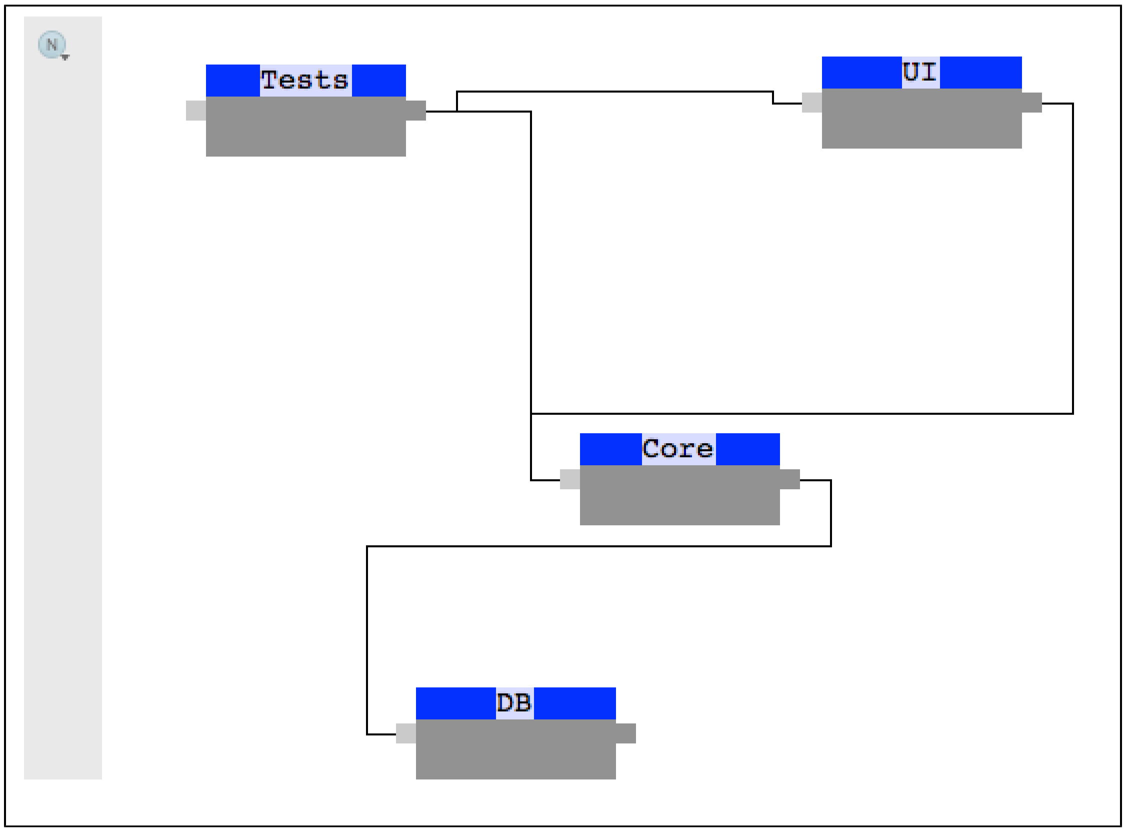
Diagramming Editor

The diagramming support in MPS allows the language designers to provide graphical editors to their concepts. The diagrams typically consist of blocks, represented by boxes, and connectors, represented by lines connecting the boxes. Both blocks and connectors are visualization of nodes from the underlying model.



Ports (optional) are predefined places on the shapes of the blocks, to which connectors may be attached to. MPS allows for two types of ports - input and output ones.

Optionally, a palette of available blocks may be displayed on the side of the diagram, so the user could quickly pick the type of the box they need to add to the diagram.

Adding elements

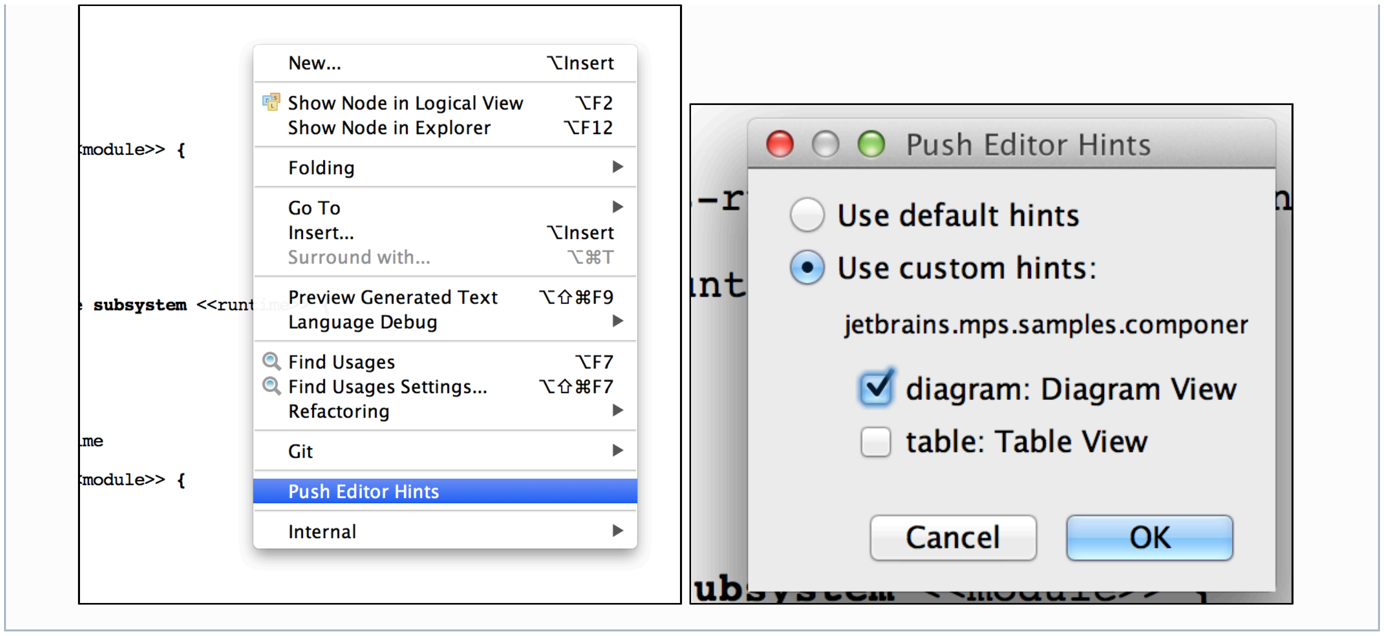
Blocks get added by double-clicking in a free area of the editor. The type of the block is chosen either by activating the particular block type in the palette or by choosing from a pop-up completion menu that shows up after clicking in the free area.

