RUNA WFE. Graphical Process Designer (GPD) Developer's Guide

Version 2.1

© 2004-2008, ZAO Runa. RUNA WFE is an open source system distributed under a LGPL license (http://www.gnu.org/licenses/lgpl.html).

Contents

1 INTRODUCTION....1

2 GRAPHICAL DESIGNER MODULES....4

3 CONFIGURING ECLIPSE FOR WORK WITH THE GRAPHICAL DESIGNER....6

4 ORG.JBPM.UI MODULE PACKAGES....6

4.1 ORG.JBPM.UI.FIGURE....7
4.2 ORG.JBPM.UI.EDITOR....9
4.3 ORG.JBPM.UI.MODEL....10
4.4 ORG.JBPM.UI.PART CONTROLLER PACKAGE....15
4.4.1 Graphical controller package org.jbpm.ui.part.graph....15
4.4.2 Hierarchical controller package org.jbpm.ui.part.tree....18
4.5 POLICY PACKAGE ORG.JBPM.UI.POLICY....21

5 CHANGING GPD FUNCTIONALITY....22

5.1 CHANGING GRAPHICAL ELEMENT REPRESENTATION....22
5.2 ADDING A NEW GRAPHICAL ELEMENT....23
5.2.1 Creation of a Model Element....24
5.2.2 Creation of a Graphical Representation of a Model Element....24
5.2.3 Adding a Graphical Representation to the Tool Palette....25
5.3 ADDING A NEW MENU ITEM....25
5.4 ADDING A NEW ELEMENT IN THE "V" ELEMENT OF THE FORM DESIGNER....27
5.5 ADDING A NEW ELEMENT IN THE "F" ELEMENT OF THE FORM DESIGNER....28
1.1.5.6. USING FREEMARKER IN FORMS....29

2.6 INTERACTION WITH INFOPATH (WINDOWS ONLY)....29

6.1 Architecture....29
6.2 Creation of InfoPath ActiveX Elements....30
6.3 Creation of InfoPath Template Parts Elements....32

3.7 RCP APPLICATION ASSEMBLY IN GPD....32

4.8 REFERENCES....34

1 INTRODUCTION

RUNA WFE Graphical Process Designer (GPD) is based on the JBoss jBPM Graphical Process Designer, modified according to the requirements of RUNA. Technologically, RUNA WFE is based on the Graphical Editing Framework (GEF) that is part of the Eclipse platform. Eclipse implements the OSGi (OSGi Framework) services model on a Java platform.

OSGi Framework provides a unified environment for applications (“bundles”), connecting:

– a bundle execution environment;

– modules that complement Java class loading policies with private classes for a module and controlled module binding;

– application module lifecycle management, allowing to dynamically install, start, stop, update, and delete modules;

– registration services, allowing for dynamic sharing of objects by applications.

The Eclipse platform is a set of subsystems, implemented with a small run-time kernel and a number of modules (plug-ins), extending the functionality of the platform. For purposes of this document, the terms "module" and "plug-in" are equivalent and interchangeable. The use of these terms is basically determined by style considerations.

The running Eclipse kernel dynamically discovers, configures, and starts the plug-ins of the platform. Eclipse supports dynamic connection of plug-ins, described by plug-in descriptors (in MANIFEST.MF and plugin.xml files). To extend the functionality, the platform plug-ins define extension points in plug-in descriptors. An extension point is an xml description of the interface of the extended plug-in component. Extending plug-ins use extension points to add functionality. The Eclipse platform does not distinguish between user plug-ins and those native to the platform.

The Eclipse platform is implemented in Java, which makes developed applications portable to different platforms with different operating systems.

GEF (Graphical Editing Framework) provides an environment for development of graphical designers. GEF is implemented as a set of plug-ins, extending the plug-ins of the Eclipse platform. GEF connects the elements of the application model with their graphical views that are created, using graphical components from the Draw2d library. GEF controllers support visual representation of model elements in the MVC (Model-View-Controller) architecture. For each element of the view, the controller for this view interprets the events of the user interface and converts them into processing commands for the corresponding element of the model.

A general overview of the GEF architecture is shown in Figure 1. General view of the GEF architecture. A description of GEF components is presented in Table 1. GEF architecture components.

Table 1. GEF architecture components.

Component	Description
Model	Holds data. Must have a mechanism to notify about changes.
View	A visual representation of the model. Consists of figures

	representing the elements of the model. A model can be represented both graphically and as a hierarchical (tree) structure.
Controller	Controllers connect model elements with corresponding view elements. Controllers can be graphical or hierarchical depending on the type of view they provide. They are responsible for editing model elements through a view and also for displaying changes in model elements in a view. Controllers use editing policies – elements performing the majority of editing tasks.
Action	Elements that process data input. Convert user interface events into requests that use controller APIs.
Request	Requests are elements, encapsulating user interface events. Allow to abstract from the source of the event.
Command	Commands encapsulate data on changes in the model. Returned by controllers in response to requests. Also contain information on possibility of interaction.
Event	Events are changes in the user interface, causing changes in a view or model.

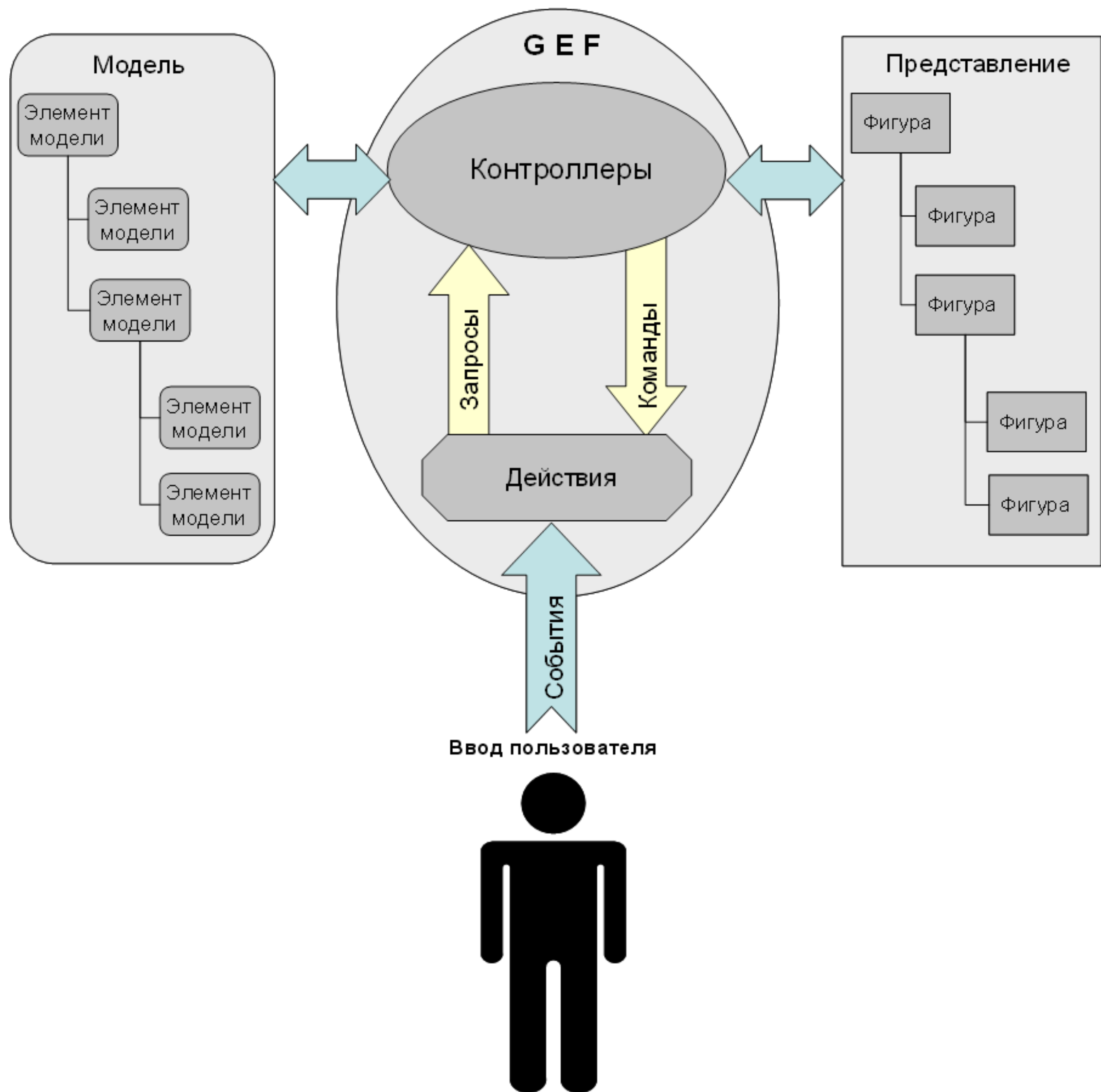

Figure 1. General view of the GEF architecture.

