Implementation of ULC Visual Editor for Eclipse

Janak Mulani & Sibylle Peter
Canoo Engineering AG
Kirschgartenstrasse 7
CH 4051 Basel
janak.mulani@canoo.com
sibylle.peter@canoo.com

Abstract

Eclipse's visual editor for Java is a code-centric editor that enables visual design of applications with Swing, AWT or SWT GUIs. ULC (UltraLightClient from Canoo Engineering AG) is a Java library of GUI components that provide server side API to Swing. ULC enables building of server centric rich Internet applications. ULC Visual Editor helps to visually design applications with ULC. ULC Visual Editor is an Eclipse plug-in extended from the Eclipse visual editor plug-ins. This paper gives a functional and architectural overview of the ULC Visual Editor and describes how it was extended from the Eclipse Visual Editor.

1. Introduction

This paper describes the implementation of ULC Visual Editor as an extension of Eclipse's visual editor for Java (Visual Editor). It aims to achieve the following goals:

1. Provide a functional overview of ULC Visual Editor (Section 2, p. 3)
2. Provide a architectural overview of ULC Visual Editor (Section 3, p. 6)
3. Provide a documentation about extending Visual Editor (Section 4, p. 12)
4. List difficulties and recommendations for the future development of Visual Editor (Section 5, p. **Error! Bookmark not defined.**)

In this section we give an overview over Visual Editor for Java, UltraLightClient and ULC Visual Editor.

1. 1. UltraLightClient

UltraLightClient[1] is a library to build Rich Internet Applications (RIA) in Java. This standards based Java library enables rich yet thin client, highly responsive graphical user interfaces (GUIs) for enterprise web applications within J2EE and J2SE infrastructures. ULC provides a standard set of server side GUI components (ULC Widgets like Tree, Table, List, Buttons, Text, Menus, etc.) with an API similar to Swing (Server side Swing). The ULC library handles application distribution and client/server communication thus shielding the developer from the complexities of client/server code split and communication optimization. ULC applications are deployed on the server in every J2EE compliant web or EJB container. The user interface is handled by a thin, application-independent UI presentation engine implemented in Java.

The UI Engine can be distributed using any deployment mechanism e. g.:

- as an applet
- using Java Web Start
- integrated in Eclipse RCP

ULC application can be deployed as easily as HTML based web applications but provides the same richness and interactivity of fat client applications.

1. 2. Visual Editor for Java

Eclipse's Visual Editor for Java[2] (Visual Editor) is a code-centric editor that enables visual design of applications with Swing, AWT or SWT GUIs. The visual editor is based on the Java Beans component model and works with .java source files. It combines a source view with design view. It enables visual design, layout and preview of visual components in a graphical editor (design view) as well as direct editing of Java source code in the Java Editor (source view). With the Visual Editor, users create a class using Swing/AWT/SWT visual components (containers, components, menus) from a design palette. This class can be an executable application (a class with a main method) or it can be a Java Bean that can be included in another class. To be used as a Java Bean by Visual Editor the only requirement is that the editor knows how to instantiate the class.

[1] See http://www.canoo.com/ulc/ for more information about UltraLightClient
[2] See for more information about the Eclipse Visual Editor Project

1. 3. ULC Visual Editor

ULC Visual Editor extends Eclipse's Visual Editor plug-in and provides a visual editor for UltraLightClient. It enables visual design of GUIs from a palette of ULC widgets. ULC Visual Editor provides a WYSIWYG graphical editor and execution facility for rapid development of ULC applications. ULC application classes can be executed in ULC *DevelopmentRunner*, a standalone launcher for ULC applications.

Section 2 describes various functional parts of the Visual Editor. Section 3 gives an overview of the architecture underlying those functional parts. Section 4 explains the implementation and is divided into the following parts:

- declarative specifications that define and link different parts of Visual Editor (section 4. 1)
- framework hook class hierarchies that implement each specification item including the APIs implemented for each class (section 4. 2)

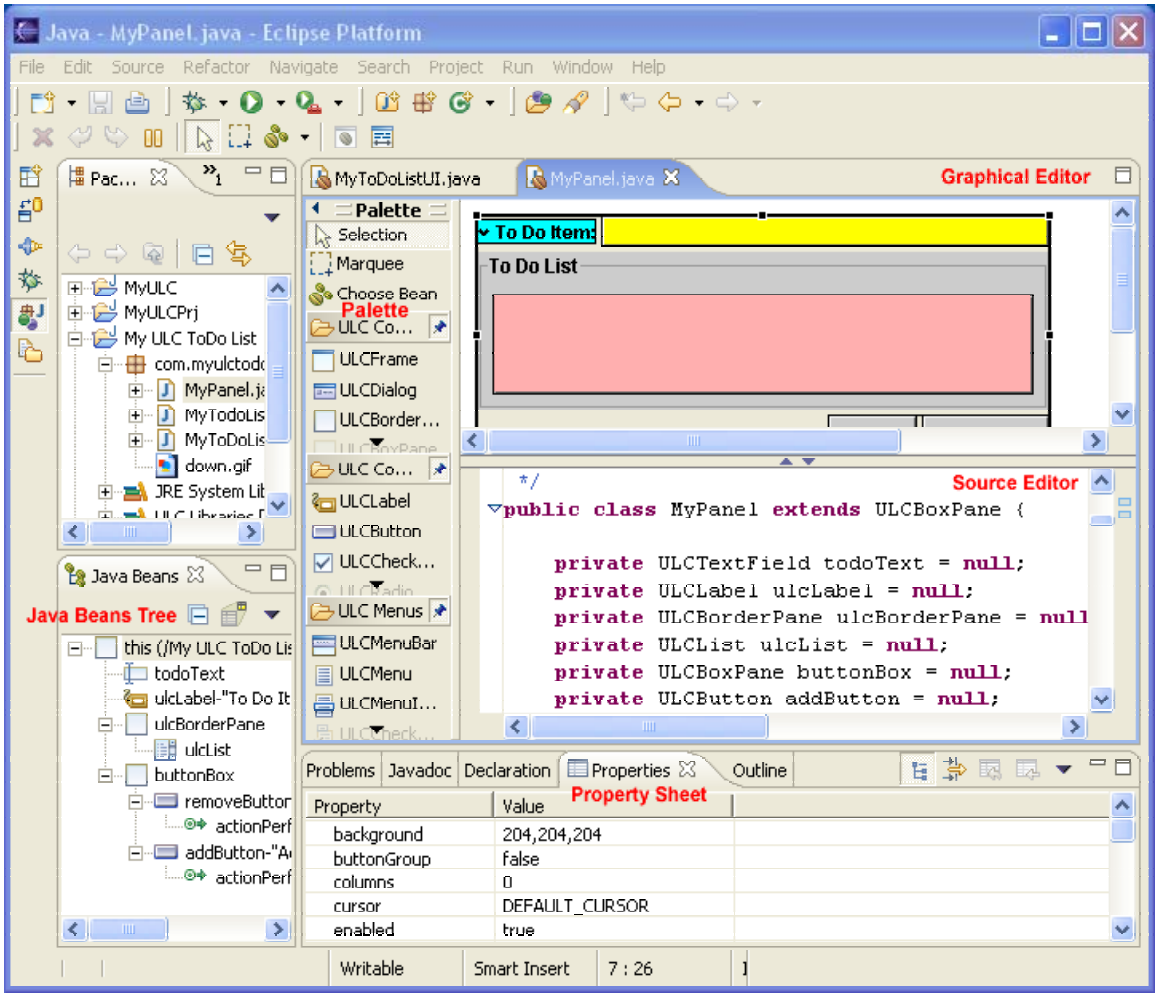
The description of the specifications and implementation also lists the framework classes and APIs that need to be generalized, opened up and documented for the ease of extension of the Visual Editor plug-in. Finally we also describe some open issues.

2. Visual Editor functional overview

The Eclipse Visual Editor consists of the following parts:

- **Palette**
- **Graphical Editor** (canvas)
- **Source Editor**
- **Property Sheet**
- **Java Beans Tree**

The picture below identifies these parts within the Eclipse window. The next sections describe the functionality of each part.



2. 1. Palette

The **Palette** contains beans that can be instantiated and visualized in the **Graphical Editor**. The user drags the beans from the **Palette** and instantiates them by dropping them on the **Graphical Editor** or in the **Java Beans Tree**. Relevant code is generated and shown in the **Source Editor**.

2. 2. Graphical Editor

The **Graphical Editor** is a WYSIWYG canvas, which previews the beans of the GUI being composed in a graphical representation. Beans are shown in the **Graphical Editor** together with their graphical representation. The source is updated whenever the beans are modified in the **Graphical Editor**. The widgets can be modified directly or by popup menu actions. Changes in the source files are reflected in the **Graphical Editor**. Due to this round tripping of the **Graphical Editor** and **Source Editor** Visual Editor can be used not only as a tool to generate code but also as an editor to show the effect of source code modifications during development.

2. 3. Java Beans Tree

The **Java Beans Tree** displays the structure of the beans representing the containment hierarchy. The selection between the entries in the **Java Beans Tree** and the **Graphical Editor** is synchronized both ways. The **Java Beans Tree** view displays each bean

- as a node with the same icon as in the palette
- with the name of the instance variable used in the source code
- with some properties like *text* or *label*.

In addition to displaying the structure of the existing components, **Java Beans Tree** is used to

- reorder and nest components.
- manipulate the beans through popup menu actions (events, renaming etc.)

2. 4. Property Sheet

The **Property Sheet** shows the properties for the bean that is selected in either the **Graphical Editor** or the **Java Beans Tree**. The properties are obtained from introspection of the beans. The value of each property is shown in the **Property Sheet**. It can be edited with an associated property editor. It is also possible to implement a filter to toggle the view between all properties and the most common properties.

2. 5. Source Editor

Visual Editor is a source-centric editor that reads and writes .java files. Changes made in the **Graphical Editor**, **Java Beans Tree** view or **Property Sheet** are reflected in the source code and vice versa. The round-trip updates between the **Graphical Editor**, **Java Beans Tree** and **Property Sheet** and the **Source Editor** are incremental.

The .java source file is the only persistent entity in the workbench, no other meta data is stored. When Visual Editor is opened, to determine the initial state of the beans, the .java file is parsed and analyzed for certain patterns[3]. These are used to create a model of the beans and their initial property settings and any relationships between them. Modifications to the source are reflected in the **VE Model** as long as the source still adheres to the recognized patterns. Most expressions are successfully parsed, but not all expressions can be correctly evaluated. In this case the editor displays warnings.