Connectors get created by dragging from an output port of a block to an input port of another or the same block.

Samples

MPS comes with bundled samples of diagramming editors. You can try the componentDependencies or the mindMaps sample projects for initial familiarization with how diagrams can be created.

This document uses the componentDependencies sample for most of the code examples. The sample defines a simple language for expressing dependencies among components in a system (a component set). Use the "Push Editor Hints" option in the pop-up menu to activate the diagramming editor.



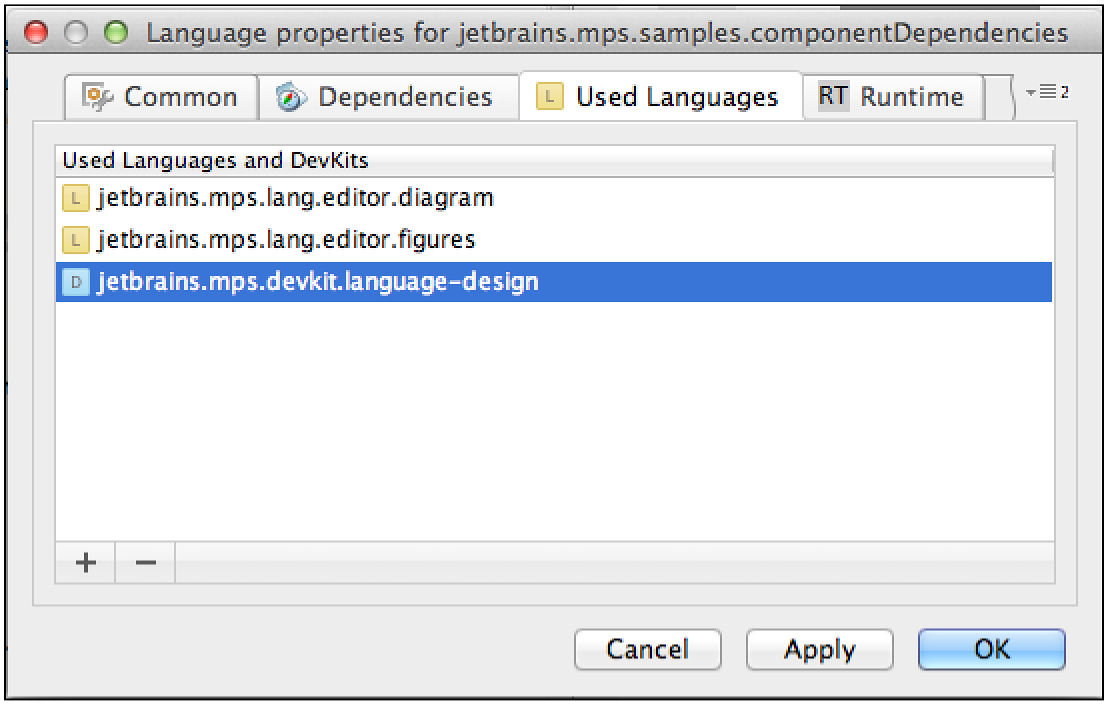
Push Editor Hints: ( ) Use default hints [✓ Use custom hints:]

jetbrains.mps.samples.componer

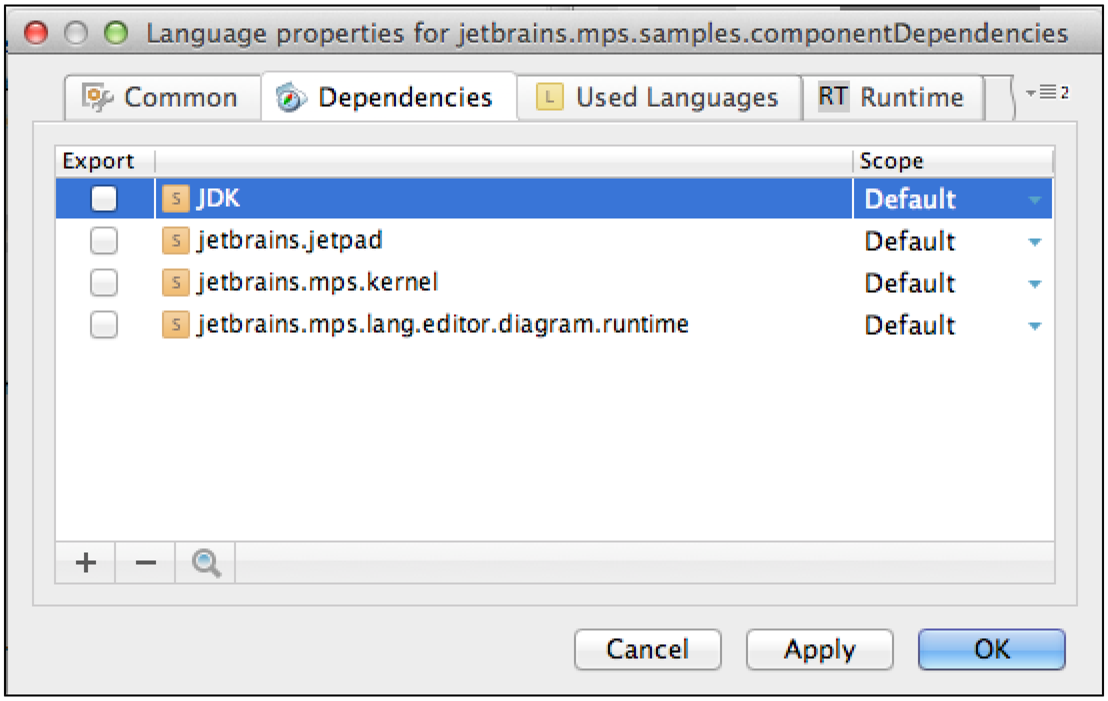
- [✓] diagram: Diagram View
- [ ] table: Table View

