1. Component description: *.wbp-component.xml
   Table 1.1 – tags in *.wbp-component.xml

Tag	Description
`<component>`	Root element in component description.
`<description>`A component that allows the user to select one or more objects from a list.`</description>`	Set the description of the component to display on the palette.
`<model class="">`	Set the class of the model – a subclass of JavaInfo.
`<order>last</order>`	The following component order parameters are supported: ● «default» - to put component after any sibling. ● «last» - to put component after any sibling. ● «last» - to put component before any sibling, as first child of parent. ● «beforeSibling nextComponentClass» - to put component before components with given class. For example in table-like components it is often requirement that you should create columns before items.

`<!-- CREATION -->`
`<creation>`
`    <source><![CDATA[new org.eclipse.swt.widgets.Button(%parent%,`
`org.eclipse.swt.SWT.NONE)]]></source>`
`    <invocation signature="setText(java.lang.String)"><![CDATA["New Button"]]></invocation>`
`</creation>`
`<creation id="empty">`
`    <source><![CDATA[new org.eclipse.swt.widgets.Button(%parent%,`
`org.eclipse.swt.SWT.NONE)]]></source>`
`</creation>`
`<creation id="check" name="Check Button">`
`    <source><![CDATA[new org.eclipse.swt.widgets.Button(%parent%,`
`org.eclipse.swt.SWT.CHECK)]]></source>`
`    <invocation signature="setText(java.lang.String)"><![CDATA["Check Button"]]></invocation>`
`    <invocation signature="setSelection(boolean)"><![CDATA[true]]></invocation>`
`    <description>Instances of this class represent a selectable user interface object that issues notification when checked and unchecked.</description>`
`</creation>`
`<creation id="radio" name="Radio Button">`
`    <source><![CDATA[new org.eclipse.swt.widgets.Button(%parent%,`
`org.eclipse.swt.SWT.RADIO)]]></source>`
`    <invocation signature="setText(java.lang.String)"><![CDATA["Radio Button"]]></invocation>`
`    <description>Instances of this class represent a selectable user interface object that issues notification when selected and unselected.</description>`
`</creation>`
	Specifies source code and method invocations that should be used to create this component using the constructor creation support (it is expected that the code in `<source>` is an object created using the constructor).
	
	Optional `<invocation>` tags allow method invocations to be added directly after the component has been added. This is useful when setting the default text, selection, etc.
	
	The «id» attribute can be used with the «creationId» attribute of the `<component>`, in the `<palette>` in plugin.xml to specify several variants of creating components with the same type. For example, a SWT Button can be a push/check/radio button, but the only difference is the style specified in the constructor.

`<standard-bean-properties/>`	Adds standard bean properties with getters and setters. Property, is created only if Designer has an editor for this property type. For example, String, Icon, int, etc. Each «standard» property has an id «`signatureOfSetterMethod`». In the following tags, use «`name`» to specify property - rules of matching are following: ● «`m:nameOfMethod`» - specifies that only the setter should be considered, and the name of the method is given; ● «`f:fieldName`» - specifies the exact name of the field; ● «`setText(java.lang.String[])`» - specifies the exact signature for setter; ● «`nameOfMethodOrField`» - it first tries to find the setter with the given name, then tries to find a field with the given name.
`<public-field-properties/>`	Adds field based properties. Property is created only if Designer has an editor for this property type, for example String, Icon, int, etc. Each «field» property has id «`property:field: nameOfField`».
`<properties-preferred names="text icon"/>`	Specifies that the listed properties are preferred, i.e. properties often used with this component. For example text and image on button. These properties are highlighted in the properties table (by default the bold font is used).
`<properties-normal names="align valign"/>`	Specifies that the listed properties are normal, so undo any special category settings specified in descriptions for superclass.
`<properties-advanced names="bounds location size maximumSize minimumSize preferredSize"/>`	Specifies that the listed properties are advanced. Advanced properties are not displayed by default in the properties table. They are also painted with a different font and color (defaults to italic font and gray color). Specifying advanced properties reduces the number of properties listed in the properties table, and also speeds up searching.
`<properties-hidden names="UI label"/>`	Specifies that listed properties should not be used at design time. These can cause some bad effects, so we don't want to show them to the user. There is no way for user to make them visible.
`<properties-noDefaultValue names="pageLeft pageTop right bottom"/>`	Specifies that default value for these properties should not be asked. This feature is used in GWT libraries, such as GWT-Ext and SmartGWT. These libraries have specific quirks - they create JavaScript object for reading value, and in same time don't allow to set many properties after creating object.

