A graphical diagram editor plugin for Eclipse

Preface

Software development has come a long way. The ideal of «code wizards», locked away in their bedrooms writing code of arcane complexity which only they can understand, is long gone. Today, programmers are expected to make code which is reusable, understandable and easy for other developers to extend. Many software development projects in the modern world are conducted cooperatively, with developers spread across the globe. Yet more are based on previous development efforts, extending existing frameworks rather than building complete systems from the ground up.

With increased integration between software units, comes the need for more powerful tools to support it. One such tool is the Eclipse platform. The Eclipse platform is many things, but primarily, it is a developer environment. It features many useful tools to aid developers in structuring and understanding complex projects. One of its most attractive features, however, is its extensibility. Not only can new tools be added, they can be fully integrated with the rest of the environment, as well as with other tools.

The UIML Diagram plugin described in this document is such a new tool for Eclipse. The purpose of the plugin is to significantly ease the process of building a graphical editor for a modelling language. Instead of creating the entire editor from scratch, it allows developers to focus only on the parts which are unique to a specific modelling language, leaving responsibility for the generic functionality to the Diagram plugin. The UIML DiaMODL plugin is the first language-specific plugin for the UIML Diagram plugin. It adds the functionality related the the DiaMODL UI modelling language, and also serves as a practical example of how language specific plugins for the UIML Diagram plugin can be made.

My contributions to this project forms my master thesis as a computer science student. The majority of my work has consisted of adding to the project's code base, which can be browsed in the CVS repository at:
http://opensource.idi.ntnu.no/projects/diamodl

In addition, I have been responsible for writing the documentation for the project, which makes up the main part of this document.

Acknowledgements

I would like to thank associate professor Hallvard Trætteberg, who in addition to being the leader of the project, has been my tutor for the last six months. Skilled with both whip and carrot, he has kept me moving when I would have been tempted to give up. I am also grateful for his tutoring, which has allowed me to make significant academic progress over the last six months.

I would also like to thank my fellow students in computer room ITV-060 for creating a pleasant work environment.

Abstract

This document serves the dual purpose of being the first version of the documentation for the UIML DiaMODL and UIML Diagram plugins, as well as being the written part of my master thesis. The documentation part is the main body of the document. It covers the background for the project, and describes each of the plugins in detail, in terms of structure and behaviour. In addition, the documentation features a relatively detailed practical guide on how the UIML Diagram plugin can be extended with language specific plugins. As this is also part of my master thesis, there is also short summary of my personal experiences working on the project.

Contents

1. Introduction

In this chapter I will describe the background and purpose of the graphical diagram editor project, as well as give a brief introduction to the main supporting technologies.

1.1 Project background and goals

This project was started by associated professor Hallvard Trætteberg, with the aim of moving the editor for his UI modelling language, DiaMODL, to an opensource framework suited for sharing within research communities and industry, as well as educational use. Earlier versions had been developed which were based on JGraph (www.jgraph.com) and later Piccolo (www.cs.umd.edu/hcil/jazz/). In late december 2004, after a visit to Prof. Peter Forbrig in Rostock, who demonstrated an Eclipse-based task modelling editor, the work started on porting the editor to the Eclipse platform. Eclipse was chosen both for its technical features, such as OS independence and a highly extensible plugin architecture, and for its broad support in many development communities. In january 2005 I, Kay Are Ulvestad, joined the project. In addition to programming work, which has taken most of my time, I am also charged with the task of writing the documentation. This documentation forms the main body of this document.

In addition to developing the DiaMODL editor, it is a goal for the project to separate the modelling language independent parts into a separate plugin, which may be used as a base for developing editors for other modeling languages.

1.2 Supporting technologies

A graphical editor is a reasonably complex system, and would take a significant amount of time to build from the ground up. Fortunately, many of the problems have already been solved by other developers. The editor is thus based on several existing software modules, most prominently the Eclipse platform, the Eclipse Modelling Framework, and the Graphical Editor Framework. These are briefly introduced below. A simplified illustration of the relationship between these modules and our editor is shown in figure 1 on page X.

1.2.1 The Eclipse Platform

The eclipse.org website defines the Eclipse Platform as «an open extensible IDE for anything and yet nothing in particular». In basic terms it is a developer platform, which provides a broad set of strongly integrated development tools, as well as a sophisticated integrated development environment (IDE). Probably the greatest quality of the Eclipse Platform is in its expandability. Eclipse features a plugin architecture which enables developers to make tools which are integrated

with, and often expand upon the functionality of, other tools. This integration happens through clearly defined mechanisms, without the need for explicit cooperation between developers. A plugin for Eclipse can range in size from small decoration in an excisting Eclipse Workbench view, to a full-featured editor.

1.2.2 The Eclipse Modelling Framework

The Eclipse Modelling Framework (EMF) is a tool for working with structured data models. Given a model specification, it generates a set of Java classes which provide support for model viewing and manipulation. Among the problems solved by EMF, are the following:

- Generating a runtime representation of the model.
- Providing methods for model manipulation.
- XMI serialization of model data.

1.2.3 The Graphical Editor Framework

The Graphical Editor Framework (GEF) provides functionality to create graphical editors based on an excisting application model. It is based on the Model-View-Controller (MVC) architectural pattern, and provides several useful features for our editor:

- A 2D drawing API, with many special features which are useful in a graphical editor, such as the concept of anchor and connection routing.
- A framework for the graphical editing and viewing of a model.

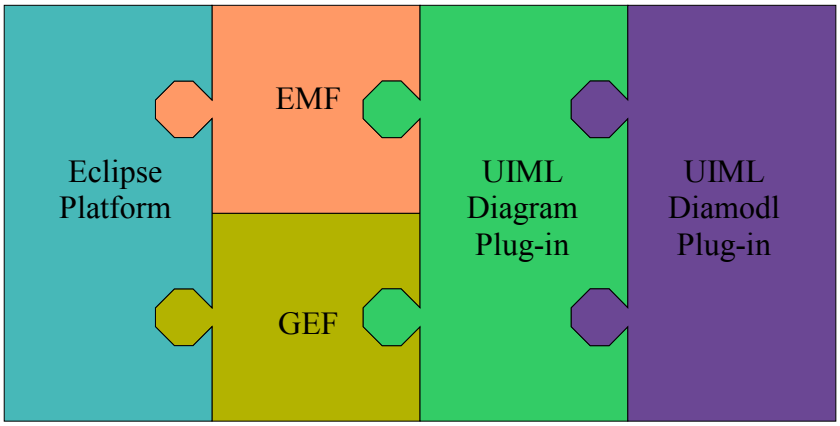


Figure 1 - Dependencies between the Eclipse Platform , EMF, GEF, and the UIML plugins

1.3 Similar projects

There are other projects underway which have goals that are similar to those of ours. The most significant of these is probably GMF. The GMF, or «Graphical Modelling Framework», project aims to make a generic bridge between GEF and EMF, to allow automatic generation of graphical editors from language meta-models. GMF was launched after our own project, and is currently in the early validation phase. The biggest difference between this project and our own, is that our project is based on the concept of a combined diagram model and core generic semantic model. Rather than generating an editor for a specific language, we offer a *generic plugin* which provides the functionality which is common to most model-based editors. To add support for a given modelling language, a *language specific plugin* must be developed. Obviously, this plugin will be significantly smaller and easier to make than it would have been if it had to provide all of the editor functionality by itself.

2. Plugin documentation

In this chapter I present a detailed description of both plugins. This includes a description of the overall architecture of the system, descriptions of the most of the classes, and explanations of some central parts of the system's behaviour.

Chapter 2.1 describes the UIML Diagram plugin.

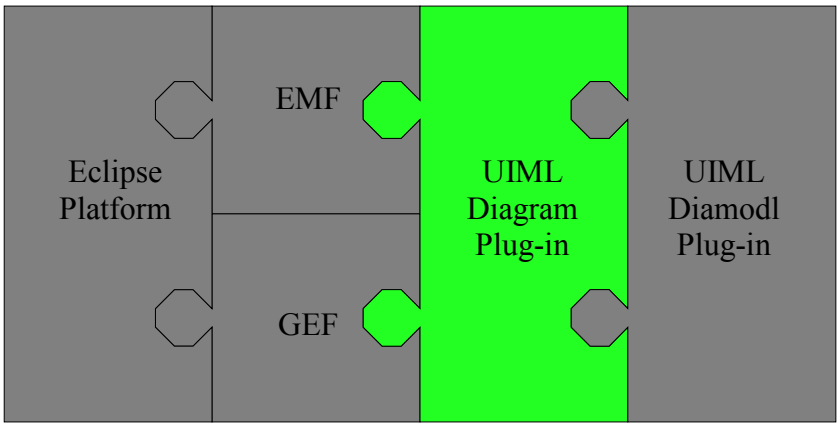
Chapter 2.2 describes the UIML DiaMODL plugin.

2.1 The UIML Diagram Plugin

The diagram plugin provides the base functionality of the editor, the part which is not language specific. It is intended for reuse by editors for other modelling languages. This chapter will give some insight into what it provides, and how it works.

Chapter 2.1.1 gives an overview of the architecture of the plugin, and presents the concrete packages and classes briefly.

Chapter 2.1.2 explains some of the most important aspects of the plugin's run-time behaviour.



2.1.1 Architecture overview

As mentioned briefly in chapter 1, GEF is built to use the Model-View-Controller pattern, and this is reflected in the diagram plugin. I assume the reader has some knowledge of this pattern, as it is widely known and used, and I will therefore not describe it in any great detail here. Instead I will describe how it relates to the specific components of the diagram plugin. If more general information of the pattern is required, I recommend a web search, or a textbook on architectural patterns.

The model

The implementation of the model in the diagram plugin is provided by EMF. As it is, at the moment, specific to to the modelling language (see figure 2), it is not part of the UIML Diagram plugin as such. Instead, it is contained in a separate «diamodl-emf» project. In time, it is planned to split up the model into a generic and a language specific part, as illustrated in figure 3 below.

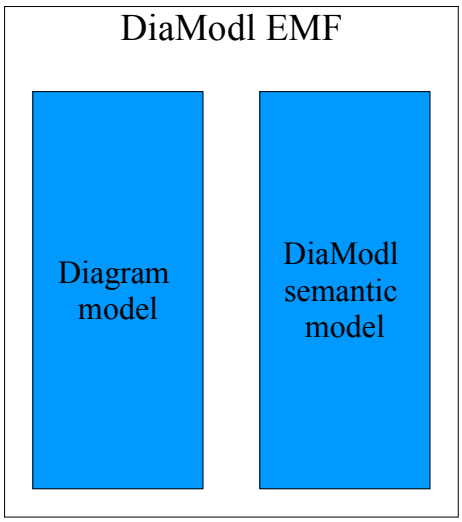


Figure 2 - DiaMODL specific model

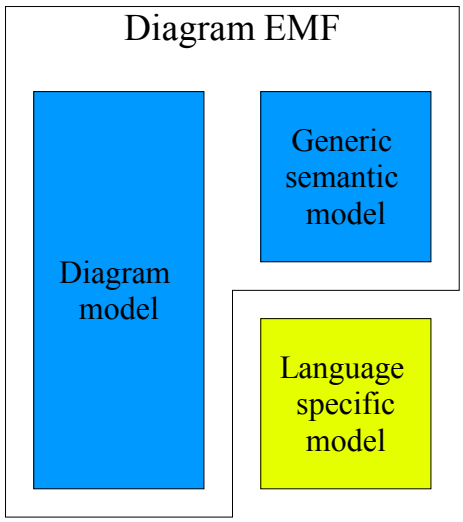


Figure 3 - Generic model with separate language specific model

The model is split into two parts: The *semantic* model and the *diagram* model. The semantic model contains the information represented by the modelling language, such as objects, their relationships, and semantic attributes. The diagram model, on the other hand, contains the information related to the *visual representation* of the semantic model. This can include on-screen coordinates, object sizes, colors and labels.

As required by the MVC pattern, the model implementation is completely oblivious to the other parts of the system. Whenever there is a need for the model to «push» information to other parts of the system, this is handled by EMF's built-in change notification mechanism. This mechanism is based on the Publisher-Subscriber pattern, in which there is no need for the publisher (in this case the model) to have any specific knowledge of its subscribers.

The view

The view part of the editor is built using GEF's Draw2D drawing API. The basic building block of a Draw2D view is the Figure, of which every graphical element displayed on screen is a subclass. The Figure class provides two important things:

- A basic implementation for all graphical elements, which includes a fairly significant amount of functionality.
- A container in which other Figures may be stored and displayed, to make composite figures.

As most of the view component of the editor is dependent on the specific modelling language, the majority of the functionality will have to be provided by the language-specific plugin. The UIML Diagram plugin's contribution to the view is limited to two classes: GraphNodeFigure and GraphEdgeFigure, which are intended to be subclassed by the language specific figures. These two classes are described in more detail in chapter 2.1.1.2.

The figure hierarchy is illustrated in figure 4. Not that there is one simplification to this figure. GraphEdgeFigure is not a direct subclass of Figure. The full chain of heritage is actually:

Figure<-Shape<-Polyline<-PolylineConnection<-GraphEdgeFigure

For the purpose of keeping the diagram simple, this detail was sacrificed.

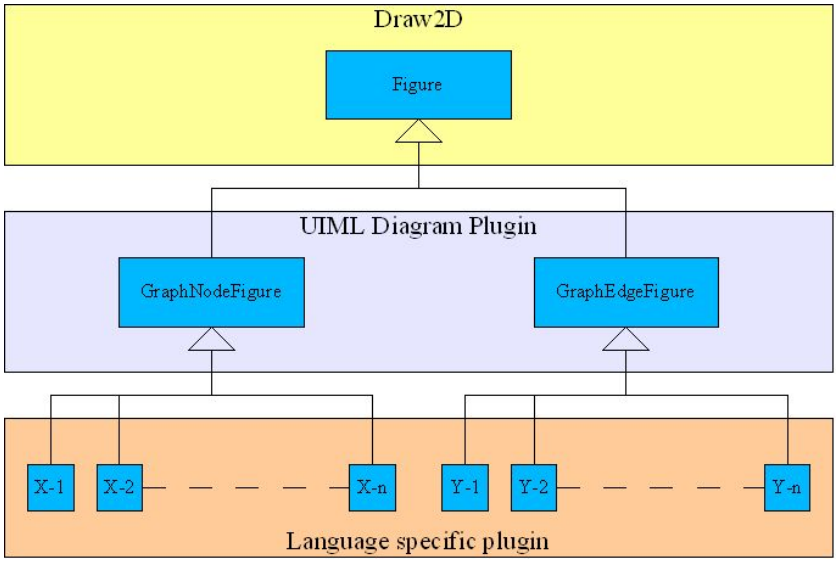


Figure 4 - The figure hierarchy

Like the model, the view has no knowledge of the other parts of the system. This is one point where the GEF architecture strays from the most common implementation of the MVC pattern. Rather than having the view listen for changes in the model and update itself automatically, all notifcations are handled by the controller. Each figure in the view is completely owned and controlled by a controller component, and only logic which is directly related to graphical layout is handled by the figure itself.

