TextGen

TextGen language aspect

Introduction

The TextGen language aspect defines a model to text transformation. It comes in handy each time you need to convert your models into the text form directly. The language contains constructs to print out text, transform nodes into text values and give the output some reasonable layout.

Operations

The append command performs the transformation and adds resulting text to the output. You can use found error command to report problems in the model. The with indent command demarcates blocks with increased indentation. Alternatively, the incre ase depth and decrease depth commands manipulate the current indentation depth without being limited to a block structure. The indent buffer command applies the current indentation (as specified by with ident or increase/decrease depth) for the current line.

Operation	Arguments
append	any number of: • {string value}, to insert use the " char, or pick constant from the completion menu • \n • $list{node.list} - list without separator • $list{node.list with ,} - with separator (intentions to add/remove a separator are available) • $ref{node.reference}, e.g. $ref{node.reference<target>} - deprecated and will be removed • ${node.child} • ${attributed node}$ - available in attribute nodes, delegates to the attributed node
found error	error text
decrease depth	decrease indentation level from now onwards
increase depth	increase indentation level from now on
indent buffer	apply indentation to the current line
with indent { <code> }	increase indentation level for the <code>

The parameters to the append command may have the with indent flag in the Inspector tool window set to true to get prepended with the current indentation buffer.

Indentation

Proper indentation is easy to get right once you understand the underlying principle. TextGen flushes the AST into text. The Te xtGen commands simply manipulate sequentially the output buffer and output some text to it, one node at a time. A variable holding the current depth of indentation (indentation buffer) is preserved for each root concept. Indentation buffer starts at zero and is changed by increase/decrease depth and with indent commands.

The "indentation", however, must be inserted into the output stream explicitly by the append commands. Simply marking a block with with indent will not automatically indent the text generated by the wrapped TextGen code. The with indent block only increases the value of the indentation buffer, but the individual appends may or may not wish to be prepended with the in dentation buffer of the current size.

There are two ways to explicitly insert indentation buffer into the output stream:

- indent buffer command
- with indent flag in the inspector for the parameters of the append command

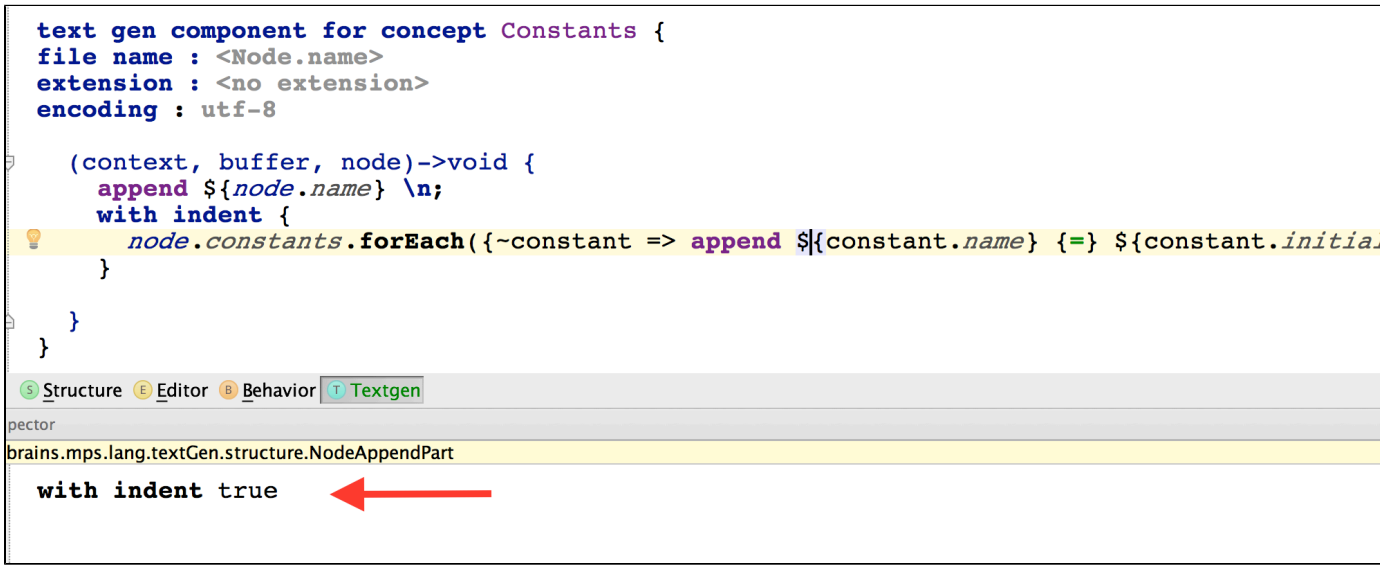
For example, to properly indent Constants in a list of constants, we call indent buffer at the beginning of each emitted line. This ensures that the indentation is inserted only at the beginning of each line.

```
text gen component for concept Constants {
file name : Constants
extension : <no extension>

encoding : utf-8

 (context, buffer, node)->void {
  append ${node.name} \n ;
  with indent {
   node.constants.forEach({~constant =>
    indent buffer ;
    append ${constant.name} {=} ${constant.initializer} \n ;
   });
  }
 }
}
```

