Information Visualization in Eclipse

FREDRIK ASP
PETER ODÉN

Examiner
Dr. Christian Schulte

Master of Science Thesis
Stockholm, Sweden 2004

ICT/ECS-2006-74

Abstract

Technology Companies today need evermore data that is vital to their function; this can be product, customer or other sorts of databases. Very few companies have any user friendly interface for browsing these vast quantities of data, instead they rely on query – response schemes with a very basic interface that have evolved very little over last 40 years.
The current research in the field of data visualization has focused on data mining and displaying how the data is stored, not how to effectively display an object and the structure it is a part of.
What we propose to do is a simple relational database browser that can both speed up finding data and most importantly helps users to easy get a broad understanding of the structures of data.

With a query – response scheme it is very easy and fast to get information about an object in the database, that is if the object is known by the user.
But if the information wanted is about the context or structure of objects then several consecutive queries will be needed and only a little of the information will be shown at any given time, meaning that the user has an hard time figuring out the exact structure. This is largely dependant on the fact that the user only can look at one object and its relations at a time. This effectively narrows the users understanding of the structure down to objects one step from the object he is currently browsing.
With our browser we will address this problem as its surrounding objects will be shown. Not only the objects in the immediate vicinity, rather the user can choose how many steps from the object he will see.

We did this browser as a plug in for eclipse as it is one of the fastest growing open source platforms available right now. It is maintained by the Eclipse group witch has the support of over 50 companies [7], so it is probable that it will be kept updated for some time to come giving our system a stable platform that doesn't become outdated.
Also eclipse is a contender for being the next de facto standard in software development as it is not only a development environment but rather a framework that allows great configurability. This is due to the plug in structure that allows the user to modify eclipse into the tool that fills his every need.

We have also used this property of eclipse in our project as we use GEF or the Graphical Editing Framework to give our tool its graphical representation. GEF is a plug in that gives a powerful framework for graphics so that we didn't have to code a graphical engine from scratch. Lastly we have done some tests for performance and functionality to show that our tool does what it is supposed to do within reasonable time.

Acknowledgments:
We would like to thank Dag Rende and Peter Roos at Total Eclipse AB for all their help during this project and also Christian Schulte for supervising our thesis. Also we would like to thank all the members of SPX for their constant support.

Table of contents

List of Abbreviations

SWT - Standard Widget Toolkit
UI – User Interface
GUI – Graphical User Interfaces
GEF – Graphical Editing Framework

Key Expressions

Focused object	The object shown in the center of the focus view.
Selected object	The object that is selected. Properties of this object are shown in the object info view. And the object is highlighted in the focus view.
Relation type	A label on the relation between two objects in the database. “Consists of” and “replaces” are two examples.
Focus view	The central view of G-dip. It shows the object currently in focus and its neighbors.
Search view	A view that lists results of a search. From which a found object can be focused.
Object info view	A view that displays relevant info about the object currently in focus.
Changes view	A view that lists the change messages that the user subscribes to.
Active aspect / V view	A view that shows in what development aspect the object in focus resides.
Object	An entry in the database is shown in the focus view as a figure with a label.
Ping message	A notification message that informs a user that an database entry for which they subscribe to, has been changed.
Content provider	The model structure of some viewer or editor, it may be shared by several viewers and/or editors.

List of Figure

List of Tables

List of Formulas

1. Introduction

All industrial design and production processes today grow ever more complex and larger. For every bit of complexity there is much data added to the documentation of the project and also more data to be taken into account during the design. There is also more and more devises and software that are supposed to work together in larger groups cooperating through any number of media, further there are many standards that anything produced needs to consider. There is no sign of any diminishing of this growth of complexity. All this complexity and any device a product needs to cooperate with add even more data that has to be considered during the design. As the design phase is traversed it itself adds data that needs to be considered in later stages of design and construction. This data may be of many different types and very different in nature, much of this data or info on where to access it is stored in various relational databases.