The Controller

The controller component of the system is somewhat more complex than the model and view. It is composed of several different types of classes, the most notable of which are *editparts*, *editpolicies* and *commands*.

Editparts play an important role in synchronizing the model and view. For every element in the model, there is a corresponding editpart, and for each such editpart, a figure in the view. As with the figures, the majority of the editparts will have to be provided by the language-specific plugin. The UIML Diagram plugin provides a set of generic ediparts which should be subclassed by the language specific plugins. These are descibed in more detail in chapter 2.1.1.1. The editpart hierarchy is shown in figure 5.

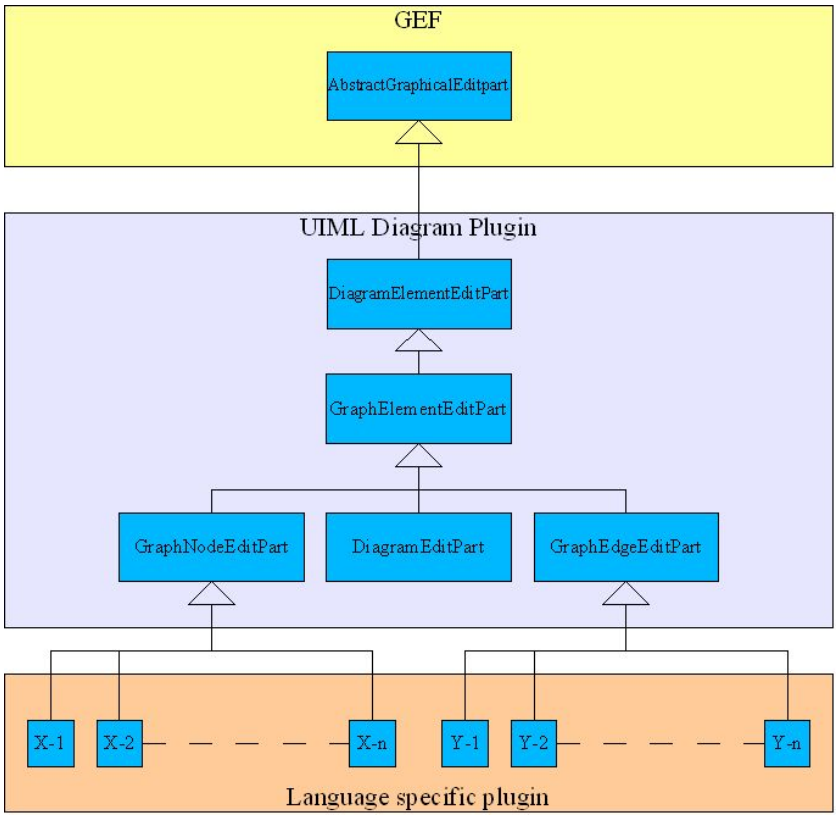


Figure 5 - The editpart hierarchy

An *editpolicy* is what enables an Editpart to react to a certain set of inputs from the user. Exactly how this works is somewhat intricate, and will be explained in detail in chapter 2.1.2.1. For now, it is sufficient to know that each editpart representing a model object has a set of editpolicies installed. Each of these editpolicies is tied to a certain *role*, that is, a type of behaviour the object should be able to support. For example, if an editpart has an editpolicy associated to the CONTAINER_ROLE, that editpolicy should contain code to handle actions related to an object working as a container for other objects, such as another object being dragged into it. The UIML Diagram plugin provides several editpolicies to support most common editor behaviour, which can

be used by the language specific plugin with minimal effort.

One of the things an editpolicy will do when it recieves a request for some action, is to create a *command*. A command is an object which contains code to make an actual change to the model, as prescribed by the «Command» architectural pattern used by GEF. The UIML Diagram plugin provides commands to support most common editor actions, such as the creation and removal of objects, repositioning and resizing, as well as creating associations between objects.

A more detailed description of the editpolicies and commands of the UIML Diagram plugin is given in chapters 2.1.1.3 and 2.1.1.4.

Figure 6 summarizes the relationships between the the main MVC object structures in the system.

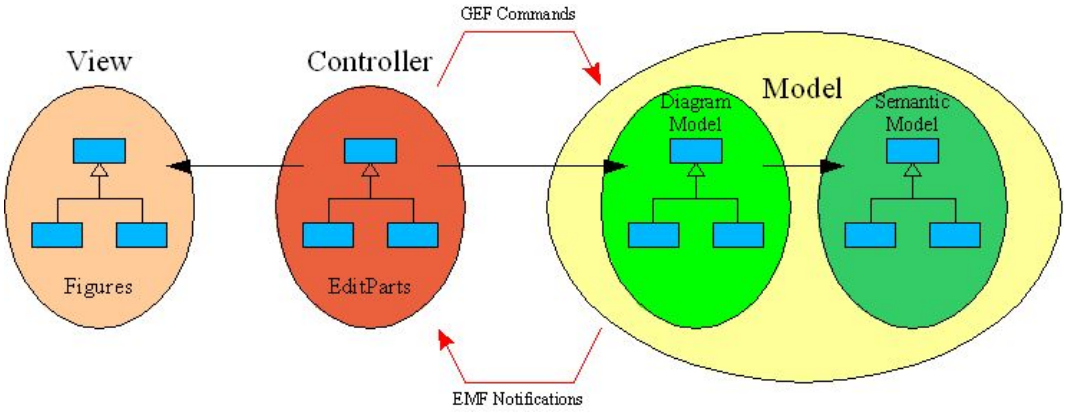


Figure 6 - Overall system architecture

The next five subchapters will provide an overview of the actual classes in the UIML Diagram plugin. Each chapter describes the contents of one of the packages:

2.1.1.1: The *no.hal.uiml.diagram.editparts* package

2.1.1.2: The *no.hal.uiml.diagram.figures* package

2.1.1.3: The *no.hal.uiml.diagram.editpolicies* package

2.1.1.4: The *no.hal.uiml.diagram.commands* package

2.1.1.5: The *no.hal.uiml.diagram.util* package

2.1.1.1 Editparts

The UIML Diagram plugin provides 5 editparts, of which 2 are intended for direct subclassing in the language specific plugin. Each of these editparts corresponds to a class in the generic diagram

model (briefly described in chapter 3.1), and handles behaviour related to the attributes and relations of that model class. What follows is a brief description of the editparts, with emphasis on which functionality is provided by each of them. At the end of this subchapter some attention is given to the non-editpart members of the no.hal.uiml.diagram.editparts package.

DiagramElementEditPart:

The DiagramElementEditPart is at the root of the UIML Diagram plugin's editpart hierarchy, and is a direct subclass of GEF's AbstractGraphicalEditPart (see figure 5). It covers the base functionality needed by all other editparts in the plugin. The most important responsibilities of this editpart are:

- Listening for changes in the semantic model. The DiagramElementEditPart contains an inner class, ModelAdapter, which extends the EMF class AdapterImpl, thus implementing the EMF Adapter interterface. An «adapter» in EMF terminology, basically means an observing class. The ModelAdapter class does not really do much besides forwarding notifications to the DiagramElementEditPart. The motivation for using an inner class here is to be able to subclass AdapterImpl, which is not possible for the DiagramElementEditPart itself, as Java does not support multiple inheritance. For a description of how model change notifications are propagated through the system, see chapter 2.1.2.2.
- Finding connections. There are two methods, getModelSourceConnections and getModelTargetConnections, which returns a list of, respectively, outgoing and incoming connections to the model object represented by the editpart.
- Getting anchors. Anchors are the start- and endpoints of a the graphical representation of a connection. As this is closely related to the view, the actual creation of anchors is not handled here, but rather in the GraphNodeFigure class, which is described in chapter 2.1.1.2. DiagramElementEditPart simply stores and provides access to the anchors.

GraphElementEditPart:

The next class in the editpart hierarchy is the GraphElementEditPart, which is a subclass of DiagramElementEditPart. This editpart adds quite a bit of functionality, including:

- The ability to listen for changes in the *diagram* model. This is done using the mechanisms provided by the DiagramElementEditPart superclass.
- Installing an editpolicy, DiagramDirectEditPolicy, for the DIRECT_EDIT_ROLE, which allows direct editing by the users. This editpolicy is described in chapter 2.1.1.3.
- Providing methods for handling *properties*. Properties appear in the Eclipse «Properties» view when an object is selected in the editor.
- Catching some notifications of model changes, including changes to the incoming and outgoing edges, as well as changes to child objects.
- Providing methods for handling labels.

DiagramEditPart:

The DiagramEditPart is the editpart for the diagram itself, and basically works as a container for the other editparts. The figure created by this editpart is the canvas on which the other figures in the editor will be painted. As the DiagramEditPart does not really contain a lot of functionality, and is not intended for subclassing in a language specific plugin, I will not describe it in further detail here.

If you are curious, I suggest reading the source code. It is quite short and simple.

GraphNodeEditPart:

GraphNodeEditPart is the superclass for all editparts in the language specific plugin which represent *nodes* in the model. Among other things, it adds the following functionality:

- It installs the remaining necessary editpolicies to allow the object to work as a proper graphical node. Specifically, this means:
    - The ability to work as a terminal for edges.
    - The ability to work as a container for other nodes (containing a sub-graph).
    - The ability to manage the graphical layout of contained nodes.
    - The ability to *be contained*, either by another node or by the diagram.
- It provides methods to support direct editing.

Combined with the functionality in the superclasses DiagramElementEditPart and GraphNodeEditPart, the GraphNodeEditPart has all the functionality which is required for an editpart to work in the graphical editor. Indeed, it is possible to use this editpart as it is. With this basic implementation, the editpart will create a rectangular figure with a label, which will support most basic editing actions. For most modelling languages you will probably want to make your own subclasses, though. Information on how to do this can be found in chapters 2.2.2.1 and 3.2.1.

GraphEdgeEditPart:

The GraphEdgeEditPart is the generic superclass for editparts in the language specific plugin which represent *edges* in the model. Most of the new functionality in this editpart is related to managing the concepts of target and source for the an edge. Like GraphNodeEditPart, it is possible to use GraphEdgeEditPart as it is. The basic implementation will give you a simple GraphEdgeFigure (see chapter 2.1.1.2 between two anchors, routed with the ModifiedManhattanConnectionRouter (described in chapter 2.1.1.5). You may want to subclass this class to add features such as labels, specific connection points on the target/source nodes, decorations, and so on. Information on extending the GraphEdgeEditPart can be found in chapers 2.2.xx and 2.3.xx.

In addition to the editparts described above, the no.hal.uiml.diagram.editparts package contains a few other classes. The most interesting of these are those which are related to the creation of editparts, the editpart-*factories*. The responsibility of these classes is to create the correct editparts for each model element (a GraphEdgeEditPart for a GraphEdge, a GraphNodeEditPart for a GraphNode, etc). For the basic editparts provided in the UIML Diagram plugin itself, editpart creation is handled by the DiagramEditPartFactory class. This class is very simple, and I suggest reading the source code if a better understanding of how it works is required. In order to create editparts for language specific model objects (for example «gate», «workflow», «data unit», and so on), the language specific plugin needs to provide its own editpart factory. How to do this will be explained in chapters 2.2.2.1 and 3.2.1. The reason I mention it here is to explain the function of the ComposedEditPartFactory class. When the editor is launched, GEF is designed to use only one factory for editparts, which will be the one returned by the getEditPartFactory() method in the

no.hal.DiagramPlugin class (the main source file for the plugin). In order to use both the DiagramEditPartFactory and the language specific factory, DiagramPlugin builds a ComposedEditPartFactory which holds both.

2.1.1.2 Figures

As mentioned in chapter 2.1.1, the UIML Diagram plugin has only two figure classes: GraphNodeFigure and GraphEdgeFigure. These are intended for subclassing in the language specific plugin.

GraphNodeFigure:

GraphNodeFigure is the superclass for any figure which represents a *node* in the modelling language. Its position in the figure-hierarchy is equivalent to GraphNodeEditPart's position in the editpart-hierarchy. The purpose of the class is to provide a basic set of functionality for all node-figures, both in order to make it easier to create language specific figures, and to guarantee a minimal set of capabilities for use by the editparts. Among other things, the following is provided:

- Label handling.
- Providing anchors, to which edges may be connected.
- Basic support for putting a node inside another.
- Basic support for scaling the figure.

These features, and how to use them when making the language specific plugin, will be explained in greater detail in chapters 3.2.2.

It is worth noting that at the time of writing, the GraphNodeFigure class is abstract, and thus can not be instantiated. It is likely that this will be changed in the near future.

GraphEdgeFigure:

GraphEdgeFigure is the superclass for any figure which represents an *edge* in the modelling language. It extends Draw2D's PolylineConnection, and provides a basic set of functionality for all figures representing edges in the model:

- Getting and setting source and target figures.
- Getting and setting source and tarket *keys*. The key can be used to determine which anchor will be used, for figures which has more than one anchor point.
- Drawing and refreshing the graphical representation of the figure.

Unlike the GraphNodeFigure, GraphEdgeFigure is quite complete on its own, and therefore there might not be a need to extend it in the language specific plugin. This is the case for the DiaMODL plugin described in chapter 2.2 in this document. If you are writing a plugin for a language which requires more functionality for its connections, chapter 3.2.2 describes how to extend the GraphEdgeFigure.

2.1.1.3 Editpolicies

Editpolicies are what makes it possible for an editpart to react to user input. Each editpart has a set of editpolicies installed, which largely determines its editing capabilities. Figure 7 shows which editpolicies are installed for each of the editparts in the UIML Diagram plugin.