For a visual class, the source code is generated as per the templates specified in the Java Emitter Template (JET) framework[4]. If the edited class extends a bean, the instance being edited is represented with a special bean called a 'this' part and the code looks as follows:

---

[3] (a description of these patterns is available at http://www-106.ibm.com/developerworks/websphere/library/techarticles/0303_winchester/winchester.html).

[4] Eclipse's Java Emitter Templates (JET) is an open source tool for generating code within the Eclipse Modeling Framework (EMF). For more information see http://www-128.ibm.com/developerworks/library/os-ecemf2/.

```
public class MyFrame extends ULCFrame {
     public MyFrame() {
          super();
          initialize();
     }

     public void initialize() {
          this.setTitle("My To Do List");
     }
}
```

If an ULC Widget is created as a member of a visual class then an instance variable of the class and a getter method is created.

```
private ULCFrame ulcFrame = null;  //  @jve:decl-index=0

private ULCFrame getUlcFrame() {
     if (ulcFrame == null) {
          ulcFrame = new ULCFrame();
          ulcFrame.setTitle("ULCFrame");
     }
     return ulcFrame;
}
```

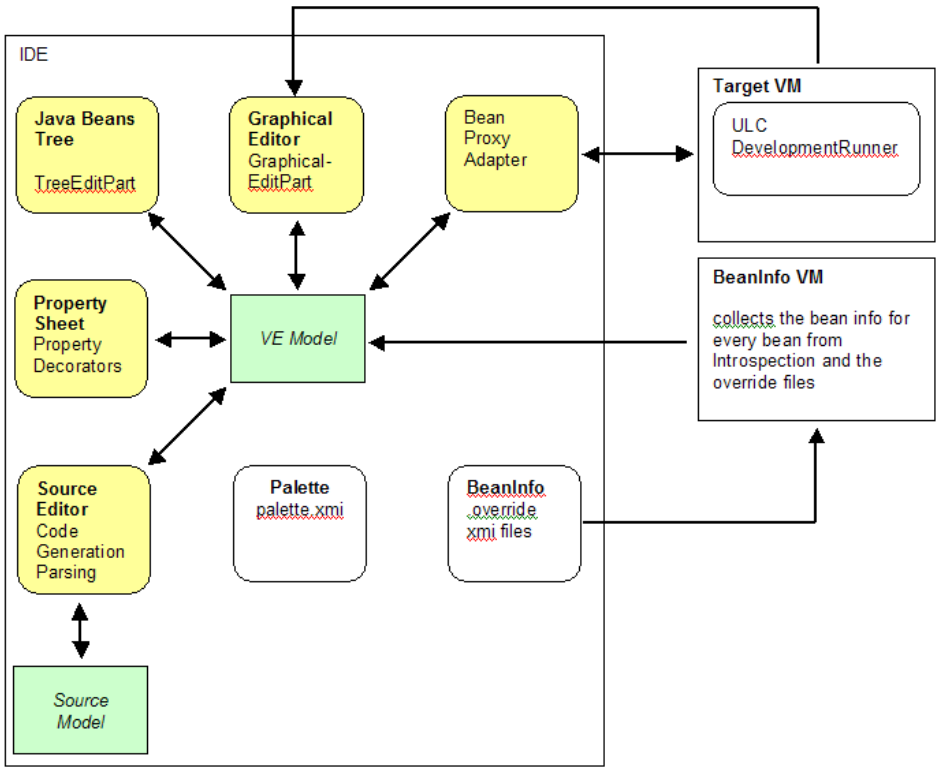
3. Visual Editor Architectural Overview

The goal of this section is to provide an overview of the architecture of Visual Editor. The main focus lies on the areas, which were important when extending Visual Editor and developing ULC Visual Editor.

Behind each functional part of Visual Editor lies an architectural component that implements its functionality. The following table lists the functional parts and their corresponding architectural components:

Graphical Editor	Graphical Editing Framework (GEF)
Java Beans Tree	Tree Editor Framework (Part of GEF)
Source Editor	Code Generation / Parsing
Property Sheet	JavaBeans BeanInfo and Property Adapters
Palette	GEF, .xmi specification file

The following diagram shows the architectural components of the Visual Editor and the relationships between them. Visual Editor is based on the Model View Controller (MVC) architecture. It maintains two internal models, the VE and Source Model. The **VE Model** is implemented using the Eclipse Modelling Framework (EMF)[5]. The views **Java Beans Tree**, **Graphical Editor** and **Property Sheet** are based on the Visual Editor model. In the diagram the related controller components are written below. The **Source Model** is built from the .java source file. Outside the IDE infrastructure separate Java VM's are running to provide images for the **Graphical Editor**.



[5] For further information see http://download.eclipse.org/tools/emf/scripts/home.php

Since ULC Visual Editor extends Visual Editor the views are implemented by the framework. Mostly controllers had to be extended therefore they are described in more detail below. To create a **Palette** with ULC Widgets only a specification file had to be created. The syntax of this specification file is explained in section 4. 1. 2.

Whenever a widget is selected from the **Palette** its Visual Editor model is created. The class information of the widget is obtained using introspection. This process is executed in the **BeanInfo VM** and only once per widget type. The **VE Model** is extended with the attributes and relationships specified in *BeanInfo* class and in a .override file. This file also connects the various views to the **VE Model** through adapters (see section 4. 1. 3) . The next sections describe the architectural parts of the diagram in more detail.

3. 1. VE Model

Visual Editor consists of a model that keeps track of the Java Beans, their properties and any relationships between them. This model is implemented using the Eclipse Modeling Framework (EMF). The **VE Model** is neutral to implementation details. Adapters are responsible for the serialization of beans into the .java source file and for their visual representation on the **Graphical Editor**. ULC Visual Editor has implemented its own adapter objects to display the model of ULC widgets in various views. These are defined in the override file for that ULC widget.

An override file is an XMI file that specifies Visual Editor specific annotations to the Visual Editor model class (*JavaClass*). This model class is created using introspection and the *BeanInfo* class provided for the bean.

EMF is used to model the structure of the class being edited i.e. its beans and their properties. Every item in the model is an instance of *JavaObjectInstance* and has a *JavaClass* that can provide details such as its attributes, methods and events. This information comes to the **VE Model** used by Eclipse as part of its Java Development Tooling (JDT) and also by introspection of the *BeanInfo* class for the bean. The *JavaObjectInstance* and *JavaClass* are held inside the Eclipse IDE (they don't have a .class file). *JavaObjectInstance* is never specialized but is decorated with adapters which handle all custom behavior for particular bean.

The *JavaClass* contains attributes for each property that comes from introspection. Each of these has details from the *java.beans.PropertyDescriptor* such as the name (used for the **Properties Sheet**), and the set method that is called on the **Target VM** by the *BeanProxyAdapter* when the attribute changes or when it has to first creates the instance on the **Target VM**.

Parent-child or container-contained relationships between beans can also be captured in the model. Examples of such relationships are *ULCTable* and *ULCTableColumns*, or *ULCBoxPane* (a layout container) and *ULCComponents* (ULC widgets laid out in the container). These relationships allow a table to hold its columns or a box pane to hold components. Such relational attributes of beans are not available from *BeanInfo* introspection because it only allows single valued attributes or relationships (such as *title*, *background*, *enabled*, etc.). Those relationships are described in the .override file that extends the **VE Model** of the bean.

3. 2. BeanInfo VM

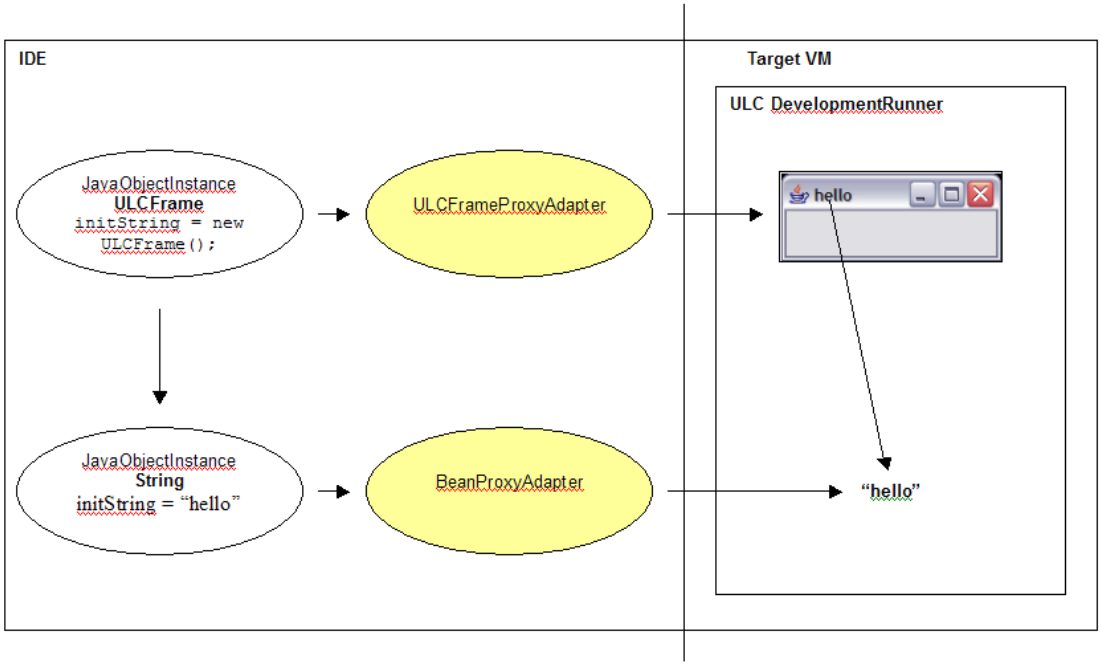
For each ULC Widget, a *BeanInfo* class is defined. This *BeanInfo* class returns information about the attributes, method and events of the Java Bean. Visual Editor creates a separate VM where the Java Beans selected from the **Palette** are introspected to return their attributes, methods and events. For each project using Visual Editor a separate VM is created. The BeanInfo **VM** also reads the .override file and extends the model of the Java Bean with external attributes (such as relationships) and adapters which serve as controllers

3. 3. Target VM

The task of the **Target VM** is to provide an image of the JavaBeans to the **Graphical Editor**. For each .java file opened with the visual editor a **Target VM** is created. It has its class path set to the build path of the Java project of the file. Communication between Visual Editor and the **Target VM** is done through a socket. Each widget selected from the **Palette** or parsed from the source file is instantiated in this application running on the **Target VM**. All property changes are reflected through a *BeanProxyAdapter*. The location of the application in the **Target VM** is set off-screen to hide it from the user. A *java.awt.Graphics* image is captured, sent to Visual Editor through the socket, and recreated as an SWT image. It is displayed on the canvas. For ULC the application is executed using ULC *DevelopmentRunner*, which enables to run the client and server part of ULC in one VM.