Alternatively, we could specify the with indent flag in the inspector for the first parameter to the append command. This will also insert the indentation only at the beginning of each line.



Root concepts

TextGen provides two types of root concepts:

- text gen component, represented by the ConceptTextGenDeclaration concept, which encodes a transformation of a concept into text. For rootable concepts the target file can also be specified.
- base text gen component, represented by the LanguageTextGenDeclaration concept, which allows definition of reusable textgen operations and utility methods. These can be called from other text gen components of the same language as well as extending languages

TextGen in extended concepts

MPS does not create files for root concept automatically. Even sub-concepts of a concept that has TextGen defined will have no file created automatically. Only exact concept matches are considered. If an extending concept desires to re-use textgen component of an ancestor as is, it shall declare its own empty TextGen component, stating the essentials as the file name, encoding and extension, and leaving the body of the component empty.

Layout

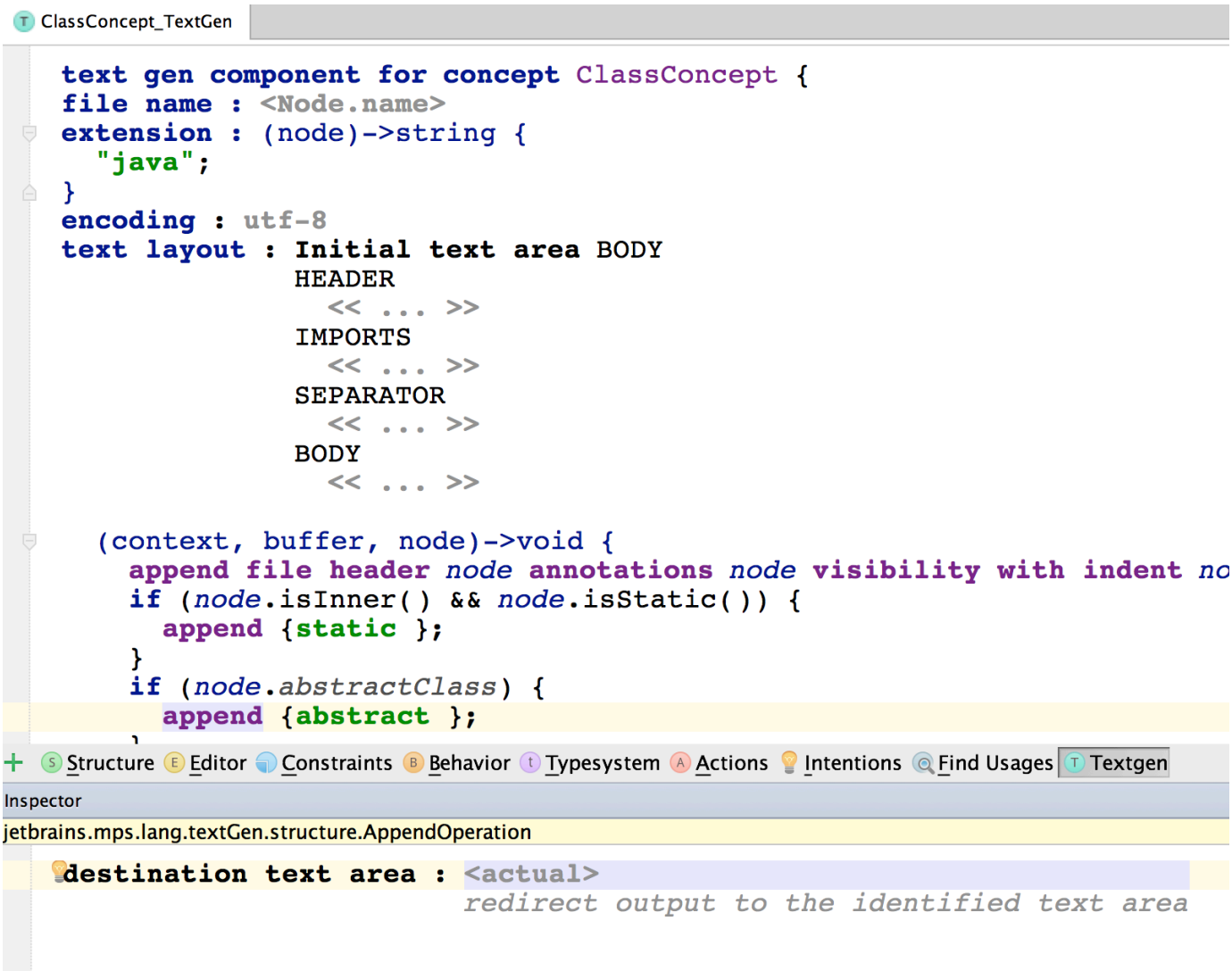