2 GRAPHICAL DESIGNER MODULES

The graphical designer is based on the JBOSS JBPM engine whose main module jbpm.core loads and unloads business process definitions, creates business process instances and execution flows and stops them. Other modules of the graphical designer use the services of the jbpm.core engine for their functions.

The graphical designer modules and their descriptions are shown in Table 2. Graphical designer modules. The relationships of the modules are shown in Figure 2. Relationships between RUNA WFE modules..

Table 2. Graphical designer modules.

Module	Description
org.jbpm.core	Contains JBOSS JBPM engine libraries as well as interfaces to work with the engine.
org.jbpm.db	Not used in the current implementation.
org.jbpm.feature	Arranges designer modules into a group.
org.jbpm.help	GPD help subsystem. Contains no help data in the current implementation.
org.jbpm.ui	Contains JBOSS JBPM graphical designer packages, including GEF, model elements and graphical views. JBOSS JBPM packages are used in RUNA WFE Graphical Process Designer.
ru.runa.jbpm.ui	RUNA WFE Graphical Process Designer module. Based on org.jbpm.ui.
tk.eclipse.plugin.htmleditor	HTML editor module.
tk.eclipse.plugin.wysiwyg	Visual (WYSIWYG) editor module. Extends the functionality of tk.eclipse.plugin.htmleditor.

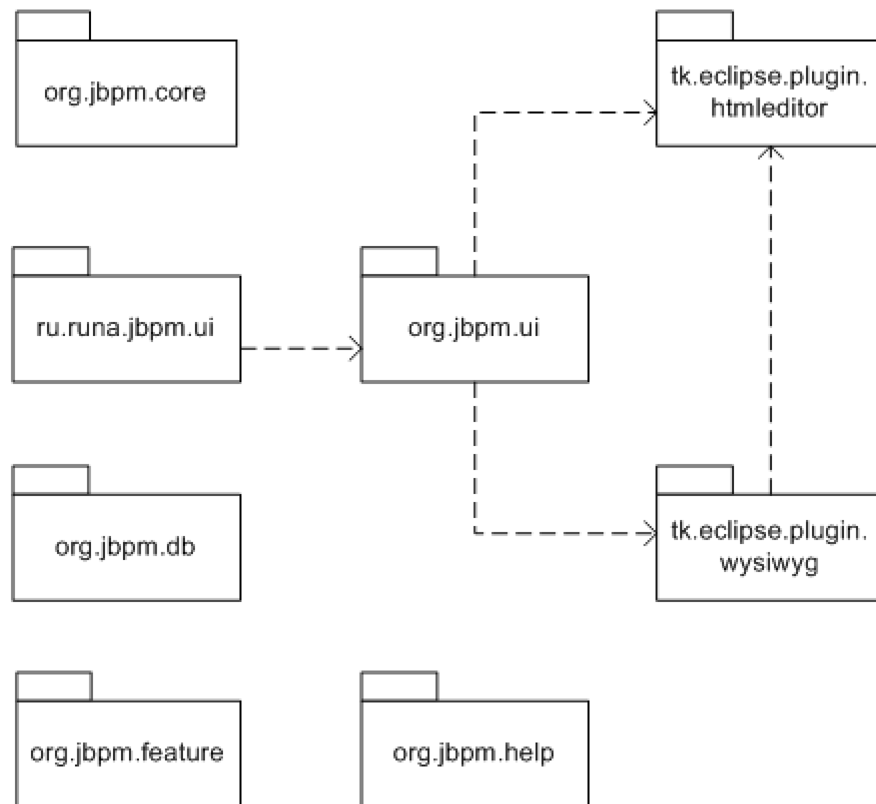

Figure 2. Relationships between RUNA WFE modules.

3 CONFIGURING ECLIPSE FOR WORK WITH THE GRAPHICAL DESIGNER

Eclipse 3.1.2 is required. The following plug-ins must be installed in Eclipse:

- ✓ WTP
- ✓ WST
- ✓ GEF

Note. All these packages are already included in the Eclipse distribution pack wtp-all-in-one-sdk.

4 ORG.JBPM.UI MODULE PACKAGES

org.jbpm.ui module packages implement the base functionality of the graphical process designer on the GEF platform. The packages correspond to GEF architecture components and contain classes implementing them. Besides, the module contains packages, implementing the user interface. The modules and their descriptions are shown in Table 3. org.jbpm.ui module packages..

Table 3. org.jbpm.ui module packages.

Package	Description
org.jbpm.ui.action	Contains classes, implementing GEF actions while interacting with the user interface of the graphical designer.
org.jbpm.ui.command	Contains classes, inheriting from org.eclipse.gef.commands.Command. These classes implement the commands that are executed by controllers and change the model in response to the requests from the user interface of the designer.
org.jbpm.ui.contributor	Contains classes that create model element objects, figure objects, as well as corresponding graphical and hierarchical controllers.
org.jbpm.ui.dialog	Contains a descriptor handler class for elements, selected in the dialog.
org.jbpm.ui.editor	Contains classes of visual component editors of the graphical editor's GUI.
org.jbpm.ui.factory	Contains factory classes of elements and adaptors.
org.jbpm.ui.figure	Contains classes, implementing figure images in the graphical designer window.
org.jbpm.ui.model	Contains classes of business process model elements.
org.jbpm.ui.outline	Contains classes, implementing an hierarchical view in the graphical designer window.
org.jbpm.ui.part.graph	Contains classes, implementing graphical controllers of model objects.
org.jbpm.ui.part.tree	Contains classes, implementing hierarchical

	controllers of model objects.
org.jbpm.ui.policy	Contains classes, implementing data processing policies (behavior) of controllers.
org.jbpm.ui.prefs	Contains parameter classes for the elements of the “Preferences” window of the module.
org.jbpm.ui.properties	Contains classes of the cell property editor.
org.jbpm.ui.resource	Contains a class of messages, as well as business process descriptors and forms.
org.jbpm.ui.util	Contains auxiliary classes of business process designer.
org.jbpm.ui.view	Contains classes to display the window with a hierarchical view of the model.
org.jbpm.ui.wizard	Contains classes of wizards for creation of graphical designer objects.

Figure 3. Relationships between classes of WFE graphical editor.

The following subsections describe the main packages of the org.jbpm.ui module.

4.1 org.jbpm.ui.figure

This package contains classes for a visual presentation (figures) of business process model elements. The classes of the package inherit from the org.eclipse.draw2d.Figure base class from the Draw2d library. A class inheritance diagram is shown in Figure 4. Inheritance of classes of graphical element views of the org.jbpm.ui.figure module. The descriptions of the classes are shown in Table 4. Classes of org.jbpm.ui.figure..