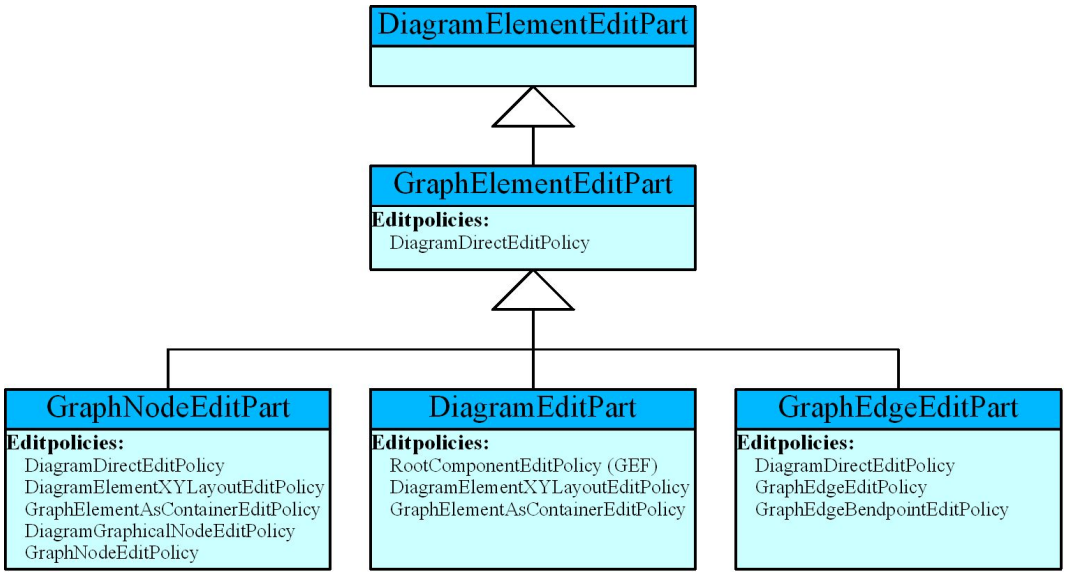


Figure 7 - Editpolicies installed on editparts

These editpolicies, with the exception of RootComponentEditPolicy (which is part of GEF), are briefly described below.

DiagramDirectEditPolicy:

DiagramDirectEditPolicy provides support for *direct editing*. In practice, this means the ability to edit the text of a label in the diagram. DiagramDirectEditPolicy extends GEF's DirectEditPolicy.

DiagramElementXYLayoutEditPolicy:

This editpolicy adds support for changing the *constraints* (sizes) of objects in the diagram, as well as for handling the constraints when *adding* one object to another (typically when the user drags a figure into another). It extends GEF's XYLayoutEditPolicy.

GraphElementAsContainerEditPolicy:

The most important responsibility of GraphElementAsContainerEditPolicy is to support creating objects inside the editpart the policy is installed on. Notice that this editpolicy is installed both on DiagramEditPart and on GraphNodeEditPart, meaning it is possible to create new objects both separately, and inside other objects in the diagram. To prevent inconsistency, the editpolicy will ask the model if the action is legal.

The second responsibilty of the GraphElementAsContainerEditPolicy is to support *orphaning*, that is, to detach children objects from their parents. This is used when an object is moved from one parent object to another.

GraphElementAsContainerEditPolicy extends GEF's ContainerEditPolicy.

DiagramGraphicalNodeEditPolicy:

This editpolicy adds support for creating *connections* (edges) between model objects. It extends GEF's GraphicalNodeEditPolicy.

GraphNodeEditPolicy:

GraphNodeEditPolicy adds support for *deleting* node objects, and extends GEF's ComponentEditPolicy.

GraphEdgeEditPolicy:

This editpolicy adds support for deleting *edges*. At the time of writing, it is very similar to GraphNodeEditPolicy, the only difference being that it extends GEF's ConnectionEditPolicy rather than ComponentEditPolicy.

GraphEdgeBendPointEditPolicy:

As the name suggests, GraphEdgeBendPointEditPolicy provides support for editing the *bendpoints* of an edge in the diagram. It extends GEF's BendpointEditPolicy.

2.1.1.4 Commands

While the editpolicies are responsible for reacting on user input, it leaves making the actual changes in the model to the *commands*. The UIML Diagram plugin provides 9 commands, of which 5 are direct subclasses of GEF's generic Command class, while the other are subclasses of StructureChangedCommand. The hierarchy of commands in shown in the figure below, which is then followed by a brief description of each command.

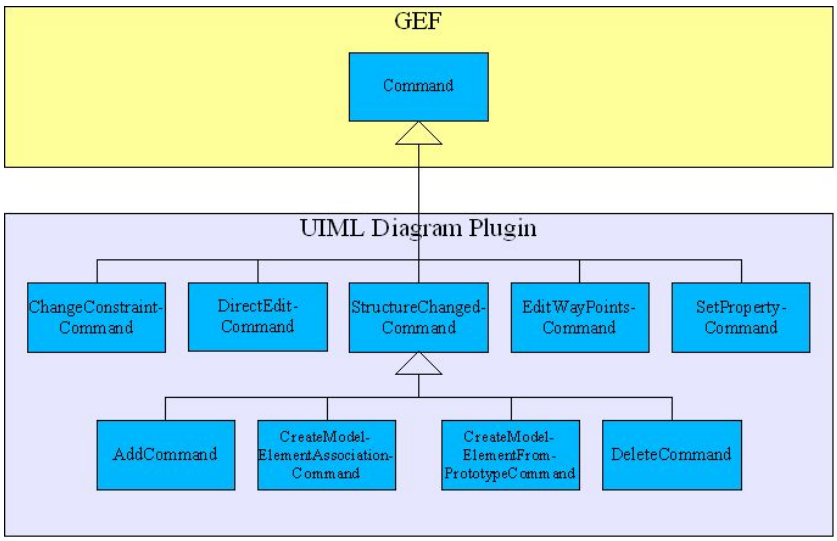


Figure 8 - The command hierarchy

ChangeConstraintCommand:

The ChangeConstraintCommand does exactly what it sounds like it does: it changes the constraints (position and size) for a model object.

DirectEditCommand:

This command changes the corresponding value in the model when an on-screen label has been modified through direct editing.

EditWayPointsCommand:

The EditWayPointsCommand changes the waypoints of a graphical edge in the diagram.

SetPropertyCommand:

This class is currently not in use.

StructureChangedCommand:

StructureChangedCommand is the superclass for all commands which change the structure of the semantic model. Examples of such changes are: Object creation, object removal, and changing parent-child relationships. It is a rather complex command, which handles such things as making it possible to undo structural changes, and propagating consequences changes. For example, deleting a node will have consequences for all incoming and outgoing edges.

AddCommand:

AddCommand adds a model object as a child of another model object. It is used together with the DeleteCommand to handle moving object from one parent to another.

CreateModelElementAssociationCommand:

This command handles the creation of *edges* in the model.

CreateModelElementFromPrototypeCommand:

This is the command which is responsible for creating the model elements which are not edges, the *nodes*. It does this by copying a prototype object from an object library. Exactly how this works is not entirely trivial, and is therefore explained separately in chapter 2.1.2.3.

DeleteCommand:

The DeleteCommand detaches a model object from its parent and connected objects. If it is directly followed by an AddCommand, the object will be moved to a new parent. If not, it is simply deleted. Note that if the object in question is a node, all children, as well as all incoming and outgoing edges, will also be deleted.

2.1.1.5 Utility classes

The no.hal.uiml.diagram.util package contains all those little classes which does not fit anywhere else, utilities which provides simple services to the other classes. At the time of writing there are 3 of these classes, all of which are described below.

ClassFilter:

The ClassFilter is an *iterator* (java.util.Iterator), which is used to filter the objects contained in either a collection or another iterator. It does this by comparing the class types of the objects to a class type given as an input to the filter. Depending on the «logic» variable, which may be given as a parameter when instantiating the ClassFilter, it will either return all objects which matches the class type (if «logic» is set to true), or all objects which does not match («logic» set to false). By default, the «logic» variable is set to true.

LocalPointAnchor:

LocalPointAnchor is a simple point-oriented anchor, which is used by GraphNodeFigure for figures which want connections to attach to a specific point. It is similar to Draw2D's XYAnchor, but with two important differences:

The point where the anchor is located is relative to a given parent figure in the diagram. In comparison, XYAnchor uses a point which is relative to window the editor is displayed in.

LocalPointAnchor has a concept of *direction*, which indicates the desired direction from which an edge should connect to the anchor. As an example, if the anchor is attached to the left edge triangular figure, which points to the right, you will propably want an incoming edge to connect from the left (see figure 9).

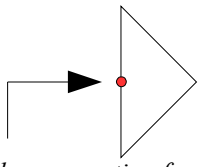


Figure 9 - Edge connecting from the left side

ModifiedManhattanConnectionRouter:

ModifiedManhattanConnectionRouter is a «quick hack» of the ManhattanConnectionRouter in Draw2D. The modified router differs from the orignal in that it is possible to lock the direction of the first and last line segments of the graph.

2.1.2 System behaviour

In this chapter I will explain some of the more important aspects of how the system actually works while it is running. While the emphasis in chapter 2.1.1 was on *structure*, this chapter focuses on *sequence*. For obvious reasons, it is beyond the scope of this document to document every single operation, so priorities have been made. The topics which have been selected are those which are of significant importance to the system, and of non-trivial complexity.

Chapter 2.1.2.1 describes the general way in which user input is captured and processed, and eventually ends up as a change in the model.

Chapter 2.1.2.2 explains how a change in the model is then propagated through the system, until it manifests itself visually.

Chapter 2.1.2.3 covers how objects are created. As I mentioned briefly in chapter 2.1.1.4, this is done by copying and adapting objects from an object library, which makes the process a little less straightforward than simply making fresh instances of the required objects.

2.1.2.1 The initiation and execution of commands

Unquestionably, the single most important feature of any editor, as opposed to a viewer, is the ability to actually edit something. I have mentioned earlier that *commands* are the units responsible for modifying the model. However, this still leaves the question of how commands are created and executed. In order to understand this, there are two other entities which needs to be explained: *tools* and *requests*.

The concept of *tools* should be well known to anyone who has used any type of direct manipulation editor. From the user's perspective, a tool is something which enables you to modify the data you are editing. It is typically accessed through a button in a toolbar or tool pallette. In GEF, each tool is represented by a an entry in the tool palette. Figure 10 shows an example of a tool in the UIML DiaMODL editor (a work-in-progress version).

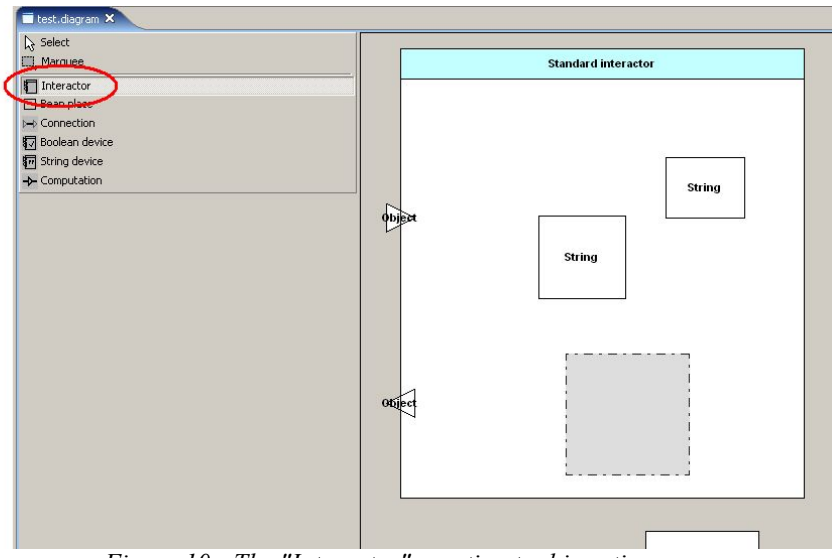


Figure 10 - The "Interactor" creation tool is active

The exact way in which tools in the UIML Diagram plugin are created, will be discussed in more detail in chapter 3.3.2. For now, it is sufficient to know that there is indeed a «tool» object for each entry in the palette, which will be invoked when the user clicks on that entry.

Requests are created by a tool when it is activated. A request holds the information which is necessary in order to create a command. Exactly what information this is depends on the tool, as well as the stage the user interaction. I will explain what this means shortly. But first, take a look at figure 11 below.

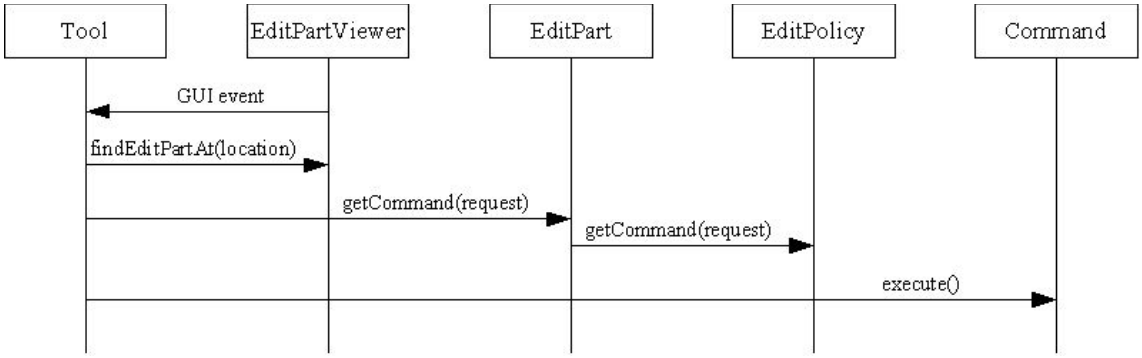


Figure 11 - Command creation and execution (simplified)

This sequence diagram show what happens when a tool is used. I will explain step-by-step:

1. First, a GUI event is detected by the user interface, and the EditPartViewer notifies the tool that it has been invoked. The EditPartViewer is part of the GEF framework. Explaining how it works and all the things it does is beyond the scope of this document. If you want to know more about it, I suggest looking it up in the GEF documentation. For this document, all you need to know about it is the following:
    - It is the entry point for input from the user interface.
    - It is capable of locating editparts from a screen coordinate.
2. When the tool is invoked, it sends a request to the EditPartViewer, to ask which is the top-most editpart currently under the mouse pointer. Remember that the canvas itself is an editpart (the

DiagramEditPart), so this should this should always return something as long as the mousepointer is inside the model view.

3. Next, the tool sends a request to the editpart returned by the viewer, asking it for a command.
4. While it is possible for the EditPart to handle the request itself, this is rarely the best way of doing things. Because most commands will be very similar for many editparts, it is better to leave this to an editpolicy, which can then be installed on the editparts which needs the associated behaviour. This is the way GEF is designed to work, and it is how the UIML Diagram plugin works. When the request is received by an editpart, it is simply passed on to an editpolicy which is capable of handling it (assuming one is installed).
5. The EditPolicy then does the job of creating the command, using the information in the request.
6. Finally, once the user interaction is over, the command will be executed. Provided the suggested action is legal according to the consistency requirements of the modelling language, this will result in a change in the model.

Before I finish this chapter, there are a couple of things which are worth noting. Firstly, some details have been left out of the sequence diagram and description above, in order to keep things simple. Most significantly:

- In addition to requesting a command, the tool will also tell the involved editparts to show or erase any visual feedback as needed. It does this through the methods showSourceFeedback(), eraseSourceFeedback(), showTargetFeedback(), and eraseTargetFeedback(). While it is not done at the time of writing, this could be used, for example, to highlight an anchorpoint when the mousepointer is over it while making an edge in the graph.
- The tool does not really tell the command to execute directly, it is done through another GEF entity called the «CommandStack». The great advantage of using the CommandStack is that it makes it possible to undo/redo the actions. For more information on the CommandStack, refer to the GEF documentation.

The second thing I want to clarify, is that the vast majority of commands which are created are in fact never executed. GEF is extremely eager when it comes to requesting commands. A command is requested the instant the mousepointer touches an editpart while a tool is active. New commands are then requested repeatedly as the pointer moves across the view, when the mousebutton is pressed, and when it is finally released. Only for the last command in an interaction (typically when the mousebutton is released), execution will be attempted.

2.1.2.2 Propagation of changes in the model

A command changes the model, but it is not involved in updating the view, nor is responsible for informing the editparts of the change. So how does a model change become known to the rest of the system? As I mentioned briefly in the introduction to chapter 2.1.1, the publisher-subscriber pattern is used to allow the model itself to notify editparts when a change has been made. What this means is that each model object maintains a list of «subscribers», and when a change is made, a «notification» object is sent to each of these. In the UIML Diagram plugin, each editpart is set up to subscribe itself to the model objects it represents when the editor is launched. When the editparts

are created by their factory (either the DiagramEditPartFactory or the language specific factory), they are given a reference to the semantic model object they represent. They are then capable of subscribing or unsubscribing themselves from this object (and the associated *diagram* model object) as needed. In the UIML Diagram plugin, this is done by the activate() and deactivate() methods in DiagramElementEditPart.

Once an editpart is subscribed to its model object, the propagation of changes to the model is quite simple. Figure 12 below shows the general principle.

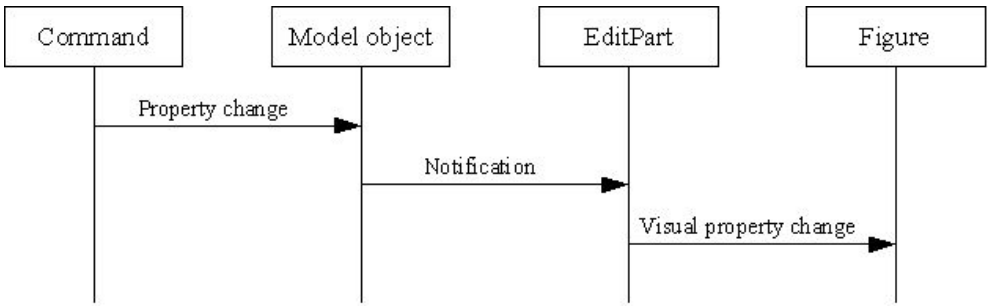


Figure 12 - The general principle of how model changes are propagated

In practice, what happens in the UIML Diagram plugin is this:

1. First, the EMF model object sends a notification to the subscribed editpart, by calling the notifyChanged() method. This method is implemented in the DiagramElementEditPart class.
2. The notification is then examined to determine its basic type. In most cases, it will be of the type «SET», indicating that the value of a field has been *set*. SET and the other possible values are defined in the org.eclipse.emf.common.notify.Notification interface.
3. Next, the notifyChanged() method will call notifySet(). This method is implemented in several classes, all the way down to the language specific editparts. The task of the of the notifySet() method is to examine which field has actually been changed.
4. Finally, notifySet() makes the necessary updates on the figure.

Figure 13 shows the sequence described above.

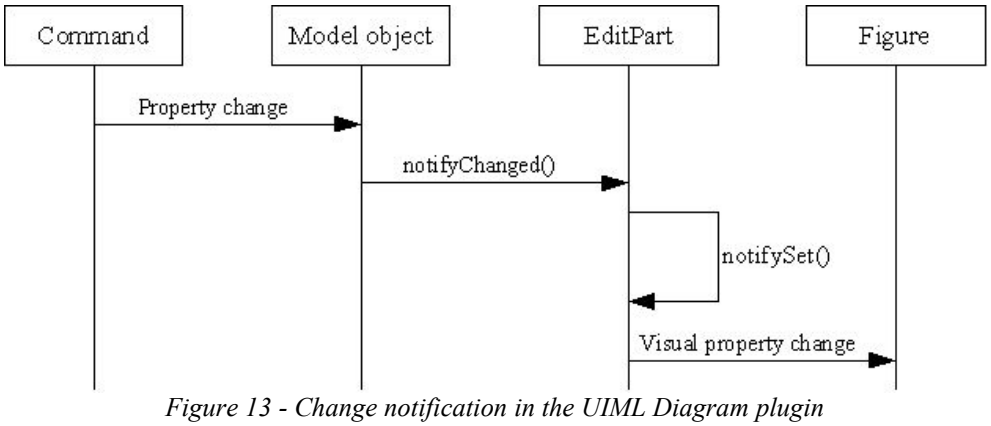


Figure 13 - Change notification in the UIML Diagram plugin

As I mentioned, the notifySet() method is implemented in several classes in the hierarchy. In the UIML Diagram plugin, the notifySet() method in each class will handle changes to certain set of fields. If the changed field does not match any in the set the class is able to handle, the notification will be passed on to its direct superclass. This will then be repeated by each notifySet() method, until one is found which is capable of handling the change, or the end of the notifySet() method in DiagramElementEditPart is reached. Figure 14 show an example of how this could happen. Here, the notification is forwarded until a notifySet() which is able to handle it is found in GraphElementEditPart. This could be the case if the change was to the «Name» field of the model object, as many figures use this field in their label, and GraphElementEditPart has methods to handle labels.

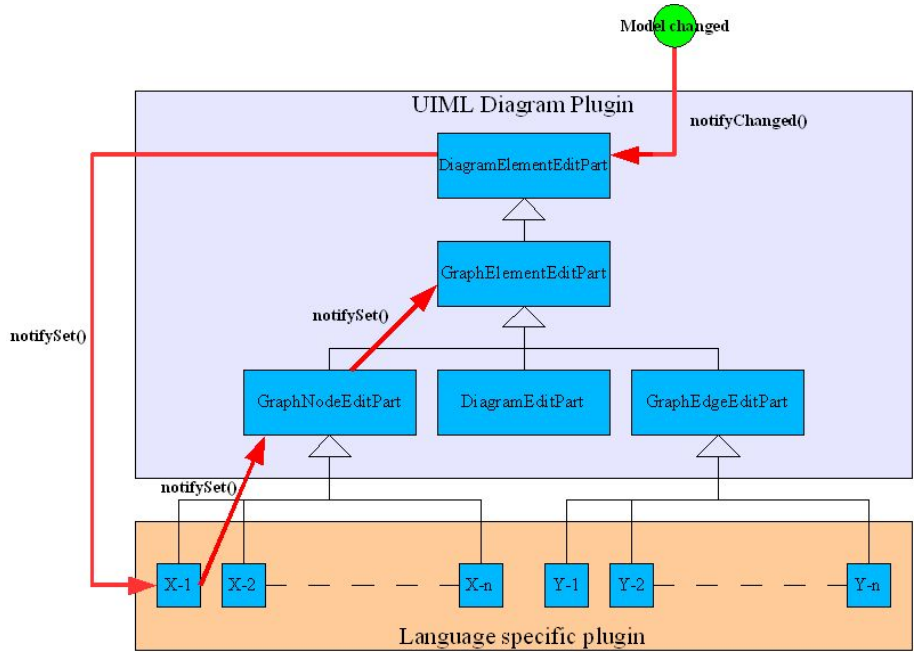


Figure 14 - The change notification is forwarded through the editpart hierarchy

Finally, figure 15 summarizes what happens when the user performs an editing action, from user interaction to visual confirmation. This diagram combines the information from figures 11 and 13. Note that I have merged «EditPartViewer» and «Figure» into the more generic «View/GUI», in order to make the diagram more readable.

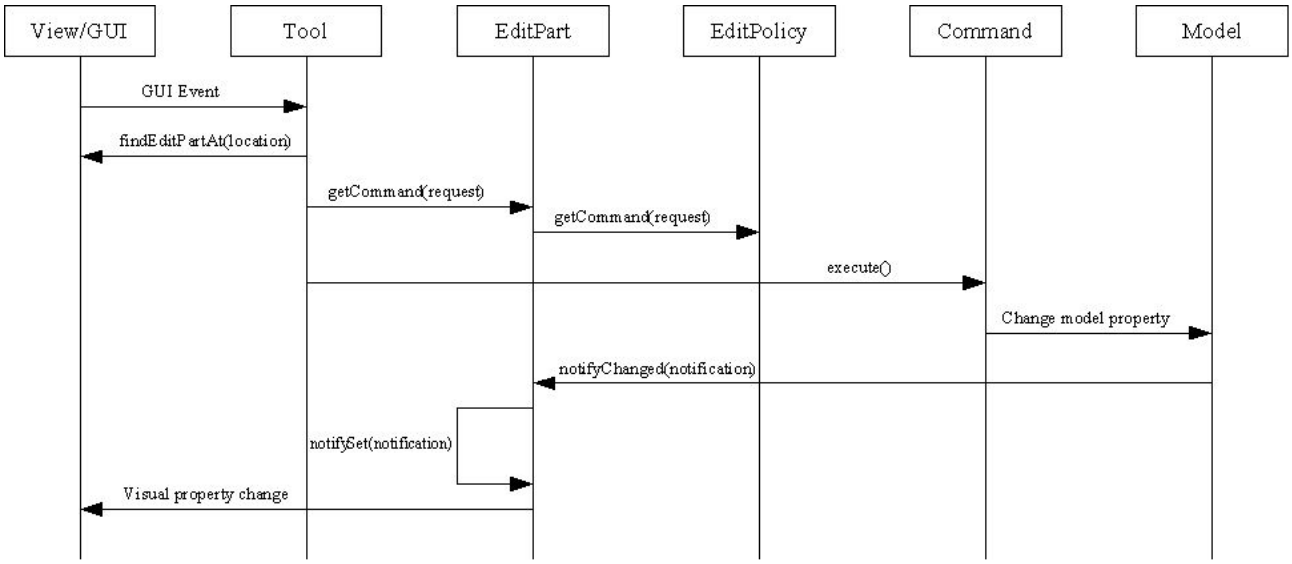


Figure 15 - User interaction summarized

2.1.2.3 Object creation

The primary way in which the UIML Diagram plugin creates new model objects is by cloning objects from a library file, rather than making new objects based on the language model. There are several reasons for doing it this way:

- It makes it possible to provide default values for fields without having to touch the underlying language model.
- It means you can easily make several different «types» of objects based on a single element in the language model. To clarify, I will use the «Interactor» in the DiaMODL language as an example: An «Interactor» is a generic entity in the language, which can behave very differently depending on how it is configured. The library file used by the DiaMODL editor has three types of interactors: «Standard Interactor», «String Interactor» and «Boolean Interactor». Each of these have their own tool in the tool palette, and can thus be created directly, even though only the basic «Interactor» class is described in the language syntax.
- It means you can make many changes to the editor without having to modify the code. The tool palette entry is defined in the «plugin.xml» file (explained in chapter 3.3.2), and the elements you can create are defined in the library file.

Exactly how object creation works is complicated, but I will try to give a simplified explanation. Fortunately, a complete understanding is not required in order to create a language specific plugin. Figure 16 shows a (very) simplified illustration of what happens when a new object is created.

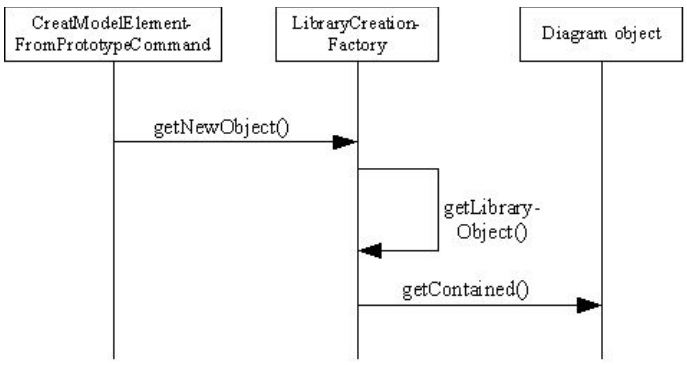


Figure 16 - Object creation

In words, this is an approximation of how it works:

When the editor is launched, a LibraryCreationFactory is instantiated. This factory, in turn, reads the contents of the library file (the file which contains the object prototypes) into a diagram object. When a request for a new object is received, the factory will search the diagram object for a model object which matches the given description. The description consists of a class name, which is the name of the basic class in the language syntax (for example «uiml.diamodl.Interactor»), and a specific object name, which is the name of the specific variant of that class (for example «String Device»). If an object which matches the description is found, a copy of it will be returned.

If no match is found in the prototype library, it is the intended behaviour of the factory that it should

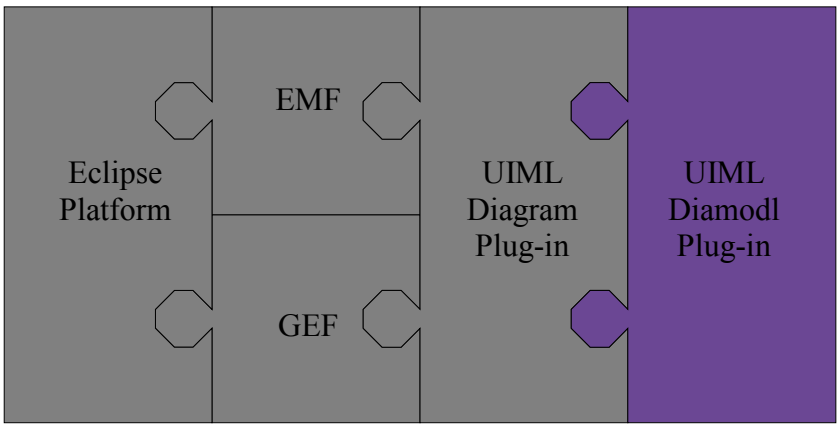
fall back to making a fresh instance of the object class. At the time of writing, this does not actually work, but it will most likely be fixed in the near future.

2.2 The UIML DiaMODL Plugin

The UIML DiaMODL plugin is the very first language specific plugin for the UIML Diagram plugin described in the previous chapter. It has been developed simultaneously with the generic plugin, by the same authors, and should serve as a practical example of how to build a language specific plugin. Note that the aim of this chapter is to document the DiaMODL plugin as it is, rather than to give general guidelines and instructions on building language specific plugins. Such instructions are the topic of chapter 3. This chapter merely shows one possible solution.

Chapter 2.2.1 gives an introduction to the DiaMODL language.

Chapter 2.2.2 describes the plugin.



2.2.1 Introduction to DiaMODL

DiaMODL is a language for user interface modelling. It was proposed by Hallvard Trætteberg in his doctoral thesis, «Model-based User Interface Design» (see [Trætteberg, 2002]), and is based on the Pisa interactor abstraction, described in [Markopoulos, 1997].

The two main features of DiaMODL are the following:

- It can be used as practical UI design tool.
- It is integrated with UML.

