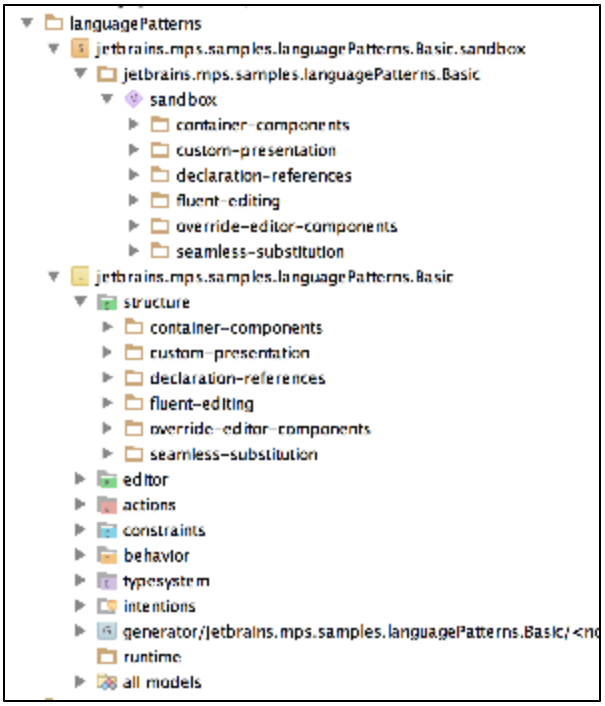
Common language patterns

This chapter covers common language design patterns that the beginners learning MPS would frequently come across. We've identified these over time as the most frequent questions that MPS users ask in the on-line forum and so we decided to summarise the answers in a single place for easy reference.

Sample project with language patterns

MPS comes bundled with sample projects. Many of the patterns discussed here have been implemented in the languagePattern s sample project. You can open it up in MPS (click on Open sample project in the MPS welcome screen) and check the actual implementation of the patterns of this chapter that refer to the sample project.



The patterns are organised into virtual packages both in the language definition and in the sandbox solution, to help you identify the elements that form the individual patterns.

Initial tips

Naming concepts

Many concepts need to provide a string property to hold the name of the nodes. Since a name of a concept has certain special qualities and needs to be recognised and handled by MPS in a special way (e.g. to label nodes in the code completion menu as

well as in the Project View, Node Explorer, Debugger Tool Window and others), it is advisable to inherit the property from the I NamedConcept concept interface. Many of your concepts and almost all root concepts should thus declare to implement IName dConcept.

Valid identifiers

If you are extending BaseLanguage and the name of the concept translates directly to a Java identifier, you should consider inheriting the name property from the IValidIdentifier concept interface, which extends INamedConcept. It will ensure through constraints that the name meets the criteria for a valid Java identifier.

Unique names

Frequently you want to ensure uniqueness of names for a certain group of nodes of the same concept. You may use either constraints or non-typesystem rules. Not-typesystem rules give you the option to customise the error message and may even offer quick-fixes to the user for automatic problem resolution. A sample uniqueness check for the name of an InputField in a Ca lculator of the Calculator tutorial would look like this:

```
checking rule check_InputField {
  applicable for concept = InputField as inputField
  overrides false

  do {
    if (inputField.parent : Calculator.inputField.any({~it => it.name :eq: inputField.name && it :ne:
inputField; })) {
      error "Duplicate name " + inputField.name -> inputField;
    }
  }

}
```

The same constraint specified with constraints would look this way:

```
concepts constraints InputField {
  can be child <none>

  can be parent <none>

  can be ancestor <none>


  property {name}
    get:<default>
    set:<default>
    is valid:(propertyValue, node)->boolean {
      node.parent : Calculator.inputField.where({~it => it.name :eq: propertyValue; }).size <= 1;
    }


  <<referent constraints>>

  default scope
    <no default scope>

}
```

Please note that constraints are probably less optimal here, for several reasons:

- you cannot specify a quick-fix
- you cannot customise the error message
- while the non-typesystem rule only underlines the duplicate names to indicate the error, constraints prevent invalid values to be inserted into the model and display the invalid values in red font, which makes the errors look more severe

Not seeing type-system errors

The type-system runs in the background and may sometimes be slow to deliver its results into the editor. Hit F5 in order to refresh all error and warning messages in the editor. Also make sure you have the Power Save Mode setting switched off. In Po wer Safe Mode the type-system only runs on explicit demand from the user (F5).