Dependencies

In order to be able to define diagramming editors in your language, the language has to have the required dependencies and used languages properly set:


Language properties for jetbrains.mps.samples.componentDependencies
Common
Dependencies
Used Languages
Runtime
Cancel
Apply
OK

Used Languages and DevKits
- jetbrains.mps.lang.editor.diagram
- jetbrains.mps.lang.editor.figures
- jetbrains.mps.devkit.language-design



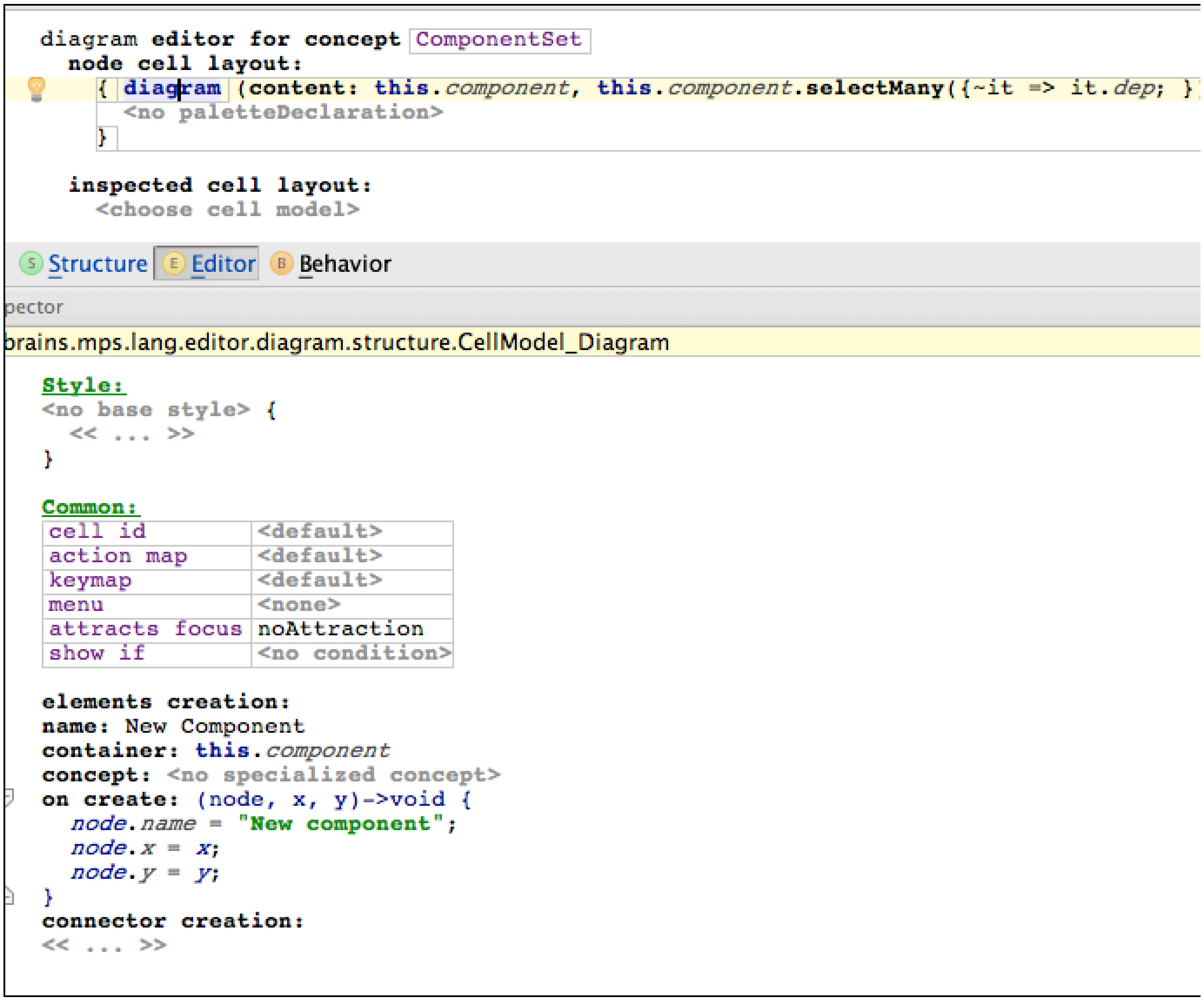
Language properties for jetbrains.mps.samples.componentDependencies

Export		Scope
☐	JDK	Default
☐	jetbrains.jetpad	Default
☐	jetbrains.mps.kernel	Default
☐	jetbrains.mps.lang.editor.diagram.runtime	Default