3. 4. BeanProxyAdapter

The BeanProxyAdapter mediates between the **VE Model** and the **Target VM**. It is responsible to create instances of ULC widgets in the **Target VM** and to set their properties. It listens to events where necessary, and deals with error conditions. Visual Editor communicates with the **Target VM** using the *IBeanProxy* API that is similar to the *java.lang.reflect* API. This API enables the creation of instances and is responsible for sending methods on the **Target VM** from *BeanProxyAdapter* objects in Visual Editor. The ULC Visual Editor uses a number of custom *BeanProxyAdapters* for ULC widgets to deal with widget specific requirements. Accordingly, these custom *BeanProxyAdapter* subclasses are specified in the ULC widget's override file.



The above diagram illustrates the Visual Editor - **Target VM** interaction. In the IDE an *ULCFrame* is created and the title property is set to **hello**. In the **VE Model** two objects (*JavaObjectInstances*), one for the ULCFrame and for a String representing the title property are created. The *JavaObjectInstance* has access to the initialization string required to instantiate the instances, e.g. **new ULCFrame()** for the *ULCFrame* and **hello** for the *title*. For each *JavaObjectInstance* the *BeanProxyAdapter* defined in the .override file is called which creates the required bean on the **Target VM** using the initialization string.

3. 5. Graphical Editing Framework

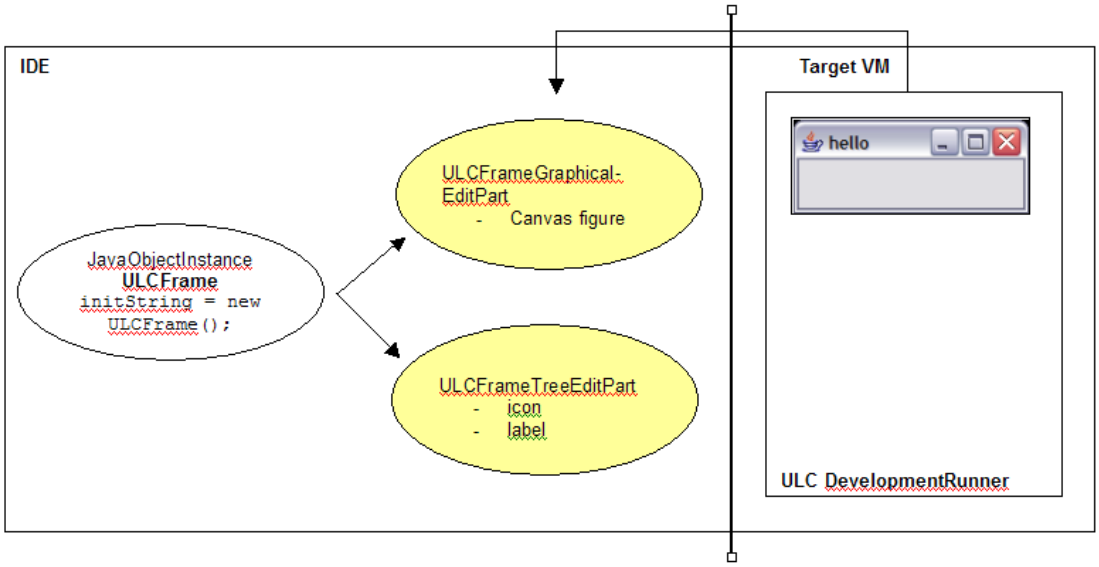
Underlying the **Graphical Editor** and the **Java Beans Tree** is Eclipse's Graphical Editing Framework (GEF)[6]. GEF enables the creation of a graphical editor from an application model. GEF supports the MVC (Model View Controller) Pattern providing the viewer (*EditPartViewer*) and the controller (*EditPart*) parts but does not know anything about the model.

Visual Editor uses GEF to display the beans in the **Graphical Editor** and the **Java Beans Tree** using the two viewer types of GEF (graphical and tree-based). The graphical viewer uses figures to display the view. Figures are defined in the org.eclipse.draw2d (Draw2d) plug-in, a standalone rendering and layout package for an SWT Canvas. Figures in Draw2d are lightweight objects which can be nested to create a complex graphical representation. GEF provides a layout management framework that handles the layout of the set of figures which reflect the model. The Tree Viewer uses an SWT *Tree* and *TreeItems* for its view.

The controllers are called *EditPart* and are responsible both for mapping the model to its view, and for making changes to the model. *EditPart* does not update the model directly but uses commands. Commands can be used to validate the user's interaction and to provide undo and redo support. Commands are defined in a *EditPolicy*. Each *EditPolicy* implementation understands a particular request and produces a command (e.g. "add a column to the table"). An *EditPart* may have more than one *EditPolicy* registered with it to handle different editing actions.

An *EditPart* also contains the list of child model objects if any. This list is used to create the corresponding *EditParts*. The newly created *EditParts* are added to the part's list of children *EditParts*. This in turn adds each child's figure to the **Graphical Editor**.

In Visual Editor *EditParts* are representing beans (ULC widgets for ULC). Every bean which is added to the **Graphical Editor** is added as a child to the edit parts of the **Graphical Editor** and of the **Java Beans Tree**. The default *EditParts* created are *org.eclipse.ve.internal.java.core.JavaBeanTreeEditPart* and *org.eclipse.ve.internal.java.core.JavaBeanGraphicalEditPart*. They inherit from the *DefaultGraphicalEditPart* and *DefaultTreeEditPart,* which are part of the Graphical Editing Framework (GEF).



The above diagram shows the relationship between a *JavaObjectInstance* (representing a bean in the **VE Model** and the edit parts. The use of GEF in Visual Editor is summarized in the following paragraphs.

[6] see http://www.eclipse.org/gef/

The *DefaultGraphicalEditPart* is responsible for

- obtaining the image of the ULC Widget from the **Target VM**
- displaying the figure on the **Graphical Editor**.
- listening to model changes and refreshing the figure accordingly
- translating user interactions on the edit part to the model using commands.

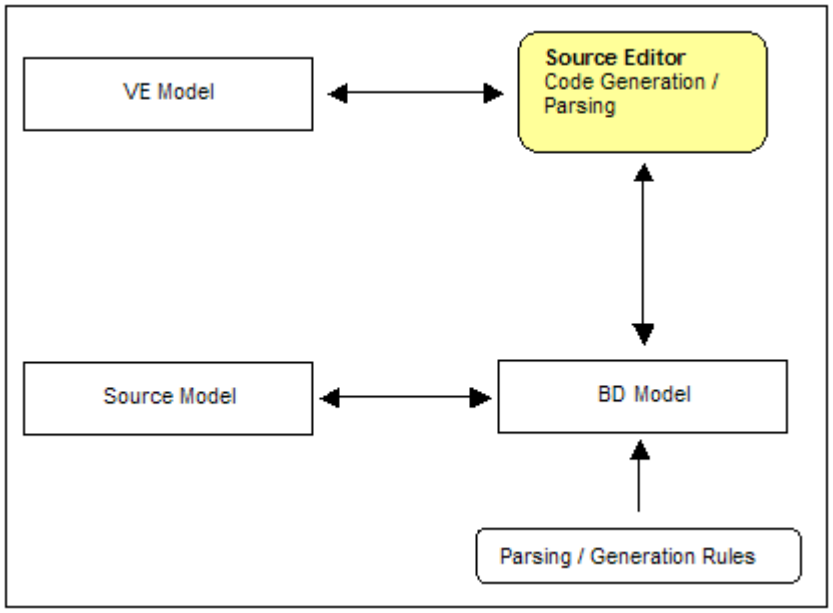
The *DefaultTreeEditPart* is responsible for the display of the edit part with a label and an icon.

The mapping of the GEF and the **VE Model** happens in the .overrides files for the beans. If a bean has no own *EditPart* specified, inheritance is used to get the *EditPart* from the superclasses (see also section 4. 1. 3)

ULC Visual Editor implements a special *ULCComponentGraphicalEditPart* for *ULCComponent* Java Bean that has a visual representation on the **Target VM** that allows the GEF figure to show this on the **Graphical Editor**. This *EditPart* talks to the *BeanProxyAdapter* which implements the interface *IVisualComponent* which returns the bounds of the figure and also has a listener interface so the *EditPart* knows when the figure is moved or resized and needs update. The *IVisualComponent* interface extends *IImageNotifier* which is responsible for returning the SWT image data of the raw image and also notifying when it changes and when the figure requires updating.

3. 6. Code Generation and Parsing

The **VE Model** is independent of the source code, which has its own model (**Source Model**). The code generation component of Visual Editor adapts the **VE Model** to the **Source Model** of the **Source Editor**. The **VE Model** is designed to be independent to the type of serialization means. In order to enable synchronization between the Visual and Java Source editors, another model is introduced: *org.eclipse.ve.internal.java.codegen.model.IBeanDeclModel*, also referred to as the **BD Model**. The **BD Model** holds specific source information regarding EMF instances and their attributes. It maps an instance and its attribute to a given class, method, and expressions.



The **BD Model** also associates each expression in the source to a specific decoder. A decoder of the type *org.eclipse.ve.internal.java.codegen.java.IExpressionDecoder* is responsible to decode a

single source expression and to update the **VE Model**. It is also responsible for the generation of source code for a given attribute or for an instance definition in the source code. A decoder is associated with a particular widget and is specified in its override file.

The Code Generation/Parsing engine (org.eclipse.ve.codegen plug-in of Visual Editor) also consults a set of rules to determine elements such as code patterns (names, style, placement, scope, etc.) and filter policies for code portions.

4. Implementation

The following sections describe the implementation of ULC Visual Editor. As stated earlier, ULC Visual Editor plug-in extends Eclipse Visual Editor for Java Plug-in. ULC Visual Editor implementation can be explained in two parts: specifications and classes. The specifications define various extensions made to Visual Editor and the ULC Widgets. The classes extend Visual Editor framework classes to customize their behavior for ULC Visual Editor.