``xml
<property-tag name="title" tag="isText" value="true"/>
``
	Sets some tag for standard property. Designer uses these tags to interpret properties in some special way. For example «isText == true» means that this property is displayed as text for this component, so when the user requests direct edit, property «title» should be edited.
	
	See other tag in table below.

``xml
<property id="setComponentOrientation(java.awt.ComponentOrientation)">
  <editor id="staticField">
    <parameter name="class">java.awt.ComponentOrientation</parameter>
    <parameter name="fields">UNKNOWN LEFT_TO_RIGHT
RIGHT_TO_LEFT</parameter>
  </editor>
</property>
``
	Specifies the editor for the property.

``xml
<property id="setDisplayedMnemonic(char)">
      <category value="preferred"/>
</property>
 	
<property id="setDisplayedMnemonic(int)">
      <category value="advanced"/>
      <editor id="swing.displayedMnemonic"/>
</property>
``
	Specifies category for the property. Possible values are: preferred, normal, advanced, hidden.

``xml
<property id="setUseHashlookup(boolean)">
      <defaultValue value="false"/>
</property>
<property id="setLabelAlign(com.gwtext.client.core.Position)">
      <defaultValue value="com.gwtext.client.core.Position.LEFT"/>
</property>
``
	Specifies the default value for the property. This is useful, for example, when some «standard bean property» has only a «set» method, but not a «get» method, so we can not get the default value from the component instance.
	Any MVEL expression/script can be used, editor ClassLoader is set as context ClassLoader, so classes from the project can be referenced directly.

``xml
<property id="setFoo(int)">
      <getter name="getMyFoo"/>
</property>
``
	Specifies the alternative «getter» method. Sometimes a «normal» setter method does not have a corresponding getter method. For example, for «setFoo» we expect «getFoo» as the getter method. To use a different getter method, specify <getter and give the name of the getter method.

``xml
<methods-include signature="/set\w+\([^,]+\)/"/>
<methods-exclude signature="setVisible(boolean)"/>
``
	Specifies that methods with given signature should be included/excluded in the set of executable methods. Designer executes only the methods included in the executable set, invocations of other methods that are present in the source code are ignored.

	If the signature starts with «/» then it is handled as a regular expression.

`<method name="setLocation" order="beforeAssociation">`
    `<parameter type="int"/>`
    `<parameter type="int"/>`
    `<tag name="tagName" value="tagValue"/>`
`</method>`
	Specifies executable method with parameters of the given type. See the description for the «parameter» tag below:
	
	«`order`» - specifies the location for invoking this method. See the «`method-order`» for a full description.
	
	«tag» element provides a way to set some tag for a method. Known tags are:
	
	1. «noFactory» used to indicate that this method should not be marked as "extracted" by default, when we open the «Extract Factory» dialog.
	2. and much more... no description yet...

`<parameter type="" [name=""] [parent="true/false"] [child="true/false"] [parent2="true/false"] [child2="true/false"] [property="id"] [defaultSource=""]>`
      `<editor .../>`
`</parameter>`
	Describes the parameters for the method or constructor.
	
	    «type» – the fully qualified type of the parameter, for example «int» or «java.lang.String»; «parent» - is «true» if the parameter is a parent of the component, for example in SWT parent is passed in constructor; «child» - is «true» if parameter is child of this component, for example in Swing the method «add(java.awt.Component)» receives the child. «parent2»/«child2» - if a pair of parameters is defined, the invocation of the method creates a parent/child link. For example, it is possible to create a method like «addInGrid(x, y, parent, child)» that adds the given child on the given parent in the specified grid cell. «property» specifies the id of property to which this parameter is bound. It is usually used in the constructor description. Oftentimes, in Swing, there are several constructors with text/image/etc properties. «defaultSource» - another attribute that is useful mainly for constructors. When the user asks for «cleaning» property, we use this source (because we can not remove the argument of the constructor). If there is no «defaultSource», «cleaning» will do nothing. «editor» tag that allows to specify the editor for this parameter, usually in the constructor. If there is no «editor» attribute defined, type based editor will be used.

`<method-order>`
    `<default order="afterAssociation"/>`
    `<method signature="setScrollPosition(int)" order="last"/>`
    `<methods order="beforeAssociation">`
        `<s>setEnabled(boolean)</s>`
	The following order parameters are supported:
	    «last» - after any related statement and any child. «afterCreation» - directly after the creation statement. This order is used by default. «beforeAssociation» - before the statement that associates this component

`</methods>`
 	
`<method-property title="alignment" method="setAlignment(int,int)"/>`	

`<method-single-property title="myFoo" method="foo(int)"/>`	Specifies that a property with the title «myFoo» should be created. This is the same as the property created by `<standard-bean-properties/>`, but the method is not a setter, therefore we need to specify it directly. Id of such property is «foo(int)».