Today the standard for accessing a relational database is different types of searches resulting in zero to several hits that can be displayed one at a time. If the user knows what he wants or at least its properties this is a very good way of finding info, but if she wants to find out what subcomponents a product has or the subcomponents subcomponents it starts to get less and less efficient. For every new object in the database the user wants to look at she needs to do another search and if she wants to get a grip on the component structure of a product it will be much work in a search based interface. It also has the drawback in the structure case that very little of the information can be shown at any given time.

What we propose to do is not replace or change the query - response scheme but to add a tool that has its strength where the query – response scheme is at its weakest. When our Graphical Database Interface Plug-in (GDIP) displays the info of a database element it will also show its related elements and their related elements and so on. So a user can easily get an idea of the structure of whatever she is looking at. GDIP is made to be very general so that it easily can be used with any relational database, and it also is very adaptable to the users needs, it is possible to change all parameters that control the drawing of the view of the database. Being built to be easily used with any relational database it is not very difficult to adapt it to access several different databases at once so that companies that have different databases for different kinds of data easily can access it all from a single application. GDIP is not supposed to take over all database manipulation, instead it is a browser allowing the designers easy access to the data they need

2. GUI

2.1 What are GUI's

GUI's or graphical user interfaces replaced the text based interfaces in what now can only be considered as the ancient times of personal computers. The GUI paradigm changed personal computers completely, granting an ease of use never before seen in computing and thus laying the foundation for the computer boom with a computer in virtually every home. Today GUI's is one of the pillars on which most of the IT industry stands. The idea behind the GUI is point and click. The user don't need to remember the name of each little subroutine, all he needs to do is select what he wants to work with point and click. To give the point and click paradigm more depth allowing more then one thing to be done to each selectable item there are several different ways to make point and click multiple action possibilities.

The most important generic features in GUI's are.

- Buttons
- Icons
- Text fields
- Checkboxes
- Menus
- Frame
- Dialogs

Table 2.1

Each of these comes in almost infinite number of variations but they are the basic blocks of all GUI's [8].

2.2 Building a GUI

Building a working GUI is not difficult, only somewhat time consuming. However building a GUI that just works is not the objective of any program designer, what is needed is an easily usable GUI and that is much harder to create [8]. A user that does not already know how to use the program must be able to use it without having to read the manual for every action that he needs to make. A program where users need to check the help guide too often will be considered difficult to use and few people will buy it. There is however no easy way to make an easy to use GUI, no recipe to follow. The designer will need to know how the user thinks, how the cultural background will influence reactions to different colors and innumerable other little things [8]. There are also many things that may seem to be good until the actual users start using the program. All these problems have to be solved to get a usable GUI. The GUI creation is one of the major parts of any program construction today.

2.3 Eclipse

Since man started programming the programs has become more and more complex and man wanting to produce these complex programs, without making the task extremely time consuming, understood that if the languages themselves where more powerful the burden would be eased. First there was raw bytecode which evolved into assembly language which in turn evolved into higher level languages. These evolved into the object oriented languages which finally produced the platform independent language Java.

Eclipse takes the evolution one step further by introducing a framework environment in open source where much of the basic functionality already exists. A programmer constructs a plug-in, a small eclipse dependant program or library. These are detected by eclipse when it's first started and it notes which extension point the plug-in extends. A plug-in can also use the functionality of an existing plug-in, as is the case with our plug-in since it uses GEF which is a plug-in for generating graphics. This solution eases the burden of writing code, but places a lot of responsibility on the programmer for understanding the mechanisms of eclipse and the supporting plug-ins.

One more benefit of using eclipse with its plug-in structure is that a user will do most of his work inside eclipse within different plug-ins. Since these all resides within eclipse he doesn't have to learn a new interface as all the standardized functions will be in the same familiar places as he is used to have them [6].