Regular expressions

By default properties can be of one of three types - integer, boolean and string. If you want a more customized datatype, you have to define one and restrict the allowed values by a regular expression. BaseLanguage defines float types this way, foe example. Open the FloatingPointConstant concept (Control/Cmd + N) and see the property value - it has a type of _FPNumber_ String. Navigate to the definition of _FPNumber_String (Control/Cmd + B) to see an example of a constrained data type:

```
constrained string datatype: _FPNumber_String

    matching regexp: -?[0-9]+\\.[0-9]*([Ee][\\+\\-]?[0-9]+)?[dD]?
```

Safe operators and operations

BaseLanguage and its core extensions offer several handy shortcuts for common operations that in Java require more efforts to get right - for example, checking for null before comparing, distinguishing empty and null lists, calling equals instead of "==", etc. Here's a list of the most commonly useful ones:

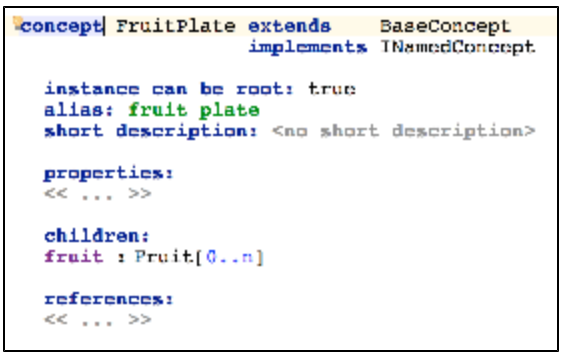
- :eq: - null-safe equals
- :ne: - null-safe not-equals
- isNull
- isNotNull
- size
- isEmpty
- isNotEmpty

Basic patterns

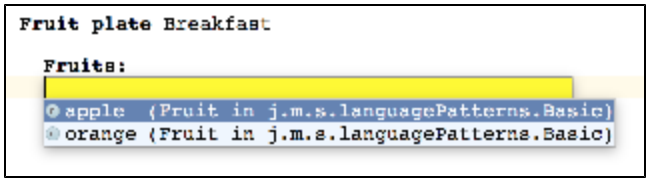
Container - component

Illustrated as the container-component virtual package in the languagePatterns sample.

A common scenario where a container (FruitPlate in our case) contains elements (Fruit) of multiple kinds (Apples, Oranges). An abstract concept (Fruit) is used as a placeholder in the container and concrete sub-concepts (Apples, Oranges) are then provided by the language user explicitly. The sub-concepts may have completely different visual appearance.



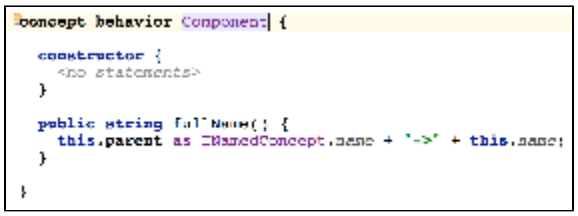
See also the seamless-substitution pattern to learn about how to allow the user to switch Apples to Oranges and back with little effort. The current implementation requires the user to first select the whole node (Apple or Orange) and only then code-completion offers the alternatives.

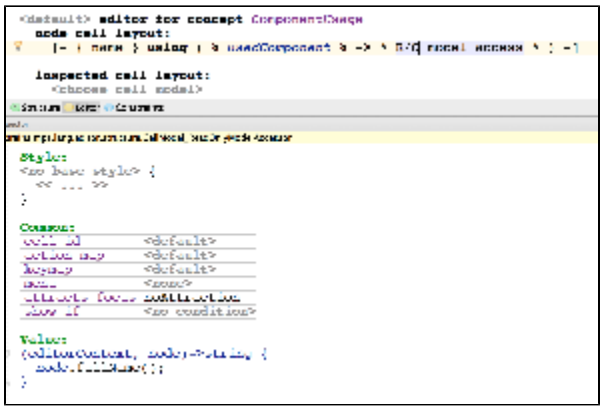


Customized presentation

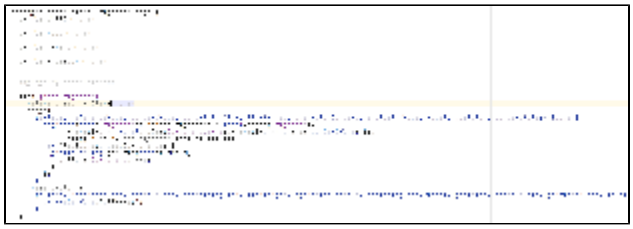
Illustrated as the custom-presentation virtual package in the languagePatterns sample.