`<constructors>`
`  <constructor>`
`    <parameter type="javax.swing.Icon" property="setIcon(javax.swing.Icon)"/>`
`  </constructor>`
`  <constructor>`
`    <parameter type="java.lang.String" property="setText(java.lang.String)"/>`
`  </constructor>`
`  <constructor>`
`    <parameter type="java.lang.String" property="setText(java.lang.String)"/>`
`    <parameter type="javax.swing.Icon" property="setIcon(javax.swing.Icon)"`
`defaultSource="null"/>`
`  </constructor>`
`</constructors>`
	Describes one or more constructors, similar to methods with a \<constructor\> tag and one or more \<parameter\> tags. You don't have to describe all the constructors. Descriptions are useful if you want to bind parameters to properties, specify editors, parent/child attributes, etc.

`<exposing-rules>`
`    <exclude package="java.awt"/>`
`    <exclude package="javax.swing"/>`
`    <include method="getContentPane"/>`
`</exposing-rules>`
	Describes the rules for exposing children components (similar to getXXX() methods). You can include or exclude a set of methods. Note, that rules are executed in reverse order, so if you later have some rules that includes (enables) using some method, it will be used for exposing, even if previous rules exclude it.
	• «package» accepts any method of classes exactly in the same package, without subpackages;
	• «method» accepts method with exactly the same name, without parameters in any package.
	Of course, all these filters will work only for components that correspond to this description or its subclasses.

Table 1.2 component parameters.

`<parameters>`
`    <parameter name="dontCacheDescription">true</parameter>`
`</parameters>`
	Specifies that description for this component should not be cached even if package has «`.wbp-cache-descriptions`» file, so requires caching by default.

`<parameters>`
`    <parameter name="layout.has" value="false"/>`
`</parameters>`
	Parameter «layout.has» specifies if this component has layout manager (supported by Swing anf eSWT).

`<parameters>`
`    <parameter name="variable.name" value="tree"/>`
`    <parameter name="variable.acronym" value="tree"/>`
`</parameters>`
	Parameters «variable.name» and «variable.acronym» specify default variable name and acronym for this component. So, component developer can tweak code generation. For example for javax.swing.JTree we can specify that acronym «tree» should be used, not auto generated «tr».

`<parameter name="variable.validateID"><![CDATA[`
`    import com.instantiations.designer.core.model.variable.description.*;`
`    if (id == FieldInitializerVariableDescription.ID) {`
`      id = FieldUniqueVariableDescription.ID;`
`    }`
`    return id;`
`]]></parameter>`
	Parameter "variable.validateID" is script that can validate that given variable «id» can be used for this component, and return alternative variable.

`<parameter name='visible.inTree'>false</parameter>` `<parameter name='visible.inGraphical'>false</parameter>`	These parameters specify if component should be visible in components tree and design canvas.
`<parameter name="objectReadyValidator">component.parent != null</parameter>`	ExpressionAccessor's get default values from component object, but sometimes just fact that component has object is not enough to allow invocation of getters and reading from fields. For example GWT-Ext requires that component should be bound to parent (so its DOM Element rendered) before we can invoke most of its methods. So, we need some way to allow component specify that it is ready. Following variables are defined for script: ● «`component`» - the toolkit object.

``xml
<parameters>
    <parameter name="gridLayout.grabHorizontal" value="true"/>
    <parameter name="gridLayout.grabVertical" value="true"/>
    <parameter name="gridLayout.rightAlignment.isLabel" value="true"/>
    <parameter name="gridLayout.rightAlignment.isTarget" value="true"/>
</parameters>
``
	In grid-based layouts (such as SWT GridLayout, Swing GridBagLayout and Jgoodies FormLayout), components may be automatically aligned when dropped into layout.
	• «gridLayout.grabHorizontal» means that component's column will grab excess space, and component will fill column horizontally.
	• «gridLayout.grabVertical» means that component's row will grab excess space, and component will fill row vertically.
	• «gridLayout.rightAlignment.isLabel» means that this component is «label» like component (Label in SWT, Jlabel in Swing), so when it is on same row as «text» like component, and directly on left of it, then «label» should be aligned right.
	• «gridLayout.rightAlignment.isTarget» means that this component is «text» like component, for example Text in SWT, JtextField in Swing.