4. 1. Specifications

ULC Visual Editor is implemented through the following specification files:

- plugin.xml: defines extensions points and dependencies.
- palette/ulcbeanscats.xmi: defines the **Palette**.
- .override: xmi files which define extensions to beans such as attributes, relations, and controllers, which extend the framework and hook into it.

The following section depicts the specification files in more detail.

4. 1. 1. Plugin.xml file

ULC Visual Editor Plug-in includes the following plug-ins:

- **com.canoo.ulc.eclipse.ulccontainer** – ULC Visual Editor uses this plug-in to define a classpath container (*ULC_Container*). It contains all ULC libraries and license files. By adding the **ULC Libraries** to the classpath of the project this container is installed. This plug-in is also part of the ULC Eclipse IDE Integration plug-in.[7]
- **com.canoo.ulc.visualeditor** – ULC Visual Editor plug-in.
- **com.canoo.ulc.visualeditor.doc** – ULC Visual Editor help and online documentation

ULC Visual Editor Plugin depends on the following plug-ins:

- org.eclipse.ve – Eclipse Visual Editor version 1.0.1
- org.eclipse.emf – Eclipse Modeling Framework version 2.0.1
- org.eclipse.gef – Graphical Editing Framework version 3.0.1

4. 1. 1. 1. Extension Points

The following extension points are implemented in ULC Visual Editor.

- org.eclipse.ve.java.core.contributors
- org.eclipse.jem.proxy.contributors
- org.eclipse.jem.beaninfo.registrations
- org.eclipse.ve.java.core.newStyleComponent
- org.eclipse.ui.popupMenus

Each extension point is described in detail in the next sections.

org.eclipse.ve.java.core.contributors

```
<extension point="org.eclipse.ve.java.core.contributors">
    <contributor
```

[7] http://ulc-community.canoo.com/snipsnap/space/Contributions/Integration+Snippets/Eclipse+IDE+Integration

```
            class = "com.canoo.ulc.visualeditor.
                ULCVisualEditorContributor"
            container="ULC_CONTAINER">
        </contributor>
        <palette
            container="ULC_CONTAINER"
            loc="first"
            categories="palette/ulcbeanscats.xmi"
            plugin="com.canoo.ulc.visualeditor">
        </palette>
    </extension>
```

This extension point provides configuration contributors for the Visual Editor. A contributor can be supplied for a specific class path container or for a plug-in. The *class* attribute has to be fully qualified. It needs to implement *org.eclipse.ve.internal.java.codegen.editorpart.IVEContributor*. ULC Visual Editor provides the class *ULCVisualEditorContributor* for its class path container *ULC_CONTAINER*.

The implementation of *ULCVisualEditorContributor.contributePalleteCats()* removes the Swing and AWT categories from the **Palette** as Swing and AWT beans should not be used in the ULC context.

In addition, this extension point specifies a simple palette category addition for *ULC_CONTAINER*. The *loc* attribute is the location of the categories in the **Palette** (first, last etc.) and the *categories* attribute is an .xmi file containing the description of the ULC Visual Editor **Palette**. The ulcbeanscats.xmi file is described later (see section 4. 1. 2).

org.eclipse.jem.proxy.contributors

```
    <extension point="org.eclipse.jem.proxy.contributors">
        <contributor
            container="ULC_CONTAINER"
            class ="com.canoo.ulc.visualeditor.
                ULCVisualEditorContributor">
        </contributor>
    </extension>
```

This extension point provides configuration contributors for proxy support. A contributor can be supplied for a specific class path container or for a plugin. ULC Visual Editor provides a contributor class *ULCVisualEditorContributor* that implements *org.eclipse.jem.internal.proxy.core.IConfigurationContributor* for its class path container *ULC_CONTAINER*. When a **Target VM** for a .java source file is started with *ULC_CONTAINER* in its class path, *ULCVisualEditorContributor* is asked to contribute.

IConfigurationContributor.contributeClasspaths(): adds **`ulcvm.jar`** to the class path of the **Target VM**.

IConfigurationContributor.contributeToRegistry(): initializes the environment for the **Target VM**. It starts a ULC dummy application using ULC *DevelopmentRunner* on the **Target VM**. The registry represents a handle to the **Target VM**.

org.eclipse.jem.beaninfo.registrations

```
    <extension  point="org.eclipse.jem.beaninfo.registrations">
        <registration container="ULC_CONTAINER">
            <beaninfo path="/com.canoo.ulc.visualeditor/
                ulcbeaninfo.jar">
                <searchpath
                    package="com.canoo.ulc.visualeditor.beaninfo"/>
```

```
            </beaninfo>
            <override package="com.canoo.ulc.visualeditor.extension"
                 path="overrides/com/canoo/ulc/visualeditor/
                 extension"/>
            <override package="com.ulcjava.base.application"
                 path="overrides/com/ulcjava/base/application"/>
        </registration>
    </extension>
```

This extension point is used to register contributors, *BeanInfo*, and overrides for *BeanInfo* contributions. These registrations are identified with the class path container *ULC_CONTAINER*. The *BeanInfo* registrations supply standard *BeanInfo* and overrides.

The *path* attribute provides path to the **`ulcbeaninfo.jar`** file and *searchpath* gives the package within the **`ulcbeaninfo.jar`** to look for the *BeanInfo* classes. This information is added to the search path for *BeanInfo*.

The *override* tag describes the path of the override files for each package. When the package name of the class being introspected matches the specified package name, the override file for the class will be found in the specified path relative to the plug-in.

org.eclipse.ve.java.core.newStyleComponent

```
    <extension point="org.eclipse.ve.java.core.newStyleComponent">
        <category name="ULC" id="com.canoo.ulc.visualeditor.cat"
            priority="300" defaultExpand="false">
        </category>
        <visualElement type="com.ulcjava.base.application.ULCFrame"
            icon="icons/clcl16/frame_obj.gif"
            category="com.canoo.ulc.visualeditor.cat"
            contributor="com.canoo.ulc.visualeditor.
                ULCVESourceContributor" name="ULCFrame">
        </visualElement>
```

This extension point is used to register new visual class styles and component extensions. New styles and components appear as choices within the **Style** tree view of the **Visual Class Creation Wizard**, and are typically used to create new classes for the Visual Editor. In ULC Visual Editor new style classes are provided for all ULC containers such as *ULCFrame*, *ULCDialog*, *ULCBoxPane*, etc.

The ULC container classes are grouped under the specified category defined in the *category* tag. Each container class is defined in a *visualElement* tag. The *type*, *name* and *icon* attributes specify the type of the widget, its name, and its corresponding icon to be displayed in the **Style** tree view. The *contributor* attribute defines the class that implements *org.eclipse.ve.internal.java.codegen.wizards.IVisualClassCreationSourceContributor*.

ULCVESourceContributor.getTemplateLocation() returns the location of the Java Emitter Template (JET) template file that contains the code template for that ULC class.

org.eclipse.ui.popupMenus

```
    <extension point="org.eclipse.ui.popupMenus">
        <objectContribution
            objectClass="org.eclipse.ve.internal.java.core.
                IJavaBeanContextMenuContributor"
            id="org.eclipse.ve.internal.jfc.core.editorpart.
            settextaction.popup.object">
            <filter name="BEANTYPE"
                value="com.ulcjava.base.application.ULCComponent"/>
```

```
        <filter name="PROPERTY" value="text"/>
        <action label="%PopupMenus.SetTextAction.Label Set Text"
            class="org.eclipse.ve.internal.jfc.core.
                SetTextObjectActionDelegate"
            id="org.eclipse.ve.internal.jfc.core.settextaction"/>
    </objectContribution>
```

This extension point is used to add actions to context menus. ULC Visual Editor defines a **Set Text** context menu entry in the *objectContribution* tag. The *action* tag is responsible to provide an *id*, a *label* for the menu entry and the *class* that implements the interface *org.eclipse.ui.IObjectActionDelegate* for object contributions. The *filter* tags restrict this action to objects of the type *ULCComponent* which have a property named **`text`**. For other label properties (e.g. **`title, label`**) similar *objectContribution* tags are defined.

4. 1. 2. Palette XMl file

The **Palette** contains all editable ULC widgets. ULC Visual Editor **Palette** definition is contained in the file palette/ulcbeanscats.xmi, which is specified as part of the extension point org.eclipse.ve.java.core.contributors. The file ulcbeanscats.xmi file defines the following three categories to be added to the visual editor **Palette**.

- ULC Containers (all container classes e.g. *ULCFrame, ULCBoxPane, ULCTabbedPane* etc).
- ULC Components (all simple beans e.g. *ULCButton, ULCLabel, ULCTable, ULCTableColumn, ULCTree, ULCProgressBar*)
- ULC Menus (*ULCMenuBar, ULCMenu, ULCMenuItem*)

The following is a sample specification from the *ulcbeanscats.xmi*:

```
<palette:CategoryCmp xmi:id="ULCContainers">
    <categoryLabel xsi:type="utility:ConstantString" string="ULC
        Containers"/>
    <cmpGroups xsi:type="palette:GroupCmp">
    <cmpEntries xsi:type="palette:AnnotatedCreationEntry"
        icon16Name="platform:/plugin/org.eclipse.ve.jfc/icons/
            full/clcl16/frame_obj.gif">
        <objectCreationEntry
            xsi:type="palette:EMFCreationToolEntry"
            creationClassURI="java:/com.ulcjava.base.application#
            ULCFrame" />
        <values xmi:type="ecore:EStringToStringMapEntry"
            key="org.eclipse.ve.internal.cde.core.
                nameincomposition"
            value="ulcFrame" />
        <entryLabel xsi:type="utility:ConstantString"
            string="ULCFrame" />
    </cmpEntries>
```

The specification starts with the definition of the category *ULCContainers*. The tag *categoryLabel* defines the string displayed in the **Palette**. The following tag *cmpGroups* defines the category as a group. Each component entry is defined in a tag *cmpEntries* which contains the following elements:

- *icon16Name*: The icon to be displayed in the **Palette** for the component
- *objectCreationEntry*: The class to be instantiated when this component is chosen.
- The *nameincomposition* key specified in *values*; it defines the name of the instance variable to be generated for the bean in the source code.
- *entryLabel*: The label to be displayed on the palette for the component.

To add a new widget to the **Palette** a new tag *cmpEntries* has to be specified at the location where the new widget should show on the **Palette**.