Eclipse is so much more then the regular GUI, it is the entire framework reducing much of the work of program construction. Instead of writing the entire program all that is needed is the essential computations and writing how to represent info on screen, All else eclipse takes care of. Because of this a user that has worked with eclipse based programs will have an idea of how to work a new eclipse based application. The eclipse platform has pre made stubs for many different functions such as load, save, undo, redo and many more leaving only the functional parts to the programmer. There is however difficult to get used to the very different programming for eclipse as opposed to a program written from scratch. In eclipse all methods and classes written are called by the eclipse framework, to get eclipse to do anything a stump of code doing it must be placed at a spot that is called by the framework. There are many tricks a programmer must learn before writing programs for eclipse will be effortless but when mastered it is a tool of much power leaving the programmer to write few lines of code to achieve much function.

Eclipse began as an IBM project. In November 2001, an IBM-independent foundation was formed to further the development of Eclipse. And it is this organization that has driven the development of Eclipse since [7].

2.3.1 SWT

Today any operating system (OS) need to have a almost full set of widgets for handling GUI:s and there are a few basic widgets that all OS have their implementation of [8]. There are some more complex widgets, the designers of each operating system have chosen those they wanted which means all operating systems with different tools for building GUI:s. It is desirable to use the systems widgets making the program look the way the user is used to and keeping a coherent look on screen. This hasn't been any problem for regular programming languages such as C++ where every system has its own compiler. Java on the other hand has to work on all systems with the same bytecode having to use only those widgets supported by all systems. More complex widgets have to be emulated by java which makes them not quite fit in among the other, emulated widgets are also slower in updating making them stand out even more. SWT is the Standards Widget Toolkit for eclipse have remedied this flaw by defining a common API for all supported window systems and for each separate system SWT uses its native widgets when possible, making eclipse programs look more like the system it is run on [6]. Whenever a window system doesn't support a widget that widget will be emulated with all the drawbacks mentioned earlier. SWT goes even further than this it has support for native system features such as drag and drop, it also can use components developed with OS component models such as ActiveX controls in windows.

2.3.2 JFace

JFace is a window system independent toolkit for common User Interface (UI) handling. Among other things it has classes for dialogs and wizards, but JFace also handles actions [5].

Actions are a way of abstracting user commands from how they are triggered; an action can be connected to a menu item or a button. The action keeps track of the widgets needed to trigger it like a menu item but it can easily be moved to another menu or to a toolbar button without changing or moving the code of the action. This helps mostly if the design or grouping of the GUI needs to be changed late in the construction, perhaps as a result of user testing.

2.3.2.1 Viewers

Viewers handle the higher levels of logic for SWT widgets, and are used to display modeled data using said widgets. There are a few standard viewers for common structures such as tables and lists. In eclipse a viewer is shown as a frame displayed on a part or on the entire main eclipse frame. Usually in eclipse there are many viewers displayed simultaneously, many of them show different contents but there are no problems with having multiple viewers having the same content provider. Viewers sharing their content provider don't necessarily show the same part of the data provided as they can have different logic for choosing what to show and how to show it [6].

2.3.3 Eclipse UI paradigm

The eclipse UI paradigm is based around three things: perspectives, views and editors. Views and editors are the types of user to data interfaces allowing the user to view and manipulate data, perspectives are a collection of views and editors with a common function such as the java programming perspective. Eclipse can handle several perspectives at any given time but to avoid confusion only views and editors belonging to the same perspective can be visible at any given time [6].

Editors have an open - save - close lifecycle, standard editors for text editing and some other simpler tasks are provided in eclipse and some more advanced are available in other plug-ins. Editors may have toolbar buttons and menu items bound to them so that they become available only when the editor is selected [6].

Views show data to assist the user in whatever task it is supporting. The java programming perspective has among others a view which shows the hierarchy of sub and super classes to the class currently edited. Views have less modifying power then the editors, but some things can be done in views. Most common is the changing of some property or name. The alterations made in a view are instantaneous as opposed to the editor where progress has to be manually saved. If there are several views showing the same information, change is propagated directly to all views without need for user initiated refreshes. There are generally several views for every editor visible in a perspective and there are many standard views provided with eclipse [6]. These standard views lessen the amount of repetitive work as almost every application needs some of the standard views. A property view shows the properties of the currently selected object. To use the provided property view it is only required that the selected object implements an interface and its methods, a task that once learnt is finished in a matter of minutes.