``xml
<parameters>
    <parameter name="tabOrder.isNotOrdered" value="true"/>
</parameters>
``
	Mark components that does not accept focus using this parameter, so they will be excluded from setting tab order.

``xml
<parameter name="tabOrder.disable">true</parameter>
``
	Disables displaying «tab order» property for this component.

``xml
<parameter name="databinding.disable">true</parameter>
``
	Disables displaying «bindings» property for this component.

``xml
<parameters>
    <parameter name="double-click.listener" value="action/performed"/>
</parameters>
``
	Specifies that on double click on component event listener with name «action» and its method «performed» will be opened. So, you can easily configure your custom components to generate/open often used event listener using double click.

``xml
<parameter name="events: no listeners of class">javax.swing.JPanel</parameter>
``
	WindowBuilder automatically discovers methods «addXListener» and «addXHandler» using reflection and shows them as event listeners. Using this parameter you specify that you don't want to show user listeners declared in specified class and any of its superclasses. This is useful if you build specialized component and don't want to overload user with bunch of listeners inherited from Swing (for example).

``xml
<parameters>
    <parameter name="viewer.control.method" value="getTable"/>
</parameters>
``
	For JFace Viewer, specifies name of method (without arguments) that returns Control.

``xml
<parameters>
    <parameter name="layoutData.exclude-properties" value="Class Constructor"/>
</parameters>
``
	For SWT «layout data» objects, we may exclude some properties, because they don't add any new information. For example «class» is not needed for LayoutData, it is already displayed as text. Also for RowData all constructor properties are mapped on fields, so no need to show «Constructor» property. So, we can perform some «fine tuning». Not important, just nice to have.

``xml
<parameters>
    <parameter name="layout-data.has" value="true"/>
``
	For SWT Layout, specifies that this layout has LayoutData, and its type.

```xml
<parameter name="flowContainer.1.component">TableColumn</parameter>
<parameter name="flowContainer.1.reference">TableColumn</parameter>
<parameter name="flowContainer.2">true</parameter>
<parameter name="flowContainer.2.horizontal">false</parameter>
<parameter name="flowContainer.2.component">TableItem</parameter>
<parameter name="flowContainer.2.reference">TableItem</parameter>


<!-- flowContainer for tree -->
<parameter name="flowContainer.tree">true</parameter>
<parameter name="flowContainer.tree.association">%parentAccess%add(%child
%)</parameter>


<!-- flowContainer defaults -->
<parameter
name="flowContainer.defaultComponent">com.google.gwt.user.client.ui.Widge
t</parameter>
<parameter
name="flowContainer.defaultReference">com.google.gwt.user.client.ui.Widge
t</parameter>


<!-- list of types -->
<parameter
name="flowContainer.component">org.eclipse.nebula.widgets.grid.GridColumn
org.eclipse.nebula.widgets.grid.GridColumnGroup</parameter>


<!-- MVEL script as validator -->
<parameter name="flowContainer.component-validator"><![CDATA[
    isComponentType('org.eclipse.nebula.widgets.grid.GridColumn') ||
    isComponentType('org.eclipse.nebula.widgets.grid.GridColumnGroup')
]]></parameter>
<parameter name="flowContainer.reference-validator"><![CDATA[
    isReferenceType('org.eclipse.nebula.widgets.grid.GridColumn') ||
    isReferenceType('org.eclipse.nebula.widgets.grid.GridColumnGroup')
]]></parameter>

<!-- simple container -->
```

«isHorizontal()» that will be evaluated using JavaInfo model for container as context.

Parameters «flowContainer.component» or «flowContainer.reference» specify what kind of components can be dropped on this container and can be used as reference to drop before. You can use single name of type or space separated list of them. When only «flowContainer.component» specified, same validator is used for references.

If you need to perform more complex check for component/reference, you can specify «flowContainer.component-validator» and «flowContainer.reference-validator». In these scripts you can use variables «container», «component» and «reference». Also, you can use methods «isComponentType()» and «isReferenceType()» to check that component or reference are subclasses of given one. You can also call any methods of «container», «component» and «reference».

Often all containers in same toolkit accept same type of components and references, for example «org.eclipse.swt.widgets.Control» is SWT or «com.google.gwt.user.client.ui.Widget» in GWT. So, it is convenient to use «flowContainer.defaultComponent» and «flowContainer.defaultReference» for superclass of all containers and don't specify them separately in each place (but you may if you want). Note, that defaults are used only when you don't specify both – component and reference validators.