Motivated by the discussion at http://forum.jetbrains.com/thread/Meta-Programming-System-2138?message=Meta-Programming-System-2138-3
Components held within Containers are referred to from ComponentUsages. The reference, however, should contain the name of the containing Container together with the name of the Component - both in the completion menu and in the editor.



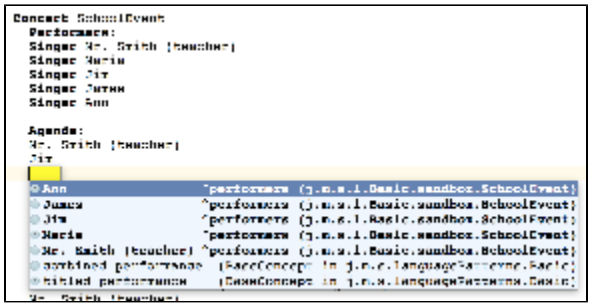


Additionally, scoping rules should guarantee that only one reference to any Component may exist and the completion menu is filtered so that already referred Components are not offered.



Declaration - references

Illustrated as the declaration-references virtual package in the languagePatterns sample.



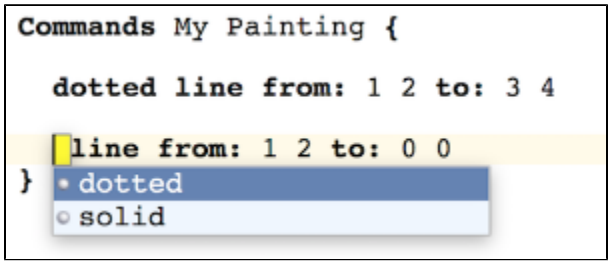
A typical pattern on declarations and references to them - Singers at an event can be organised into agendas with their Perfor mances. Different types of Performances are available.

Scoping rules ensure that:

- only singers listed in the current event can be added to its agenda
- each singer can only be listed once in a combined performance
- a handy intention (IntroduceSinger) is available to create a singer from a string typed in the performance (frequently known as "create from usage" or "introduce variable" refactoring).

Fluent editing

Illustrated as the fluent-editing virtual package in the languagePatterns sample.



An example of creating a text-like editing experience and implements many of the recommendations mentioned in the Editor cookbook (https://confluence.jetbrains.com/display/MPSD32/Editor+cookbook). It implements a simple language for specifying drawing commands (line, rectangle):

- an empty line is inserted upon hitting Enter, empty lines can be placed anywhere thanks to a default node factory specified for the child collection
- empty line does not appear as an option in the completion menu thanks to implementing the IDontSubstituteByDefault concept interface
- typing on an empty line will insert the desired item (line or rectangle)
- when the body of a block of code is empty, the cursor is positioned on the first line (next to the header), editing can be started on this first line
- NodeFactories take care of propagating values to new nodes if a nodes is replaced with another one
- an optional "line style" child can be specified on the left of the draw commands thanks to left-side transformations
- wrappers allow the desired line style to be typed on an empty line and an intermediate "IncompleteCommand" will be created automatically our of it
- IncompleteCommand expects either line or rectangle to be typed and will be substituted into the desired draw command instantly
- draw commands define aliases and editors so that they start with the same word
- an editor in the abstract super-concept of the draw commands is reused by concrete sub-concepts
- typing on the left hand-side of a command will offer the allowed prefixes
- deleting a prefix (solid, dotted) will correctly delete only the prefix

Override editor component

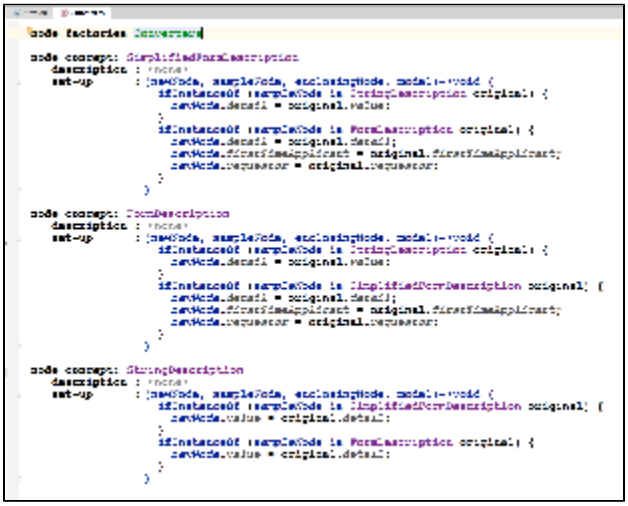
Illustrated as the override-editor-component virtual package in the languagePatterns sample.