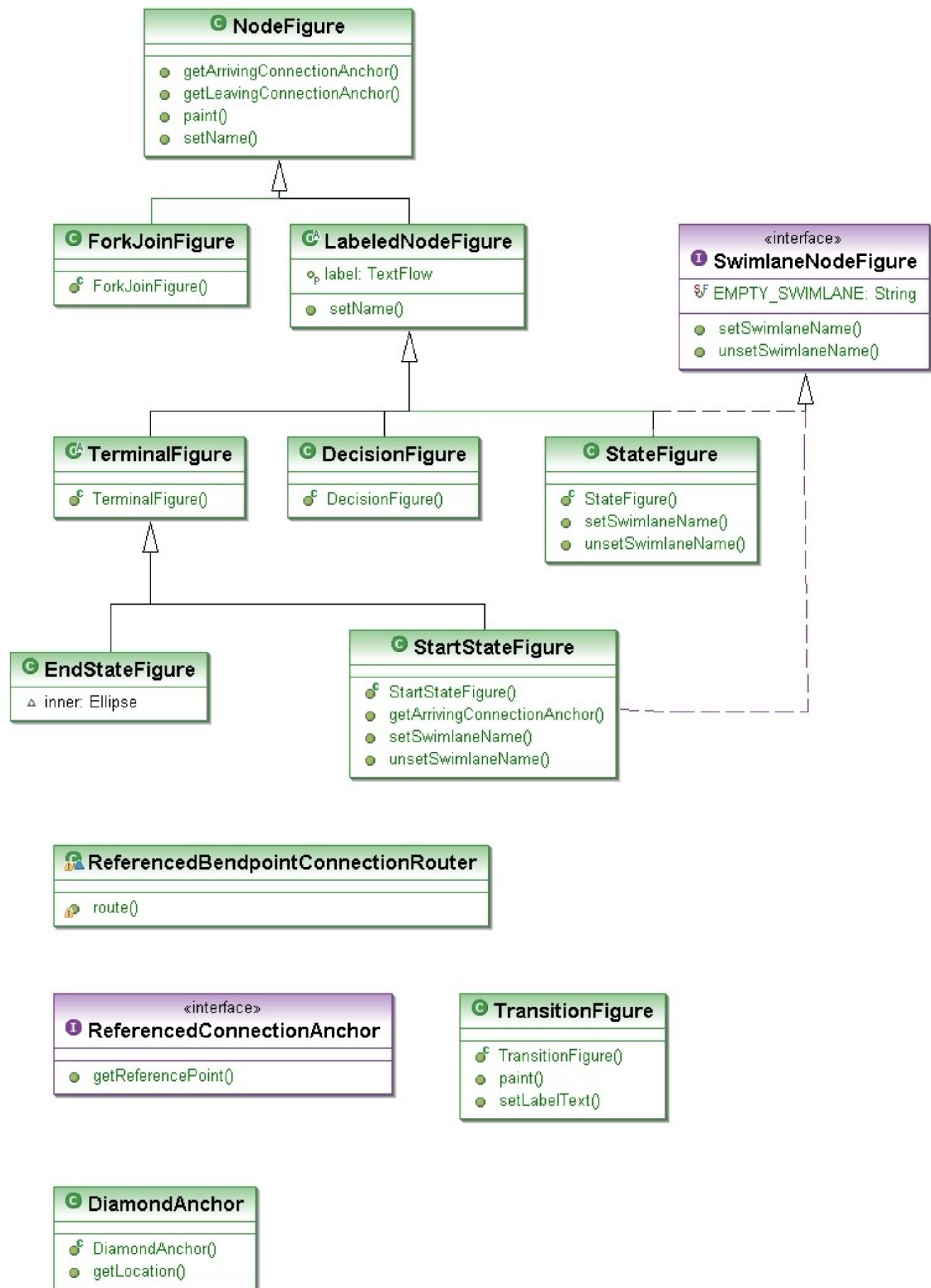

Figure 4. Inheritance of classes of graphical element views of the org.jbpm.ui.figure module

Table 4. Classes of org.jbpm.ui.figure.

Class	Description
DecisionFigure	Implements an image of the node figure Decision.
DiamondAnchor	Implements an anchor of a connection within a figure.
EndStateFigure	Implements an image of node figure End State.

ForkJoinFigure	Implements images of the node figures Fork and Join.
LabeledNodeFigure	An abstract base class. Controls the label of a node figure.
NodeFigure	A base class. Determines the base behavior of a node figure.
ReferencedBendpointConnectionRouter	Implements the routing of the line for a given connection.
ReferencedConnectionAnchor	This interface overrides the method of getting the reference point.
StartStateFigure	Implements an image of node figure Start State.
StateFigure	Implements an image of node figure State.
SwimlaneNodeFigure	This interface defines a static constant EMPTY_SWIMLANE and declares the setSwimlaneName(String swimlaneName) set method and the unsetSwimlaneName unset method.
TerminalFigure	An abstract base class for figures Start State and End State.
TransitionFigure	Implements an image of the Transition figure.

4.2 org.jbpm.ui.editor

Class	Description
DesignerActionRegistry	A container for actions executed in the graphical designer. Adds actions to the action stack.
DesignerContentProvider	Implements methods for providing data, describing model elements.
DesignerDropTargetListener	Extends class TemplateTransferDropTargetListener for conversion into a factory.
DesignerEditor	Contains methods, implementing graphical editor functionality.
DesignerEditorActionBarContributor	A class to install, deinstall and control menu items and corresponding windows of GPD.
DesignerGraphicalEditorPart	A class for graphical representation of business processes in WFE RUNA GUI.
DesignerPaletteRoot	A class for the palette of business process graphical elements in WFE RUNA GUI.
DesignerSwimlaneEditorPage	A class for a business process role editor.
DesignerVariableEditorPage	A class for a business process state editor.
ImageHelper	An auxiliary class forming an image for graphical representation of business processes in WFE RUNA GUI.
PaletteFlyoutPreferences	A class to save/load preferences for the palette of business process graphical elements in WFE RUNA GUI.

4.3 org.jbpm.ui.model

The org.jbpm.ui.model package contains classes of model elements of the graphical designer. An inheritance diagram for model element classes is shown in Figure 5. Inheritance of graphical element classes of the org.jbpm.ui.model module. The descriptions of the classes are shown in Table 5. Classes of org.jbpm.ui.model package.

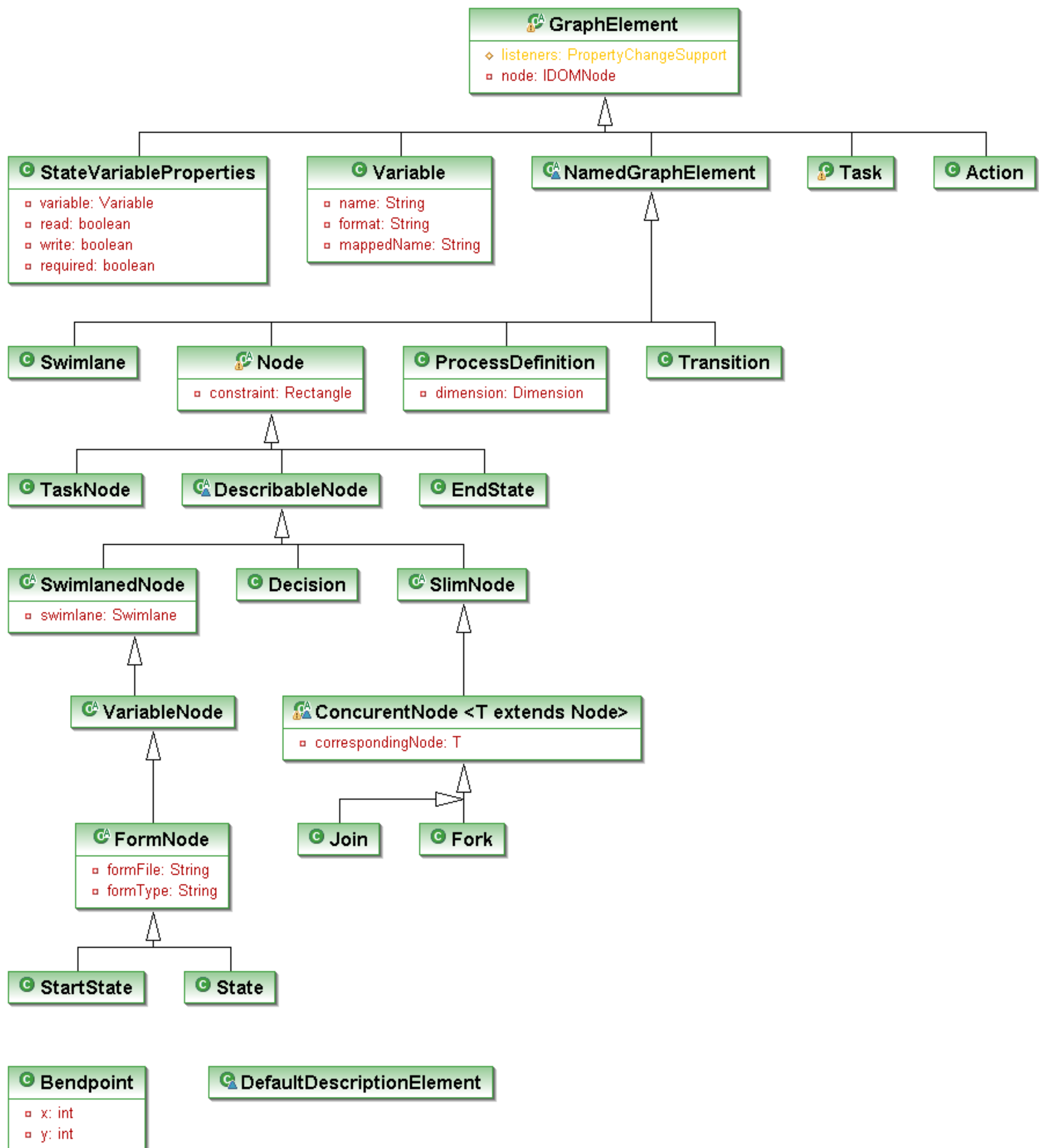

Figure 5. Inheritance of graphical element classes of the org.jbpm.ui.model module