Parameter «flowContainer.association» specifies for child component is associated with this flow container. It is required for add/move operations in JavaInfoUtils. Following types of associations are supported:
- «invocationChild» - usual association in Swing and GWT, when child is passed into «add» method, it requires exact specification of method after space «%parentAccess%add(%child%)». As this type of associations is often used, it does not require separate «invocationChild» prefix, you can just write invocation itself.

WindowBuilder supports declarative «simple» containers, i.e. containers that

`<parameter name="simpleContainer">true</parameter>`
`<parameter name="simpleContainer.association">%parentAccess`
`%setWidget(%child%)</parameter>`
	accept only one component.
	
	When just «`simpleContainer`» is specified, then simple container for both design canvas and components tree is declared. When «`simpleContainer.canvas`» and «`simpleContainer.tree`» you can specify separate settings for design canvas and components tree.
	
	Parameter «`simpleContainer.component`» specifies what kind of components can be dropped on this container. You can use single name of type or space separated list of them.
	
	Often all containers in same toolkit accept same type of components and references, for example «org.eclipse.swt.widgets.Control» is SWT or «com.google.gwt.user.client.ui.Widget» in GWT. So, it is convenient to use «`simpleContainer.defaultComponent`» for superclass of all containers and don't specify it separately in each place (but you may if you want).
	
	Parameter «`simpleContainer.association`» specifies for child component is associated with this flow container. It is required for add/move operations in JavaInfoUtils. Following types of associations are supported:
	● «`invocationChild`» - usual association in Swing and GWT, when child is passed into «`add`» method, it requires exact specification of method after space «`%parentAccess%add(%child%)`». As this type of associations is often used, it does not require separate «`invocationChild`» prefix, you can just write invocation itself.

`<!-- bounds for DialogBox -->`
`<parameter name="applyTopBoundsScript"><![CDATA[`
`    widget.getWidget().setSize(size.width, size.height);`
`    widget.show();`
`]]></parameter>`
 	
`<!-- bounds for Composite -->`
`<parameter name="applyTopBoundsScript"><![CDATA[`
`    rootPanel.add(widget);`
`    widget.setSize(size.width, size.height);`
`]]></parameter>`
	For some components in GWT you can not set size directly, see http://code.google.com/p/google-web-toolkit/issues/detail?id=2595, so you can specify script to do this. Following variables are defined in script:
	● «`DOM`» - the GWT DOM class;
	● «`rootPanel`» - the instance of RootPanel;
	● «`widget`» - the instance of top level widget;
	● «`size`» - the size of widget requested by user, has properties «`width`» and «`height`»;

`<parameter name="setTopBoundsScript"><![CDATA[`	Allows to perform some operation when user resizes this top level component.

``
        content = model.getWidget();
        if (content != null) {
                panelSize = model.bounds.size;
                contentSize = content.bounds.size;
                newContentWidth = size.width - (panelSize.width -
contentSize.width);
                newContentHeight = size.height - (panelSize.height -
contentSize.height);
                model.widget.setSize(newContentWidth,
newContentHeight);
        }
]]>
``
	Following variables are defined in script:
	   ● «DOM» - the GWT DOM class;
	   ● «rootPanel» - the instance of RootPanel;
	   ● «widget» - the instance of top level widget;
	   ● «size» - the size of widget requested by user, has properties «width» and «height»;

``
<parameter name="copyChildPropertyTop from=Association/direction
to=Direction category=system(7)"/>
 	