- jetbrains.mps.lang.editor.diagram - the language for defining diagrams
- jetbrains.mps.lang.editor.figures (optional) - a language for defining custom visual elements (blocks and connectors)
- jetbrains.jetpad and jetbrains.mps.lang.editor.diagram.runtime - runtime libraries that handle the diagram rendering and behavior

Diagram definition

Let's start from the concept that should be the root of the diagram. The diagramming editor for that node will contain the diagr am editor cell:


```
diagram editor for concept ComponentSet
  node cell layout:
    { diagram (content: this.component, this.component.selectMany({~it => it.dep; }
      <no paletteDeclaration>
    }

  inspected cell layout:
    <choose cell model>
```

```
Style:
<no base style> {
  << ... >>
}
```

Common:

cell id	<default>
action map	<default>
keymap	<default>
menu	<none>
attracts focus	noAttraction
show if	<no condition>

```
elements creation:
name: New Component
container: this.component
concept: <no specialized concept>
on create: (node, x, y)->void {
  node.name = "New component";
  node.x = x;
  node.y = y;
}
connector creation:
<< ... >>
```

Note that the diagram editor cell does not have to be the root of the editor definition. Just like any other editor cell it can be composed with other editor cells into a larger editor definition.

The diagram cell needs its content parameter to hold all the nodes that should become part of the diagram. In our case we pass in all the components (will be rendered as blocks) and their dependencies (will be rendered as connectors). The way these nodes are rendered is defined by their respective editor definitions, as explained later.

Down in the Inspector element creation handlers can be defined. These get invoked whenever a new visual block is to be created in the diagram. Each handler has several properties to set:

- name - an arbitrary name to represent the option of creating a new element in the completion menu and in the palette
- container - a collection of nodes that the newly created node should be added to
- concept - the concept of the node that gets created through the handler, defaults to the type of the nodes in the contai ner, but allows sub-types to be specified instead
- on create - a handler that can manipulate the node before it gets added to the model and rendered in the diagram. Typically the name is set to some meaningful value and the position of the block on the screen is saved into the model.

There can be multiple element creation handlers defined.

Similarly, connector creation handlers can be defined for the diagram cell to handle connector creation. On top of the attributes already described for element creation handlers, connector creation handlers have these specific attributes:

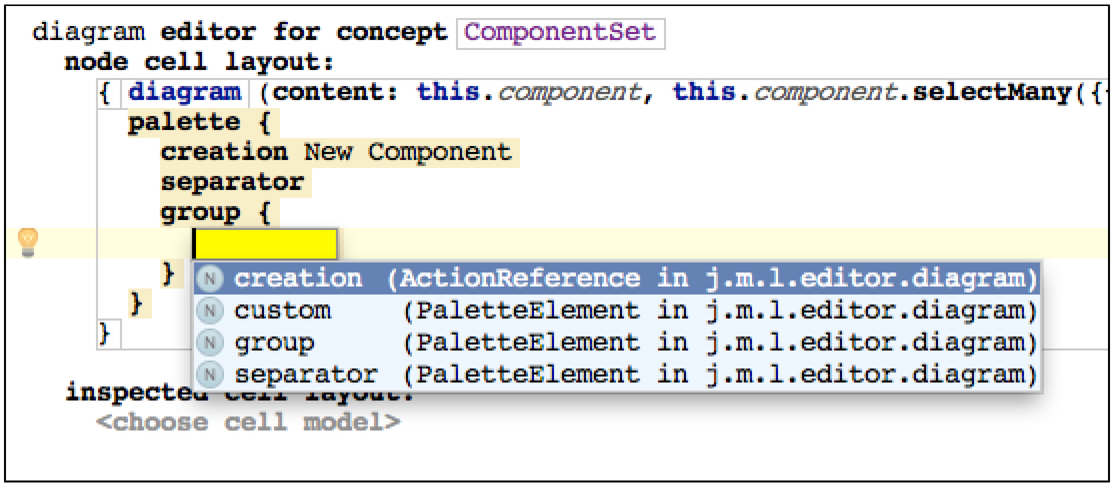
- can create - a concept function returning a boolean value and indicating whether a connector with the specified properties can be legally constructed and added to the diagram.
- on create - a concept function that handles creation of a now connector.
- the from and to parameters to these functions specify the source and target nodes (represented by a Block or a Port)

for the new connection.
- the fromId and toId parameters to these functions specify the ids of the source and target nodes (represented by a Bloc k or a Port) for the new connection.

Elements get created when the user double-clicks in the editor. If multiple element types are available, a completion pop-up menu shows up.

Connectors get created when the user drags from the source block or its output port to a target block or its input port.

