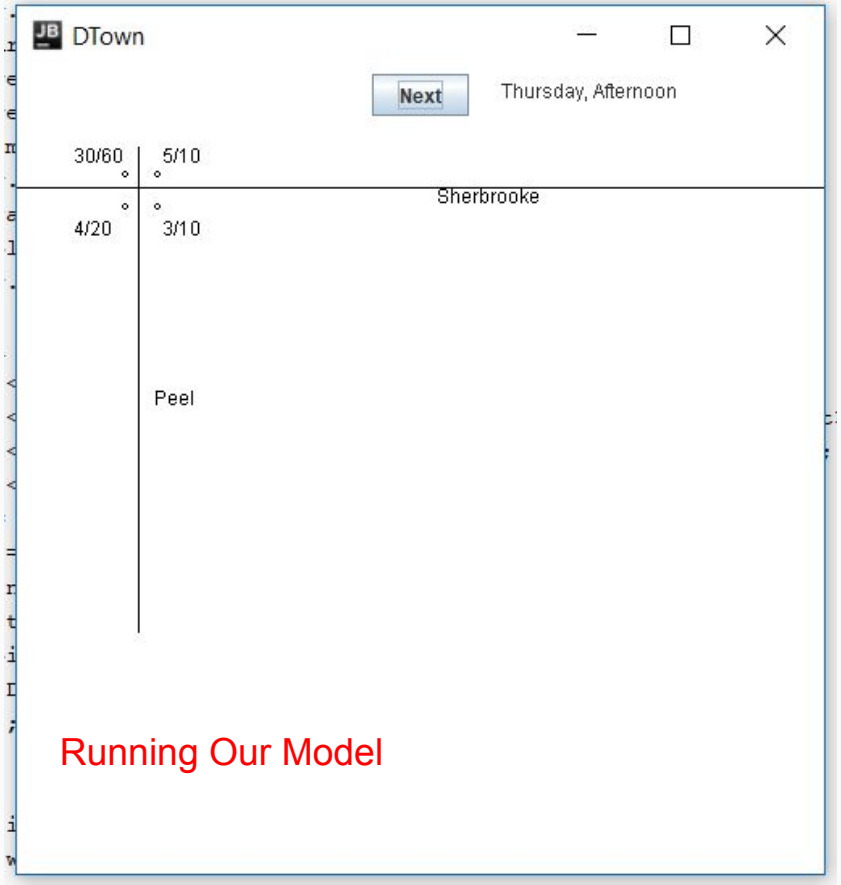
```
property value

comment : <none>
value : (templateValue, genContext, node, operationContext)->string {
          node.name;
        }
```

Generator Module Overview (Cont'd)



- After we have implemented all the mapping rules, we rebuild the whole project.
- We then right click on the sample model that we created using our DSL (DTown)
- We can click "Run 'Node DTown'" to run our model which is shown in the slide below.
- We can also click "Preview Generated Text" to see the generated text file (java file) which is given in the slide below

Generator Module Overview (Cont'd)