4. 1. 3. .override XMI Files

An .override file is defined for each ULC widget. On the one hand it extends the **VE Model** with attributes that are not available from introspection e.g. container-contained relationships. These attributes are specified as structural feature. On the other hand the .override file annotates the **VE Model** with adapters for various parts of the visual editor such as **Graphical Editor**, **Java Beans Tree**, **Source Editor, Target VM**, and **Property Sheet**. These adapters are specified as annotations. The following sections describe relevant annotations and structural features.

4. 1. 3. 1. Annotation for Graphical Editing Framework

For GEF two classes which serve as controllers need to be specified, one for the **Graphical Editor** and the second for the **Java Beans Tree**. Below we give an example for *ULCComponent*. The attribute *treeViewClassnam*e specifies the class *ULCComponentTreeEditPar*t that adapts the **VE Model** for the **Java Beans Tree**. The attribute *graphViewClassname* specifies the class *ULCComponentGraphicalEditPart* that adapts the **VE Model** for the **Graphical Editor**. This specification holds for each ULC widget that extends *ULCComponent* unless it is explicitly specified in the override file for that widget.

```
<addedEObjects
     xsi:type="org.eclipse.ve.internal.cde.decorators:
          ClassDescriptorDecorator"
     treeViewClassname="com.canoo.ulc.visualeditor/com.canoo.ulc.
          visualeditor.gef.tree.ULCComponentTreeEditPart"
     graphViewClassname="com.canoo.ulc.visualeditor/com.canoo.
          ulc.visualeditor.gef.graphical.
          ULCComponentGraphicalEditPart"
/>
```

The icon to be displayed for the ULC widget in the **Java Beans Tree** is specified as *graphic* tag (e.g. *ULCComboBox*):

```
<graphic
     xsi:type="org.eclipse.ve.internal.cde.utility:GIFFileGraphic"
     resourceName="platform:/plugin/org.eclipse.ve.jfc/icons/
          full/clcl16/combobox_obj.gif"
/>
```

To set a value for a property at creation time a creation policy has been defined. In ULC Visual Editor the class *SetPropertyCreationPolicy* is responsible to set a *String* value to a property. Property and value are defined as key value pairs. In the following example *SetPropertyCreationPolicy* will create a command to set the **`title`** property of *ULCFrame* to the string **`ULCFrame`**:

```
<keyedValues xsi:type="ecore:EStringToStringMapEntry"
     key="org.eclipse.ve.internal.cde.core.creationtool.policy"
     value="com.canoo.ulc.visualeditor/
          com.canoo.ulc.visualeditor.propertysheet.
          SetPropertyCreationPolicy:title=ULCFrame"
/>
```

4. 1. 3. 2. Annotation for Target VM

The **VE Model** is adapted to the **Target VM** through the *BeanProxyAdapter* object. It enables the creation of beans on the **Target VM** and the setting properties on those instances. The default behavior for *BeanProxyAdapter* calls the set method from the *java.beans.PropertyDescriptor* on the **Target VM**.whenever a property is set using Visual Editor. Any special rules over and above this, need to be coded in a custom *BeanProxyAdapter*.

Custom *BeanProxyAdapter* subclasses are created and described against a *JavaClass* (the meta object of a *JavaObjectInstance*) of a bean. It is specified as follows in the .override file:

```
<addedEObjects xsi:type="org.eclipse.ve.internal.jcm:BeanDecorator"
      beanLocation="GLOBAL_GLOBAL"
     BeanProxyClassName="com.canoo.ulc.visualeditor/com.canoo.ulc.
          visualeditor.proxy.ULCComponentProxyAdapter"
/>
```

ULCComponentProxyAdapter is specified as the *BeanProxyAdapter* for *ULCComponent*. This class acts as *BeanProxyAdapter* for each ULC widget that extends *ULCComponent* unless it is explicitly spedified in the widget's .override file.

The *beanLocation* attribute determines the scope of the instance variable generated for the bean in the source code of the visual class in which the bean is included. There are several options:

- *GLOBAL_GLOBAL* means that the bean corresponds to an instance variable and has its own get method that initializes and returns it.
- *GLOBAL_LOCAL* means that the bean corresponds to an instance variable but it does not have its own get method, instead it is initialized in the method that initializes its container.
- *LOCAL* means that the bean is local to the method that uses it.

4. 1. 3. 3. Annotation for Code Generator and Code Decoder

The **VE Model** is adapted to the Java Source Editor's model through *CodeGenHelperClass*. The BDM model also associates each expression in the source to a specific expression decoder. The following is an example specification for *ULCWindow*:

```
<addedEObjects xsi:type="codeGenHelpers:CodeGenHelperClass"
     source="codegen.CodeGenHelperClass"
     expDecoder="com.canoo.ulc.visualeditor/
          com.canoo.ulc.visualeditor.codegen.ULCWindowDecoder"
     modelled="true"
/>
```

The attribute *expDecoder* is set to the class *ULCWindowDecoder*. It is an expression decoder for the code generation and for building the source model for certain methods. It inherits from *org.eclipse.ve.internal.java.codegen.java.ObjectDecoder*. *ULCWindowDecoder* initializes a special decoder helper which is responsible for generating and decoding code concerning the container-contained relationship (e.g. *ULCWindow.add(String, ULCComponent)*). Code generation and decoding of JavaBeans properties are handled by the *ObjectDecoder* and by *org.eclipse.ve.internal.java.codegen.java.SimpleAttributeDecoderHelper*. Each ULC container has a custom expression decoder.

The attribute *modelled* is set to *true* in a class where an instance of the class has to be in the **VE Model**. This means that when any component or subclass of the component is dropped on the canvas it will be added to the model.

4. 1. 3. 4. Structural Features

When a bean's **VE Model** is created, i.e., when a *JavaClass* is instantiated to create a *JavaObjectInstance*, the properties of the bean appear as Structural Features of the *JavaObjectInstance*. These properties are obtained by introspecting the bean. The values of these properties are stored as settings in the *JavaObjectInstance*.

Properties which are not available from introspection of the bean, i.e. the ones which do not follow the JavaBeans specification[8], need to be specified in the .override file. Consequently, they are added as structural features. Such properties include relationships between objects (e.g. container-contained relations) and the attributes acquired by an object as result of a relationship.

[8] http://java.sun.com/products/javabeans/docs/spec.html

For example there exists a container-contained relationship between an *ULCBoxPane* and its *ULCComponents*. Because ULC Visual Editor needs to know about these children, the structural feature *components* had to be introduced in the *ULCBoxPane.override*:

```
<event:Add  featureName="eStructuralFeatures">
     <addedEObjects xsi:type="ecore:EReference" name="components"
          eType="ecore:Eclass java:/com.ulcjava.base.application#
               ULCComponent"
          upperBound="-1"
          changeable="true"
          unsettable="false"
          containment="false">
          <eAnnotations
               xsi:type="org.eclipse.ve.internal.cde.decorators:
                    PropertyDescriptorDecorator"
               hidden="true"/>
          <eAnnotations
               xsi:type="org.eclipse.ve.internal.jcm:
                    BeanFeatureDecorator"
               linkType="CHILD"/>
     </addedEObjects>
</event:Add>
```

The type of the value of this structural feature is *ULCComponent* (defined in the attribute *etype*). It is a one to many relationship as specified by *upperBound=-1*. The attribute *unsettable* means that it is valid to set the value to NULL and have it still marked as set. The attribute *changeable* defines if this property can be set, i.e., it is not a read-only property.

The attribute *containment* needs to be set to *false*. This is necessary to avoid that the container owns the children in the **VE Model**. The components are instance variables, so they will actually be children of the composition itself and not of the container. The container-contained relationship is expressed by the *BeanFeatureDecorator* (see below).

The structural feature *components* is annotated with a *PropertyDescriptorDecorator*. This is used to add custom behavior for the appearance in the **Property Sheet** such as a cell editor for a layout property, or a specific label provider. In the above example we set *hidden=true* to indicate that the *components* property of *ULCBoxPane* should not be displayed in the **Property Sheet**.

The *components* structural feature is further annotated with *BeanFeatureDecorator* with *linkType="Child"* indicating that this feature is a parent/child relationship versus just a property setting. This solution is typically used if the child appears as an edit part in the visuals.

With this structural feature added, the references to *JavaObjectInstance* objects representing the child (*ULCComponent*) are stored in the **VE Model** against the *JavaObjectInstance* representing the parent (*ULCBoxPane*). This is used by the **Java Beans Tree,** by the **Graphical Editor** and for code generation and parsing.

Another example for the need of a structural feature is shown in *ULCComponent.override*. If an *ULCComponent* is contained in container, it needs to store the containment information like alignment, position etc. in a special attribute. This special attribute is defined as a structural feature with the name *containment*:

```
<event:Add  featureName="eStructuralFeatures">
     <addedEObjects xsi:type="ecore:EReference" name="containment"
          eType="ecore:EClass java:/java.lang#Object"
          upperBound="1"
          changeable="true"
          unsettable="false"
          containment="true">
    </addedEObjects>
</event:Add>
```

The above specification extends the **VE Model** for *ULCComponent* with a new property *containment* whose value can be of type *Object*. ULC Visual Editor either uses *String* values for con-

tainment parameters within a *ULCWindow* or it uses a special object of types defined in *com.canoo.ulc.visualeditor.extension* package. These are *BoxPaneContainment*, *CardPaneContainment* and *TabbedPaneContainment* classes which define the containment parameters for respective containers. *ULCComponentPropertySourceAdapter.includeFeature()* is decides that the structural feature *containment* is only displayed in the **Property Sheet** if the widget is contained in a container. It also implements which property descriptor should be used to display the different containment types.

4. 1. 3. 5. Annotation for Property Sheet

The **VE Model** is annotated with org.eclipse.ve.internal.cde.properties.PropertySourceAdapter. A *PropertySourceAdapter* is responsible for the following

- It defines which features to display on the property sheet.
- It specifies special property descriptors (e.g. for structural features which are defined in the .override file and which are not JavaBeans properties).
- It can get the value of a structural feature.
- It contains all structural properties of a *JavaClass.*

The following is the specification for *ULCComponent*:

```
<addedEObjects
     xsi:type="org.eclipse.ve.internal.cde.decorators:
          PropertySourceAdapterInformation"
     propertySourceAdapterClassname="com.canoo.ulc.visualeditor/
          com.canoo.ulc.visualeditor.propertysheet.
          ULCComponentPropertySourceAdapter"
/>
```

The *propertySourceAdapterClassname* specifies *ULCComponentPropertySourceAdapter* as the adapter for **Property Sheet**.