Palette



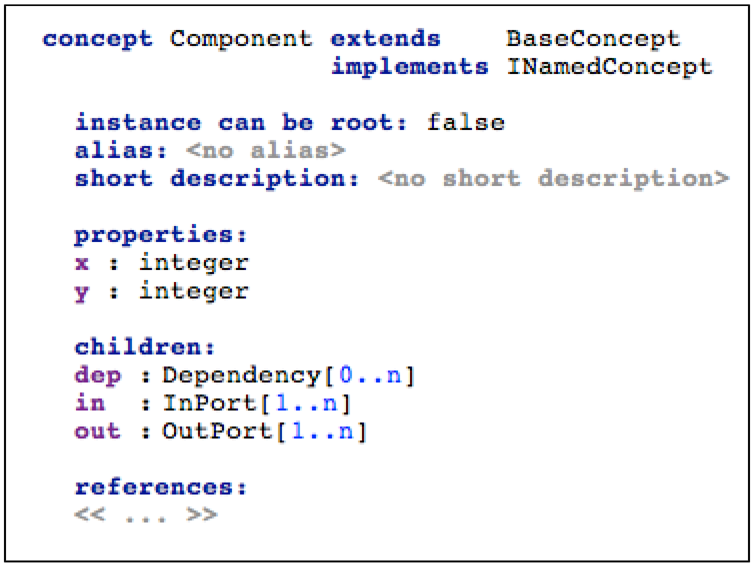
```
diagram editor for concept ComponentSet
  node cell layout:
    { diagram (content: this.component, this.component.selectMany({
      palette {
        creation New Component
        separator
        group {
          
        }
      }
    }
  inspected cell layout:
    <choose cell model>
```

```
creation  (ActionReference in j.m.l.editor.diagram)
custom     (PaletteElement in j.m.l.editor.diagram)
group      (PaletteElement in j.m.l.editor.diagram)
separator  (PaletteElement in j.m.l.editor.diagram)
```

The optional palette will allow developers to pick a type of blocks and links to create whenever double-clicking or dragging in the diagram. The palette is defined for diagram editor cells and apart from specifying the creation components allows for visual grouping and separating of the palette items..

Blocks

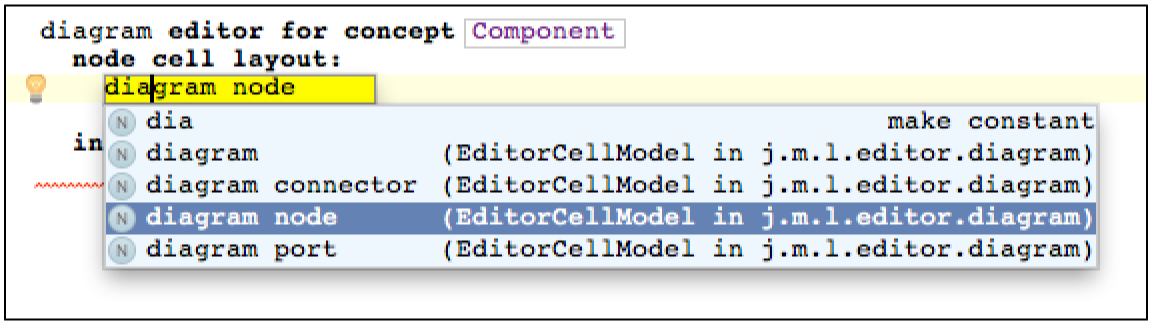
The concepts for the nodes that want to participate in diagramming as blocks need to provide properties that will preserve useful diagramming qualities, such as x/y coordinates, size, color, title, etc.



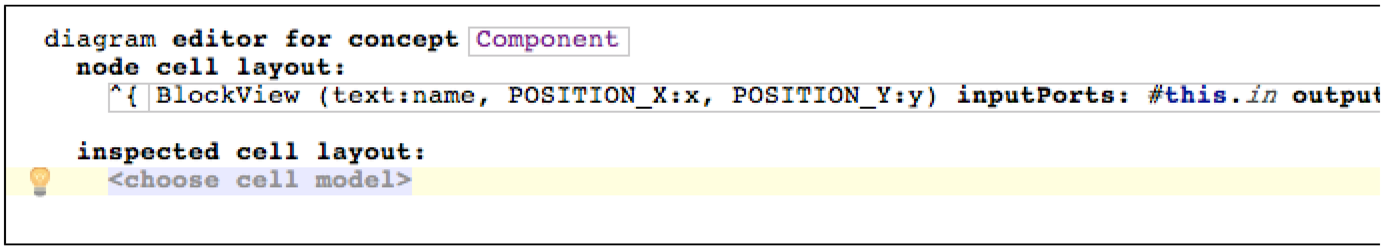
```
concept Component extends    BaseConcept
                  implements INamedConcept

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  x : integer
  y : integer

  children:
  dep  : Dependency[0..n]
  in   : InPort[1..n]
  out  : OutPort[1..n]

  references:
  << ... >>
```

Additionally, the nodes should provide input and output ports, which connectors can visually connect to.

The editor will then use the diagram node cell:


```
diagram editor for concept Component
  node cell layout:
    diagram node
    dia                                    make constant
    diagram           (EditorCellModel in j.m.l.editor.diagram)
    diagram connector (EditorCellModel in j.m.l.editor.diagram)
    diagram node      (EditorCellModel in j.m.l.editor.diagram)
    diagram port      (EditorCellModel in j.m.l.editor.diagram)
```

The diagram node cell requires a figure to be specified. This is a reference to a figure class that defines the visual layout of the block using the jetpad framework. MPS comes with a set of pre-defined graphical shapes in the jetbrains.mps.lang.editor.figure s.library solution, which you can import and use. Each figure may expose several property fields that hold visual characteristics of the figure. All the figure parameters should be specified in the editor definition, most likely by mapping them to the node's properties defined in the concept:


```
diagram editor for concept Component
  node cell layout:
    ^{ BlockView (text:name, POSITION_X:x, POSITION_Y:y) inputPorts: #this.in output

  inspected cell layout:
    <choose cell model>
```