There's provisional mechanism to control layout of output files. The text layout section of ConceptTextGenDeclaration (available only in rootable concepts) allows the authors to define multiple logical sections (with a default one) and then optionally specify for each append, to which section to append the text.

Text generation is not always possible in a sequence that corresponds to lines in a physical file. E.g. for a Java source, one could distinguish 2 distinct areas, e.g. imports and class body, where imports is populated along with the body. A passionate language designer might want to break up the file further, e.g. up to file comment, package statement, imports, and class body that consists of fields and methods, and populate each one independently while traversing a ClassConcept. That's what we call a layout of an output file, and that's what we give control over now. MPS veterans might be aware of two buffers (TOP and BOTTOM) that used to be available in TextGen for years. These were predefined, hard-coded values. Now it's up to language designer to designate areas of an output file and their order.

Note, distinct areas come handy especially when generating text from attributes, as they change order of execution. With them, it's even more tricky to make sure flow of text gen corresponds to physical text lines, and designated areas make generation a lot more comfortable.

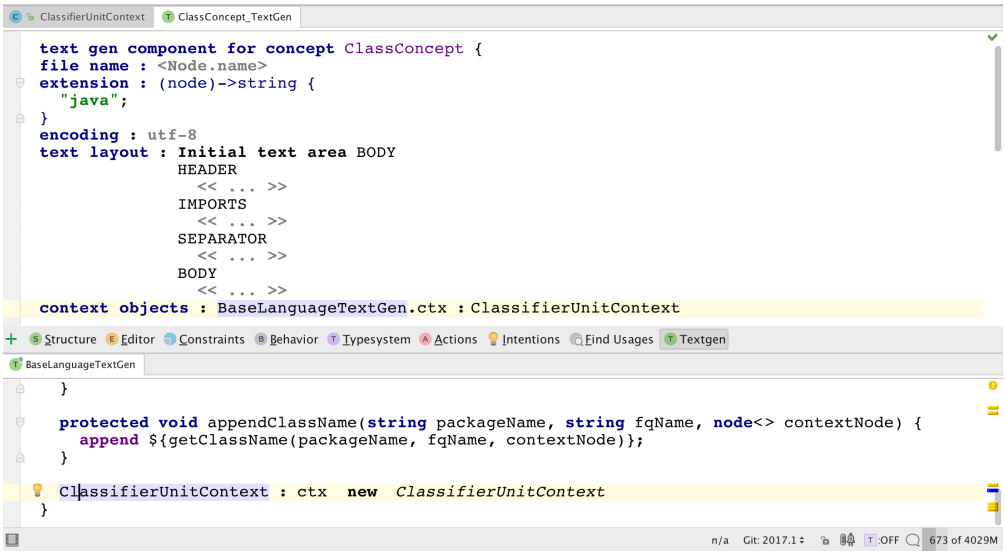
Layout of the file could be specified for a top text gen, the one that produces files.