<parameter name="copyPropertyTop from=Constructor/columnWidth to=width
category=normal"/>
``
	Adds copy of given property (usually part of some complex property) as top level. Following attributes are supported:
	   ● «from» - the «/» separated path to source property;
	   ● «to» - the name of top level target property;
	   ● «category» - the category of target property, may be «preferred», «normal», «advanced», «hidden» or «system(int)»;

``` <parameter name="modelMethodProperty getter=getAnchor setter=setAnchor type=java.lang.String title=anchor"/> <parameter name="modelMethodChildProperty getter=getWidth setter=setWidth title=Width category=system(7) type=java.lang.String child=com.instantiations.designer.gwt.model.widgets.Widget_Info"/> ```	Adds new property to component/its_child that is bound to getter/setter method in JavaInfo model.

``
<parameter name="GEF.requestValidator.parent">true</parameter>
 	
<parameter name="GEF.requestValidator.child"><![CDATA[
  isComponentType(parent, 'org.eclipse.swt.widgets.Table') &&
parent.object.itemCount <= 2
]]></parameter>
``
	Sometimes you need to add limitations on what children can be dropped on parent (during create/paste/move/add operations).
	
	So, you can specify two MVEL scripts:
	   ● «GEF.requestValidator.parent» - for parent, to check all possible children. If no such script, then «true».
	   ● «GEF.requestValidator.child» - for child, to check that this child can be dropper only on some specific parents. For example, here we check that «parent» is SWT Table that has no more than 2 items. If no such script, then «true».

	Following variables and functions can be used: ● «parent» - the parent where we are going to drop child; ● «child» - the child to drop; ● «isComponentType(javaInfo, componentClassName)» - checks that give object is JavaInfo and its component type is instance of given one. ● «ReflectionUtils» - class with many Class/Method/Field utils. In addition WindowBuilder always check that constructor/factory of child is compatible with parent (i.e. you can pass parent as parameter into constructor/factory, if needed). If one of the conditions (compatibility or parent/child scripts) is false, then parent becomes «transparent» for this child, so request will be passed to the parent parent.
`<parameter name="GEF.transparentOnBorders.always">true</parameter>`	Specifies that EditPart for this model is «transparent» on borders, in 3 pixels. This isuseful when we know that container places these children without gap between them, so to hit container between children we need to make them transparent. NOTE: may be we should replace this with some parameter for container, because container knows how it lays out its children.

`<parameter name="refresh_afterCreate"><![CDATA[`
`    import com.google.gwt.user.client.ui.*;`
`    if (object.getItemCount() == 0) {`
`        object.addItem("Empty Tree.");`
`        object.addItem("Drop items in structure.");`
`    }`
`]]></parameter>`
	Script «refresh_afterCreate» allows you to tweak created component object, for example show some message if component can accept some children or some properties should be set.

`<parameter name="parser.preferredRoot">true</parameter>`	Sometimes after parsing we have more than one possible root JavaInfo. Usually we select root that has biggest hierarchy. However sometimes we know that there is better choice, for example GWT 1.6 examples use RootPanel and also create DialogBox. We want to show RootPanel, because it is entry point, and DialogBox is something dynamic. So, we mark RootPanel with this tag and it is always selected as preferred root. Note, that if there are more than one preferred root, actual root again will be selected by size of hierarchy.
`<parameter name="GEF.clickToParent">true</parameter>`	If some model is marked with this parameter, then click on it on design canvas will cause moving selection request to parent. So, in most cases this will cause parent

	selection instead of this model selection. This is used for example in ApplicationWindow, where because of binary execution flow we have several model for same Shell object – window itself, «parent» parameter for method «createContents()». Without this parameter click on window header will cause «parent» selection. But user expects ApplicationWindow selection, for example to resize it.
`<parameter name="topBounds.pack">true</parameter>`	Specifies that superclass of this Swing Window has pack() invocation, so default size (450x300) should not be applied to this Window.
`<parameter name="Wrapper.method">getControl</parameter>`	Specifies method used to access wrapped component of viewer.
`<parameter name="Wrapper.noWrapped.inConstructors">true</parameter>`	Specifies that this viewer (and its subclasses) has no constructors that accept wrapped component (for example CellEditor-s have no such constructors, but Jface Viewer-s have them). Or you can use `"Wrapper.notWrapped"` tag for single constructor or factory parameter.

`<parameter name="liveComponent.forcedSize.use"><![CDATA[`
`    isStrict && isExplorer`
`]]></parameter>`
`<parameter name="liveComponent.forcedSize.width">100px</parameter>`
`<parameter name="liveComponent.forcedSize.height">auto</parameter>`
	WindowBuilder needs to create «live component image» when you drop component on absolute based containers/layouts. However some component don't have reasonable preferred size, so use have to force it in description. In GWT size can be any string which is valid CSS size, in other toolkits – only integer. Also in GWT you can use script «`liveComponent.forcedSize.use`» to check if size should be applied, if problem happens only with specific OS/Browser.

Table 1.3 method tags.

`<method name="getDocument">`
`    <tag name="exposeDisconnectedComponent" value="true"/>`
`</method>`
	Specifies that method `"getDocument()"` should be used to expose component, even if it is not connected to the host component. For example JTextComponent (such as JTextField) has method getDocument(), that returns non-visual object Document, that can not be child of visual component. But we still want to show it as child, so force this.

Table 1.4 standard property tags.

`<property-tag name="title" tag="isText" value="true"/>`	Tag «isText == true» means that this property is displayed as text for this component, so when user requests direct edit, property «title» should be edited.
`<property-tag name="image" tag="isImage" value="true"/>`	Tag «isText == true» means that this property contains displayed image, so when (on

	absolute layout) user change one of these properties, component will be resized to its (new) preferred size.

Table 1.5 property editors with id's.

editor id	example	description
| staticField | ```xml
<editor id="staticField">
javax.swing.SwingConstants
        TOP CENTER BOTTOM
</editor>
``` | Editor that allows select one static field from given class. |
| style | ```xml
<editor id="style">
        org.eclipse.swt.SWT
        BORDER FLAT
        type PUSH PUSH CHECK RADIO
TOGGLE ARROW
        align LEFT LEFT CENTER
RIGHT
</editor>
``` | Complex editor for «int» property that supports two sets of sub-properties – set properties (zero or more can be selected) and select properties (only one can be selected).

For «select» properties name should start with word «select» and then some unique string, in «values» first word is title, second — default field name, rest words – names of static fields from class. |
| innerClass | ```xml
<editor id="innerClass">
        inner
        Sorter
org.eclipse.jface.viewers.ViewerSorter
        <![CDATA[
private static class ${name} extends
org.eclipse.jface.viewers.ViewerSorter {
        public int compare(org.eclipse.jface.viewers.Viewer
viewer, Object e1, Object e2) {
                Object item1 = e1;
                Object item2 = e2;
                return 0;
        }
}
``` | Editor that allows select existing class that extends type with «class» parameter (class or interface), or create new inner class using configured code.