Table 5. Classes of org.jbpm.ui.model package

Class	Description
Action	The Action element class. Inherits from abstract class GraphElement. Implements methods that: – get the delegate class name and configuration of the element; – get and set the event-type attribute.
Active	The interface declares methods that: – add, delete and get an action list of the element.
Bendpoint	A model element class. Defines a bendpoint of a Transition element (a bendpoint of the line, connecting the nodes). Defines methods for: – getting bendpoint coordinates; – setting the location of a bendpoint.
ConcurentNode	An abstract class of a Node element of the model. Inherits from class SlimNode. The base class for the Join and Fork classes. Defines a class for corresponding Join or Fork nodes. Implements the propertyChange method of the PropertyChangeListener interface.
Decision	A class of the Decision node element of the model. Inherits from the abstract class DescribableNode. Implements methods that: – get the action list for a node; – add and delete an action in an hierarchy of actions; – get and set node properties; – check for possibility to add and attach a transition.
defaultActionElement	Implements methods of the Active interface that: – add and delete actions and get the action list of an element.
defaultDelegationElement	Implements methods of the Delegable interface that: – get and set the name of the delegate class of the element; – get and set the configuration of the delegate class of the element;
defaultDescriptionElement	Implements methods of the Describable interface that: – get and set the Description property of an element. The class uses static constants of the NotificationMessages interface.
defaultStateActionElement	Inherits from class defaultActionElement. Overrides the method that adds an action.
Delegable	This interface declares methods that: – get and set the name of the delegate class of the element; – get and set the configuration of the delegate class of the element;
Describable	The interface declares methods that:

	– get and set the Description property of the element.
DescribableNode	An abstract class of a Node element of the model with a Description property. Inherits from abstract class Node. The base class for classes Decision, SlimNode, SwimlanedNode. Defines the Description property for a model element; Implements methods for getting and setting the value of this property.
ElementType	Defines methods that configure types of model elements: – initialize and display element types from the module manifest; – get element types from collections; – create ElementContributor interfaces for element types.
EndState	Defines the End node element of the model. Inherits from class Node. Defines methods that: – get the prefix of the End element name; – set the size of the graphical element area; – define the possibility to add and attach input and output transitions.
EventTypes	This interface defines static constants for event types.
Fork	Defines the Fork node element of the model. This class: – defines the prefix of the Fork element name; – defines corresponding class of the Join node (Synchronization); – checks for possibility to add and attach input and output transitions.
FormNode	An abstract class of a Node element of the model. Inherits from VariableNode. The base class for classes StartState and State. Defines a form for the State and Start elements.
GraphElement	An abstract class. Implements methods of interfaces EventTypes, IPropertySource, NotificationMessages and INodeAdapter. Defines methods that: – initialize a node element of the model from an XML file; – get the hierarchy level and element type; – get, add and delete elements in a hierarchy of elements; – add and delete listeners of property change events;
InternalState	A marker interface for marking process states between the Start and End states.
Join	Defines the Join node element of the model. The class: – defines the prefix of the Join element name; – defines corresponding class of the Fork node; – checks for possibility to add and attach input or output transitions.
NamedGraphElement	An abstract class. Inherits from class GraphElement. The base class for classes Node, ProcessDefinition, Swimlane, Transition. Implements methods for

	getting and setting the name of a node.
Node	An abstract class of a Node element of the model. Inherits from class NamedGraphElement. The base class for classes DescribableNode, EndState and TaskNode. Defines a node element of the model. Implements methods to do the following: – get the prefix of the name; – get and set a node area; – get and form the names of outward transitions; – check an element as the parent of a given element; – get lists of input and output Transition elements; – add and delete output transitions. This class declares methods that check for possibility to add and attach input and output transitions.
NotificationMessages	This interface defines static constants for messages.
ProcessDefinition	Defines a process. Inherits from class NamedGraphElement. Implements the methods of the interfaces Active and Describable. The methods of the Active interface are implemented, using class defaultActionElemen methods that add actions, delete actions and get an action list. The methods of the Describable interface are implemented, using class defaultDescriptionElement methods that get and set the Description property of the process. This class defines methods that: – get and set the size of the Process element; – initialize a node and set its name to "process"; – form the names of process nodes, swimlanes and state variables; – add, delete and get lists of nodes and swimlanes of the process; – define the Descriptor property of the process; – determine equivalence of objects.
SlimNode	An abstract class of a node element of the model. Inherits from class DescribableNode. The base class for ConcurentNode. Implements methods that: – get the action list for a node; – add and delete actions in a hierarchy of actions; – set the geometrical size of the node area.
StartState	Defines the Start node element of the model. Inherits from abstract class FormNode. Defines methods that: – get the prefix of the Start element name; – set the size of the area of a graphical element; – get input and output transitions; – get, add and delete tasks; – get and delete a swimlanes; – check for possibily to add and attach input and output transitions.
State	Defines the Start node element of the model. Inherits from abstract class FormNode. Defines methods that: – add and delete an action;

	– get a list of actions; – get the prefix of the State element name; – get and delete swimlanes; – assign and get swimlanes; – check for possibility to add and attach input and output transitions.
StateVariableProperties	Defines a State graphic element of the model. Inherits from abstract class GraphElement. Defines methods that: – set and check the properties of variables of a State element.
Swimlane	Defines a Swimlane model element. Defines methods that: – get and set the node configuration property; – get and set the delegate class; – get and set the node description property.
SwimlanedNode	An abstract class of a node model element containing a swimlane. Inherits from class DescribableNode. The base class for class VariableNode. Defines the Swimlane property for a node. Defines methods to add, get and delete the Swimlane property. Overrides the getPropertyValue and setPropertyValue methods for getting and setting PROPERTY_SWIMLANE.
Task	Defines a Task graphic element of the model. Inherits from abstract class GraphElement. Defines methods that: – get and set the name of the element; – get the name of the parent element; – get and set the date of task execution; – get, add and delete Assignment and Controller nodes; – get and set Assignment and Controller nodes; – get and set configuration type of Assignment and Controller nodes; – get and add configuration information for Assignment and Controller nodes; – add and get variables and lists of variables for a controller; – check for a Blocking attribute and set this attribute; – check for possibility to assign the specified name.
TaskNode	Defines a node element of the model. Inherits from abstract class Node. Defines methods that: – form the name of an element; – get, add and delete an element; – get a list of elements; – get a child object of the Task class ; – get a child object of the Transition class; – check for possibility to add and attach input and output Transition elements.

Transition	Defines a Transition model element. Inherits from abstract class NamedGraphElement. Overrides its abstract method canSetNameTo (String name) that checks whether the parent element of a given Transition element has a Transition element, named “name”. Overrides methods to add, delete actions and get an action list. Defines methods to add, delete and set bendpoints of a Transition figure, as well as get a list of bendpoints. Defines methods to get and set the Source element and the Target element for a Transition element. Defines a method to get the name of a Transition element, the names of the Source and Target elements. Defines a method to set the name of a Transition element.
Variable	Defines a State Variable model element. Inherits from class GraphElement. Defines methods that: – set and get values of the variables “name”, “format” and “mappedName”; – determine equivalence of objects.
VariableNode	An abstract class of a node element of the model. Inherits from class SwimlanedNode. The base class for class FormNode. Defines methods that: – add and delete properties of State variables of State and Start elements; – get and set lists of State variables.

4.4 org.jbpm.ui.part controller package

GPD controllers are grouped into graphical controller packages org.jbpm.ui.part.graph and hierarchical controller packages org.jbpm.ui.part.tree.

4.4.1 Graphical controller package org.jbpm.ui.part.graph

A graphical controller package contains controller classes to represent graphical model elements on a diagram of business processes in the graphical designer. Inheritance of classes of graphical controllers is shown on a diagram in Figure 6. Inheritance of graphical controller classes. A description of these classes is shown in Table 6. org.jbpm.ui.part.graph package classes.