Running Our Model

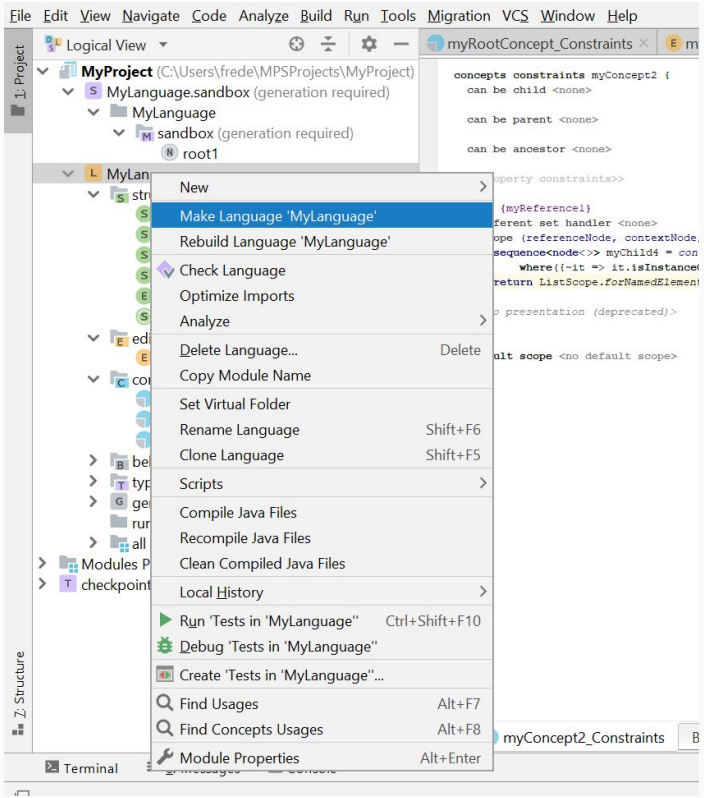
DTown.java | map_Neighbourhood | HorizontalStreet | BikeRack | MemberMovement_Be

```
1  package Bixi.sandbox;
2
3  /*Generated by MPS */
4
5  import javax.swing.JFrame;
6  import java.util.ArrayList;
7  import java.awt.event.ActionListener;
8  import java.awt.event.ActionEvent;
9  import java.awt.Dimension;
10 import javax.swing.JPanel;
11 import java.awt.Graphics;
12 import java.awt.Color;
13 import javax.swing.JButton;
14
15 public class DTown extends JFrame {
16   public ArrayList<DTown.Street> streets = new ArrayList<DTown.Street>();
17   public ArrayList<DTown.Movement> movements = new ArrayList<DTown.Movement>();
18   public ArrayList<DTown.BikeRack> racks = new ArrayList<DTown.BikeRack>();
19   public ArrayList<DTown.Member> members = new ArrayList<DTown.Member>();
20   public int day = 8;
21   public int time = 100;
22   public boolean initialized = false;
23   public boolean iterate = false;
24   public static void main(String[] args) {
25     DTown n = new DTown();
26     n.initialize();
27   }
28
29   public void initialize() {
30     streets.add(new DTown.Street("Sherbrooke", 15));
31     streets.add(new DTown.Street("Peel", 15));
32     members.add(new DTown.Member(12, "Alec"));
33     DTown.Street hStreet = null;
34     DTown.Street vStreet = null;
```