　Other parameters:

- name — default name of created class;
- class — the type to find subclasses;
- source — the source for creating new class. |

2. Static/instance factory description: *.wbp-factory.xml

```xml
<?xml version='1.0' encoding='UTF-8'?>
<factory>
      <method name='createButton_xmlDesc' factory='true'>
            <parameter type='java.lang.String' property='setText(java.lang.String)'/>
      </method>
      <method name='createButton2' factory='true'>
            <parameter type='java.lang.String'/>
            <parameter type='java.lang.String'/>
            <parameter type='java.lang.String' defaultSource='"222"'/>
            <parameter type='java.lang.String'/>
      </method>
      <method name='createButton_factoryFalseOverride' factory='true'>
            <parameter type='java.lang.String' property='setText(java.lang.String)'/>
      </method>
</factory>
```

```xml
<?xml version='1.0' encoding='UTF-8'?>
<factory>
      <method name="createLabel">
            <parameter type="org.eclipse.swt.widgets.Composite" parent="true"/>
            <parameter type="java.lang.String" defaultSource="&quot;New Label&quot;" property="property.setter: setText(java.lang.String)"/>
            <parameter type="int" defaultSource="org.eclipse.swt.SWT.NONE">
                  <editor id="style">
                        <parameter name="class">org.eclipse.swt.SWT</parameter>
                        <parameter name="set">BORDER SEPARATOR WRAP</parameter>
                        <parameter name="select0">dir VERTICAL HORIZONTAL VERTICAL</parameter>
                        <parameter name="select1">shadow SHADOW_OUT SHADOW_IN SHADOW_OUT SHADOW_NONE</parameter>
                        <parameter name="select2">align LEFT LEFT CENTER RIGHT</parameter>
                  </editor>
            </parameter>
            <description>Creates a label as a part of the form.</description>
            <name>Some name for palette entry.</name>
      </method>
</factory>
```

Table 2.1 – tags in *.wbp-factory.xml

Tag	description
`<method name="createLabel" factory="true/false">`	Describes method with given name, parameters (see below) and flag «factory» that specifies if this method is factory.
`<parameter>`	See description for <parameter> in *.wbp-component.xml

`<description>`Creates a label as a part of the form.`</description>`	The description text to use for palette entry (may be overridden in it).
`<name>`Some name for palette entry.`</name>`	The name to use for palette entry (may be overridden in it). If no name, then signature of method used.