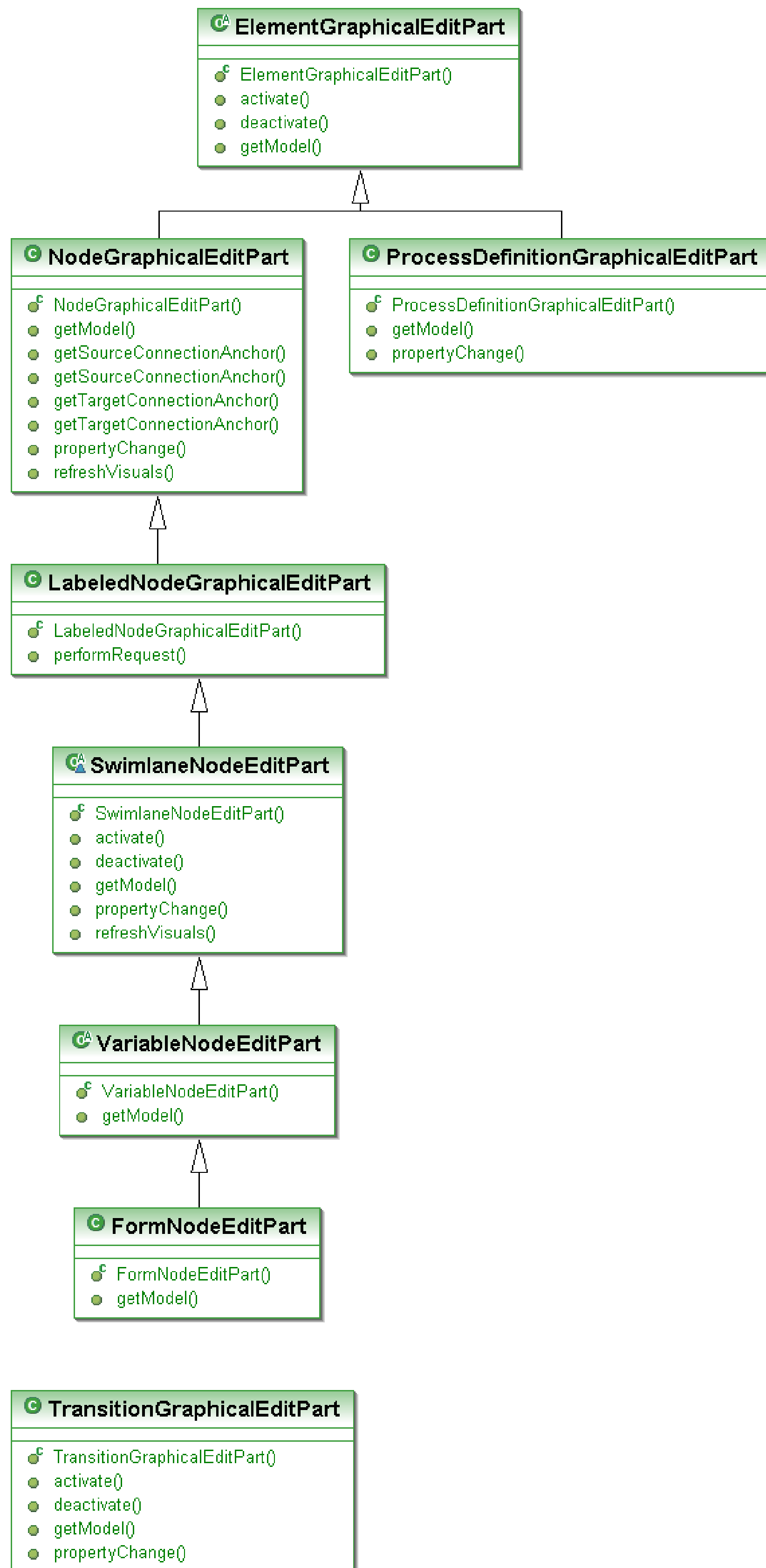

Figure 6. Inheritance of graphical controller classes

Table 6. org.jbpm.ui.part.graph package classes.

Class	Description
ElementGraphicalEditPart	Defines the controller of a graphical element. Inherits from org.eclipse.gef.editparts.AbstractGraphicalEditPart. Overrides methods that: – get an object element of the model; – get a visual representation of the element; – activate and deactivate the element controller.
FormNodeEditPart	Defines the controller of a node element containing a form. Inherits from class VariableNodeEditPart. Overrides the method that gets a model object, so as to get an object with a form.
LabeledNodeGraphicalEditPart	Defines the controller of a node element with a label. Overrides the request execution method and defines private methods for label editing.
NodeGraphicalEditPart	Defines the controller of a node element. Inherits from class ElementGraphicalEditPart. Redefines and implements methods that: – get a node object element of the model; – get and create a visual representation of a node; – create policies (behavior) of the controller; – create anchors of the source and target for input and output connections; – get lists of input and output transitions; – update the visual representation of a node; – modify node properties.
ProcessDefinitionGraphicalEditPart	Defines the controller of the process. Inherits from class ElementGraphicalEditPart. Redefines and implements methods that: – get a node element “Process” of the model; – get a list of child elements; – create policies (behavior) for the controller of the process; – modify node properties.
SwimlaneNodeEditPart	An abstract class defining the controller of a node element containing a swimlane. Inherits from class LabeledNodeGraphicalEditPart. Overrides base class methods that: – update the visual representation of a swimlane; – get a node element of the model, containing a swimlane; – get the Swimlane object of the model; – activate and deactivate the controller; – modify swimlane properties.
TransitionGraphicalEditPart	Defines the controller of a Transition element. Inherits from org.eclipse.gef.editparts.AbstractConnectionEditPart. Overrides methods that: – get a Transition object of the model; – create and update the visual representation of a transition;

	– get a list of bendpoints for a transition; – create policies (behavior) for a transition controller; – activate and deactivate the transition controller; – modify transition properties.
VariableNodeEditPart	This abstract class defines the controller of a State node element with State variables. Inherits from class SwimlaneNodeEditPart. Overrides the method that gets a model object, so as to get an object with State variables.

4.4.2 Hierarchical controller package org.jbpm.ui.part.tree

This hierarchical controller package contains controller classes for hierarchical representation of model elements in the graphical designer window.

Inheritance of controller classes for hierarchical representation of model elements is shown in Figure 7. Inheritance of hierarchical controller classes.. A description of these classes is shown in Table 7. org.jbpm.ui.part.graph package classes.

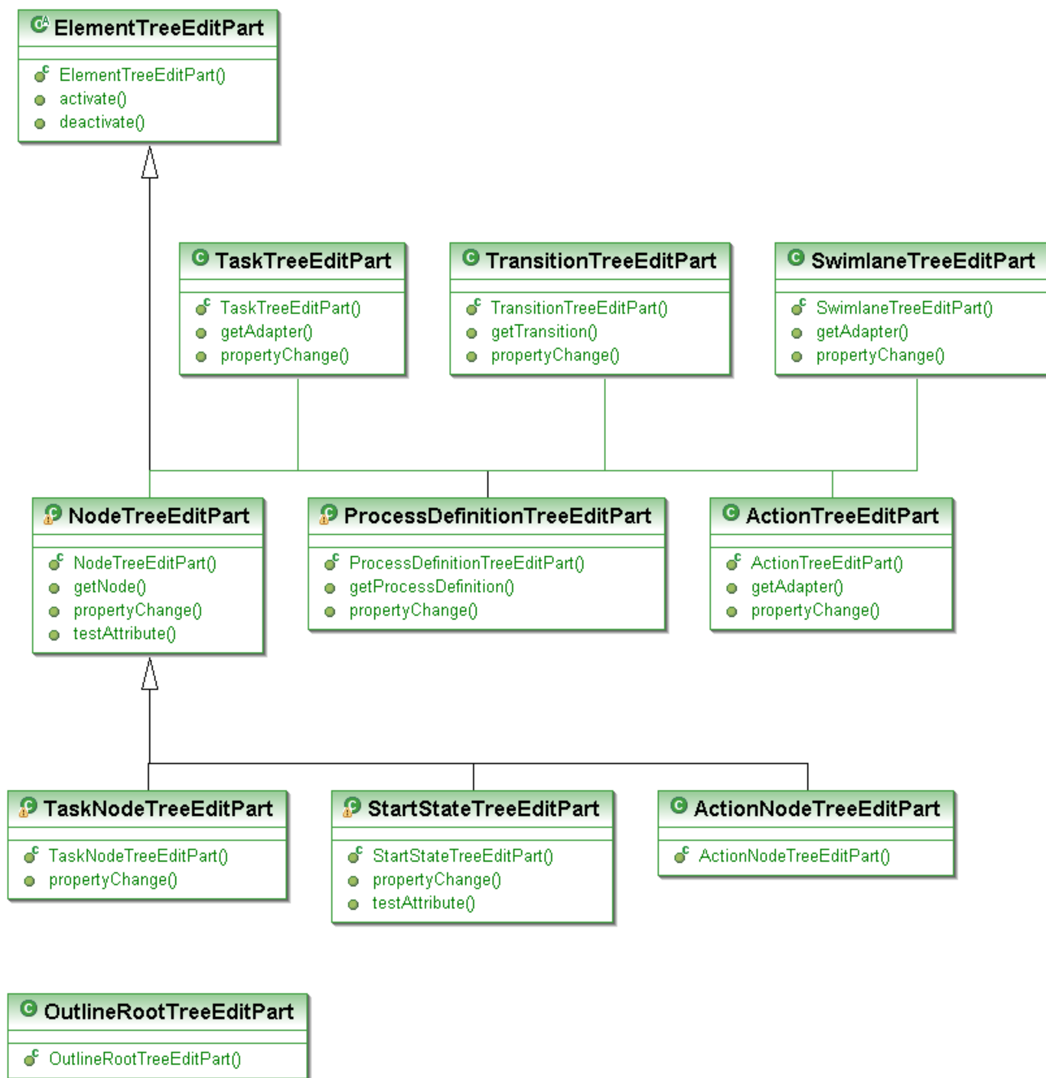

Figure 7. Inheritance of hierarchical controller classes.

