How to extend WindowBuilder to support new components.

Here is an overview on how to add new components to be used within WindowlBuilder. Note, that there are at least three levels of support for components:

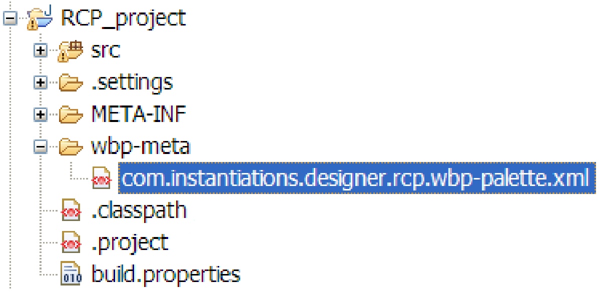
- Add component to the palette;
- Describe the component using *.wbp-component.xml;
- Write a Java code for special model/layout features.

Let's look at the 3 levels step by step:

1. Add component to palette

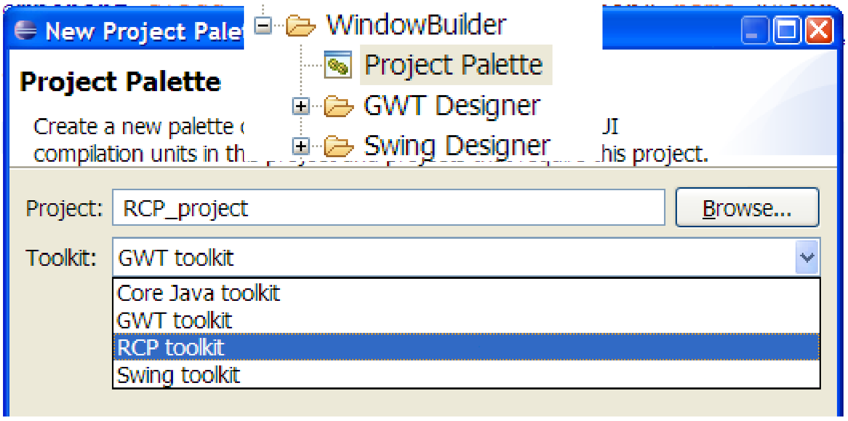
1.1. Palette contribution in project

You define your Palette contribution in an XML file called *toolkitID.wpb-palette.xml* within your project's «wbp-meta» folder. Your palette must have «category» and «component» attributes.



```xml
<?xml version="1.0" encoding="UTF-8"?>
<palette>
        <category id="someUniqueId" name="Custom category" description="Category added for project RCP_project"
                open="true">
                <component class="javax.swing.JButton"/>
                <component class="javax.swing.JRadioButton" name="Your name"
                        description="You can write any description here."/>
        </category>
</palette>
```

You can use the Project Palette wizard to generate such file for you (this is just a sample so ignore the fact that it always generates Swing components).



Project Palette

Create a new palette (… JI

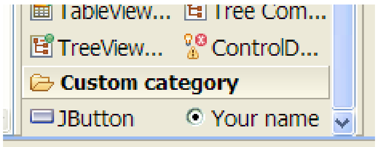
compilation units in th… his project.

Project: RCP_project

Toolkit: GWT toolkit

Most of the attributes in *.wbp-palette.xml are self-explanatory. The `next` attribute specifies the id of another category (in the same file or in a contribution from plugin.xml) before which this category should be added.

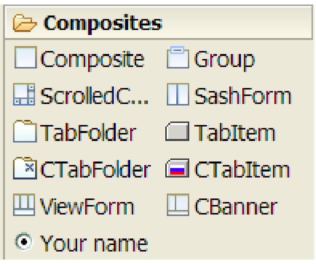
Here is the result of the previously shown contribution:



Note, that you can define `component` outside of new `category`, so put it into some existing standard category, but in this case you should specify `category` attribute.

```
<component category="com.instantiations.designer.rcp.composites"
         class="javax.swing.JRadioButton" name="Your name"/>
```

Icon for component can be placed in a png or gif format directly near the component class, with the same name as the component, or in wbp-meta folder, in the same package/folder and same name, but with .png/.gif extension.



1.2. Palette contribution in jar

You can place your ***.wbp-palette.xml** file within the project where you want to edit GUI using WindowBuilder or in any project required by your GUI project or directly in a jar file with contributed components. Just by adding **my-components.jar** to the classpath will automatically add the components to the palette. If your project does not include the jar, you will not see the corresponding entries on the palette, so they will not consume space.

1.3 Palette contribution from plugin.xml

If you want to use more features, for example specify the location of a category relative to other categories (use the `next` attribute) or always show components and add corresponding jar to the classpath automatically, then you should create a plugin and use Eclipse extensions to contribute to WindowBuilder palette.

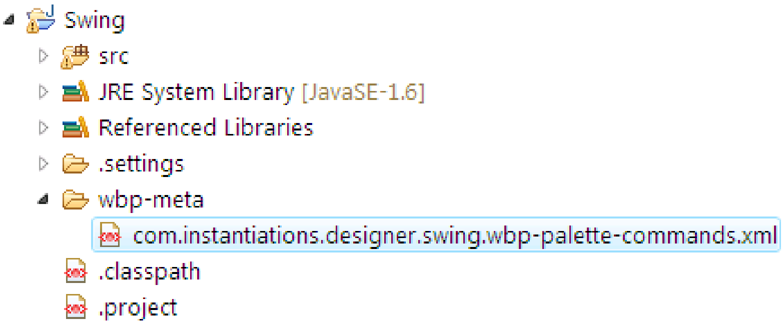
```
<extension point="com.instantiations.designer.core.toolkits">
  <toolkit id="com.instantiations.designer.rcp">
    <classLoader-library bundle="com.instantiations.designer.rcp.nebula.lib" jar="cdatetime-0.9.0.jar"/>
      <palette>
        <category id="com.instantiations.designer.rcp.nebula" name="Nebula" description="Nebula custom widgets"
         next="com.instantiations.designer.rcp.FormsAPI">
          <!-- CDateTime -->
          <component class="org.eclipse.swt.nebula.widgets.cdatetime.CButton">
            <library type="org.eclipse.swt.nebula.widgets.cdatetime.CButton"
         bundle="com.instantiations.designer.rcp.nebula.lib" jar="cdatetime-0.9.0.jar"/>
          </component>
```

In the XML file shown above, the `classLoader-library` tells WindowBuilder that the “cdatetime-0.9.0.jar” file should always be added into the ClassLoader (even if it is not in the classpath). This makes it possible for the components to load from the jar file and always displayed on the palette. You should also specify the `library` element for the `component` to tell WindowBuilder that some library should be added to the classpath the first time you try to use this component.

For a full description of palette related attributes see the toolkits.exsd schema.

1.4 Palette commands in project

There are times when you want to use different palettes for different projects (of same toolkit), or if you want to share the same palette across all the developers working on the same project. You can add an empty xml file called, *toolkitID.wbp-palette-commands.xml*, within your project's «wbp-meta» folder. When WindowBuilder finds this file, it will save all the operations that you perform on the palette (like moving categories and components, adding new components, etc) into this file and read from it later.


Swing
src
JRE System Library [JavaSE-1.6]
Referenced Libraries
.settings
wbp-meta
com.instantiations.designer.swing.wbp-palette-commands.xml
.classpath
.project

For example, after moving the «Layouts» category above the «Containers» category, this file will have the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<commands>
        <moveCategory
                id="com.instantiations.designer.swing.layouts"
                nextCategory="com.instantiations.designer.swing.containers"/>
</commands>
```

2. Describe component using *.wbp-component.xml

WindowBuilder 7.0 allows you to describe as much information about your component that can be used to provide for convenient editing. Just like the `<icon>`, `<description>` for component can be placed directly near a component class, with the same name as the component, or in wbp-meta folder, also in the same package/folder and same name.

The simplest component description looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<component xmlns="http://www.instantiations.com/D2/WBPComponent">
        <model class="com.instantiations.designer.swt.model.widgets.LabelInfo"/>
        <description>Instances of this class represent a non-selectable user interface object that displays a
                string or image. When SEPARATOR is specified, displays a single vertical or horizontal line.</description>
        <!-- CREATION -->
        <creation>
                <source><![CDATA[new org.eclipse.swt.widgets.Label(%parent%, org.eclipse.swt.SWT.NONE)]]></source>
                <invocation signature="setText(java.lang.String)"><![CDATA["New Label"]]></invocation>
        </creation>
        <creation id="separatorHorizontal" name="Horizontal Separator">
                <source><![CDATA[new org.eclipse.swt.widgets.Label(%parent%, org.eclipse.swt.SWT.SEPARATOR |
org.eclipse.swt.SWT.HORIZONTAL)]]></source>
                <description>Horizontal separator.</description>
        </creation>
</component>
```

As you can see, you can specify only the `<description>` attribute and one or more `<creation>` elements. The `<description>` attribute can also be skipped, but not recommended, as it is used by default as the description for the component on the palette.

Note, that plugin com.instantiations.designer.core includes a XSD schema file, schema/wbp-component.xsd that is used for validating *.wbp-component.xml files. It sets fairly strong rules on format, to ensure that your XML editor validates description files as you write them.