Generated Text File

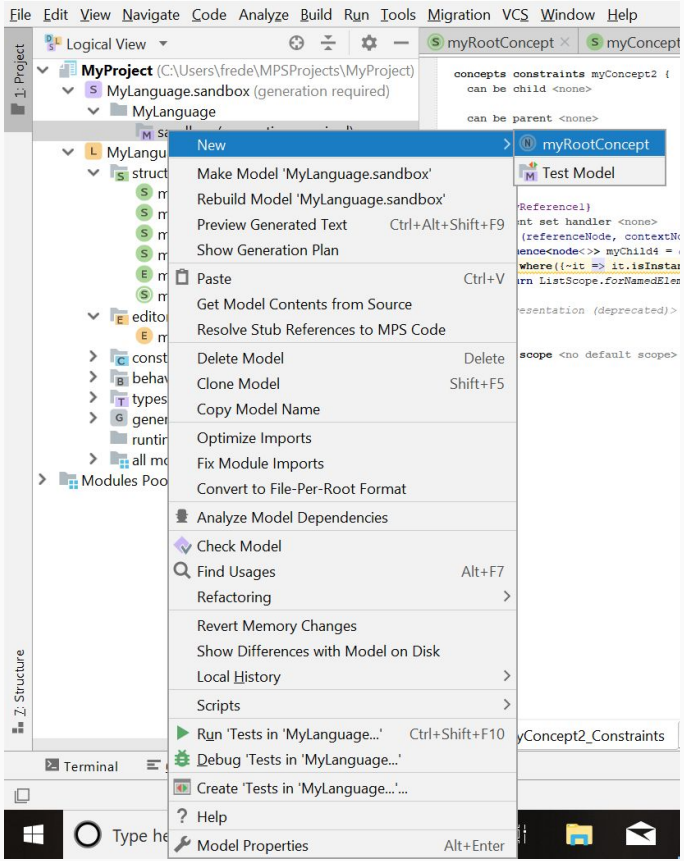
Test your Language: Create a Model

Make the Language Module



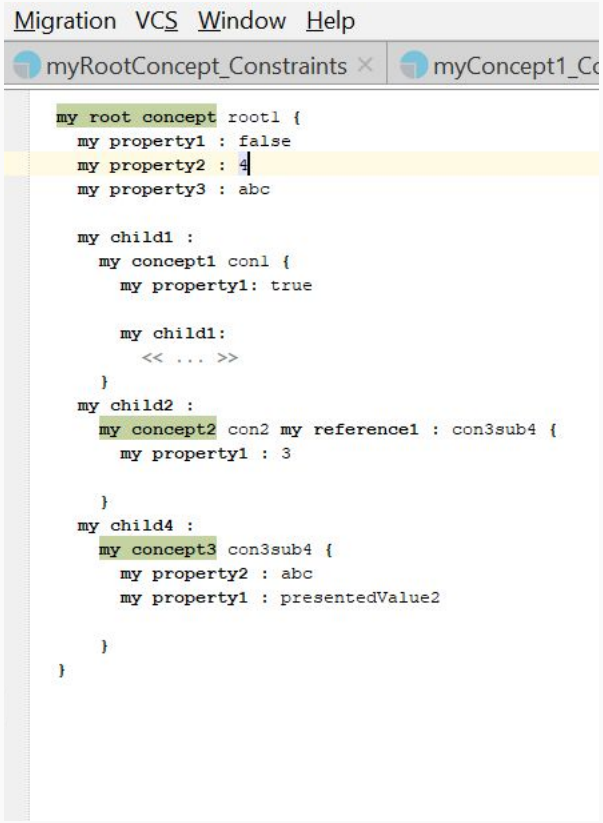
- First make the language, so that it can be used to implement the model.

Create a Solution Model



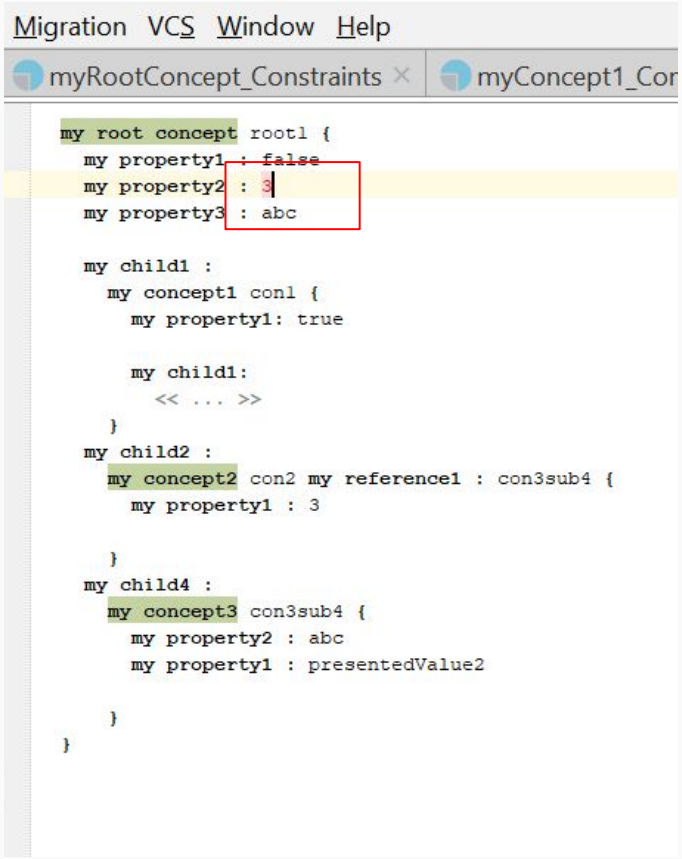
- Instantiate *myRootConcept* by creating a node of the root concept.

Instantiate Concept Nodes



- Use the projectional editor that is specified in the *Editor* module to create nodes of the different concepts that the language contains.
- The editor allows the user to add properties, children and references to the nodes.
- The editor asserts that the constraints and the typesystem rules are respected.

Test your Constraints



- This image shows a case of a constraint that is not respected.
- The editor refuses to accept this value.

Conclusion

Conclusion to this Brief MPS Tutorial

- MPS's organized file structure makes it optimal for creating potentially large languages, while keeping different aspects of the language separate from each other, thus making quick edits easy;
- Its auto-generation abilities allows for quick code generation. Once a user is familiar with the ins and outs of the software, coding can become a remarkably quick activity;
- The projectional nature of the editor enforces proper syntax. If any errors are present, the user will be notified of them very quickly.

JetBrains MPS. (2018, November 11). Retrieved from https://en.wikipedia.org/wiki/JetBrains_MPS