Table 7. org.jbpm.ui.part.graph package classes.

Class	Description
ActionNodeTreeEditPart	Defines the Action controller of a node element of the model. Inherits from NodeTreeEditPart. The constructor of the class checks that the element can perform actions (implements the Active interface).
ActionTreeEditPart	Defines the controller of an Action element. Inherits from ElementTreeEditPart. Defines methods that: – get the Action model element from the current controller; – update the visual representation of the Action element of the current controller when its properties are changed; – get the adapter of the visual representation of the element;
ElementTreeEditPart	This abstract class defines the controller of an element. Inherits from AbstractTreeEditPart. Defines methods that:

	– get a graphical object element of the model; – activate and deactivate the controller;
NodeTreeEditPart	Defines the controller of a node element of the model. Defines methods that: – get a node element of the model; – get the list of child elements of an element, including the actions; – update the visual representation of an element; – update child elements while changing the properties of the element; – checking that the element is a node.
OutlineRootTreeEditPart	Defines the root controller. Inherits from AbstractTreeEditPart. Defines methods to get a list of processes of the model.
ProcessDefinitionTreeEditPart	Defines the controller of a process definition element. Inherits from ElementTreeEditPart. Defines methods that: – get a Process Definition model element from the current controller; – get a list of child elements (nodes, swimlanes, actions); – update the element and its child elements while changing the properties of the element.
StartStateTreeEditPart	Defines the Start element controller. Inherits from NodeTreeEditPart. Defines methods that: – get a list of the Start element children; – update child elements while changing the properties of the element; – check that the element and its parents have no tasks (the "hasTask" attribute is equal to "false").
SwimlaneTreeEditPart	Defines the controller of a Swimlane element. Inherits from ElementTreeEditPart. Defines methods that: – get a Swimlane model element from the current controller; – update the visual representation of a Swimlane element of the current controller; – update the element and its child elements while changing the properties of the element; – get the adapter of the visual representation of the element.
TaskNodeTreeEditPart	Defines the controller of a Task node. Inherits from NodeTreeEditPart. Defines methods that: – get a list of child elements of Task and Output Transition types; – update the visual representation of the Task node of the controller; – get a task list from a node element of the model; – get a task list from the current controller element for the node; – match the task list of the current Controller element for the node with the corresponding model element,

	when elements are added or deleted.
TaskTreeEditPart	Defines the controller of a Task element. Inherits from ElementTreeEditPart. Defines methods that: – get the Task model element from the current controller; – update the visual representation of the Task element of the current controller when its properties are changed; – get the adapter of the visual representation of the element.
TransitionTreeEditPart	Defines the controller of a Transition element. Inherits from ElementTreeEditPart. Defines methods that: – get a Transition model element from the current controller; – update the visual representation of the Transition element of the current controller; – get the task list of the Transition element of the current controller; – update the child elements while changing the properties of the element.

4.5 Policy package org.jbpm.ui.policy

This policy package contains classes that implement execution of edit tasks (requests) by controllers. A description of these classes is shown in Table 8. org.jbpm.ui.policy package classes..

Table 8. org.jbpm.ui.policy package classes.

Class	**Description**
NodeComponentEditPolicy	Implements the policy of node deletion on request. Contains the createDeleteCommand method that creates a NodeDeleteCommand command (node deletion).
NodeDirectEditPolicy	Implements the policy of direct node editing on request. Contains methods that: – get a NodeSetNameCommand command for the node; – display changes.
NodeGraphicalNodeEditPolicy	Implements the policy of creation and attachment of a connection. Contains methods that: – get a command to attach connection to the target; – get a command to attach connection to the

	sources; – get a command to reattach the connection to the source; – get a command to reattach the connection to the target.
ProcessDefinitionXYLayoutEditPolicy	Implements the policy of placing a graphical element on a diagram. Contains methods that: – create a command to add a descendant; – get a command to change the node area of a descendant; – get a command to create a descendant node; – create a command to delete a descendant.
TransitionConnectionBendpointEditPolicy	Implements the policy of bendpoint management for a transition. Contains methods that: – get a command to create a bendpoint for a transition; – get the command to delete a bendpoint for a transition; – get the command to move a bendpoint for a transition.
TransitionConnectionEditPolicy	Implements the policy of transition editing. Contains a method that gets a command to delete the attachment of a transition to an element.
TransitionConnectionEndpointsEditPolicy	Implements the policy to manage the endpoints of a transition connection. Contains methods that: – add descriptors of the endpoints (attachment points) of a transition; – delete descriptors of the endpoints (attachment points) of a transition.

5 CHANGING GPD FUNCTIONALITY

5.1 Changing Graphical Element Representation

Representations of graphical elements are contained in package org.jbpm.ui.figure. An inheritance diagram and a description of representation classes of the org.jbpm.ui.figure package can be found in section 4.1 of this document.

To set up parameters of representation figures for the graphical elements of the org.jbpm.ui.figure package, the figure classes override the methods of class org.eclipse.draw2d.Figure. In a general case, to change a graphical representation it is necessary to override the paintFigure method in the descendants of class Figure from the draw2d library.

To display figures, the methods of class org.eclipse.draw2d.Graphics are used.

Table 9. Figure classes and image creation methods. lists classes of the org.jbpm.ui.figure package and their methods that implement the figures, used for process modeling in RUNA WFE. To change the representation of graphical elements it is necessary to make changes to the classes provided.

Table 9. Figure classes and image creation methods.

Class	Image creation methods
StateFigure	The void paintFigure(Graphics g) method gets a Graphics class object from the draw2d library to draw a figure. The method forms the figure area (a minimal rectangle surrounding the figure), based on the top left-hand corner coordinates, width and height specified. The drawRoundRectangle method of the Graphics class draws a softbox (a rectangle with rounded corners), formed and passed to it.
DecisionFigure .	The void paintFigure(Graphics g) method gets a Graphics class object from the draw2d library to draw a figure. The method forms a minimal rectangle around the figure (the figure area) and calculates the coordinates of the middle points of the sides of the rectangle that are passed to the drawPolygon method of class Graphics in the form of an array. The drawPolygon method draws a diamond based on the specified corner coordinates.
ForkJoinFigure	The void paintFigure(Graphics g) method gets a Graphics class object from the draw2d library to draw a figure. The method forms the figure area (a minimal rectangle surrounding the figure), based on the top left-hand corner coordinates specified. The setBackgroundColor and fillRectangle methods of the Graphics class specifies black as the fill color and fills the rectangle with this color.
StartStateFigure	The void addEllipse() method draws a circle, using the Ellipse class constructor from the draw2d library, fills it with black and sets its size and layout.
EndStateFigure	The void addEllipse() method draws a circle, using the Ellipse class constructor from the draw2d library, fills it with black and sets its size and layout. To draw an internal black circle, the method calls the Ellipse class constructor again, fills the circle with black, sets a smaller size and adds the new circle into the circle, created earlier.

5.2 Adding a New Graphical Element

New elements of the RUNA WFE Graphical Process Designer must be based on GEF. GEF controllers link elements of a business process model with their graphical representations.

5.2.1 Creation of a Model Element

Classes of GPD model elements are contained in the org.jbpm.ui.model package.

To create a new model element, it is first necessary to select the base class, depending on the purpose of the new element. A class inheritance diagram is shown in Figure 5. Inheritance of graphical element classes of the org.jbpm.ui.model module. A list of classes of the org.jbpm.ui.model package and their description are shown in 4.3 of this document. End classes («leaves») in a class hierarchy implement the current set of elements of a RUNA WFE model.

To create a new model:

1. Create a model element class that inherits from a class with the most appropriate properties (see Table 5. Classes of org.jbpm.ui.model package);
2. Override base class methods that do not fit or are not implemented;
3. Define methods to add new functionality to the class being created.

5.2.2 Creation of a Graphical Representation of a Model Element

Graphical representations of model elements depend on the model element notation selected in RUNA WFE.