4. 2. Classes

To implement the functionality ULC Visual Editor extends classes from Visual Editor for Java and other frameworks:

- JavaBeans: For defining *BeanInfo* classes for ULC Widgets
- GEF: For interaction with the **Graphical Editor** and **Java Bean Tree.**
- Visual Editor:
    - *BeanProxyAdapter*: For interaction with the Target VM
    - Decoders for code generation: For interaction with the source code
    - *PropertySourceAdapter*: For interaction with the property sheet

Most of these classes have been mentioned in the previous sections as they are specified in the override files. The following sections describe each class hierarchy.

4. 2. 1. JavaBeans BeanInfo

A *BeanInfo* class is provided for each editable ULC Widget. The name of a *BeanInfo* class take the form of a widget name suffixed with *BeanInfo*, for e.g., *ULCComponentBeanInfo*, *ULCButtonBeanInfo*. Each *BeanInfo* class implements *org.eclipse.ve.internal.jfc.beaninfo.IvjBeanInfo*. There are two important methods:

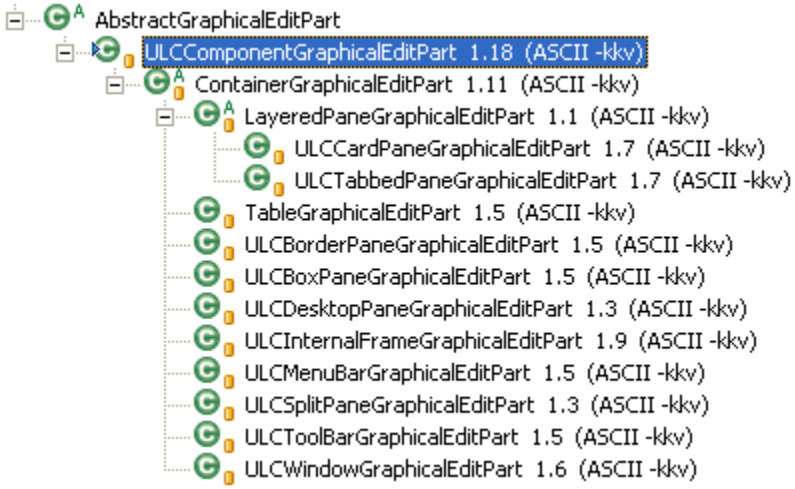
- getPropertyDescriptors(): returns property descriptors for each attribute of the bean.
- getEventSetDescriptors(): returns event descriptors for the bean if it supports events.

4. 2. 2. Graphical Editing Framework

Each top level bean is added as a child to the **Graphical Editor**. *GraphicalEditPart* adapts the **VE Model** to the **Graphical Editor**. *GraphicalEditPart* creates the GEF figure. It acts as a controller between the image and the model and is responsible for keeping them in synch. User's interaction with the figure of the bean is turned into a *Request* object by GEF. *GraphicalEditPart* processes the *Request* object to create commands using an appropriate *EditPolicy*. These commands are executed to make changes to the **VE Model**. *EditPolicy* determines the behavior of the *EditPart* in response to an editing operation by the user. An *EditPart* can install more than one *EditPolicies*, each being responsible for the creation of commands as per the user's gestures (requests) from **Palette** (drop, create) and for move and remove. An *EditPart* also handles Direct Edits where the user types text in an editable control.

4. 2. 2. 1. GraphicalEditPart

In ULC Visual Editor, *ULCComponentGraphicalEditPart* is the root class of the ULC GraphicalEditParts hierarchy. It extends *org.eclipse.gef.editparts.AbstractGraphicalEditPart*. ULC containers (e.g. *ULCFrame*, *ULCBoxPane*, etc.) implement their own *GraphicalEditPart* which inherit from *ULCComponentGraphicalEditPart* as ULC containers inherit from *ULCComponent*. The following picture shows the complete hierarchy:



Important methods of *ULCComponentGraphicalEditPart* are:

- *createFigure()*: Provides the figure for the EditPart and sets its layout manager to *org.eclipse.draw2d.XYLayout*. The layout manager handles the layout of the child figures of Draw2d *IFigure* instances.
- *createEditPolicies()*: Associates *EditPolicy* implementations with specific editing roles of the *EditPart*. An *EditPolicy* processes requests to create commands. *ULCComponent* installs the followig edit policies:
  - *DefaultComponentEditPolicy* for COMPONENT_ROLE.
  - *ComponentDirectEditPolicy* for DIRECT_EDIT_ROLE.
  - *ULCFreeFormEditPolicy* that ensure that a *FreeFormProxyAdapter* to the Target VM has been installed.
- *getAdapter()*: Returns the following model adapters
  - *IPropertySource:* for the **Property Sheet**
  - *IConstraintHandler:* handles resizing/move
  - *IActionFilter:* for direct editing
  - *IVisualComponent*: for the visual component.
- *activate()*: Adds *EditPart* as listener to the model

- *deactivate()*: Removes *EditPart* as listener from the model

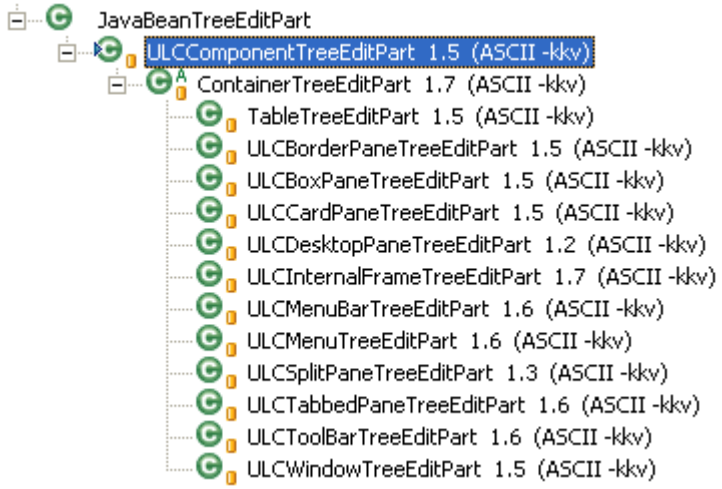
The class *ContainerGraphicalEditPart* represents ULC containers and hence handles the container-contained relationship. Each ULC container has its own *GraphicalEditPart* class. Important methods of *ContainerGraphicalEditPart* are:

- *createChild()*: Creates an *EditPart* corresponding to the model child (contained component).
- *createContainerPolicy()*: Returns an *EditPolicy* to give effect to the container-contained relationship at the model level. This method is abstract and implemented by each ULC container.
- *createLayoutEditPolicy()*: Installs *ContainerLayoutEditPolicy* for LAYOUT_ROLE to handle layout of figures of children in container's figure. (See also section 4. 2. 2. 4, LayoutEditPolicy)
- *getChildrenSfNames()*: This abstract method returns the structural feature name that represents the container-contained relationship.
- *getModelChildren()*: Returns the model children (contained components).
- *activate()/deactivate()*: Adds a listener for changes to the *EditParts* of the children in addition to the listeners added/removed by the parent class.

4. 2. 2. 2. TreeEditPart

A *TreeEditPart* adapts the **VE Model** to display an icon and a label for in the **Java Beans Tree**. It shows the container-contained hierarchy for the model in the **Java Beans Tree** by traversing the child edit parts. It also contains an *EditPolicy*, which creates commands for creating, adding, moving and removing components.

In ULC Visual Editor, *ULCComponentTreeEditPart* is the root class of the TreeEditParts hierarchy. It extends *org.eclipse.ve.internal.java.core.JavaBeanTreeEditPart*. ULC containers (e.g. *ULCFrame*, *ULCBoxPane*, etc.) implement their own *TreeEditPart* which inherit from *ULCComponentTreeEditPart* as ULC containers inherit from *ULCComponent*. The following picture shows the complete hierarchy:



Important methods of *ULCComponentTreeEditPart* are:

- *createEditPolicies()*: Installs an *EditPolicy* to handle Direct Edit and the edit policies used by *JavaBeanTreeEditPart*.
- *getAdapter()*: Returns model adapters for **Property Sheet** (*IPropertySource*).

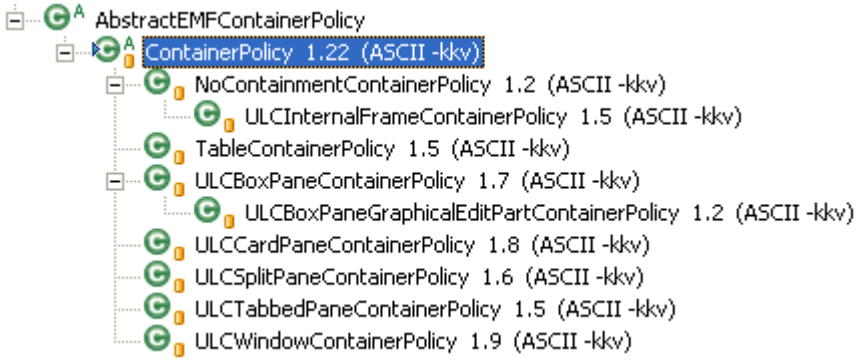
The class *ContainerTreeEditPart* represents ULC containers and hence handles container-contained relationship. Each ULC container has its own *TreeEditPart* class. Important methods of *ContainerTreeEditPart* are:

- *createChildEditPart()*: Creates an *EditPart* corresponding to the model child (contained component)

createContainerPolicy(): Returns an *EditPolicy* to give effect to container-contained relationship at the model level. This method is abstract and implement by each ULC container.

- *getChildrenSfNames()*: This abstract method returns the structural feature name that represents the container-contained relationship.
- *getChildJavaBeans()*: Returns the model children (contained components).
- *activate()/deactivate()*: Adds a listener for changes to the *EditParts* of the children in addition to the listeners added/removed by the parent class.

4. 2. 2. 3. EditPolicy



ULC Visual Editor uses *ContainerPolicy* class to process editing actions pertaining to container-contained relationships such as add, delete and move contained components to or from containers. *ContainerPolicy* extends *org.eclipse.ve.internal.cde.emf.AbstractEMFContainerPolicy*. This policy is installed by the *EditPart* for each ULC container. It generates commands to modify the **VE Model** to add, move and delete model children.