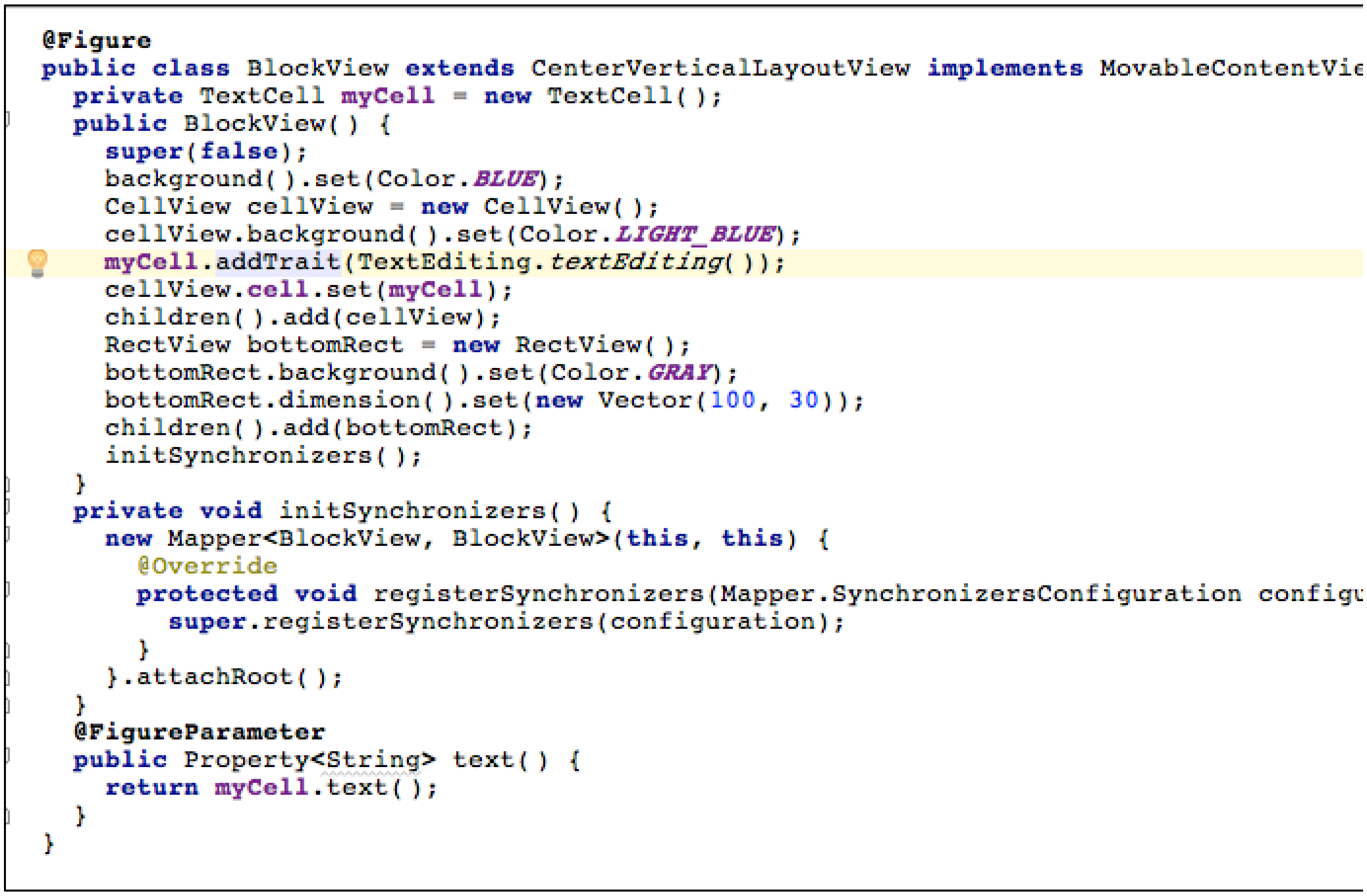
The values for parameters may ether be references to the node's properties, or BaseLanguage expressions prepended with the # character. You can use this to refer to the edited node from within the expression.

If the node defines input and output ports, they should also be specified as parameters here so that they get displayed in the diagram. Again, to specify ports you can either refer to the node's properties or use a BaseLanguage expression prepended with the # character.

As all editor cells, diagramming cells can have Action Maps associated with them. This way you can enable the Delete key to delete a block or a connector.

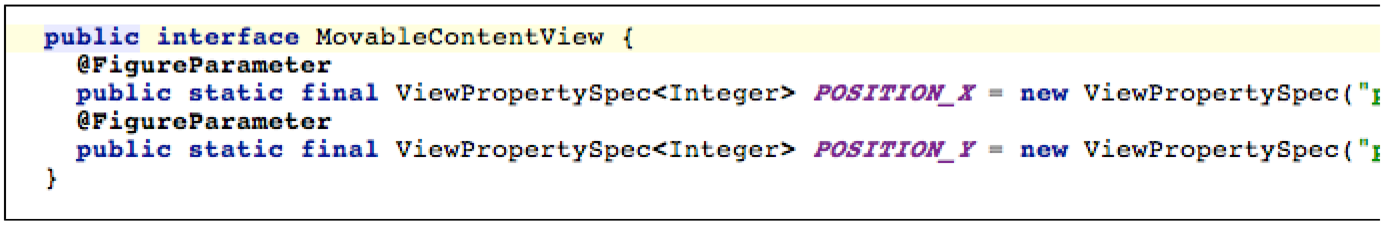
Custom figures

Alternatively you can define your own figures. These are BaseLanguage classes implementing the jetbrains.jetpad.projectional. view.View interface (or its descendants) and annotated with the @Figure annotation. Use the @FigureParameter annotation to demarcate property fields, such as width, height etc.



```
@Figure
public class BlockView extends CenterVerticalLayoutView implements MovableContentVie
  private TextCell myCell = new TextCell();
  public BlockView() {
    super(false);
    background().set(Color.BLUE);
    CellView cellView = new CellView();
    cellView.background().set(Color.LIGHT_BLUE);
    myCell.addTrait(TextEditing.textEditing());
    cellView.cell.set(myCell);
    children().add(cellView);
    RectView bottomRect = new RectView();
    bottomRect.background().set(Color.GRAY);
    bottomRect.dimension().set(new Vector(100, 30));
    children().add(bottomRect);
    initSynchronizers();
  }
  private void initSynchronizers() {
    new Mapper<BlockView, BlockView>(this, this) {
      @Override
      protected void registerSynchronizers(Mapper.SynchronizersConfiguration configu
        super.registerSynchronizers(configuration);
      }
    }.attachRoot();
  }
  @FigureParameter
  public Property<String> text() {
    return myCell.text();
  }
}
```