Graphical representation classes are contained in the org.jbpm.ui.figure package.Since the graphical representation of a model must show interdependencies between model elements, graphical representation classes are arranged in a hierarchy. An inheritance diagram for graphical representation classes is shown in Figure 4. Inheritance of classes of graphical element views of the org.jbpm.ui.figure module. A list of classes of the org.jbpm.ui.figure package and a description of these classes are shown in section 4.1 of this document**.** End classes («leaves») in a class hierarchy implement the current set of elements of a RUNA WFE model. Just like with a new model element, it is first necessary to select a base class, depending on the purpose of the element being created

To create a new graphical representation (figure):

1. Create a model element class that inherits from a class with the most appropriate properties (see Table 4. Classes of org.jbpm.ui.figure.);
2. Override base class methods that do not fit or are not implemented; The methods to create figures for leaf classes of graphical representations are described in Table 9. Figure classes and image creation methods.;
3. If necessary, define methods, adding new functionality to the Figure class being created.

5.2.3 Adding a Graphical Representation to the Tool Palette

Graphical representations of elements are added to a process diagram in RUNA WFE with the help of a tool palette. A tool is added to tool palette in the extension point org.jbpm.ui.elements, using the editor of the manifest file of the plugin.xml module.

To add an element to the palette:

1. Go to the Extensions tab of the manifest editor and add the element in the context menu of the org.jbpm.ui.elements extension point (right-click on the extension point).
2. In the Extension Element Details section enter the element name and select “contributor”. Select the following in the window of the Java Attribute Wizzard:

   – package: org.jbpm.ui.contributor;

   – contributor class name;

   – In the text window “Interfaces” add an ElementContributor interface, implemented by the class. As a result, a class to implement this interface will be generated.

3. Implement contributor class methods to create instances of the following:

   – model elements;

   – graphical controller of a model element;

   – hierarchical controller of a model element;

   – figures for graphical representation of a model element.

To create instances, constructors of corresponding base classes are often used.

5.3 Adding a New Menu Item

The RUNA WFE GPD menu is contained in the plugin.xml descriptor of the ru.runa.jbpm.ui plug-in. The ru.runa.jbpm.ui plug-in defines the items of the GPD menu by adding functionality in the extension point org.eclipse.ui.actionSets of the org.eclipse.ui plug-in.

Add and edit the items of the GPD menu on the Extensions tab of the manifest editor of the module that is part of the Eclipse development environment. Note that these menu items by themselves have no application functionality. They are rather a means to structure functional elements “Action” that can be considered as end items of the menu (“leaves” in the menu hierarchy).

To add a menu item:

1. Open the plugin.xml file of the ru.runa.jbpm.ui plug-in in the module manifest editor.
2. Create the required menu item. In the context menu of the Main Menu element of the org.eclipse.ui.actionSets extension point choose New > Menu. The module manifest editor will create a new menu extension element, and the properties of this new element will appear on the right of the editor window:

   – id – a unique identifier of the element;

   – label – a label displayed on the element;

   – path – path to the menu item in the menu hierarchy.
3. Add a separator element to the menu item just created. To make the new menu item available to other plug-ins for extension, the separator must have the name “additions”.

To add an action:

1. In the context menu of the Main Menu element of the org.eclipse.ui.actionSets extension point choose New > Action. The module manifest editor will create a new action extension element, and the properties of this new element will appear on the right of the editor window:

   – id – a unique identifier of the element;

   – label – a label displayed on the element;

   – accelerator – obsolete/deprecated;

– definitionId – the id of the command associated with this action;
– menubarPath – path to the action in the menu structure;
– toolbarPath – path to the action in the toolbar;
– icon – a relative path to the image file for this element;
– icon – a relative path to the image file of an inactive element;
– hoverIcon – a relative path to the image file of the element, when the mouse pointer is over it;
– tooltip – the text of a popup tip;
– helpContextId – the context help identifier;
– style – an action representation attribute (push, radio, toggle, pulldown);
– state – start state attribute (optional);
– pulldown – obsolete/deprecated;
– class – full path to the action handler class. The class must implement the org.eclipse.ui.IWorkbenchWindowActionDelegate or

org.eclipse.ui.IWorkbenchWindowPulldownDelegate interface. This attribute is ignored, if the "retarget" attribute is set to true;

– retarget – if set to true, the global action handler is used;

– allowLabelUpdate – used if the "retarget" attribute is set to true. If this element is set to true, the "label" and "tooltip" attributes of this action are replaced with global handler attributes;

– enablesFor – ignored if not specified. Determines the number of elements that must be selected to perform this action.

2. After entering the full path to the action handler class ("class" attribute), select "class". Eclipse will create an action handler class with stub methods, using the path specified. To specify the functionality of the action, it is necessary to implement the functionality of these methods.

5.4 Adding a New Element in the "V" Element of the Form Designer

Editing vartags

The vartags.xml file (located in tk.eclipse.plugin.wysiwyg/vartags) contains all available vartags in the following format:

<vartag type="ru.runa.wf.web.html.vartag.ActorComboboxVarTag" image="ChooseActor.png" width="160" height="27" />

type – real java type of class VarTag, mandatory

image – an image name for graphical representation only, optional

The images are stored in the tk.eclipse.plugin.wysiwyg project in the *FCKeditor2.2\editor\plugins\RunaVarTags\im* folder.

Images can also be localized, which means that different images will be used for different locales. Thus, in our example we can put the ChooseActor.png and ChooseActor.ru.png images into the required folder.

If the user locale is RU, ChooseActor.ru.png will be used, otherwise it will be ChooseActor.png.<vartag type="ru.runa.wf.web.html.vartag.ActorComboboxVarTag" image="ChooseActor.png" width="160" height="27" />

The general format for the name of an image file is ${fileName}.(locale).${fileExtension}.

width – width, optional, the default is 200

height – height, optional, the default is 30

Vartag names are also required for image display and a selection list. They must be specified in messages(.*).properties localization files in the following format:

ru.runa.wf.web.html.vartag.GroupMembersAutoCompletionVarTag=Group Members Auto Completion VarTag

That is, the string key is Java type VarTag.

If the list does not change after editing, the $
{gpd}/workspace/.metadata/.plugins/tk.eclipse.plugin.wysiwyg folder must be removed. It will be recreated.

5.5 Adding a New Element in the "F" Element of the Form Designer

Editing ftl.methods.xml.

The ftl.methods.xml file (located in tk.eclipse.plugin.wysiwyg/vartags) contains all freemarker tags, available in GPD, in the following format:

```
<tag tagName="AAA" displayName="Name" image="A.png" width="250" height="40">
      <parameter name="Param1" type="combo" values="${variables}.long" />
      <parameter name="Param2" type="combo">
            <value name="all" displayName="Presentation.ListAll" />
            <value name="raw" displayName="Presentation.ListRaw" />
      </parameter>
</tag>
```

tagName – a tag name; a tag with the same name must be registered in the WFE system.

displayName – a tag name to be displayed in the editor window; can be localized using messages(.*).properties.

image – an image name for graphical representation only, optional

The images are stored in the tk.eclipse.plugin.wysiwyg project in the *FCKeditor2.2\editor\plugins\RunaVarTags\im* folder.

Images can also be localized, which means that different images will be used for different locales. Thus, in our example we can put the A.png and A.ru.png images into the required folder. If the user locale is RU, A.ru.png will be used, otherwise it will be A.png.

The general format for the name of an image file is ${fileName}.(locale).${fileExtension}.

width – width, optional, the default is 250

height – heght, optional, the default is 40

```
parameter (0..*) — tag parameters

displayName — a parameter name to be displayed; can be localized using
messages(.*).properties
type – parameter type; the types “text”(user input) and “combo” (list) are
supported
values — a possibility to specify a list of variables of a certain type, if
“type” is “combo”, optional.
```

If the list does not change after editing, the $

{gpd}/workspace/.metadata/.plugins/tk.eclipse.plugin.wysiwyg folder must be removed. It will be recreated.

1.1.5.6. Using FreeMarker in Forms

Specify the form type FreeMarker (ftl) at the form creation time and then write HTML + FreeMarker in this form. There are tags that are executed on the side of the WFE server (you can create and add new tags for your needs, and these tags will have access to the entire system). There is also object formatting, predefined in FreeMarker (see freemarker BuiltIns) and extended in the editor by adding certain formats.

Forms can be created either in the text mode (understanding of freemarker is recommended) or in the graphical mode. Note that your possibilities in the graphical mode are limited to the functionality available from the menu items (tags, output of variables). You cannot build logic (cycles, conditions) and use a built-in EL (expression language), but this is not important for base forms.

2. 6 INTERACTION WITH INFOPATH (WINDOWS ONLY)

6.1 Architecture

Microsoft InfoPath 2007, included in the Microsoft Office 2007 distribution disk, has been selected as a graphical designer. It allows to create form templates, using a standard and a customizable form element palette. To use the form elements, specific to RUNAWFE, we will use elements that will be added to the InfoPath palette.