An example of using editor components that get overriden in a sub-concept (Truck). An editor in Car uses an editor component CarProperties also defined in Car. A sub-concept (Truck) overrides the CarProperties editor component with the TruckProperties editor component to contain its own properties. The editor in Car will use the Truck's variant of the editor component for Trucks and the Car variant for Cars.

Seamless substitution

Illustrated as the seamless-substitution virtual package in the languagePatterns sample.

An example that seamlessly switches (substitutes) between different related subconcepts. A Request contains a "description", which may be either a string, a simple form or a complex form. A completion-menu lets the user pick the requested description type. If the user simply types text, the "string"-based description is picked automatically in the "PickTheRightDescriptionType" substitute action. The first cell of editors for each of the description concepts is sensitive to substitution (set through the "menu" property of the cell) and so offers the option to switch between description types with completion-menu. The "Converte rs" node factories contain code that preserves parts of the description information and propagates it into the newly instantiated description concept.



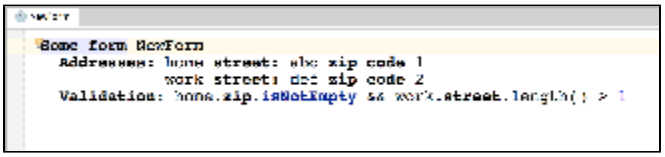
Hierarchical scopes

If you are defining references of some sort, you typically want to restrict the scope for the references so they can only point to nodes in certain locations or "distance" from the reference. Additionally, definitions may hide other references with the same names that are further away from the reference - think of Java's local variables hiding parameter declarations or fields, for example. In MPS when defining references you provide the the scope of the reference in the Constraints aspect of the language definition. For hierarchical scopes you use the inherited scope type in the reference and then let your code containers (aka classes, methods, block statements, etc.) implement ScopeProvider concept interface. In its getScope() method you implement the logic of retrieving candidate targets of certain kind for references that want to point to them. Check out the Scopes documentation for detailed how-to.

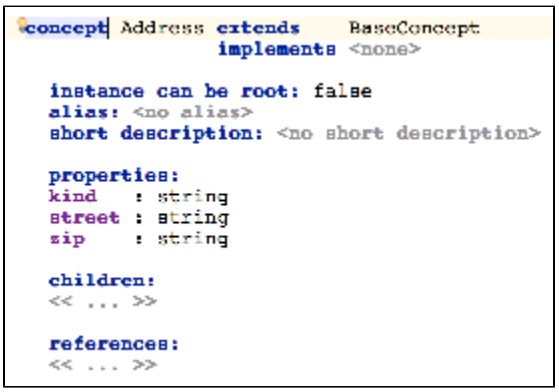
Extending the DotExpression for your own references

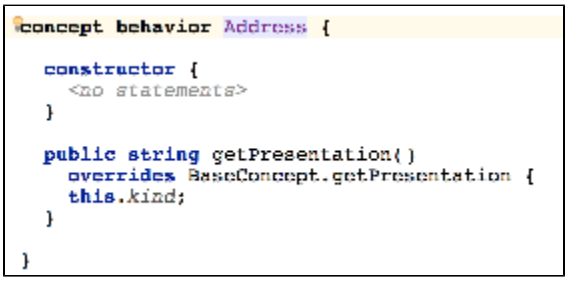
Illustrated as the dotexpression virtual package in the languagePatterns sample (bundled in MPS 3.3 and later).

De-referencing elements by "dot" notation is a very common practise in many languages. BaseLanguage offers the DotExpressi on concept to mimics Java's "dot" operator and you can leverage it in your languages, if they're extending BaseLanguage's expressions. Let's assume a simple form with several addresses that need to be validated by validation expressions. The expressions need to refer to the street and zip code of each address in order to include their values in the validation expression:



Each address is represented by a node of Address concept. Notice that it also customises the presentation so that it displays the "kind" property in the code-completion menu:





The AddressReference concept allows referring to addresses from within BaseLanguage expressions in the validation section.

All BaseLanguage expressions are converted to DotExpression as soon as you append "dot" to them, with the left operand being the original expression. What remains is the right side of the DotExpression, called operation. Operations implement the I Operation concept interface. We will need two operations, one for street and one for the zip code, thus we'll start with a common abstract super concept for these:

The concrete operations then only specify a meaningful alias and a type-system rule so as to participate properly in the whole surrounding expression: