STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLVII**, Number 1, 2002

DIAGRAM DESIGN IN OCL EVALUATOR

HORIA CHIOREAN

Abstract. OCL Evaluator is a tool developed by the Computer Research Laboratory of the Babes-Bolyai University (LCI), designed for checking UML models stored in XMI format. The main purpose of this software is that of checking (verifying). The general use case scenario is: the user designs his application using a case tool like Rational Rose, Together, etc. Once this design faze is finished, the user has a model for the problem. This model is exported using the same case tool or some other program into an XMI document. The XMI document is inputted into our tool and verified. These verifications are done according to certain rules, that can be very complex and are grouped into several categories.

On the other hand, once errors are reported for a model, it's natural to allow the user to correct these errors. This is where the diagrams "kick in". Diagrams can simplify very much the process of error correcting, because they can provide a visual representation for a model.

The problem was to implement these diagrams in OCL Evaluator, using the Java programming language.

1. Achievements regarding OCL

The Object Constraint Language (or OCL) first appeared in 1997 as part of UML's 1.1 specification. OCL is a formal language used to express constraints. These typically specify invariant conditions that must be satisfied by the system being modeled.

Although it has been developed some years ago, very little support has been given to OCL in the sense that there are very few software tools that give the user the possibility to check his/hers model using the object constraint language. Well known CASE tools, such as Rational Rose, have plugins that allow certain verifications to be done using OCL, but these plugins are by no means enough in order to use OCL's the full power.

2000 *Mathematics Subject Classification.* 68N30.

1998 *CR Categories and Descriptors.* D.2.3 [**Software**] : Software Engineering – *Coding Tools and Techniques*; D.2.7 [**Software**] : Software Engineering – *Distribution, Maintenance and Enhancements* .

2. The importance of UML diagrams

A diagram (in general) could be considered a graphical representation of certain elements together with their relations. The reason why diagrams are so important is because they provide a graphical representation for a system (or for a part of the system). Having such a representation, makes it a lot easier to understand how that system works.

The Unified Modeling Language (or UML) is, as its name states, a modeling language. In other words, it is a language that is used to design models for problems. (by problems, we mean software problems, that can be solved using an Object Oriented approach) It is natural for such a language to have diagrams in its specification and therefore UML (in the 1.4 specification), defines the following kinds of diagrams:

(1) Static Structure Diagrams – class diagram and object diagram.
(2) Use Case Diagrams – the use case diagram.
(3) Interaction Diagrams – collaboration diagram and sequence diagram.
(4) State Charts Diagrams – the state chart diagram.
(5) Activity Diagrams – the activity diagram.
(6) Implementation diagrams – the component diagram and the deployment diagram.

Each of these diagrams contains several elements and relations, according to their type, that are abstract elements defined in UML's specification. (elements like classes, actors, use cases, messages, objects, associations, etc) Each and every one of those, has a graphical notation like: rectangles for classes, lines for associations, ovals for use cases, etc, graphical notations that are used in a diagram.

The diagrams should give the user the possibility of representing only parts of the model or the model in it's entirety. This means that a diagram should have the following features:

(1) To allow one or more elements to be represented in more then one diagram. In other words, an element can be represented in multiple diagrams.
(2) Deleting one/more elements from a diagram, without affecting the model.
(3) Undo/Redo functionalities.
(4) Element filtering – this is a very important feature, because it allows the user to see only the part of that element that he is interested in. Plus it can simplify a great deal the situation when you have very large diagrams.

Most of the well known CASE tools: Rational Rose, Together, Use, etc. support diagrams, but each has their drawbacks.

3. Our solution: UML diagram design in OCL Evaluator using JGraph

We developed the only tool, OCL Evaluator, that is based on the object constraint language, with the sole purpose of providing efficient support for model checking. Although there are several OCL compilers out there (like the Dresden OCL Compiler or IBM's OCL Compiler), none provide an adequate user interface and are therefore very difficult to use. OCL Evaluator is based on our own OCL compiler, but also has an extensive user interface that facilitates the process of checking. Moreover, the user not only has the possibility of checking a system, but also to correct that system based on eventual errors.

In this context, it was decided that to include a graphical representation of a system, by means of diagrams, was very important because firstly, it would facilitate the checking process a great deal by giving a visual representation and secondly, no other OCL software had this facility.

Although only class diagrams and use case diagrams have been implemented, most of these diagrams have a thing in common: they can be looked at similar to a graph. In other words, the structure of a diagram is similar to that of a graph where the objects are vertices (cells) and the relations between them are edges.

Therefore, when it was decided to include diagrams in the OCL Evaluator, there were 2 options: either to implement a graphical library from scratch or to use an existing graph library and to modify it so that it would fulfill the need of representing a good UML diagram. The later was chosen in the end and the graph library chosen was JGraph (`http://www.jgraph.com`) .

JGraph is a freeware, Java based library, used to represent graphs. The intention behind it, is to provide a freely available and fully Swing compliant implementation of a graph component. As a Swing component for graphs, JGraph is based on the mathematical theory of networks, called graph theory, and the Swing user interface library, which defines the architecture. By combining these two, JGraph becomes a Swing user interface component used for visualizing graphs.

The design of the JGraph is similar to that of a Swing component. In other words, besides the fact that JGraph **is** a Swing component (because it subclasses JComponent), its architecture is based on Swing Model View Controller (or MVC).

Figure 1 shows, according to [1], the diagram of a JComponent (JTree), which shows the way in which the MVC pattern is applied.

On the diagram, you clearly see all the participants (except the controller): the component itself - JTree, its UI (which has the responsibility of rendering the component) - TreeUI, and its model which encapsulates all the information – DefaultTreeModel. The control in Swing MVC is encapsulated in the UI-delegate, which is in charge of rendering the component in platform-specific manner, and mapping the events from the user interface to transactions, that are executed on the model.

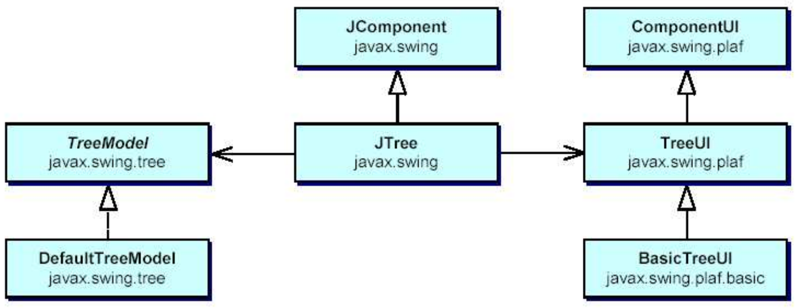


FIGURE 1. JTree MVC architecture

The JGraph component inherits this basic setup from the JComponent class and its UIdelegate, which implements the ComponentUI abstract class.

As in the case of text components, the split between platform-dependent and non-platform Design and Implementation of JGraph dependent attributes, is implemented using the concept of views, which are independent from the elements that appear in the model.

In Swing's text components, the elements of the model implement the Element interface, and for each of these elements, there exists exactly one view. These views, which implement the View interface, are either accessed through a mapping between the elements and the views, or through an entry point called root view, which is referenced from the text component's UIdelegate.

JGraph has an analogous setup, with the only difference that a graph view is referenced by the JGraph instance. The cells of the graph model implement the GraphCell interface, which is JGraph's analogy to the Element interface. The cell views implement the CellView interface, in analogy to Swing's View interface. The cell views are either accessed through the CellMapper mechanism, or through the graph view, which is an instance of the GraphView class. However, since the GraphView class works together with other classes, the analogy with Swing's text components is more helpful to understand the separation between the cell and the view.

In contrast to text components, where the geometric attributes are stored in the model only, JGraph allows to store such attributes separately in each view, thus allowing a graph model to have multiple graphic configurations, namely one for each attached view.

Figure 2 shows JGraphs's model view controller diagram, according to [1].

The key elements in a JGraph are the GraphCells. These cells are inserted into the GraphModel and using the GraphUI they are given a visual representation. There a 3 kinds of GraphCells: Vertexes, Edges and Ports. While vertexes and

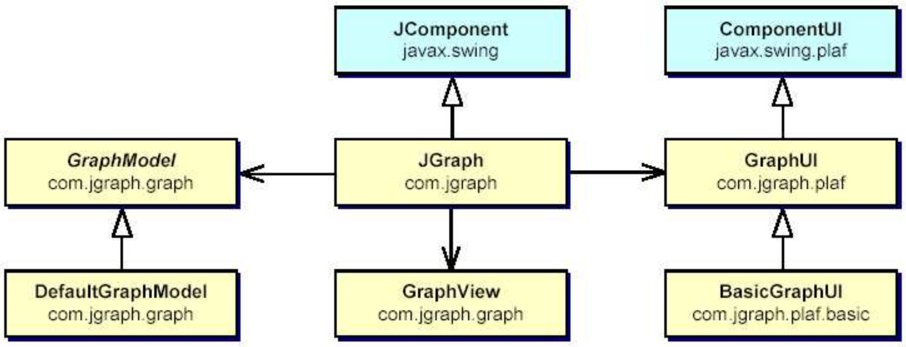


FIGURE 2. JGraph MVC

edges are the common elements of a graph, the port is new concept. In JGraph, a port is connection point for an edge. Ports are added as children to vertexes, providing a way for an edge to connect to a vertex. The graphical representation of a GraphCell is achieved using CellView. CellView is the base class for all the special views such as: VertexView, PortView and EdgeView.

The graph cell has 2 very important attributes: it holds a reference to an Object (referred to as the user object) and it has a corresponding View. This view is in fact an instance of the JGraph's CellView interface and it also holds a reference to a Renderer (which normally is a subclass of JComponent). This renderer is responsible for the painting. For each cell in the graph model, there exists exactly one cell view in each graph view, which has its own internal representation of the graph model. The renderers are instantiated and referenced by the cell views.

Renderers are based on the idea of the TreeCellRenderer class from Swing, and on the Flyweight design pattern. The basic idea behind this pattern is to "use sharing to support large numbers of fine-grained objects efficiently."

Because having a separate component for each CellView-instance would be expensive, the component is shared among all cell views of a certain class. A cell view therefore only stores the state of the component (such as the color, size etc.), whereas the renderer holds the component's painting code (for example a JLabel instance – in the case of Vertex).

The CellViews are painted by configuring the renderer, and painting the latter to a CellRendererPane , which may be used to paint components without the need to add them, as in the case of a container. The renderers in JGraph are used in analogy to the renderer in JTree, just that JGraph provides more than one renderer, namely one for each type of cell. Thus, JGraph provides a renderer for vertices, one for edges, and one for ports. For each subtype of the CellView interface, by overriding the getRenderer method, you may associate a new renderer. The renderer should be static to allow it to be shared among multiple instances of a class.

The renderer itself is typically an instance of the JComponent class, with an overridden paint method that paints the cell, based on the attributes of the passed-in cell view. The renderer also implements the **CellViewRenderer** interface, which provides the getRendererComponent method to configure the renderer. According to [1], Figure 3 shows the architecture of the renderers.

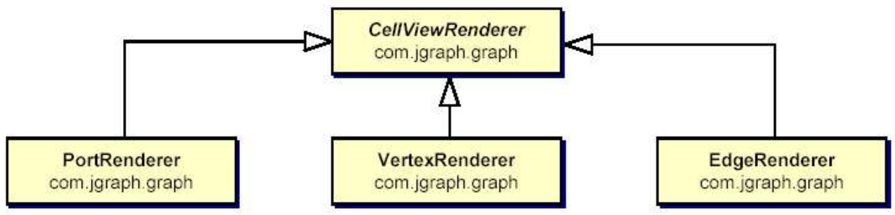


Figure 3. JGraph's renderers

The three default implementations of the CellViewRenderer interface are the VertexRenderer, EdgeRenderer and PortRenderer classes.

4. Our solution

There were 2 main challenges in using this library: firstly fixing some bugs that were present in the implementation of JGraph (the most important of which being an annoying flicker when dragging cells) and secondly, adapting JGraph so that it could be used for UML diagrams.

There were two main bugs in JGraph 1.0:

(1) When dragging a cell or an edge, or when changing the size of a cell, there would be a visible and annoying flicker on the screen.
(2) When trying to bend an edge, the connection point would not be inserted correctly. (bending edge is achieved by adding connection points to an edge and dragging those points).

Fixes:

(1) The overlay() method in the BasicGraphUI class did not use Swing's double buffering technique, this begin the reason for the flicker. The method was modified so that the painting would be done in an off-screen buffer and only after that displayed on screen.
(2) The OnMouseClick() method in the EdgeHandler class did not make correctly the calculations about the location of the control point.

In the current implementation of the Evaluator, the diagrams were implemented graphically by using the JDektopPane and JInterFrame Swing classes. This meant that a diagram would contain an instance of a JGraph and this instance would actually work as a canvas for JInternalFrame. This design allows (and so in should) the existence of multiple diagrams.

Figure 4 shows what the diagrams look like (when they are empty).

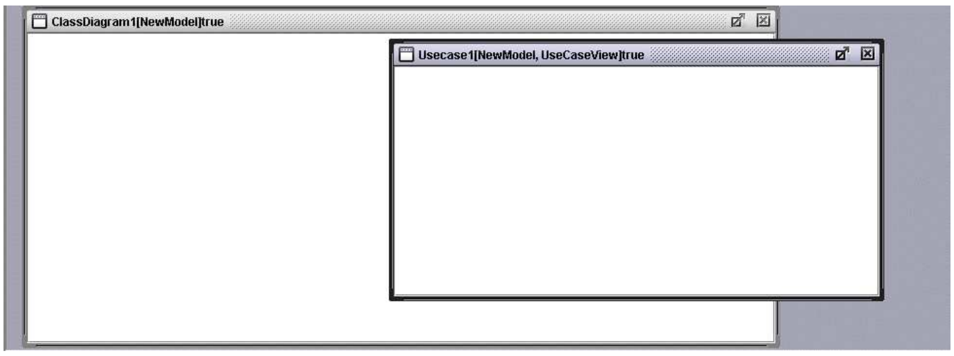

ClassDiagram1[NewModel]true
Usecase1[NewModel, UseCaseView]true

FIGURE 4. Diagram Overview in OCL Evaluator

Each element in a diagram, would have to be a cell in a graph. However, because JGraph's support for cells was limited to only one kind of cell - the Vertex, which is a rectangle with an optional text in the center, the cells implemented by us were: Class Cell, Package Cell, Actor Cell and UseCase cell.

Because JGraph uses the Factory method to create views for each type of cell, (this is achieved trough a method called createView in the JGraph class), we sub-classed JGraph and created our own DiagramGraph. This class represents the graph behind the diagram and it's responsible for creating the correct views for each kind of cell. We didn't implement a new GraphModel because the one provided by the library (DefaultGraphModel) was general enough for what we needed.

Therefore, we implemented only the necessary types of cells. Each one of Class Cell, Use Case Cell, Package Cell and Actor Cell represents the graph cell, and holds the user object.

Every custom cell, also has a corresponding view related to it, as shown in Figure 6:

Every renderer, has an appropriate paint() method. This method will be invoked by BasicGraphUI, when the rendering mechanism takes place. This is the place where we wrote the code that displays each cells according to its abstract counterpart from UML 1.4. The cell class holds a reference to an object for the model (the object which it represents). So when the paint() method is invoked by

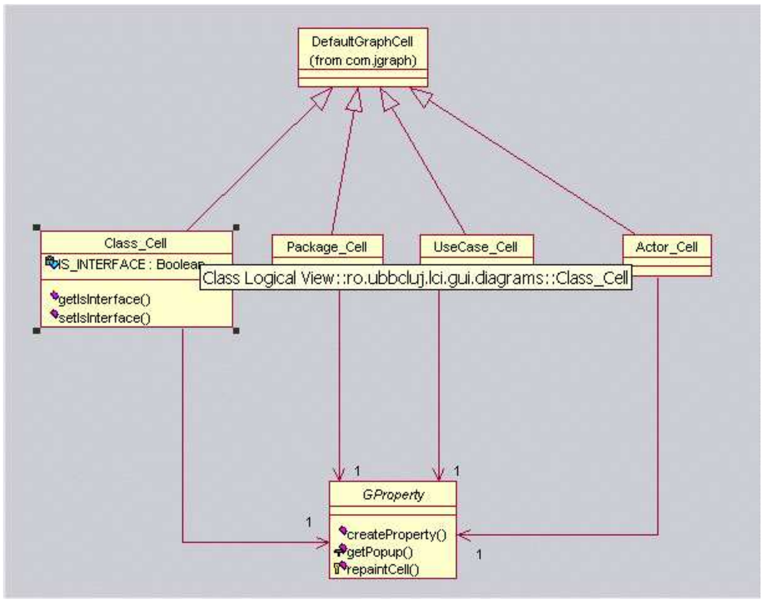


FIGURE 5. Diagram Cells implemented for OCL Evaluator

the view, the user object is accessed, and the painting is done according to the information extracted from this object. For example, in order to draw a class, before drawing it's outline (the rectangle), we take the name of the class, its attributes and methods from the object (which must be an instanceof Classifier), and draw them using the drawString() and drawImage() methods from java.awt.Graphics. This painting mechanism is used for every kind of cell, the only difference being the user object and the shape of the cell (shape which conforms to the UML 1.4 specification).

In addition to the having a view, we implemented two other classes for our cells. The first one called GProperty and encapsulates the graphical property for that cell – meaning the fill color, the outline color, the title font size, the body font size, etc. The second one is called AbstractFilter and represents a filter. This filter has been only implemented for ClassCell, and it allows the filtering of classes according to the visibility of the attributes and methods.

Holding a model element as a user object, provides greater flexibility in the sense that when this user object, which is always an instance of the Element interface from UML1.4, is modified, a simple repaint is enough to visualize the change in all the corresponding diagrams.

As far as the relations between elements are concerned, in any kind of UML diagram, these relations are represented by straight lines with possible decorations

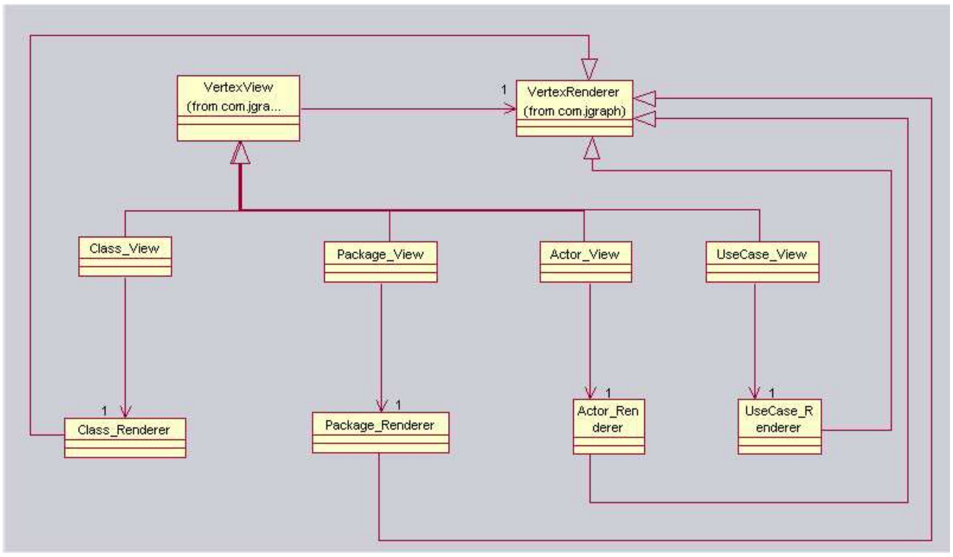


FIGURE 6. Custom Views and Renderers

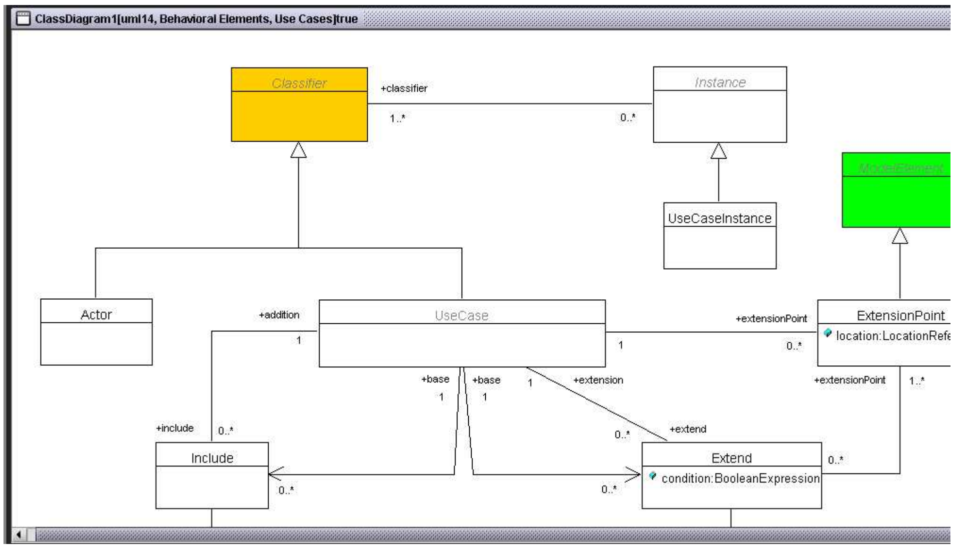


FIGURE 7. A class diagram

at the extremities. Already JGraph already had the support for this, there were two important issues that were solved:

(1) An Association in UML, besides being represented as a straight line, also has several additional text labels that indicate the role names and the multiplicity of that association. Therefore we implemented text labels that could be set at the end of the edge. Although these labels are not tied to the edge, by double clicking an edge, they will gather around that edge.
(2) See Figure 8.

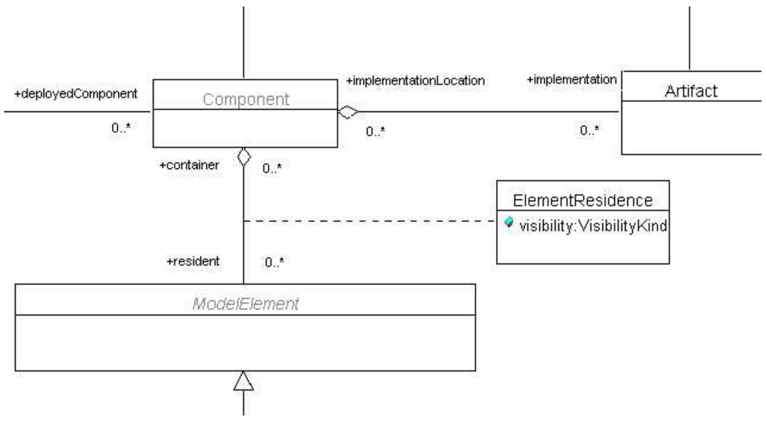


Figure 8. Diagram with an Association Class

An Association Class in UML is represented with a dashed line that connects a class with an association. This meant that we had to connect a cell with an edge using a second edge. We stated previously that the key to connecting edges are the ports. These ports act as a sort of "glue" in the sense that they keep edges connected. However, in the JGraph library the only cells that are allowed to have ports are the vertexes.

So, we needed some sort of hybrid edge that allowed the addition of ports as children on one hand, but would still have the capability to connect to vertexes together. We solved this problem by creating 2 special classes (with their corresponding views): SpecialEdge and SpecialPort, which extend DefaultEdge respectively DefaultPort. However, SpecialEdge's view class - SpecialEdgeView is similar to the view of a vertex making SpecialEdge therefore both an edge and a vertex. Figure 9 illustrates this.

The OCL evaluator is divided in 5 major parts, as shown in Figure 10.

You can create a diagram by right-clicking in the browser on the desired parent for the diagram (a package usually), and from the pop-menu selecting New. In order to add elements to the diagram, you can achieve this in 2 ways:

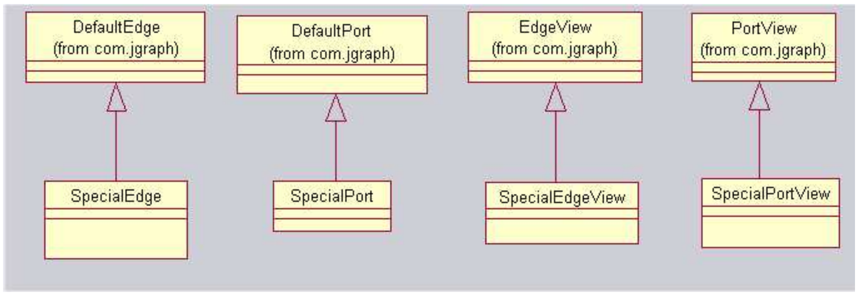
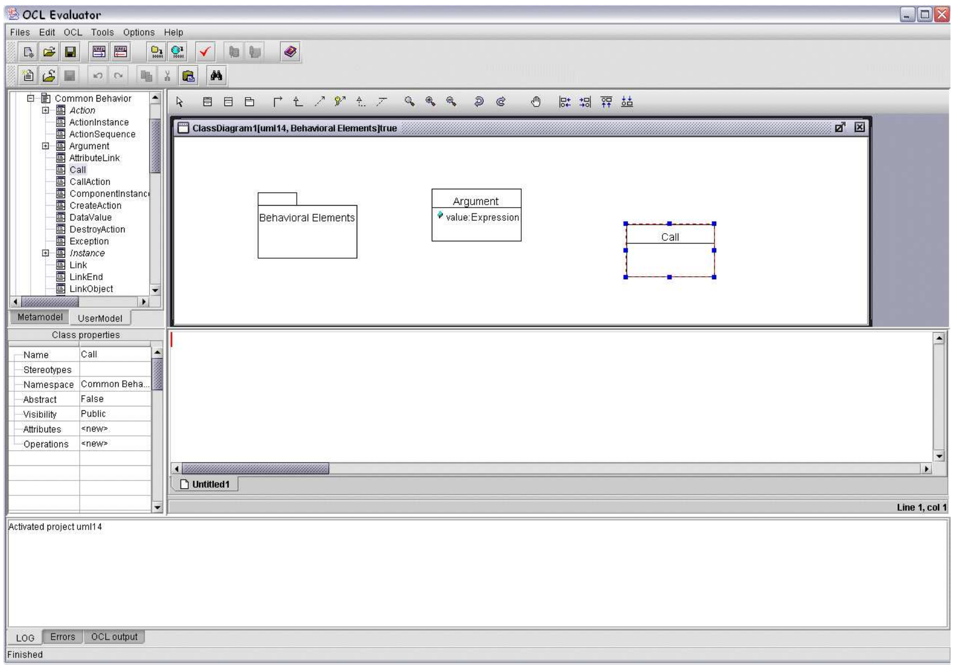

DefaultEdge (from com.jgraph)
DefaultPort (from com.jgraph)
EdgeView (from com.jgraph)
PortView (from com.jgraph)
SpecialEdge
SpecialPort
SpecialEdgeView
SpecialPortView

FIGURE 9. The Special Cells



Class properties

Name	Call
Stereotypes	
Namespace	Common Beha...
Abstract	False
Visibility	Public
Attributes	<new>
Operations	<new>

Line 1, col 1

Activated project uml14

Finished

FIGURE 10. OCL Evaluator Overview

(1) Dragging and dropping an element from the browser onto the diagram, in which case the element will be represented graphically according to its information.
(2) Using the toolbar to create new elements/relations.

5. Conclusions and future work

OCL Evaluator is not yet in a release version. It's still in beta version. However, several extensions of our project are possible such as: using OCL to verify components and using OCL for checking XMI documents.

Of course that the two types of diagrams that were implemented, although they are the most "fundamental diagrams", are not enough for a competitive tool. New diagrams will be implemented as the project extends. However, because JGraph's extensibility is limited, a decision will have to be taken whether to continue using this library, or to implement a new one.

References

[1] Alder. The JGraph Paper. `http://jgraph.sourceforge.net/paper.pdf`
[2] Alder. The JGraph 1.0 API Specification. `http://api.jgraph.com`
[3] Gamma, Helm, Johnson, Vlissides. Design Patterns. Addison-Wesley, Reading MA, 1995.
[4] Geary. Graphic Java 2, Volume II: Swing (3rd Edition). Sun Microsystems, Palo Alto CA, 1999.
[5] OMG Unified Modeling Language Specification, Version 1.4 draft, February 2001

Babeş-Bolyai University, Computer Science Research Laboratory, RO 3400 Cluj-Napoca, Str. Kogălniceanu 1, Romania