ULC Visual Editor uses a *CommandBuilder* class that extends *org.eclipse.ve.internal.java.rules.RuledCommandBuilder* to build commands to set/unset values on structural features of the **VE Model**. For instance, the *components* structural feature of a ULC container is set with the *JavaObjectInstance* of the ULC Component which is added to a ULC container like *ULCFrame*.

Important methods of *ContainerPolicy* are:

- *createContainmentCommand()*: This abstract method is implemented by *ContainerPolicy* specific to each ULC container. It is used to generate the appropriate command. Consider a scenario where a *ULCButton* is added to a *ULCFrame*. *ULCWindowContainerPolicy.createContainmentCommand()* will create a command which:
  - Sets the *components* structural feature of *ULCFrame* to the *JavaObjectInstance* of *ULCButton*.
  - Sets the *containment* structural feature of *ULCButton* to a string containing the alignment information.

  Consider another scenario where a *ULCButton* is added to a *ULCBoxPane*. *ULCBoxPaneContainerPolicy.createContainmentCommand()* will create a command to:

  - Sets the *components* structural feature of *ULCFrame* to the *JavaObjectInstance* of *ULCButton*.

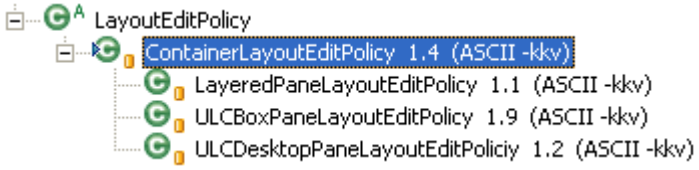
- Sets *containment* structural feature of *ULCButton* to a *JavaObjectInstance* of *BoxPaneContainment* class. This class contains *ULCBoxPane* specific containment parameters like alignment, position, span etc.

- *getAddCommand()*: creates a command when a child from **Graphical Editor** or **Java Beans Tree** is added to a container.
- *getCreateCommand()*: creates a command when a child is added from the palette to a container.
- *getDeleteDependentCommand()*: creates a command when a child is removed from a container.
- *getMoveChildrenCommand()*: creates a command when a child is moved to another position in the same container.
- *getOrphanChildrenCommand()*: creates a command when a child becomes an orphan, i. e. when it is dragged out of the container.
- *getSfFor()*:This abstract method is implemented by *ContainerPolicy* specific to each ULC container to return their structural features that represent container-contained relationship.

4. 2. 2. 4. LayoutEditPolicy

LayoutEditPolicy supports interaction of *GraphicalEditParts* with the *org.eclipse.draw2d.LayoutManager* of the container's figure. *LayoutEditPolicy* is responsible for moving, resizing, reparenting and creating children. It provides commands for all of these operations. It also creates visual feedback (e.g. grid) on ULC containers like *ULCBoxPane* to facilitate the desired placement of components within the container.

ContainerLayoutEditPolicy is the root class of the *LayoutEditPolicy* hierarchy. It extends *org.eclipse.gef.editpolicies.LayoutEditPolicy*. The following picture shows the complete hierarchy:



Important methods of *ContainerLayoutEditPolicy* are

- *activate()/deactivate()*
- *getAddCommand()*
- *getCreateCommand()*
- *getDeleteDependantCommand()*
- *getMoveChildrenCommand()*
- *getOrphanChildrenCommand().*

These are functionally similar to corresponding methods in *ContainerPolicy*.

LayeredPaneLayoutPolicy is used by the *LayeredPaneGraphicalEditPart* which is used for *ULCTabbedPane* and *ULCCardPane*.

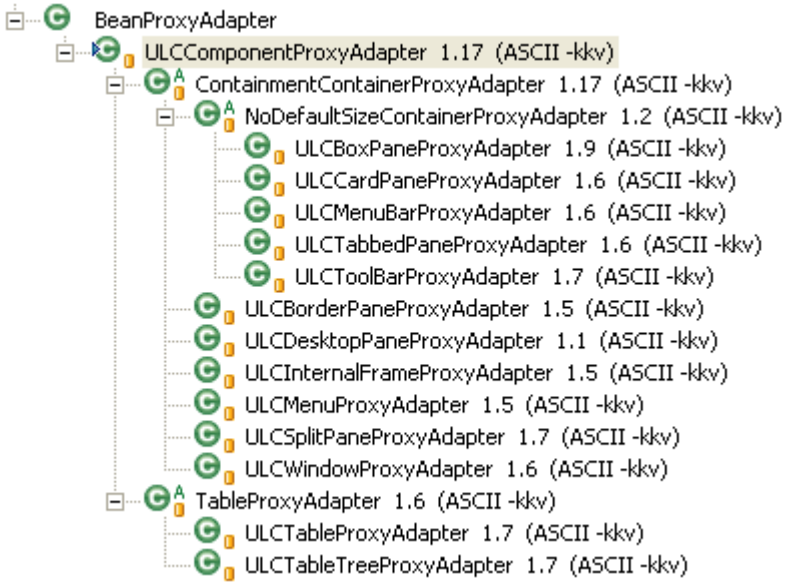
ULCBoxPaneLayoutPolicy is used by *ULCBoxPaneGraphicalEditPart*. It implements *showLayoutTargetFeedback(Request)* method to display the feedback grid in order to facilitate the placement of components.

ULCDesktopPaneLayoutEditPolicy is used by *ULCDesktopPaneGraphicalEditPart* to handle moving and resizing of contained *ULCInternalFrames*.

ULCFlowLayoutPolicy extends *org.eclipse.ve.internal.cde.core.FlowLayoutPolicy*. It is used by *ULCToolBarGraphicalEditPart* and *ULCMenuBarGraphicalEditPart*.

4. 2. 3. BeanProxyAdapter

A *BeanProxyAdapter* mediates between the objects of the **VE Model** and the **Target VM**. It creates the beans in the **Target VM** and sets the property accordingly. For *ULCComponent* and most of the containers a special *BeanProxyAdapter* had to be implemented. To be able to get the images of the instantiated bean the *visible* property should never be false in the **Target VM**. The *BeanProxyAdapters* for containers are rooted in *ContainmentContainerProxyAdapter* and handle the addition and removal of children (contained components) in the **Target VM**. *TableProxyAdapter* is used for handling columns of *ULCTable* and *ULCTableTree*. The following picture shows the complete hierarchy:



ULCComponentProxyAdapter extends *org.eclipse.ve.internal.java.core.BeanProxyAdapter*. It also implements *IVisualComponent*. A visual component is an interface to visual components of the *GraphicalEditPart*. It creates a notification if the figure, its size or its position has changed. *ULCComponentProxyAdapter* needs to have knowledge of

- how to deal with images for the GEF edit parts to be able to display them
- how to listen to them to be able to notify when the figure needs moving/resizing.

ULCComponentProxyAdapter also implements *IComponentProxyHost.* This interface completes the *IBeanProxyHost* interface with component spedific methods.

Important methods of *ULCComponentProxyAdapter* are:

- *applied()/canceled()*: This method is called whenever a structural feature changes in the VE model. It sets/unsets those properties on the bean instantiated in the Target VM that have been set/unset in the VE model.
- *add/removeComponentListener()*: Creates a *ComponentManager.* This is the callback listener for *org.eclipse.ve.internal.jfc.vm.ComponentListener* that is running in the Target VM. It adds/removes the visual component listener which listens when the component moves, or is resized or is shown.
- *getBounds(), getLocation(),getSize()*: Gets bounds, location and size of the component on the Target VM using *ComponentManager*.
- *addImageListener()/removeImageListener()*:Creates a *ImageDataCollector*. It adds/removes image listeners.
- *refreshImage()*: Gets an image from the Target VM using the *ImageDataCollector* and refreshes the image on the canvas.

- *instantiateOnFreeForm()/disposeOnFreeForm()*: The container in which the beans are placed when they are not contained in a top level window is called free form. It instantiates/Disposes the bean on the free form in the **Target VM**.

ContainmentContainerProxyAdapter class handles adding and removing of components from containers in the **Target VM**. It implements *IContainmentInfo* to get the descriptor for *containment* structural feature. It has the following important methods:

- *applied()/canceled()*: Adds/removes adapters on children. Adds/removes children on the Target VM.
- *AddChildOnTargetVM()/removeChildOnTargetVM()*: *BeanProxyAdapter* for each ULC container implements these abstract methods to handle the addition and removal of components to the container in the Target VM.
- *getContainmentPropertyDescriptor()*: Gets the property descriptor for the *containment* structural feature.

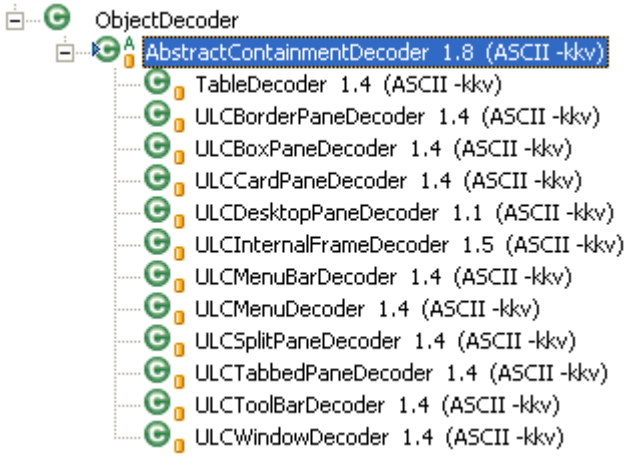
4. 2. 4. Code Generation: Decoder and DecoderHelper

A Decoder adapts the **VE Model** to the source model and is responsible for keeping them in synch. The code generation and decoding of simple properties of ULCComponents, (i.e. those having setter and getters) is handled by *org.eclipse.ve.internal.java.codegen.java.ObjectDecoder*.