Components may have different constructors with values bound to some method-based properties. Such constructors rarely used in SWT, but often used in Swing. So, it would be convenient to describe this directly. Here is the description for JLabel. As you can see, you can use the `property` attribute with method signature (or field name for public field based properties).

```
        <!-- CONSTRUCTORS -->
        <constructors>
                <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="int" property="setHorizontalAlignment(int)"/>
        </constructor>
        <constructor>
            <parameter type="javax.swing.Icon" property="setIcon(javax.swing.Icon)"/>
        </constructor>
        <constructor>
            <parameter type="javax.swing.Icon" property="setIcon(javax.swing.Icon)"/>
            <parameter type="int" property="setHorizontalAlignment(int)"/>
        </constructor>
        <constructor>
            <parameter type="java.lang.String" property="setText(java.lang.String)"/>
            <parameter type="javax.swing.Icon" property="setIcon(javax.swing.Icon)"/>
            <parameter type="int" property="setHorizontalAlignment(int)"/>
        </constructor>
    </constructors>
```

Note, that you don't have to declare all constructors of your component. You need to do this only if you want to tell WindowBuilder something useful about the constructor. For example that some parameter is bound to some property, or specify default value of parameter (when user presses DEL key on «constructor/parameter» property).

Components may have a lot of properties, more than a hundred, for Swing. But you only use a handful of them. So, as a component developer you should specify which properties should be marked as rarely used (advanced), sometimes used (normal) and often used (preferred). Some properties may be even declared as hidden, so they will never be displayed to user, even if the component has it.

```
    <!-- PROPERTIES -->
    <properties-preferred names="text icon labelFor"/>
    <properties-advanced
        names="border disabledIcon displayedMnemonicIndex horizontalTextPosition iconTextGap
verticalTextPosition"/>
    <properties-hidden names="UI"/>
```

WindowBuilder understands a special `<tag>` property, such as marking it as a «text» property, so you can easily edit its text by pressing the Space key when the component is selected.

```
    <property-tag name="text" tag="isText" value="true"/>
```

Properties have different types and for most standard types, WindowBuilder includes a special property editor to use in the properties table. However, there are times when the standard editor is not good enough, for example for integer enumerations. In this case, you should specify a «configurable property editor».

```
    <property id="setVerticalAlignment(int)">
        <editor id="staticField">
            <parameter name="class">javax.swing.SwingConstants</parameter>
            <parameter name="fields">TOP CENTER BOTTOM</parameter>
        </editor>
    </property>
    <property id="setHorizontalAlignment(int)">
        <editor id="staticField">
            <parameter name="class">javax.swing.SwingConstants</parameter>
            <parameter name="fields">LEFT CENTER RIGHT LEADING TRAILING</parameter>
        </editor>
    </property>
```

See Table 1.5 in *.wbp-component.xml documentation for a list of existing configurable editors.

If for some reason you don’t want to specify a property category using `<properties-*>` element, you can do this separately for some property.

```
    <property id="setDisplayedMnemonic(char)">
        <category value="preferred"/>
    </property>
```

Some components can not be used separately, but are containers that accept other components. Oftentimes, the description in the XML file is enough to support this. There are two types of description based containers: simple (accept only one component) and flow (accept several components, allow to order them). See the `simpleContainer` and `flowContainer` documentation.

3. Write Java code

Sometimes descriptions are not enough. For example you need some special editing behavior, so you should write GEF LayoutEditPolicy. In this case you need to write a plugin to host your models and policies.

Here is how you can specify a model for a component in its description:

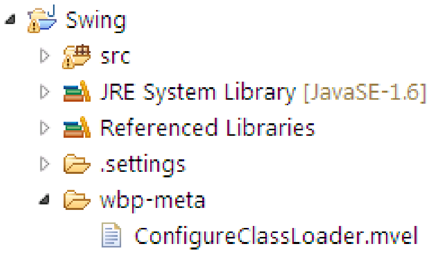
```
<model class="com.instantiations.designer.swing.model.component.ComponentInfo"/>
```

Class ComponentInfo is an indirect subclass of JavaInfo that provides basic support for all Java models. Usually all your models will be subclasses of ComponentInfo/ContainerInfo (for Swing) or ControlInfo/CompositeInfo (for SWT).

WindowBuilder provides a lot of broadcast notifications that can be used to be informed about some event, or participate in them. For example you may install a listener for adding properties to any component (by your choice), even if it is not yours, see com.instantiations.designer.core.model.broadcast.JavaEventListener.addProperties(JavaInfo, List<Property>). Or you can be informed that some JavaInfo component was deleted, so you may also want to delete some other resources/source, see com.instantiations.designer.core.model.broadcast.JavaEventListener.deleteAfter(JavaInfo, JavaInfo).

4. Configuring static objects

Sometimes you know that your application configures environment in which it executes. For example loads preferences from file/database, configure default settings of layout managers, etc. In most cases you can not execute same Java configuration code, so WindowBuilder provides support for «design time configuration». To use it, create a file called *`ConfigureClassLoader.mvel`* in your project's *`wbp-meta`* folder.



MVEL http://mvel.codehaus.org/ is a powerful expression language for Java-based applications. For example this script sets default insets (we don't recommend to use such strange values :-)) for MigLayout on JPanel containers.

```
import net.miginfocom.layout.*;
PlatformDefaults.setPanelInsets(
        new UnitValue(20),
        new UnitValue(50),
        new UnitValue(10),
        new UnitValue(5));
```