The support for this mechanism is preliminary and is quite rudimentary now. We utilize it in our BaseLanguage implementation, so this notice is to explain you what's going on rather than encourage you to put this into production.



Context objects

It is vital for certain model-to-text conversion scenarios to preserve some context information during TextGen. In BaseLanguage, for example, TextGen has to track model imports and qualified class names. The cumbersome and low-level approach of previous versions based on direct text buffer manipulation has been replaced with the possibility to define and use customized objects as part of the concept's textgen specification.



At the moment, regular java class (with a no-arg or a single-arg constructor that takes concept instance) are supported as context objects. You reference context objects from code as a regular variable.

Handling attributes in TextGen

When nodes are annotated with Attributes, the TexGen for these attributes is processed first. The ${attributed node} construct within the attribute's TextGen will then insert the TextGen of the attributes node itself.
If there are multiple attributes on a single node, they are processed in turn, starting with the last-assigned (top-most) attribute. Attributes without TextGen associated are ignored and skipped.

> The top-most attribute is technically the last in the containment (the way editor depicts the attributes is visually differ ent from the order one may notice in Node Explorer). I.e. A node with attributes A1 and A2 in the editor looks like A2(A1(N)), and TextGen processes A2 first, then A1, then N. When TextGen is asked to generate text for N, it looks up the last attribute in the containment that has textgen defined (if any), and delegates to it. Let's assume A2 has TextGen, and A1 not. Then, if the attribute's component has ${attributedNode}, TextGen would check the previous attributes for associated textgen. If there are none (and in our sample A1 has none), text generation for the actual node N is the last to receive control.

Examples

Here is an example of the text gen component for the ForeachStatement (jetbrains.mps.baseLanguage).

```
text gen component for concept ForeachStatement {
  (node, context, buffer)->void {
    if (node.loopLabel != null) {
      append \n ${node.loopLabel.name} {:} ;
    } else if (node.label != null) {
      append \n ${node.label} {:} ;
    }
    append \n ;
    indent buffer ;
    append {for (} ${node.variable} { : } ${node.iterable} {) {} ;
    with indent {
      append ${node.body} ;
    }
    append \n {}} ;
  }
}
```

This is an artificial example of the text gen:

```
text gen component for concept CodeBlockConcept {
  (node, context, buffer)->void {
      indent buffer ;
      append {codeBlock {} \n ;
      with indent {
        indent buffer ;
        append {// Begin of codeBlock} \n ;

        indent buffer ;
        append {int i = 0} \n ;

        indent buffer ;
        append {// End of codeBlock} \n ;
      }
      append {}} ;
  }
}
```

producing following code block containing a number of lines with indentation:

```
text gen component for concept CodeBlockConcept {
codeBlock {
  // Begin of codeBlock
  int i = 0
  // End of codeBlock
}
```

An example of TextGen for attribute that adds extra text to output of attributed node:

```
text gen component for concept MethodDocComment {
  (context, buffer, node)->void {
    append doc comment start node;
    append $list{node.tags};
    append doc comment end node;

    append ${ attributed node }$;
  }
}
```

Previous Next