For this document, only a brief introduction to the basic elements of the language will be given. More detailed descriptions can be found in [Trætteberg, 2002] and [Trætteberg, 2003].

The following are the basic modelling units of the DiaMODL language:

Nodes

«Interactor»:

An interactor is a unit which mediates information between the system and the user. It will usually represent some UI component. It is visually represented by a Rectangle, with a name field at the top, and «gates» along the left-hand side, as shown in figure 17. The large white section below the name field is the content pane. This area may contain either a single interface element, or a DiaMODL subgraph.

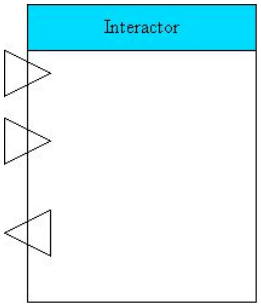


Figure 17 - A DiaMODL interactor

«Computation»:

A computation is a node which takes a set of inputs, and calculates a result. How the result is calculated, as well as the number of inputs, depends on the *function* of the computation. Visually, a computation is represented as a triangle, with the inputs entering at one side, and the result emerging from the opposite tip, as shown in figure 18.

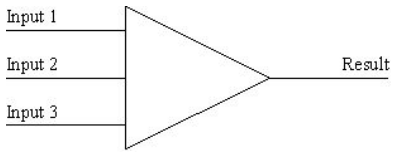


Figure 18 - A DiaMODL computation

«Gate»:

A gate is a type of computation, which is directly tied to an interactor. Apart from the special relationship to an interactor, the one significant difference between a gate and a regular computation is that the gate is divided into a base and a tip. For an output gate, the tip is inside the interactor, while the base is on the outside. For an output gate, the positions are reversed. Semantically, the difference between the parts on the inside and outside of the interactor, is that the those inside holds a value as perceived by the user, while those on the outside hold the value as perceived by the

system. Figure 17 above showed a set of gates attached to an interactor.

«BeanPlace»:

A beanplace is used to hold either a variable or a widget. Its contents are dynamically altered by its incoming edges. Figure 19 shows an example of DiaMODL beanplace.

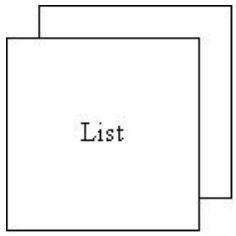


Figure 19 - A DiaMODL beanplace (multi-value)

Edges

«Connection»:

A connection is the basic edge of the DiaMODL language. It is visually represented by a simple line between two nodes in the graph. Semantically, a connection represents a flow of data between two nodes. The direction and cardinality of the flow is implied by the nodes themselves, rather than the connection, and there are therefore no arrowheads or similar decorations attached to it. Figure 20 illustrates a connection between a computation and a gate.

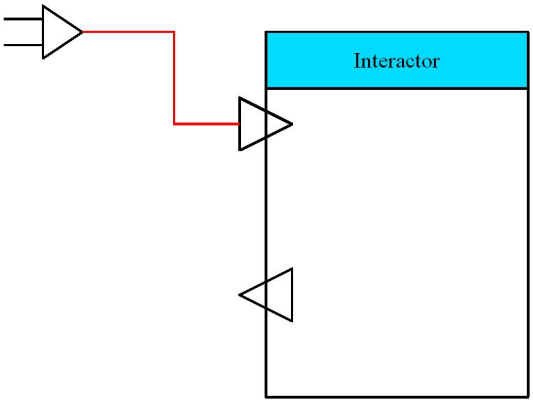


Figure 20 - A DiaMODL connection

2.2.2 Plugin overview

The overall architecture of the DiaMODL editor is, by necessity, the same as for the generic plugin, and its description will therefore not be repeated here. This chapter is dedicated to showing how the DiaMODL plugin extends and utilizes the generic plugin to provide a fully featured editor with a fairly modest amount of effort. If you look at the sourcecode of the plugin, you should notice that it is both significantly smaller and much simpler than that of the diagram plugin. This is because most of the difficult problems have already been solved, either by GEF, EMF, or the UIML Diagram plugin. What remains is providing the bits which are specific to the DiaMODL language. This is done mostly by subclassing the required classes of the generic plugin, as well as providing a few supplementary files. The following chapters describes how this has be done in the various packages and files of the DiaMODL plugin:

2.2.2.1: Covers the *no.hal.uiml.diamodl.editparts* package

2.2.2.2: Covers the *no.hal.uiml.diamodl.figures* and *no.hal.uiml.diamodl.layoutmanagers* packages

2.2.2.3: Covers the *no.hal.uiml.diamodl.commands* and *no.hal.uiml.diamodl.editpolicies* packages

2.2.2.4: Covers the other files used by the plugin

2.2.2.1 Editparts

The *editparts* package holds 6 editparts, one for each of the basic DiaMODL elements described in chapter 2.2.1, and one extra which works as a superclass for all the DiaMODL node type elements. The full hierarchy of editparts for the DiaMODL editor, including both the UIML Diagram and DiaMODL specific plugins, is shown in figure 21. Now on to the description of each editpart:

DiamodlNodeEditPart:

The DiamodlNodeEditPart is the abstract superclass of all the node type editparts in the plugin, and thus provides a place to make global changes. At the time of writing, it only does one thing: replacing the DiagramGraphicalNodeEditPolicy of the UIML Diagram plugin with a DiaMODL specific DiamodlGraphicalNodeEditPolicy. The DiamodlGraphicalNodeEditPolicy is discussed in chapter 2.2.2.3.

InteractorEditPart:

InteractorEditPart is the editpart which is used to represent an Interactor in the DiaMODL language. It is a very simple class, containing only one simple method: createFigure(). This method is responsible for creating the figure of the interactor.

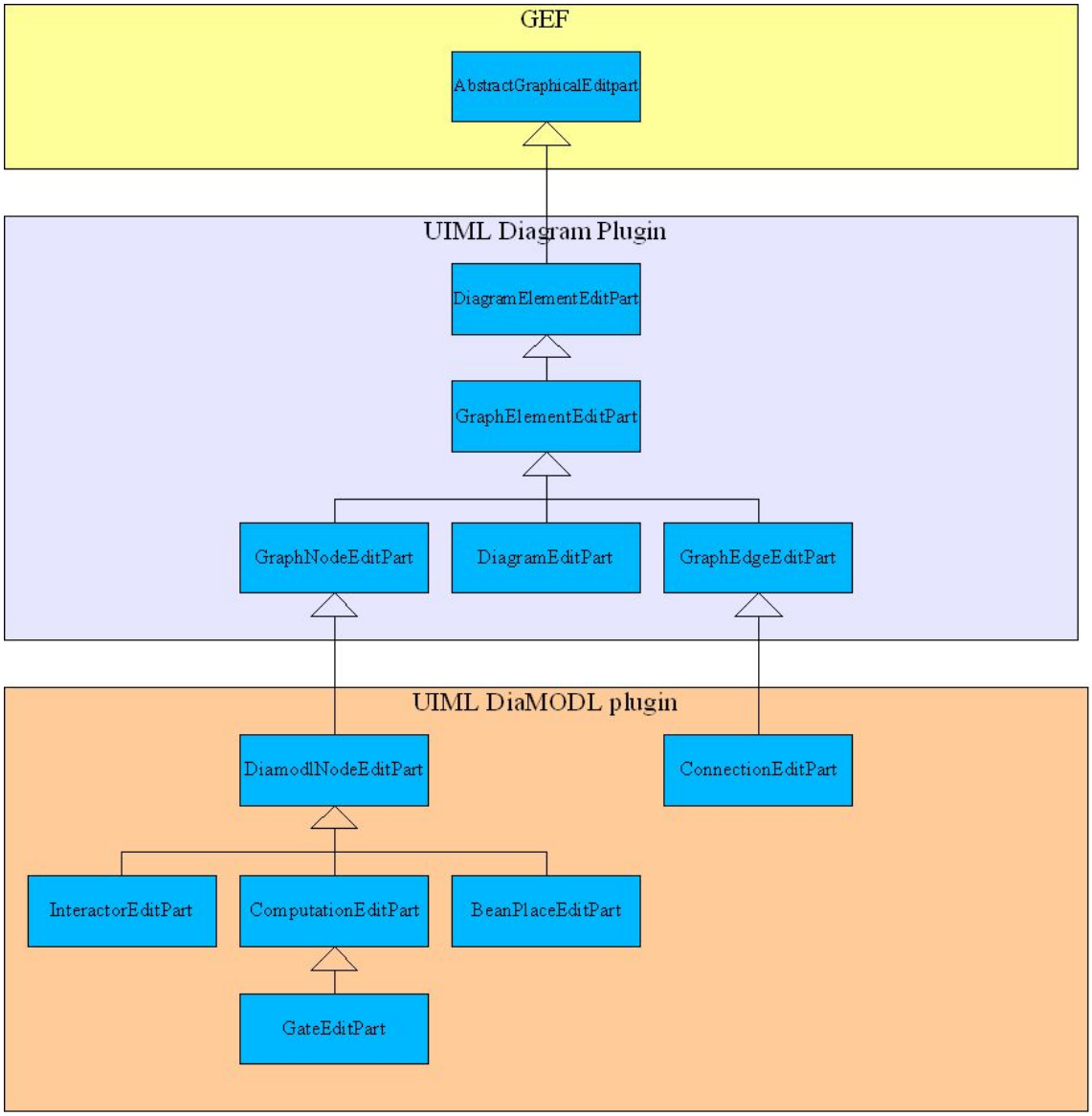


Figure 21 - The editpart hierarchy of the full DiaModl editor

ComputationEditPart:

This editpart represents DiaMODL computation nodes. As with the InteractorEditPart, it is responsible for creating the figure. However, if you look at the source code, you will notice that it is a bit longer and more complex than the InteractorEditPart. Fortunately, this is about as bad as the complexity of language specific editparts get, and it is still quite easy to understand. Since this is the first «long» language specific editpart in this document, I will explain the increased complexity in some detail:

- Firstly, you will notice that the createFigure() method is a bit longer. The reason for this is that the ComputationFigure requires a bit more more configuration than the InteractorFigure does:
    - It needs to know in which direction it should should be pointing, as opposed to the InteractorFigure, which only has one orientation.
    - It needs to know the arity of its function in order to determine how many anchorpoints it should provide. The InteractorFigure does not need to provide anchors at all, as this is handled by its gates.

- Next, the ComputationFigure overrides a few methods from GraphElementEditPart. Specifically, these are the methods realated to label handling. The reason for this is that the ComputationFigure uses the «Function» property in the model for its label, while the default methods in GraphElementEditPart use the «Name» property.
- Finally, it overrides the notifySet() function in order to update the label when the function field is changed. It also intercepts changes to name field, to prevent the redundant refresh which would have happened if the notification had been forwarded to GraphElementEditPart. Note that this is not strictly necessary for the editpart to work properly.

GateEditPart:

GateEditPart is a small subclass of ComputationEditPart. It overrides two methods to provide the behaviour which is specific to DiaMODL gates:

- The createFigure() method is overridden. While it is quite similar to the one in ComputationEditPart, there are two differences:
    - It can only have EAST or WEST as its orientation, and this is controlled by another model field than for the computation.
    - It uses a GateFigure rather than a ComputationFigure.
- ComputeBounds() in GraphNodeEditPart is overridden to return null. Since the position of a gate is determined by the interactor which owns it, there is no need for it to calculate its own bounds.

BeanPlaceEditPart:

The BeanPlaceEditPart represents DiaMODL beanplaces in the editor. Like ComputationEditPart, it overrides some methods in the UIML Diagram plugin to support using another field than the default «Name» property for its label. In the case of beanplaces, the «Value Type» property is used. One additional change is that BeanPlaceEditPart checks to see if it represents multiple values, as that would require a slightly different appearance for its figure.

ConnectionEditPart:

ConnectionEditPart represents the single *edge* type in DiaMODL, the *connection*. The differences between the ConnectionEditPart and its direct superclass, GraphEdgeEditPart, are the following:

- It overrides the setSource() and setTarget() methods to support connecting to specific anchorpoints on its source and target figures.
- It uses the «Function» field for its label, rather than the default «Name» field. This is done the same way as for ComputationEditPart.

One final thing worth noting about this editpart is that unlike the other non-abstract DiaMODL editparts, it does not have a corresponding figure in the UIML DiaMODL plugin. The reason for this is that the GraphEdgeFigure in the generic plugin provides all the functionality which is needed for DiaMODL connections.

In addition to the editparts themselves the *no.hal.uiml.diamodl.editparts* package also holds the factory used to create them, the DiamodlEditPartFactory. When the editor is launched this factory

will be merged with the editpart-factory in the generic plugin, and the composite factory will be used to create editparts from model objects. For more information on the factories in the UIML Diagram Plugin, see chapter 2.1.1.1.

2.2.2.3 Figures and layoutmanagers

The *figures* package holds 4 classes, one for each node type in the DiaMODL language (recall that the generic GraphEdgeFigure is used for the single edge type). The full hierarchy of figure classes, covering both the generic and language specific plugin, is shown in figure 22. Note that once again, I have simplified the figure somewhat by drawing GraphEdgeFigure as a direct subclass of Figure, rather than including the full chain of heritage:

Figure<-Shape<-Polyline<-PolylineConnection<-GraphEdgeFigure

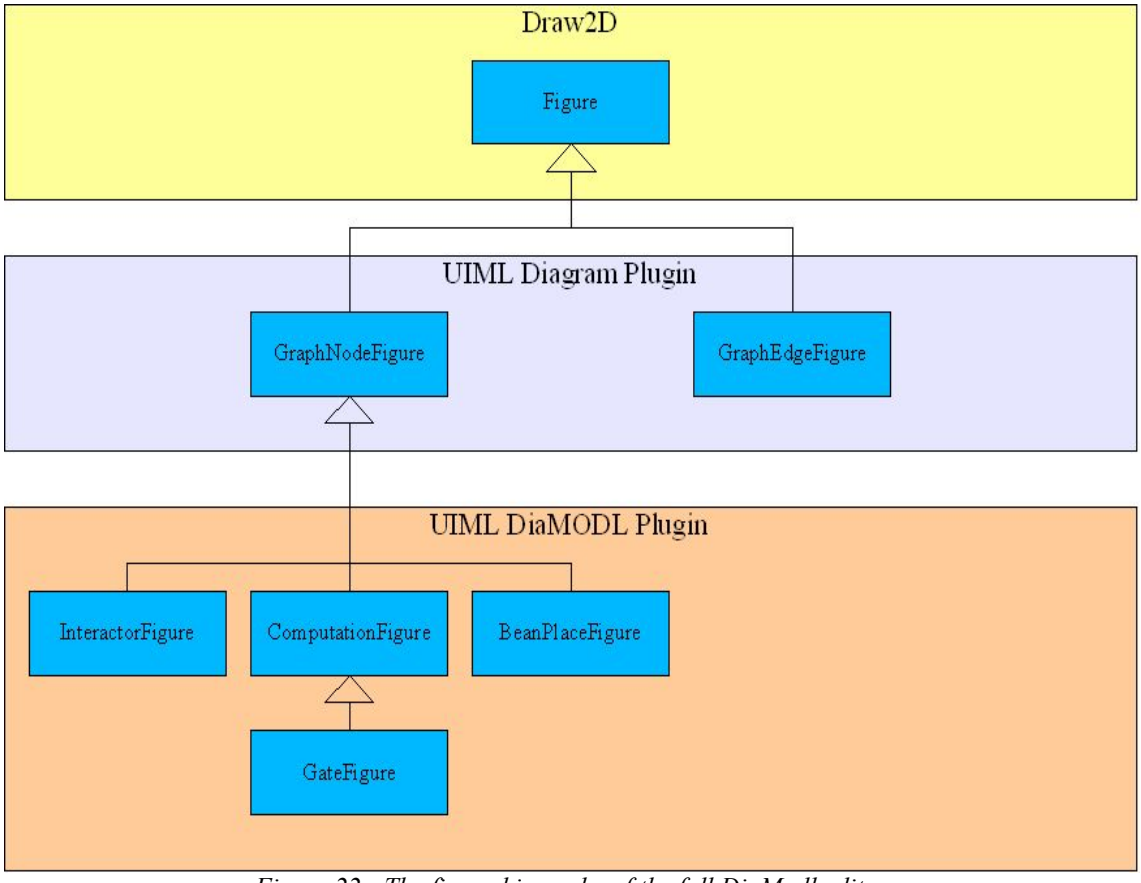


Figure 22 - The figure hierarchy of the full DiaModl editor

One of the figures, InteractorFigure, has an associated layoutmanager in the *layoutmanagers* package. These two classes are closely related, and will be discussed together in the figure overview below. Note that some basic knowledge of Draw2D is recommended for understanding the remainder of this chapter. Appendix A holds some suggestions for where such knowledge might be acquired.

InteractorFigure and InteractorLayout:

InteractorFigure is, by far, the most complicated figure in the package. The reason for this is that while the other DiaMODL node types are atomic units, interactors almost always have several child nodes. This means that the InteractorFigure has to manage the positioning of other figures as well as drawing its own features. Recently, the InteractorFigure class was simplified significantly by moving the reponsibility for positioning child figures to the InteractorLayout layoutmanager class. This has rendered many of the methods in InteractorFigure redundant. What responsibilities still remain in the InteractorFigure is as follows:

- Drawing the interactor itself. This means the two rectangles representing the «label area» and «content area».
- Attaching the label to the figure. The responsibility of positioning it, however, has been moved to the layoutmanager.
- Calculating the size of the outline bounds from the outer bounds, and vice versa. The concepts of outline and outer bounds are explained more thoroughly in chapter 3.2.2. For now, I will settle for saying that one relates to the total area the figure occopies on the canvas, while the other relates to the «core area» of the figure.

As mentioned, InteractorLayout is the layoutmanager for InteractorFigure. In basic terms, a layoutmanager is an object which is reponsible for positioning the children of a figure. If an explanation in less basic terms is required, refer to the Draw2D documentation. InteractorLayout provides the following functionality:

- It positions the label of the interactor inside the label area.
- It looks through throught the list of the figure's children, finds the ones which represent DiaMODL gates, and then resizes and positions them correctly along the left-hand side of the InteractorFigure.
- It will, in time, also make sure that the remaining contents of the figure are positioned inside the content area. This is not implemented at the time of writing.

A final note: The task of adding figures to their appropriate parent figures is handled by the GEF framework itself, and is done in such a way that the hierarchy of figures corresponds to the structure of the model. Earlier versions of InteractorFigure were built up from several different sub-figures to which the child-figures would be added. For example, all gates were added to a special «gateBox» figure, contained children were added to the «contentsBox», and so on. I do NOT recommend this approach, as GEF is quite insistant about keeping the hierarchy of figures the way the model prescribes. While it is definately *possible* to defy GEF on this point, it will most likely not be worth the trouble.

ComputationFigure:

ComputationFigure is the figure representing a DiaMODL computation in the editor. While the sourcecode might seem a bit long, it is actually not all that complicated. Much of the space is used by some lengthy switch-case statements dealing with the possibility of rotating the figure, and another large chunk of the space is occupied by a relatively simple inner class, TwoPartTriangle. In summary, ComputationFigure does the following:

- Positioning the label on the figure.

- Drawing the triangular shape which represents a computation in DiaMODL. The geometry is handled by the inner class TwoPartTriangle. The reason for using this class rather than the «Triangle» class provided by Draw2D, is the need to be able to hide either the root or the tip of the triangle (not supported by the editpart yet, but this is planned for a future revision).
- Calculating inner and outer bounds.
- Calculating the position of anchorpoints along the triangular outline.
- Providing a preferred *direction* for the anchors. The direction specifies in how edges should connect to the the anchor (see chapter 2.1.1.5)

GateFigure:

GateFigure is a subclass of ComputationFigure, which is used for DiaMODL gates. It overrides a single method in ComputationFigure, which causes the outline to be drawn with a line through the middle, separating the base and tip.

BeanPlaceFigure:

The last class in the package is BeanPlaceFigure, which represents DiaMODL beanplaces. It is a very simple figure, as a beanplace is usually just a rectangle with a centered text label. If the beanplace has a «multi-value» value type (such as a list), the figure should be drawn with slightly transposed rectangle behind the main one. BeanPlaceFigure provides the following functionality:

- Calculating inner and outer bounds.
- Toggling between showing and not showing the multi-value indicator.

2.2.2.3 Commands and editpolicies

The UIML DiaMODL plugin subclass one Command and one EditPolicy from the generic plugin. This need to be done in order to add support for creating DiaMODL *connections* rather than just simple *edges* between nodes.

DiamodlGraphicalNodeEditPolicy:

This editpolicy is a subclass of DiagramGraphicalNodeEditPolicy, which overrides a single method to add support for connections. In practice it inserts a check to see if the new edge being created matches the *Connection* model element, and if it does, the CreateConnectionCommand is used instead of the generic CreateModelElementAssociationCommand.

CreateConnectionCommand:

The CreateConnectionCommand is a subclass of CreateModelElementAssociationCommand, and works in a very similar way. The only difference is that CreateConnectionCommand also sets the «source key» and «target key» values in the model. These values are used to determine to which anchor point a connection is attached for nodes which have more than one.

2.2.2.4 Other files

In addition to the packages described in the previous chapters, the DiaMODL plugin relies on a few other elements to work. This chapter describes these elements.

The «plugin.xml» file:

The «plugin.xml» file is the main configuration file for the DiaMODL plugin. The purpose of this file is to tell Eclipse how the DiaMODL plugin connects to the UIML Diagram plugin. This is done through «extension points». The UIML Diagram plugin defines two extension points:

- *no.hal.uiml.editPartFactory*, which is used to specify the language specific editpart factory
- *no.hal.uiml.toolPaletteEntry*, which is used to specify the entries for the tool palette

Figure 23 shows how the DiaMODL plugin connects to the editPartFactory extension point, telling the generic plugin where to find the editpart factort, as well as the correct prefix for the classes in the factory. The entry for the toolPaletteEntry extension point is too long to paste here, but an exerpt is shown in figure 24. The exerpt show the entries for two tools, one to create a DiaMODL interactor, and one to create a beanplace. More information on the meaning of the entries is found in chapter 3.3.2.

```
<extension point="no.hal.uiml.editPartFactory">
    <edit-part-factory
        factory-class="no.hal.uiml.diamodl.editparts.DiamodlEditPartFactory"
        model-class-prefix="uiml.diamodl.">
    </edit-part-factory>
</extension>
```

Figure 23 - entry for the editPartFactory extension point

```
<extension point="no.hal.uiml.toolPaletteEntry">
    <tool-palette-entry
        tool-entry-class="org.eclipse.gef.palette.CombinedTemplateCreationEntry"
        short-label="Interactor"
        long-label="Create interactor"
        small-image="icons/object-device.gif"
        large-image="icons/object-device.gif"
    >
    <creation-factory
        creation-factory-class="no.hal.uiml.diagram.editparts.LibraryDiagramCreationFactory"
        object-class="uiml.diamodl.Interactor"
        object-name="Standard interactor"
    />
    </tool-palette-entry>
    <tool-palette-entry
        tool-entry-class="org.eclipse.gef.palette.CombinedTemplateCreationEntry"
        short-label="Bean place"
        long-label="Create bean place"
        small-image="icons/bean-place.gif"
        large-image="icons/bean-place.gif"
    >
    <creation-factory
        creation-factory-class="no.hal.uiml.diagram.editparts.LibraryDiagramCreationFactory"
        object-class="uiml.diamodl.BeanPlace"
        object-name="Bean place"
    />
    </tool-palette-entry>

....

</extension>
```

Figure 24 - exerpt from the entry for the toolPaletteEntry extension point

The «library.diagram» file:

The «library.diagram» file holds prototypes for the objects which can be created with the tools defined in «plugin.xml». The library.diagram for the DiaMODL plugin currently holds prototypes for three different types of interactor objects (with gates), one beanplace and one computation object. How the «library.diagram» file is used in object creation was briefly discussed in chapter 2.1.2.3, and some more information is available in chapter 3.4.

The «icons» and «resources» directories:

The «icons» and «resources» directories contain various graphics files used by the editor. The «icons» directory contains the icons used for the tool palette entries (as defined in «plugin.xml»). The «resources» directory is not currently in use.

The «DiamodlPlugin.java» file:

«DiamodlPlugin.java» is the main program file for the DiaMODL plugin. It provides the methods necessary for Eclipse to launch the plugin. This file was generated by Eclipse when creating the plugin project.

3. Making language specific plugins

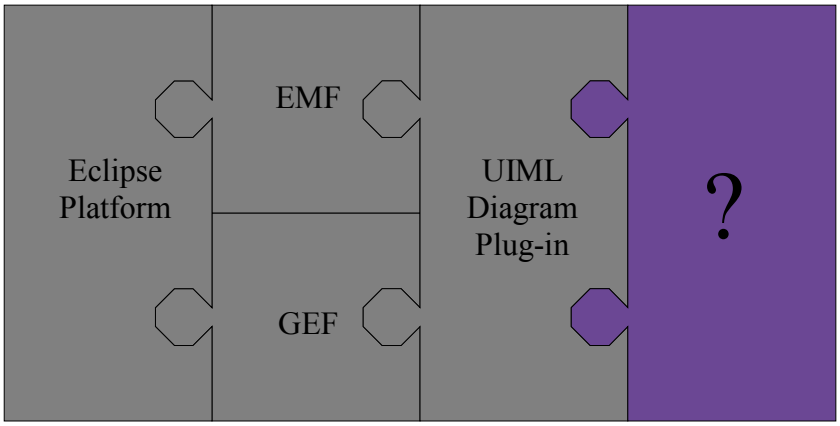
The UIML Diagram plugin is designed to be extended with language specific plug-ins. This chapter provides instructions on how to do this.

Chapter 3.1 gives a basic overview of how you can create a language model, by using EMF.

Chapter 3.2 covers how to extend the UIML Diagram classes, to add the extra functionality needed for your editor.

Chapter 3.3 describes how to configure your plugin through the «plugin.xml» file.

Chapter 3.4 discusses how to build the prototype library.



3.1 Building a language model

The first thing which needs to be provided when creating a language specific plugin, is a working implementation of the *language model*. The language model specifies the syntax of the language. As I mentioned briefly in chapter 2.1.1, it is our intention to split the model in the «diamodl-emf» project into a generic and a language specific part. This will reduce the effort required to build a plugin for a new language significantly. Unfortunately, at the time of writing, the mechanisms to support this are not yet ready. What this means, is that if you want to make a language specific plugin right now, you will have to create a complete model, which duplicates the generic parts of «diamodl-emf»'s model, but replaces the diamodl-specific part. This should not be overly difficult, but it does require some understanding of the contents of the «diamodl-emf» project. Hopefully, the following subchapter will provide some of this understanding.

3.1.1 Understanding the model of the «diamodl-emf» project

The model of the «diamodl-emf» project is stored in the file «diamodl-metamodel.ecore», which can foun in the «src/model/» subfolder of the project. This is an EMF ecore file, which can be viewed and edited by using the «Sample ecore model editor» which comes with EMF. There are also other editors which will do the job (such as the graphical Omondo EclipseUML editor), but the standard editor suffice for most basic needs. For information on creating models using the various editors, refer to their respective documentations.

Figure 25 shows the model file viewed in simple EMF editor.

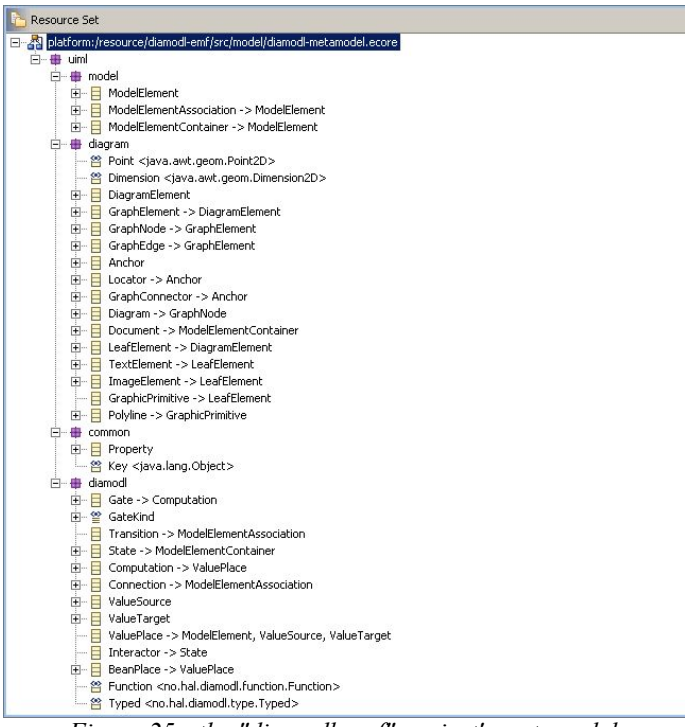


Figure 25 - the "diamodl-emf" project's metamodel

As you can see, the model is divided into four packages:

«uiml.model»:

This is the generic part of the semantic model. It is very simple, consisting of only three classes: ModelElement, ModelElementAssociation and ModelElementContainer. Any language-specific model must be mapped onto this generic model. This should not be very difficult. Normally, it will be a matter of making sure that your *nodes* are subclasses of either ModelElement or ModelElementContainer, and that your *edges* are subclasses of ModelElementAssociation.

«uiml.diagram»:

This is the diagram model of your language. As you might have suspected from the figures in chapter 2.1.1, this part is generic. This means that you should not make any language-specific extensions it. If you need to store more diagram information than what is explicitly supported by the diagram model, this can be done by adding generic «properties» to its objects (see below). The diagram model is based on the UML Diagram Interchange Metamodel specification.

«uiml.common»:

This package holds the generic «Property» class mentioned in the previous section. This class is used by the UIML Diagram plugin to allow setting and reading properties which are not explicitly listed as fields in the diagram model. This provides a simple and clean mechanism for storing extra information in a diagram, without the need to alter or add to the diagram model.

«uiml.diamodl»:

And finally, the language specific part. This is the semantic model of the DiaMODL language, which specifies its syntax. Notice how it is mapped onto the generic semantic model, with every type of node and edge being subclasses of one of the classes in the «uiml.model» package. There are three entries which are neither nodes or edges: *GateKind* is an enumeration, while *Function* and *Typed* are datatypes. These are used in the fields of various DiaMODL classes, and are not part of the model's class hierarchy.

3.1.2 Making your own model

For your own language specific plugin, the model should be the same as the one described in the previous chapter, except, of course, the «uiml.diamodl» package. This should be replaced with your own language specific package (probably something along the lines of «uiml.mylanguage»). Remember to make sure it is properly mapped onto the generic model in «uiml.model». Once you have created the ecore file, you can use EMF's code generation facilities to build a software implementation of your language model. Information on doing this can be found in the EMF documentation.

Next, you might want to make some modifications to the code generated by EMF. By default, any

field you read from the model will be interpreted as a Java *String*. If you want to receive a different type of object from the model, you will need to implement some kind of parsing functionality at some stage. The latest version of «diamodl-emf» uses a rather clever mechanism for this purpose: Instead of hardcoding the parser code directly into the methods which return values from the model, the plugin offers a «syntaxRegistry» extension point. This makes it possible to simply «plug in» the parsers you need. As an example, the latest version of the DiaMODL plugin uses this extension point to add support for the «pnuts» syntax (the pnuts scripting language), greatly extending the functionality of the plugin. Of course, the «syntaxRegistry» extension point will be included with the generic model plugin when it is finished.

3.2 Editparts, figures, and more

The majority of the time when creating your plugin will probably be spent extending the generic UIML Diagram classes with the functionality required by your language. Most of your new classes will be editparts and figures, but there is also a fair chance you might need to make one or two new commands and editpolicies as well. This chapter discusses how to do this.

3.2.1 Extending editparts

You should create, at least, one editpart for every object class (type of node or edge) in your modelling language. As an example, lets imagine a simple interaction modelling language, «MockupML», built up from three classes:

- *Users*, as shown in figure 26.
- *System modules*, as shown in figure 27.
- *Information flows*, as shown in figure 28.

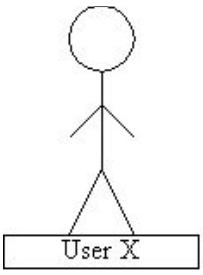


Figure 26 - a MockupML user

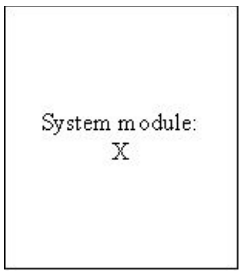


Figure 27 - a MockupML system module

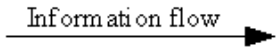


Figure 28 - a MockupML information flow

Figure 29 shows an example MockupML diagram.

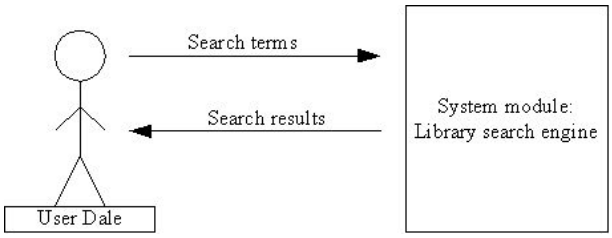


Figure 29: - an example MockupML diagram

An plugin for the MockupML language would need three editparts: UserEditPart, SystemModuleEditPart, and InformationFlowEditPart. In order to integrate properly with UIML Diagram plugin, these classes should be subclasses of the appropriate generic editparts. Since «user» and «system module» are nodes in the graph, they should extend GraphNodeEditPart. «information flow», being an edge, should extend GraphEdgeEditPart. Furthermore, each of these classes should hold, at the very least, two methods:

- «protected IFigure createFigure()», which will build a figure object.
- If the figure of the editpart uses data from a field from the model, it needs to override the notifySet() method to react to changes to that field. The exception is if that field is the «Name» field. The «name» field is, by default, listened to and set as the label of any figure in the graph.

Figure 30 shows an example skeleton implementation for the InformationFlowEditPart. We have decided that the «Information type» field in the language model is going to be used for the text above the arrow, rather than the «Name» field. «notifySet» has therefore been overridden.

```
package yourcountry.you.uiml.mockupml.editparts;

import org.eclipse.draw2d.IFigure;
import yourcountry.you.uiml.mockupml.figures.SomeFigure;

import no.hal.uiml.diagram.editparts.GraphEdgeEditPart;

import uiml.model.DiagramPackage;
import uiml.mockupml.MockupMLPackage;


public class  InformationFlowEditPart extends GraphEdgeEditPart {

    protected void notifySet(Notification notification) {
        switch (notification.getFeatureID(DiagramPackage.class)) {
            case MockupMLPackage.INFORMATION_FLOW__INFORMATION_TYPE:
                //do something!
                break;
            default:
                super.notifySet(notification);
                break;
        }
    }


    protected IFigure createFigure(){
        return new SomeFigure();
    }

}
```

Figure 30 - skeleton implementation of the InformationFlowEditPart

To learn more on creating language specific editparts, I recommend looking at the DiaMODL plugin, both the description in chapter 2.2.2.1, and the actual source code.

In addition to creating the editparts themselves, an editpart-factory is needed. The task of this factory is to create a mapping between the model objects and the correct editparts. When a model is opened in the editor, GEF will look at each model object, and create an editpart for it. The factory is quite simple to make, and is probably best understood by looking at an excisting one. For this purpose, you could either examine the DiamodlEditPartFactory in the DiaMODL plugin, or the editpart-factory for our example language, MockupML, which is shown in figure 31 below.

```
package yourcountry.you.uiml.mockupml.editparts;

import org.eclipse.gef.EditPart;
import org.eclipse.gef.EditPartFactory;

import uiml.diagram.GraphElement;
import uiml.mockupml.User;
import uiml.mockupml.SystemModule;
import uiml.mockupml.InformationFlow;


public class MockupMLEditPartFactory extends EditPartFactory{

    public EditPart createEditPart(EditPart context, Object model) {
        EditPart editPart = null;
        if (model instanceof GraphElement) {
            Object semanticModel = ((GraphElement)model).getSemanticModel();
            if (semanticModel instanceof User) {
                editPart = new UserEditPart();
            } else if (semanticModel instanceof SystemModule) {
                editPart = new SystemModuleEditPart();
            } else if (semanticModel instanceof InformationFlow) {
                editPart = new InformationFlowEditPart();
            }
        }
        if (editPart != null) {
            editPart.setModel(model);
        }
        return editPart;
    }
```

Figure 31 - example editpart-factory for MockupML

3.2.2 Extending figures

In addition to editparts, you will need to make figures for your plugin. Basically, you will need to make one figure for every type of node in your the modelling language. For edges, there is a good chance you can make do with the provided GraphEdgeFigure. It extends Draw2D's PolyLineConnection, and has a quite decent amount of functionality. If you decide you need a different figure for your edges anyway, you *must* make it a subclass of GraphEdgeFigure. Similarly, all your node figures have to be subclasses of GraphNodeFigure. The reason for this is that the editparts rely on being able to cast figures as either GraphNodeFigure or GraphEdgeFigure.

For the MockupML language described in the previous chapter, you would only need to make two figures: UserFigure and SystemModuleFigure. For the «Information Flow» object, GraphEdgeFigure would be sufficient. All that needs to be added to the default implementation is an arrow decoration at the target end of the line, and this can easily be done when creating the figure in InformationFlowEditPart. The nodes would need a little more work, though.

While you are largely free to make your figures any way you wish to, there are some abstract methods in GraphNodeFigure which need to implemented if things are to work properly. These are:

- *protected abstract IFigure createOutline():*

  This method should create and return a figure which represents the basic shape of the figure, that is, the *outline* of the figure.

- *public abstract Rectangle calculateOuterBounds(Rectangle outlineBounds):*

  This method should return a rectangle which represents what the outer bounds of the figure would be if the outline had the size given by the outlineBounds parameter. The outer bounds

of the figure is the smallest rectangle which encloses all its elements. The difference between outer bounds and outline bounds is illustrated in figure 32, using a DiaMODL interactor from the DiaMODL plugin as an example.

- *public abstract Dimension calculateOutlineSize(Rectangle newBounds):*

  This method should calculate what the *size* of the outline would be if the outer bounds were set to the value given by the parameter.

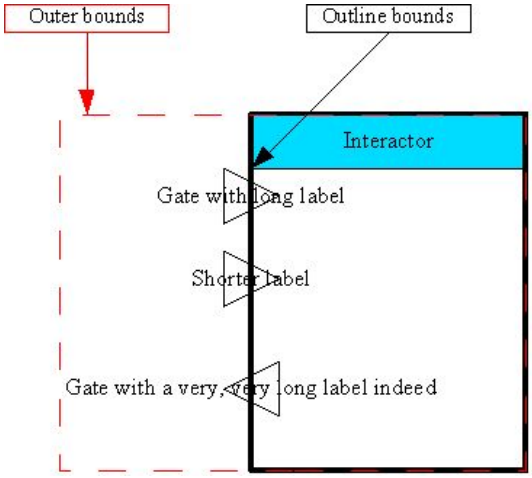


Figure 32 - Outer bounds and outline bounds

Note: Depending on which version of GraphNodeFigure you are looking at, you may see some other abstract methods, namely createChildren(), positionChildren() and arrangeChildren(), which would need to be implemented in order to subclass GraphNodeFigure. Default implementations will be made for these in the near future, and for now, they can be safely ignored.

I will not write anything here about general use of Draw2D, as that is already covered by the Draw2D documentation and other online resources (see appendix A). You may wish to have a look at the figures in the DiaMODL plugin for some ideas, but be aware that although they certainly work, these are probably not perfect examples of Draw2D usage.

3.2.3 Extending other classes

Apart from editparts and figures, you might also need to override at least one command and an editpolicy. The command you are most likely to want to overwrite, is the one used to create associations between objects. By default, any edges you create in the graph will only store a minimum of information, just enough to know which nodes it connects. To properly support richer edge types, you should do the following:

1. Make a command to create the type of edge you need. As an example, the an editor for the MockupML language could have a command called «CreateInformationFlowCommand», which

would be used when creating MockupML informationflows. The best way to make the new command is to extend CreateModelElementAssociationCommand in the generic plugin, and add the extra code to set the fields specific to your connection type.

2. Make a subclass of DiagramGraphicalNodeEditPolicy, and override the getConnectionCreateCommand method. Make it return the command from the previous step when your type of edge is being created.
3. Install the new editpolicy on the editparts in your plugin, in the GRAPHICAL_NODE_ROLE.

If the above procedure seems a little confusing, I suggest you have a look at how it is done in the DiaMODL plugin. There, the new command is CreateConnectionCommand, the new editpolicy is DiamodlGraphicalNodeEditPolicy, and it is installed on all DiaMODL editparts in the DiamodlEditPart class.

3.3 Plugging in - extension points

Extension points are the primary mechanism provided by Eclipse to allow integration of different plugins. A plugin which is designed to share its functionality with other plugins, such as the UIML Diagram plugin, provides extension points where other plugins may «connect» itself. Extension points are both defined and connected to through the «plugin.xml» file. For more information about extension points, refer to the Eclipse documentation.

The UIML Diagram plugin provides two extension points which you will need to connect to when making a language specific plugin. These are described in the chapters below.

3.3.1 The «editPartFactory» extension point

The *no.hal.uiml.editPartFactory* extension point is used to specify where the editpart-factory for your plugin can be found. There are two attributes which need to be specified:

«factory-class»: This is where you specify where the factory class can be found. It should be a complete address on the form «<package name>.<class name>»

«model-class-prefix»: This tells the Diagram plugin what the correct class prefix of the model objects in your language is. It should end with a punctuation.

As an example, see the editpartFactory entry in the DiaMODL plugin's «plugin.xml» file (figure 23), or the example entry in figure 33.

```
<extension point="no.hal.uiml.editPartFactory">
    <edit-part-factory
        factory-class="yourcountry.you.uiml.mockupml.editparts.MockupMLEditPartFactory"
        model-class-prefix="uiml.mockupml.">
    </edit-part-factory>
</extension>
```

Figure 33 - entry for the editPartFactory of the MockupML plugin

3.3.2 The «toolPaletteEntry» extension point

The *no.hal.uiml.toolPaletteEntry* extension point is used to specify which entries your plugin wants to add to the tool palette of the editor. Here you will typically have entries for each model element you want to be able to create with the editor. The information is divided into too parts: one to specify the properties of the button itself, and one to tell the Diagram plugin how to create the object. The attributes which need to be set are the following:

tool-palette-entry:

«tool-entry-class»: The type of the entry, as defined in the org.eclipse.gef.palette package.

«short-label»: The short label for the button, that is, the one which will actually be printed on it.

«long-label»: The long label, which will be used as the tooltip when the pointer hovers over the button.

«small-image»: The image which will appears next to the short label.

«large-image»: A larger version of the «small-image». This attribute is not currently used.

creation-factory:

«creation-factory-class»: This attribute tells the Diagram plugin where to find the factory which will create the new object. Set it to "no.hal.uiml.diagram.editparts.LibraryDiagramCreationFactory".

«object-class»: Here you specify the general class of the model object you are going to create.

«object-name»: This specifies which *variant* of the model object you wish to copy from the prototype library.

Again, for examples, examine the toolPaletteEntry in the DiaMODL plugin's «plugin.xml» file (figure 24), or have a look at the MockupML example below (figure 34).

```
<extension point="no.hal.uiml.toolPaletteEntry">
    <tool-palette-entry
        tool-entry-class="org.eclipse.gef.palette.CombinedTemplateCreationEntry"
        short-label="User"
        long-label="New User"
        small-image="icons/user-small.gif"
        large-image="icons/user-big.gif"
    >
    <creation-factory
        creation-factory-class="no.hal.uiml.diagram.editparts.LibraryDiagramCreationFactory"
        object-class="uiml.mockupml.User"
        object-name="Default User"
    />
    </tool-palette-entry>

....more entries here...

</extension>
```

Figure 34 - example toolPaletteEntry for the MockupML plugin

3.4 Creating a library file

In order to create new model objects, you will need to a prototype library. Refer to chapter 2.1.2.3 for general information on how objects are created by the UIML Diagram plugin. This chapter contains practical information on how to build the prototype library.

3.4.1 Creating and editing the «library.diagram» file

Once you have built the language model in EMF, there are several ways in which the library file can be written. One very direct way is to use a text editor, and write the file by hand, using the XMI language (XML Metadata Interchange). An example of what the DiaMODL plugin's library file looks like in a text editor, is shown in figure 35.

```
<?xml version="1.0" encoding="UTF-8"?>
<uiml.diagram:Document xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:uiml.diagram="http:///uiml/diagram.ecore"
xmlns:uiml.diamodl="http:///uiml/diamodl.ecore">
    <children xsi:type="uiml.diamodl:Interactor" name="Standard interactor">
        <children xsi:type="uiml.diamodl:Gate" name="Object output"/>
        <children xsi:type="uiml.diamodl:Gate" name="Object input" kind="inputSend"/>
    </children>
    <diagrams semanticModel="/" name="interface uiml.diamodl.Interactor">
        <contained xsi:type="uiml.diagram:GraphNode" position="10.0,10.0" semanticModel="//@children.0" size="60.0x120.0">
            <contained xsi:type="uiml.diagram:GraphNode" position="-5.0,30.0" semanticModel="//@children.0/@children.0" size="20.0x20.0"/>
            <contained xsi:type="uiml.diagram:GraphNode" position="-5.0,80.0" semanticModel="//@children.0/@children.1" size="20.0x20.0"/>
        </contained>
    </diagrams>
</uiml.diagram:Document>
```

Figure 35 - exerpt from the DiaMODL plugin's prototype library, in plain text

While this might be a possibility for very small languages (like MockupML), it is probably going to be very tedius (and difficult) for most. A quicker and simpler approach is to use the diagram model editor provided by the EMF.edit framework, which should have generated when you built the language model. This will give you a tree-view editor, as shown in figure 36 below.

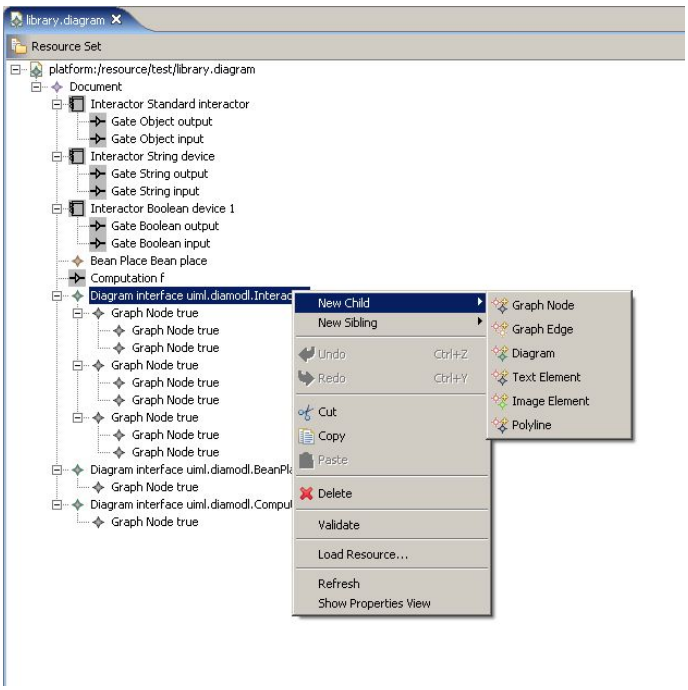


Figure 36 - the DiaMODL plugin's prototype library, in the diagram model editor

For information on how to use the diagram model editor, refer to the EMF documentation.

3.4.2 The structure of the library

In order for UIML Diagram plugin to find prototype objects when they are requested, the library file needs to follow a strict structure. This structure is probably most easily understood by looking at examples (such as the DiaMODL library shown in figure 36), but I will give a verbal summary of the rules as well. I suggest reading it very slowly, while frequently looking at the example shown in figure 36:

1. The first thing you need to do is to create the root node of the diagram. This should be of the «Document» class. In the EMF diagram model editor, the class of the root node is chosen in the «New Diagram Model» wizard.
2. Next, for every prototype you need, create a semantic model entry as a child of the root node. In figure 36, the entries labeled «Interactor Standard Interactor», «Interactor String Device», «Interactor Boolean Device», «Bean Place Bean Place» and «Computation f», are the semantic model entries. You may set the properties and build the substructure of each entry as you wish. The properties and substructures will form the default configuration for each prototype. Note that the value you choose for the «Name» property will be used to locate the prototype (this is the «object-name»-attribute described in chapter 3.3.2)
3. Now to the tricky part. Each of the semantic model entries will need a corresponding diagram model entry. The first step will be to create a node of the class «Diagram» for each semantic model class of the entries described in point 2. For the DiaMODL library in figure 36, that would mean one Diagram node for Interactor, one for BeanPlace and one for Computation. The name field for each of the nodes should be set to «interface <semantic model class name>». Again

using the DiaMODL library as an example, the node names should be «interface uiml.diamodl.Interactor», «interface uiml.diamodl.BeanPlace» and «interface uiml.diamodl.Computation». Now, set the «Semantic model» property for the *first* of these «Diagram» nodes so that it points to the root node, «Document» (you will get to choose it from a pull-down menu, where it should be right at the top).

4. Next, for each «Diagram» node, create «Graph Node» entries for each of the semantic model prototypes of the class described in that «Diagram» node's name. Confused yet? Once again, look at figure 36. Under the «Diagram» node labeled «interface uiml.diamodl.Interactor», there are three children, one for each of the interactor entries above in the «semantic» part of the tree. Once you have created the «Graph Node» entries, set their «Semantic Model» properties so that each of them so that they point to a semantic model node of the appropriate type.
5. Create child nodes for each of the nodes from point 4, so that the structure of each corresponds to that of the nodes' semantic models, and set their «Semantic Model» properties so that they each of them points to the corresponding nodes in the semantic model. Figure 37 below illustrates how this is done in the DiaMODL library. Note that you are free to set any other property of the «Graph Node» entries to whatever you like. The values you choose for each property will be the defaults for the prototypes.

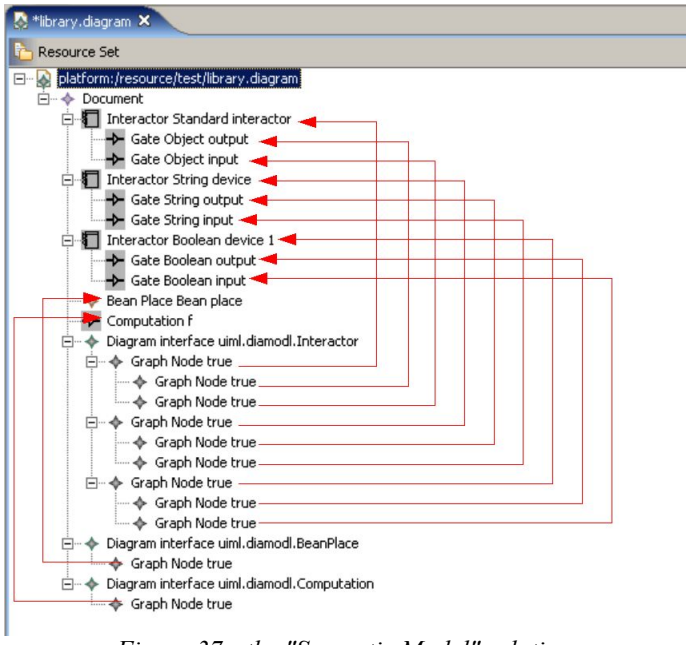


Figure 37 - the "Semantic Model" relation

4. Experiences and results

In this chapter I will sum up the status of the project, as well as share some of my own experiences from working on it.

Chapter 4.1 is about the status of the project itself, with emphasis on what progress has been made.

Chapter 4.2 contains some of my personal experiences working on the project.

Chapter 4.3 discusses future work.

4.1 Preliminary project summary

While there is still much work to be done before the project can be classified as «finished», I think it is fair to say that it has come a long way since its inception. Over the last couple of months, we have had the pleasure of seing many parts of the editor «snap into place». The majority of the central design choices have been made at this point, and what remains is mostly to refine and expand upon what has been done.

In chapter 1.3 I mentioned the GMF project, which has been running simultaneously with our own. While their goal is somewhat different from ours, many of the problems they have to solve are similar. It is therefore interesting at this point to compare the solutions which have been proposed in the GMF project with our own. The GMF project is significantly larger than ours, and is backed by many strong contributors, including IBM themselves. It is encouraging to observe that the solutions we have arrived at independently are similar to those proposed in that project on many points.

Over the coming weeks work on the UIML Diagram plugin, as well as the DiaMODL plugin, will continue at full force. It is likely that significant improvements will be made in the near future.

4.2 Personal experiences

As I mentioned in the preface, my contributions to this project form my master thesis, the final part of my five years of computer science education. I am by no means new to computer programming, having written many, many thousands of code lines in various languages since I wrote my first lines of BASIC, on the Dragon 32 I had when I was ten. Java has been my primary language for five years now, and I consider myself to be relatively proficient with it. Nothing I had encountered as a student prior to this project had caused me too much of a headache, so while I went into it looking for some challenge, I was never worried about not being up to the task.

My initial confidence was, however, soon replaced with a great deal of respect for the complexity of real-world development projects. To be honest, my first reaction when seing the project folders was that of being overwhelmed. I soon realized that this project would be different from anything I had done earlier:

- It was *much* larger than my previous projects. My previous projects had usually covered at most 30-40 source files, not more than what you can barely keep track of mentally.
- With my previous projects, I had built pretty much everything from scratch. This, I soon discovered, requires a different set of skills than building on top of large frameworks developed by someone else.
- Up until this point, my primary programming tool had been the plain text editor. This project required not only that I *used* the Eclipse IDE, but that the software I wrote should be strongly integrated with it.

Fortunately, thanks both to persistence and good tutoring, I have gradually been able to comprehend more and more of the system. It has been a challenging process, but also a very educational one. Obviously, working with a more experienced programmer, I have been able to pick up new tricks and nuances of the Java language. Additionally, I have developed better skills for understanding interaction between different software modules, and for reading other developers' source code. But apart from these purely technical skills, I have also learnt a few more general lessons.

The first I would like to mention that learning to use powerful development tools, like Eclipse, is actually worth the effort. While working on small projects I never bothered with IDEs and other tools. I wanted to get right into the code, right away. Being a practical man, I rarely use a tool before I *need* to use it, and so far I had been able to manage fine with writing the code in simple text editors and using text output for debugging. For this project, the *need* to use more sophisticated tools finally came. Without the many powerful functions provided by Eclipse, it would not have been possible for me to keep track of a project of this size. In particular, having easy access to essential structural information, like class and call hierarchies, have proved to be invaluable.

Another lesson is that building on top of ready-made frameworks is not necessarily any less difficult than building a system from scratch. Time-saving in many cases, yes, but not necessarily easier, especially for a relatively inexperienced programmer. During this project, I believe I have spent at least as much time trying to understand features of the underlying frameworks, as I have spent on actually writing code. The lesson to be learnt from this is not that you should not use frameworks, but rather that system integration is a challenging discipline, and one where I need to improve my

skills.

On a related note, this project has increased my appreciation of good documentation. During my work, I have encountered features and concepts which have been everything from completely undocumented, to almost over-documented. And of course some which are well documented.

To sum it up, this has been both a challenging and educational project for me. Fortunately, I will not have to leave it just yet, as I will be employed as a developer for the duration of the summer. The project has reached its most satisfying phase, where we start to see major functionality snapping into place. I am looking forward to seeing what progress can be made over the coming weeks.

4.3 Future work

At the time of writing, there is still a significant amount of work remaining before the project reaches full maturity. While the general structure is mostly in place, there is a definite need for expansion and refinement. In particular, there is a need for improvement in the following areas:

- Graphical features, both in the generic and DiaMODL plugins. For example, current connection routing leaves something to be desired, and there are some positioning bugs to contend with.
- API clarity and consistency, in the generic plugin. There are functions which have been created as «quick and dirty» solutions to short term problems. These will have to be considered carefully, and replaced with cleaner and more consistent mechanisms where it is required.
- Commands and general editor behaviour. Not all commands are currently working as intended, either failing to execute properly under certain circumstances, or lacking undo/redo support. Additionally, there is probably room for adding more commands and editor functionality.

The one overall design issue where a solution has not yet been developed, is that of separating the general and language-specific parts of the language metamodel. We do not believe that this is excessively difficult to do, but it has not been a priority so far in the project. It will need to be addressed at some point, though.

Finally, as the project continues to evolve, so must the documentation. The relevant parts of this document will be updated as needed.

Appendixes

Appendix A – Suggested reading

Online resources

«GEF description» at the Eclipse Wiki

Where: http://www.eclipsewiki.editme.com/GefDescription

What: A well written, easy to read introduction to GEF and Draw2D

«A shape Diagram Editor» at eclipse.org

Where: http://eclipse.org/articles/Article-GEF-diagram-editor/shape.html

What: A fairly easy to grasp example of a basic GEF editor.

«Display a UML Diagram with Draw2D» at eclipse.org

Where: http://eclipse.org/articles/Article-GEF-Draw2d/GEF-Draw2d.html

What: A short, but useful example of Draw2D usage.

«All about Eclipse Plug-ins» at Addison Wesley Professional

Where: http://www.awprofessional.com/articles/article.asp?p=370626&rl=1

What: Sample chapter from the book «Official Eclipse 3.0 FAQs». Gives good insight into how Eclipse plugins work.

Offline resources

«The Java Developers guide to Eclipse» (2nd edition)

by D'Anjou, Fairbrother, Kehn, Kellerman and McCarthy (ISBN 0-321-30502-7)

This is a very comprehensive guide to Eclipse, covering everything from the basics to advanced topics.

«Eclipse Development using the G.E.F. and the E.M.F.»

by Moore, Dean, Gerber, Wagenknecht and Vanderheyden (ISBN 0738453161)

A good book which covers EMF, GEF, and how to use the frameworks together. Also available online at http://www.redbooks.ibm.com/redbooks/pdfs/sg246302.pdf.

Appendix B – References

(Markopoulos, 1997) Markopoulos, P. *A compositional model for the formal specification of user interface software*. PhD thesis at the Department of Computer Science, Queen Mary and Westfield College, University of London, 1997.

(Trætteberg, 2002) Trætteberg, H. *Model-based User Interface Design*. PhD thesis at the Department of Computer and Information Sciences, Faculty of Information Technology, Mathematics and Electrical Engineering, Norwegian University of Science and Technology, 2002.

(Trætteberg, 2003) Trætteberg, H. *Dialog modelling with interactors and UML Statecharts – A hybrid approach*. DSV-IS 2003 (http://math.uma.pt/dsvis2003).

Appendix C – Figure index