2.3.4 Projects

Eclipse is project based, so if some file is open in an eclipse application then that file must be part of one of the opened projects. The eclipse workspace is generally mapped to a single directory where each of the projects has its own subdirectory.

Eclipse incorporates something called the project nature mechanism which is basically a tag specifying what type of content the project has. The set of project nature tags are not set and a user can add new tags when necessary, a project containing several kinds of content types can have several tags to reflect that. The project nature mechanism provides an easy way of letting projects with different content types use tools that where never meant to be used together without the tools having to be aware of each other [6].

2.3.5 Plug-ins

Eclipse itself is just a framework and without any plug-ins it wouldn't do anything. There are different sorts of plug-ins, some add building blocks for other plug-ins such as GEF or draw2d and others add some sort of functionality such as the java programming plug-in or our Gdip.

A plug-in must be connected to one or more extension points so that eclipse knows how it is supposed to handle the plug-in. As the plug-ins methods are always invoked by the eclipse framework this is very important. Extension points are of many different types.

Important Extension point types

- Actions
- Perspectives
- Editors
- Views
- Wizards

Table 2.2

These are not all of the types but the ones most regularly used. The user interfaces for these different types are most often as follows; actions are most often shown as both a button and an item in a menu but can be either one of them without the other. Editors is generally a field where different data is shown and edited, a word processor is a good example of what can be done with a editor. A view is a field that is mostly used show data that can't be directly manipulated such as an error message list. And wizards are used to make wizards for helping new users through difficult maneuvers.

Any plug-in is also allowed to define any number of new extension points allowing it to have sub plug-ins for additional or exchangeable functionality [6].

2.4 Graphical Editing Framework

The Graphical Editing Framework or GEF is a part of the Eclipse Tool Project which is a project whose aim is to build the best possible tools for eclipse and also to minimize overlapping of functionality so that the wheel doesn't have to be reinvented over and over again. GEF is a tool that makes development of graphical editors within the eclipse environment very easy when you are familiar with its workings.

GEF's primary role is to display any model graphically [5], and then allow a user to manipulate this model by user input. GEF uses the MCV paradigm by supplying a framework for the Controller and View layers and the programmer only has to attach his own model. The hard part here is that the programmer really has to understand how GEF works otherwise he will not be able to make use of it.

After the programmer has constructed his model he has to define the view; deciding on which figures and what type of layout he will use. There are many predefined figures and layouts, or the programmer can opt to write his own from scratch. After the view is properly setup the programmer will move on to the controller layer. This layer is composed of so called EditParts which acts as connectors between the model and the view and provides means to manipulate the model and when changes occur propagate them to the view. Each viewer has a factory for creating EditParts, and when the application is started the root object from where all objects can be reached is supplied to the viewer which uses the factory to construct the EditPart for that object. When that is done each EditPart will construct its own EditPart children by calling the factory and manipulating them. If a new model object is created, the EditPart in which it will be placed calls the factory which constructs an instance of the correct EditPart class. The final structure is illustrated in Fig 2.1.

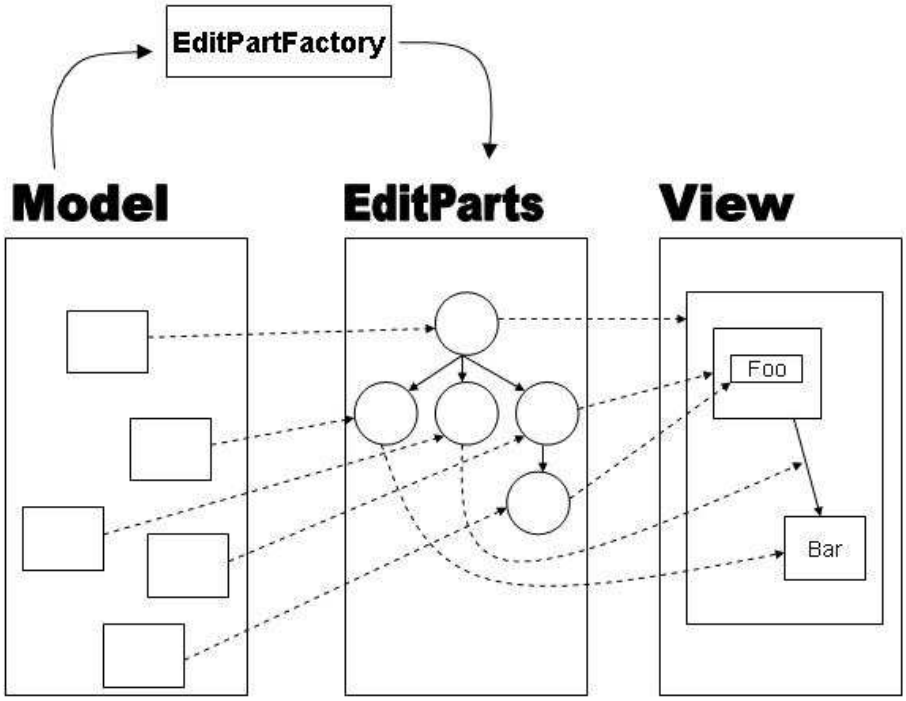

Fig 2.1

When this is done the programmer has to decide what editing functions his program will have. This is done in two steps. Firstly he has to decide which tools to use. There are predefined tools ready to use or the programmer can make custom ones from scratch. Secondly he has to decide how the EditParts should react when a tool sends a request to it. For this purpose GEF has so called EditPolicies. An EditPart without an EditPolicy cannot be edited or manipulated in any way [5]. The EditPart forwards the request to the EditPolicy that handles the appropriate type of requests. The EditPolicy replies with a command to the EditPart which sends this command back to the tool

that originally sent the request. The tool then sends the command to the command stack and where it will be executed.

2.4.1 Draw2d

Draw2d is a lightweight system that runs on SWT and it gives the programmer easy access to a rendering architecture by handling the painting and the mouse events by propagating these to the Draw2d figures [5]. There are lots of predefined figures like Polyline, Triangle and Ellipse, or the programmer can extend these figures to create his own figures. Also there exist some container figures like Panel that can be used for nested figures. The figures themselves can be opaque or transparent and ordered in layers for hiding irrelevant information.

GEF is heavily dependent on Draw2d for the graphics. Draw2d on the other hand is a standalone library that can be used without GEF or even without Eclipse although it's designed with GEF and Eclipse in mind.

2.4.2 Figures

The figure class is the central building block of Draw2d [5]. Everything that Draw2d displays is displayed through a figure. The class contains methods providing the basic functionality that is required for working with the figure. It contains methods for handling listeners so that the figure can notify the proper instances when events occur, for registering structural changes in the figures, for setting the figure's transparency.

2.4.3 EditParts

EditParts is the central building blocks of GEF. They are used as the controller layer in the MCV hierarchy which specifies how the model elements should be represented in the view. There are three types of EditParts: GraphicalEditParts, ConnectionEditParts and TreeEditParts. GraphicalEditParts role is to render a model object into graphics, ConnectionEditParts are used as connections between EditParts to represent i.e. relations and lastly TreeEditParts that are used for building trees of the model. Usually every object in the model has a corresponding EditPart, but this is not necessary.

EditParts are defined through a standard interface and when an EditPart is implemented it is recommended that it is a subclass of the AbstractEditPart which is an abstract base implementation of that interface. This class provides many methods which more or less all are used by GEF for management of the EditParts. These can be overridden for customization of how GEF handles the EditParts. Apart from these standard methods an EditPart can extend some standard interfaces for added functionality i.e. IPropertySource which allows info about the selected EditPart to be displayed in the properties view.