The MovableContentView interface provides additional parameters to the figure class:

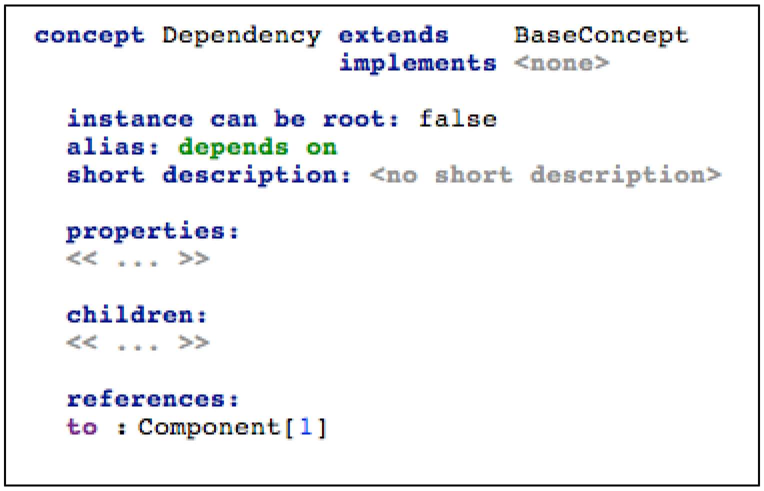


```
public interface MovableContentView {
  @FigureParameter
  public static final ViewPropertySpec<Integer> POSITION_X = new ViewPropertySpec("
  @FigureParameter
  public static final ViewPropertySpec<Integer> POSITION_Y = new ViewPropertySpec("
}
```

By studying jetbrains.mps.lang.editor.figures.library you may get a better understanding of the jetpad library and its inner workings.

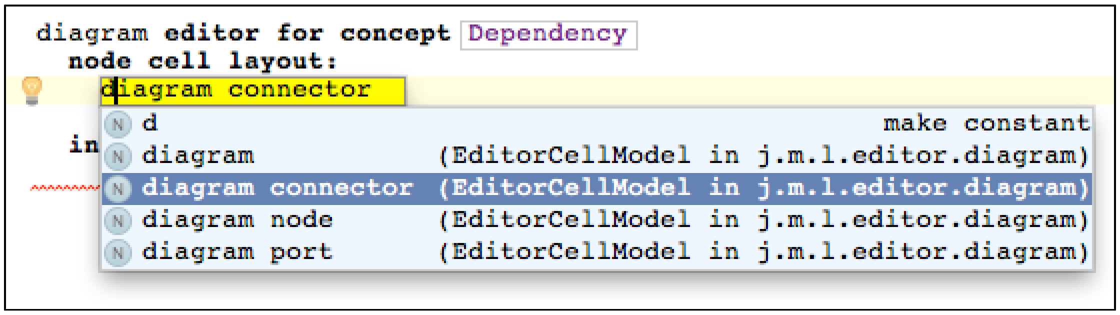
Connectors

The nodes that will be represented by connectors do not need to preserve any diagramming properties. As of version 3.1 connectors cannot be visually customized and will be always rendered as a solid black line. This will most likely change in one of the following versions of MPS.



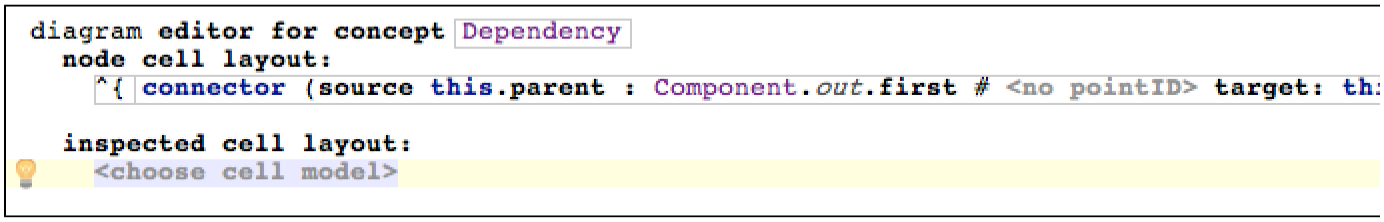
```
concept Dependency extends     BaseConcept
                   implements  <none>

  instance can be root: false
  alias: depends on
  short description: <no short description>

  properties:
  << ... >>

  children:
  << ... >>

  references:
  to : Component[1]
```

The editor for the node needs to contain a diagram connector cell:



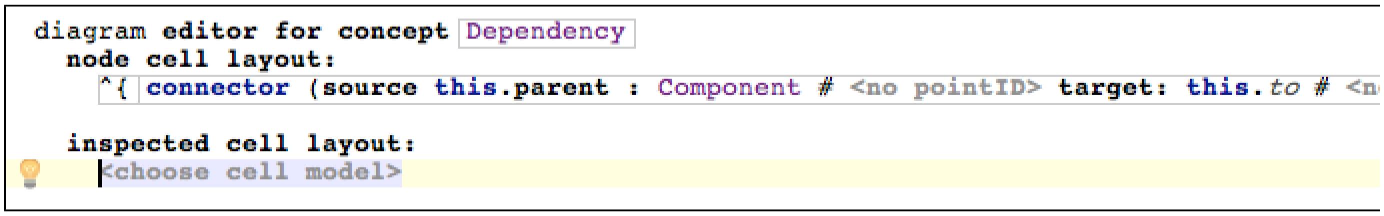
```
diagram editor for concept Dependency
  node cell layout:
    diagram connector
    d                                        make constant
 in diagram              (EditorCellModel in j.m.l.editor.diagram)
    diagram connector    (EditorCellModel in j.m.l.editor.diagram)
    diagram node         (EditorCellModel in j.m.l.editor.diagram)
    diagram port         (EditorCellModel in j.m.l.editor.diagram)
```

The cell requires a source and a target for the connector. These can either be ports:



```
diagram editor for concept Dependency
  node cell layout:
    ^{ connector (source this.parent : Component.out.first # <no pointID> target: th

  inspected cell layout:
    <choose cell model>
```

or nodes themselves:



```
diagram editor for concept Dependency
  node cell layout:
    ^{ connector (source this.parent : Component # <no pointID> target: this.to # <n

  inspected cell layout:
    <choose cell model>
```

The values may again be direct references to node's properties or BaseLanguage expressions prepended with the # character.

Rendering ports

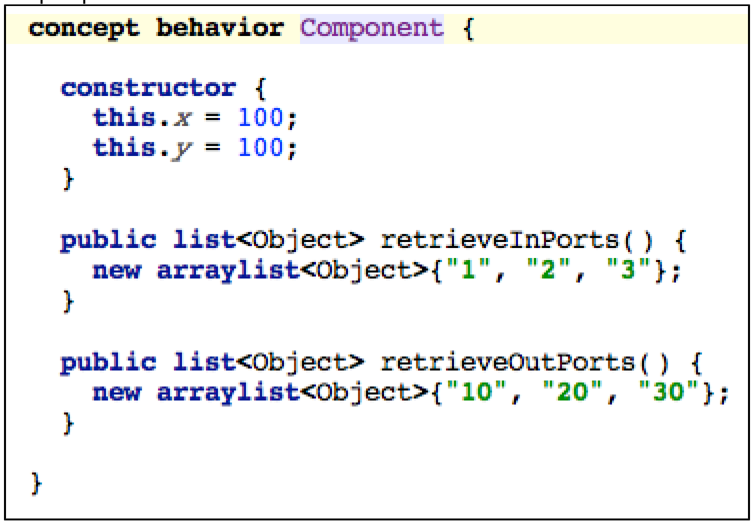
Input and output ports should use the input port and output port editor cells, respectively. The rendering of ports cannot be customized in MPS 3.1, but will be most likely enabled in later versions.

Use the T key to rotate the ports of a selected block by 90 degrees. This way you can easily switch between the left-to-right and top-to-bottom port positions.

Using implicit ports

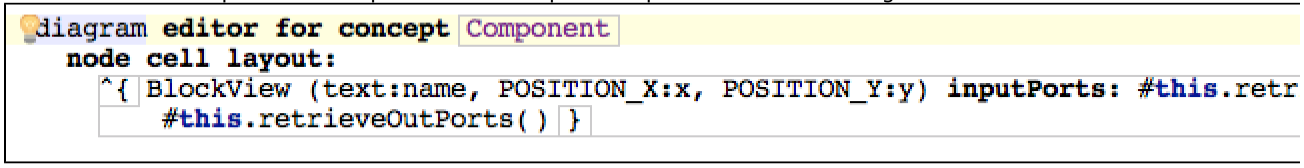
In some situations you will not be able to represent ports directly in the model. You'll only want to use blocks and connectors, but ports will have to be somehow derived from the model. This case can easily be supported:

1. Decide on the representation of ports. Each port will be represented by a unique identifier, such as number or a string
2. Have the concept for the blocks define behavior methods that return collections of identifiers - separately for input and output ports



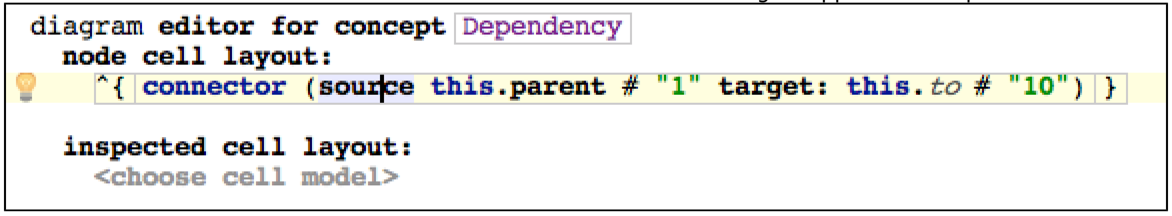
```
concept behavior Component {

  constructor {
    this.x = 100;
    this.y = 100;
  }

  public list<Object> retrieveInPorts() {
    new arraylist<Object>{"1", "2", "3"};
  }

  public list<Object> retrieveOutPorts() {
    new arraylist<Object>{"10", "20", "30"};
  }

}
```

3. Use the methods to provide the inputPorts and outputPorts parameters to the DiagramNode editor cell



```
diagram editor for concept Component
  node cell layout:
    ^{ BlockView (text:name, POSITION_X:x, POSITION_Y:y) inputPorts: #this.retr
       #this.retrieveOutPorts() }
```

4. In the connector editor cell refer to the block's node as source and target. Append the requested id after the # symbol



```
diagram editor for concept Dependency
  node cell layout:
    ^{ connector (source this.parent # "1" target: this.to # "10") }

  inspected cell layout:
    <choose cell model>
```

Previous Next