You can also specify descriptions for factory methods directly in source:

```
public final class StaticFactory {
    /**
    * @wbp.factory
    * @wbp.factory.parameter.source text "new JButton"
    * @wbp.factory.parameter.property text text
    */
    public static JButton createButton(String text, Icon icon) {
        return new JButton(text, icon);
    }
    /**
    * @wbp.factory
    * @wbp.factory.parameters.noBinding
    */
    public static JTextField createTextField(int columns) {
        return new JTextField(10);
    }
}
```

Table 2.2 – tags source for factory

Tag	description
`@wbp.factory`	Specifies if method is factory method. If this tag is used on class, then all methods of this class are considered as factories. Designer will automatically exclude methods that return «null» (such as setters) and getters, i.e. methods that starts with «set» and have no parameters.
`@wbp.factory.parameter.source parameterName expressionSource`	Sets the default source for parameter. This source will be also used during adding component using this factory method. If no such source specified, default value for type will be used – zero for numerics and «null» for objects.
`@wbp.factory.parameter.property parameterName propertyTitle`	Specifies that parameter with given name bound to some component property. This is useful for example if you want to set text of button directly in factory

	method, without additional setText() invocation.
`@wbp.factory.parameters.noBinding`	Specifies that Designer should not bind parameters to properties by names.
	Usually parameters should be bound to corresponding properties, but if there are many methods and parameters, it is tedious to describe each such binding. Instead you can just name parameters with same name as title of components property. For example, if you name parameter «text», it will be bound to property with title «text». Note that title is used, not id of property (because id is complex string).
	One special name of parameter is reserved for SWT support, parameter with name «parent» is considered as parent parameter (i.e. in same way if it would be marked as parent in *.wbp-factory.xml).

Table 3.1 – other source tags

Tag	description

``java
/**
 * @wbp.parser.entryPoint
 */
public void someMethod(Composite parent) {
    Composite composite = new Composite(parent, SWT.NONE);
}
``
	If you want to edit GUI in standalone class, which is not subclass of know component, such as Swing JPanel or SWT Composite, then you should specify WindowBuilder which method to use as entry point for execution flow. It this case WindowBuilder will not look on constructor and superclass, it will design just this specific method.

``java
mainPanel = new JPanel(); // @wbp.parser.preferredRoot
``
	Specifies that marked component should be used as root component, even when there are other possible roots, may be with bigger hierarchy (by default WindowBuilder chooses root component with biggest hierarchy).

@wbp.parser.constructor	If «this» component has only one constructor, then Designer will select it as entry point for execution flow. If «this» container has several constructors and one of them merked with this tag, then it will be used. If no tag and container has default constructor (without parameters), then it will be used implicitly. If all constructors are non-default error will be generated.

``java
@wbp.eval.method.parameter parameterName expressionSource
  /**",
  * @wbp.eval.method.parameter _int 1 + 2
  * @wbp.eval.method.parameter _true true
  * @wbp.eval.method.parameter _false false
  * @wbp.eval.method.parameter _string "ab" + "c"
  * @wbp.eval.method.parameter _null null
  */
  public Test(int _int, boolean _true, boolean _false, String _string,
Object _null) {
    super(_int, _true, _false, _string, _null);
  }
``
	If constructor (or other entry method) has parameter which value is used for creating some component or as part of some property (method invocation or field assignment) then Designer requires hint about value that will be passed with this parameter. This tag can be used to specify any statically evaluable expression.

``java
@wbp.eval.method.return parameterName
  public int root() {
    return instancePublicMethod(5);
  }
  /**
``
	Sometimes code may use some instance methods declared in same class. Designer can not evaluate these methods (at least in current implementation), so it requires some hint. In our experience just value of one of the parameters (usually even single parameter) is enough.

``
    * @wbp.eval.method.return value
    */
   public int instancePublicMethod(int value) {
     return 2 * value;
   }
``	

``
@wbp.gwt.Request
public interface MyService extends RemoteService {
   /** @wbp.gwt.Request */
   int getValue() throws Exception;
}
``
	GWT specific tag for RemoteService interface method that specifies that com.google.gwt.http.client.Request should be returned from «async» of this method.

``` /** * @wbp.nls.resourceBundle test.messages */ private static final ResourceBundle m_bundle = MyResourceBundleFactory.getMainBundle(); ```	JavaDoc tag that tells the parser where to find the resource files.

Parameters for GridLayout.

parameter	description
GridLayout.grabHorizontal	Specifies that when this component added to GridLayout, it should grab horizontally and use FILL as alignment.
GridLayout.grabVertical	Specifies that when this component added to GridLayout, it should grab vertically and use FILL as alignment.
GridLayout.rightAlignment.isLabel GridLayout.rightAlignment.isTarget	If one component is marked as «isLabel» and other as «isTarget», and «target» is in next column of same row as «label», then «label» should be align right.

Tags for «`<method>`» elements.

parameter	description
`<tag name="binaryExecutionFlow.dontVisit" value="true"/>`	Specifies that this method should not be visited when called from binary execution flow.