The container-contained relationship between ULC containers and ULC components are specified by structural features such as *components, menuBar* (for *ULCWindow*) and *columns* (for *ULCTable* and *ULCTableTree*). These attributes are not available from inspection because they do not have appropriate set/get methods. These structural features are set and unset when components are added and removed. Each container has its own special method for adding and removing components. Therefore ULC Visual Editor uses special code generation and decoder adapter classes for each container. These classes are responsible for:

- generating code for the container-contained structural features
- decoding (parsing) the methods for adding/removing components to create the model

In ULC Visual Editor, *AbstractContainementDecoder* is the root class of the decoder hierarchy. It extends *org.eclipse.ve.internal.java.codegen.java.ObjectDecoder*. The following picture shows the complete hierarchy:



Important methods of *AbstractContainmentDecoder* are:

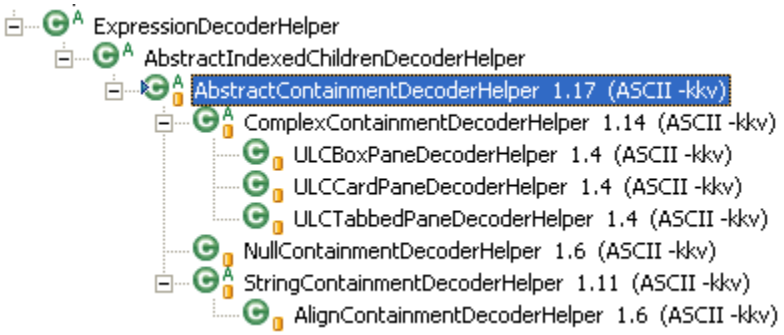
- *getChildren()*: Returns the model children.

getChildrenSfNames(): This abstract method is implemented by the decoder for each container and returns the names of the structural features that represent the container-contained relation.

- *getDecoderHelper()*:This abstract method is implemented by the decoder for each container and returns its decoder helper class (see section 4. 2. 4. 1 below).
- *getMethodName()*:This abstract method is implemented by the decoder for each container and returns the name of the method of the container bean that sets the container-contained structural feature.
- *initialDecoderHelper()*: Sets the decoder helper for a given structural feature.
- *initialFeatureMapper(EStructuralFeature)/initialFeatureMapper()*:Sets the feature mapper for a given structural feature. An *IFeatureMapper* is a *IExpressionDecoderHelper* which provides java methods to map *PropertyDecorators* to structural feature and vice versa. ULC Visual Editor uses *ContainmentFeatureMapper* that extends *AbstractFeatureMapper*.

4. 2. 4. 1. DecoderHelpers

The decoder helpers are helper classes to the decoders. They implement methods for generation and decoding of code. In ULC Visual Editor, *AbstractContainementDecoderHelper* is the root class of the decoder helper hierarchy. It extends *org.eclipse.ve.internal.java.codegen.java.AbstractIndexedChildrenDecoderHelper*. The following picture shows the complete hierarchy:



Important methods of *AbstractContainementDecoderHelper* are:

- *adaptToCompositionModel()/UnadaptToCompositionModel()*: Adapts/Unadapts the child to the composition model.

decode(), decodeArguments(MethodInvocation) : Decodes the expression and arguments to construct a container-contained relationship in the model.

- *generate(), generateArguments()*: Generates the expression and arguments for a child object.
- *getAddedInstance()*: Gets the child object.
- *getArgsHandles()*: Returns a list of unique strings for arguments that make this expression unique.
- *getIndexedEntries(), getIndexedEntry()*: Returns child(ren).
- *removeFromModel()*: Removes the child from model.

ComplexContainmentDecoderHelper is used by *ULCBoxPaneDecoder*, *ULCCardPaneDecoder* and *ULCTabbedPaneDecoder*. For each of these a special object (*BoxPaneContainment*, *TabbedPaneContainment*, *CardPaneContainment*) containing the containment parameters is set as a value of the *containment* structural feature of the contained component. The complex arguments for adding children are derived from the value of the *containment* structural feature. The following source code example shows the complex arguments when adding a *ULCButton* to a *ULCBoxPane*. The first five arguments (x, y, horizontalSpan, verticalSpan, alignment) have to be derived from *BoxPaneContainment*:

```
ulcBoxPane.set(0,0,1,1, ULCBoxPane.BOX_EXPAND_EXPAND, ulcbutton);
```

StringContainmentDecoderHelper is used by *ULCWindowDecoder*. *ULCWindow* uses a *String* object to store the alignment value in the *containment* structural feature of the contained component.

NullContainmentDecoderHelper is used by *ULCMenuBarDecoder*, *ULCToolBarDecoder*, *ULCDeskTopPaneDecoder* and *ULCInternalFrameDecoder*. These containers do not have containment parameters.

4. 2. 5. Property Source Adapters

A *PropertySourceAdapter* is used to adapt the **VE Model** to the **Property Sheet.** It helps to display and edit the properties of the bean in the **Property Sheet** by getting the property descriptor for each structural feature. *ULCComponentPropertySourceAdapter* extends *org.eclipse.ve.internal.cde.properties.PropertySourceAdapter*.

Important methods of *ULCComponentPropertySourceAdapter*:

- *includeFeature*(): decides if a structural feature is to be displayed in the **Property Sheet**. ULC Visual Editor delegates this decision to the *BeanProxyAdapter* of each container.
- *getPropertyDescriptor(EStructuralFeature sf)*: gets the property descriptor for the structural feature.

4. 2. 6. Property Descriptors

Property descriptors are responsible for the representation of a structural feature in the **Property Sheet.** They implement the interface *org.eclipse.ui.views.properties.IPropertyDescriptor*. Required attributes of this class are the property id and the display name of the property. Optionally, it holds label providers, cell editors, description and help context.

ULC Visual Editor implements special label providers and cell editors for certain types of Java classes and for structural features, which need a special representation (e.g. containment).

ULC Visual Editor provides property descriptors for the following ULC Java classes:

- com.ulcjava.base.application.util.Dimension
- com.ulcjava.base.application.util.Insets
- com.ulcjava.base.application.util.Point
- com.ulcjava.base.application.util.ULCIcon

The label providers extend *org.eclipse.jface.viewers.LabelProvider* and the cell editors extend *org.eclipse.ve.internal.java.core.DefaultJavaClassCellEditor* with the exception of *ULCIconCellEditor* that extends *org.eclipse.jface.viewers.DialogCellEditor*.

In addition ULC Visual Editor provides the following property descriptors for *containment* structural feature. These are:

- *AlignmentPropertyDescriptor*: Creates a *AlignmentLabelProvider* for the alignment property of the contained component in a ULC container (e.g *IDefaults.BOX_EXPAND_EXPAND*). It also defines *AlignmentCellEditor* to display the various aligment strings in a combo box. It extends *org.eclipse.ve.internal.propertysheet.ObjectComboBoxCellEditor*.
- *BoxPaneContainmentPropertyDescriptor*: Creates a label provider, which displays the *BoxPaneContainment* properties (position, span and alignment) in one cell.
- *CardPaneContainmentDescriptor*: Creates a label provider which displays the *CardPaneContainment* properties (name, alignment) in one cell
- *TabbedPaneContainmentDescriptor*: Creates a label provider for *TabbedPaneContainment* properties (title, icon, tool tip text, alignment) in one cell.

5. Recommendations

This paper gives a functional and an architectural overview of ULC Visual Editor for Eclipse. It further explains how ULC Visual Editor was implemented by extending Eclipse's Visual Editor for Java. Since ULC API is similar to Swing API, ULC Visual Editor was able to reuse quite a bit of code from Visual Editor. However, special code was required in the following cases:

- Unlike Visual Editor, which uses an intermediate containment object to represent the container-contained relationship, ULC Visual Editor uses a direct relationship between a container and its contained objects. Therefore the handling of the container-contained relationship between ULC containers and ULC components within them required special coding.
- ULC widgets are faceless server side widgets, i.e. they are not visible; they are rendered using Swing in the client side presentation engine (UI Engine). Special code was required for ULC Visual Editor's interactions with the Target VM. In order to instantiate beans and to obtain images from the Target VM, ULC Visual Editor needs to start a dummy ULC application in the **Target VM** by using ULC *DevelopmentRunner*.

5. 1. Open issues in Visual Editor Framework

In the following we recommend some extension and modifications for the Visual Editor Framework in order to make it easily extendable:

1. Schema of .override XMI and documentation.
2. Schema and documentation of palette XMI.
3. For each part of the Visual Editor Framework a clear hook should be provided and documented, for instance exact hook classes for **Graphical Editor**, **Java Beans Tree**, **Property Sheet**, P**alette**, **Target VM**, **BeanInfo VM** and **Code generation**. These hooks should be provided from the org.eclipse.ve.java.core plug-in, so that extensions of VE need not depend on the on Visual Editor for Swing or SWT. For example, *JavaBeanGraphicalEditPart*, *ChildRelationshipDecoderHelper are* classes which are in the org.eclipse.ve.jfc package but offer general functionality.
4. ULC Visual Editor is using many of the *org.eclipse.ve.internal.** classes like *JavaBeanTreeEditPart*, *ComponentManager* etc. These classes should not be internal classes as they provide essential functionality.
5. Adapters are a key feature of the implementation of Visual Editor. Most of the adapters have to be specified on the models (*BeanProxyAdapter*, *EditPart, etc.*), but some adapters have to be specified in specific methods of framework classes (e.g. *ExpressionDecoderAdapter* on *BeanDecoderAdapter*). This should be well documented. Even better would be if there exists only one way to specify adapters.
6. The synchronization of the **VE Model** and the **Source Model** is a key element in Visual Editor. From our experience this code contains still a lot of dependencies to Swing and SWT specific issues and was the most difficult part to extend. It is essential that the codegen.jar of the org.eclipse.ve.java.core plug-in is well documented.

The following issues are classified as bugs:

1. Better handle on fetching images from **Target VM** and displaying them on the canvas. Sometime a manual refresh is required because the state of **Graphical Editor** is not the same as the state of **Target VM** due to subtle timing problems between their threads.
2. Feedback on *JSplitPane* so that a component can be added to the right side without requiring to add a component to the left side first.

3. The Free Form window of the **Target VM** is sometimes visible in spite of the large off screen location, which confuses end users

The following issues are classified as nice to have.

1. It would be very convenient to trigger **pause round tripping** when editing in the **Source Editor** and **reload** after the editing in the Source Editor has stopped a certain time.
2. The **Palette** should be extended in a way that the user can choose which categories he wants to display when editing a visual class.

6. Acknowledgements

ULC Visual Editor would not have been possible without the help and active support of the Eclipse Visual Editor team at IBM. We are grateful to Dr. Gili Mendel, Joe Winchester, Rich Kulp, Srimanth Gunturi, Peter Walker and rest of the members of the Eclipse Visual Editor for Java team at IBM.

In addition, we thank all members of the ULC team at Canoo. Special thanks go to Daniel Grob who also worked on the first version of ULC Visual Editor for IBM WSAD.