Attention! For the system to work correctly, the RunaGPDInfoPathSupport.dll is required, stored in C:\WINDOWS\SYSTEM32. This library allows to work with archive file forms (**XSN**), stored in the **CAB format**.

Standard InfoPath Elements Supported by RUNAWFE

InfoPath element	Description
Text Box	Single-line text input
Drop-Down List Box	A drop-down list

Check Box	A checkbox
Rich Text Box	A field for multi-line text input
Date Picker	Selecting a date with the help of a calendar
File Attachment	Uploading and downloading a file
Image	An image on a form
Hyperlink	Opens in a new window

In addition to these standard elements, InfoPath allows to add other elements based on TemplatePart (templates) or ActiveX (COM components).

6.2 Creation of InfoPath ActiveX Elements

Microsoft Visual Studio 2007 development environment is used.
The added InfoPath elements are ActiveX objects, implementing UserControl, IObjectSafety and ICOMControl interfaces. They must all gave a unique GUID in the COM model. The utility to create a GUID is called GUIDGEN.EXE and delivered with MS Visual Studio. Net Framework will be used for simplicity, though it is not necessary.

A new project of type **Windows Class Library** is created in MS Visual Studio (Visual C# projects).
In the properties of the created object on the tab Configuration Properties > Build select **Register for COM Interop = true**.
Create a new **GUID**.
Make a copy of class "template", implementing the InfoPath element, and change the GUID
Change the **OnPaint** method (optional). This method is called each time an element is displayed on an InfoPath form. To give the element a beautiful appearance, draw it here.
Note the **OnSizeChanged** method of the class. It is called each time an InfoPath user tries to change the size of an element on a form. Thus, you can explicitly control the size of an element on a form.
Having done all this, press Project Build and wait for compilation and creation of a new DLL, ready to be installed on a local computer.
For installation on other computers, additional steps are required. Sign DLL (**Strong Name**) – generate a key file (sn.exe –k keyfile.snk) with **sn.exe** (a VS utility), add it into the project and

write the values into the AssemblyInfo.cs file [assembly: AssemblyKeyFile("..\\..\\keyfile.snk")], [assembly: AssemblyKeyName("ActorFullNameDisplay")] instead of default values.
The element should then be added into GAC by **gacutil.exe** (a VS utility) and registered as COM by **regasm.exe (**a .Net utility**)**.
The component business logic can access RUNAWFE and is not limited in functionality.
From the MS InfoPath 2007 interface on the Controls tab, we can register ActiveX components as elements by using Add or Remove Custom Controls, but not ActiveX components in the .Net category. For this category, the user has to write component descriptions manually and place them in a certain location.
The path to the description files of customizable InfoPath elements:

${USER_HOME}\Local Settings\Application Data\Microsoft\InfoPath\Controls

Example:
C:\Documents and Settings\dofs\Local Settings\Application Data\Microsoft\InfoPath\Controls

Element descriptors must have an **.ict** extension for InfoPath to read them correctly while loading.

A convenient name for a file in this folder is
{GUID}.ict (for example, {C4134657-1B43-4968-9913-FAE952F685A0}.ict), where GUID is the GUID of the corresponding InfoPath element.

The file must be in UTF-8 coding.
The file format is as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<ict:control name="CONTROL_NAME">
	<ict:designTimeProperties>
		<ict:iconBitmap>CONTROL_ICON</ict:iconBitmap>
	</ict:designTimeProperties>
	<ict:deployment>
		<ict:controlPackage classid="{ACTIVEX_GUID}"/>
	</ict:deployment>
</ict:control>
```

where
CONTROL_NAME = element name in the palette
CONTROL_ICON = an icon in the palette; represented in the Base64 format in the file
ACTIVEX_GUID = GUID ActiveX.

If the file has been created incorrectly, InfoPath will output an error message with details of the error at the next run. If there is no message and the element is not added into the palette, check that the .ict file is in the right folder.

Use existing components as examples.

6.3 Creation of InfoPath Template Parts Elements

These components are limited by the functionality of the DataSource, that will be processed at form initialization in the run mode. This element type is useful to output different lists (used to output actors).

Publication consists of the registration of **XTP** template files in InfoPath from the “update/delete elements” menu.

Use existing templates as an example.

3. 7 RCP APPLICATION ASSEMBLY IN GPD

GPD is designed to work as a separate RCP (Rich Client Platform) application/product. RCP application assembly is required after making changes to GPD modules.

RCP application is exported as a file with the .product extension. A product file can be created automatically by Eclipse at the time, when a module project is created, based on an RCP application template. A product file can also be created at the time when the product configuration is set up. A product file can be viewed and edited in the product file editor of the Eclipse development environment. The product file editor contains the tabs “Overview”, “Configuration” and “Brand”.

On the Overview tab of the product file editor the following is specified:

– product ID;

– the application to run when the product is launched;

– the name displayed in the application header.

The “Synchronize” link in the “Testing” section is used to update the plugin.xml descriptor of the main module of the product to reflect the changes made to the product modules

in the development environment. The "Launch the Product" and "Launch the product in Debug Mode" options allow to test an RCP application without exporting it.

In the "Exporting" section, the Eclipse Product Export Wizard allows to set up export parameters and export an RCP application, based on the configuration defined on the Configuration tab.

On the Configuration tab in the "Modules and Fragments" section, the modules constituting the RCP application are specified. After the main module is specified, the rest of the required set of modules can be defined automatically. The way in which the RCP application is to be launched is specified in the "Configuration File" and "Launch Arguments" sections.

To assemble an RCP application of the Graphical process Designer, do the following:

1. Open the org.jbpm.ui.gpd.product product file. Set the following parameters on the Overview tab in the "Product Definition" section:
   – Product ID: org.jbpm.ui.RUNA;
   – Application: ru.runa.jbpm.ui.bp editor;
   – Product Name: Runa WFE GPD;
   – The product configuration is based on a plug-in.
2. In the "Testing" section click "Synchronize" to synchronize the changes with the main module of the product.
3. To add newly created modules (if any) to the set, click Add on the Configuration tab and select the necessary modules from the list.

   Eclipse allows to recreate the list of the required modules for the RCP application. To do this:
   – delete all the modules by choosing Delete All;
   – add the main module of the RCP application to an empty list;
   – select "Add Required Modules".
4. On the "Overview" tab in the "Exporting" section select Eclipse Product Export Wizard.
5. In the wizard window:
   – specify the full path to the target export catalog;
   – specify compatibility with the target Java machine in the compiler options;
   – press Finish.

   The RCP application will be exported using the path specified.

To export RCP applications to Linux, Mac OS X and Solaris platforms, the Eclipse-RCP-delta-pack plug-in must be installed in the Eclipse development environment. The Eclipse-RCP-delta-pack plug-in can be installed by choosing Help > Software Updates > Manage

Configuration in the main menu to update the installed Eclipse platform and by choosing the plug-in from a plug-in list.

Besides, for the Eclipse 3.1.2 platform, the Eclipse-RCP-delta-pack package can be downloaded form http://archive.eclipse.org/eclipse/downloads/drops/R-3.1.2-200601181600/index.php.

Once Eclipse-RCP-delta-pack is installed in the product file editor, a tab is added to launch the RCP application on the selected platform.

4. 8 REFERENCES

1. Almost exhaustive information on the technologies mentioned in this document is available at http://www.eclipse.org/. Besides, the Eclipse development environment has a well-developed help system that includes reference information on the technologies used, links to information resources and examples. Update plug-ins, that can be loaded from http://www.eclipse.org/, usually include help data that is added into the Eclipse help system automatically.

3. The book “Building Commercial Quality Eclipse Plug-ins” by Eric Clayberg and Dan Rubel is recommended to plug-in developers. Publisher: Addison WesleyProfessional.ISBN: 032142672X; Published: Mar 22, 2006; Copyright 2006; Dimensions 7x9-1/4; Pages: 864; Edition: 2nd..

2. Information on OSGi Framework can be obtained from the OSGi alliance site at: http://www.osgi.org/osgi_technology/index.asp?section=2 .

3. GEF documentation is available at: http://www.eclipse.org/gef/reference/articles.html. Useful information for developers is contained in http://wiki.eclipse.org/index.php/GEF_Developer_FAQ and http://wiki.eclipse.org/index.php/GEF_Troubleshooting_Guide#Draw2D_common_mistakes. An example of using GEF components to create a database schema editor can be found in at http://www.eclipse.org/articles/Article-GEF-editor/gef-schema-editor.html.

4. GEF tutorials:
http://www-128.ibm.com/developerworks/opensource/library/os-gef/
http://eclipsewiki.editme.com/GefDescription