All EditParts are created in a factory which is implemented through the EditPartFactory interface which contains only one method where the programmer has to define which EditPart class any given model object should be coupled up with. When an EditPart is created GEF has to be informed of its presence to be activated, and if the EditPart should somehow become obsolete it will be deactivated and after a while garbage collected.

2.4.4 Requests and Commands

Within a framework there has to be a way for the different parts to communicate with each other. In GEF this is done by requests, which in turn creates commands. These contain the information that has to be supplied for executing the intended action. There are three main types of requests, CreateRequests, GroupRequests and LocationRequests. The common case when creating a request is that a tool creates a request for some specific purpose like changing the size of a visual object or selecting another object. This request is forwarded to the EditPart that is the target of the action. The EditPart itself doesn't know what the request means and forwards it to the corresponding EditPolicy which creates a command for the request and sends it back. The Command is then sent to the command stack where it will be executed and the effects of the request will be put into action [5]. This is shown in Fig 2.2 below.

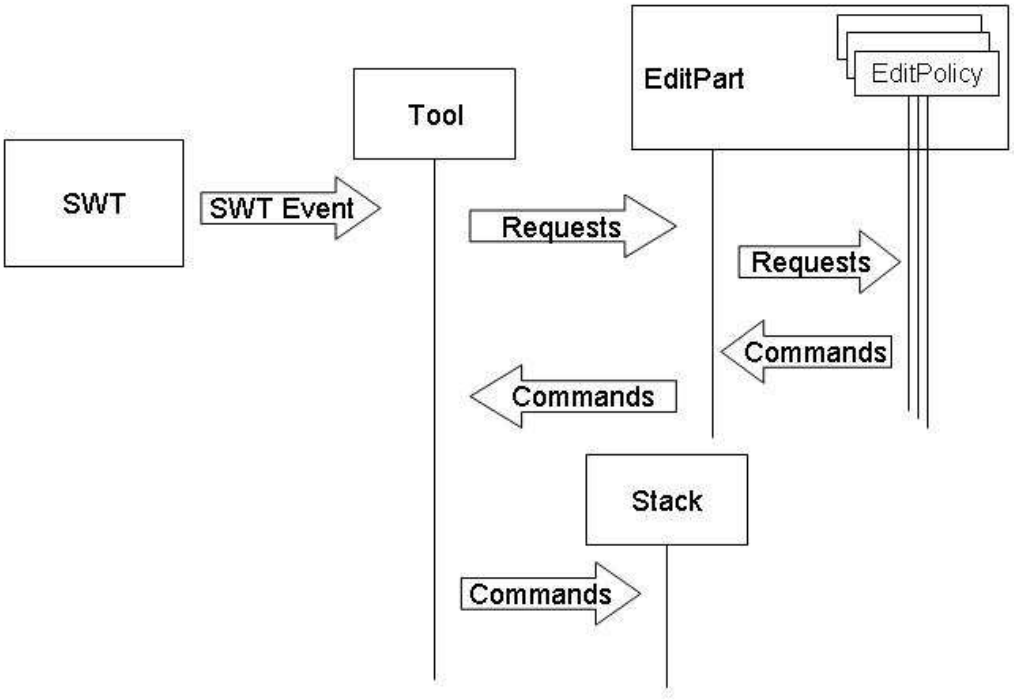

Fig 2.2 Events, Requests and Commands

2.4.5 EditPolicies

EditPolicies is essentially the component in GEF that gives editing possibilities to the EditParts. EditParts without EditPolicies are in fact static objects that users cannot interact with. An EditPolicy defines what is possible to do with an EditPart, i.e. if it is selectable, movable or resizable. It is also possible for an EditPolicy to issue requests to other EditParts, which then will forward these to their corresponding EditPolicy. Each EditPolicy has a role which defines what type of requests the EditPolicy will respond to. An EditPart can only have one EditPolicy per role.

2.4.6 GraphicalViewer

The GraphicalViewer is another component that resides within GEF to make the development of the draw2d part easier. It provides JFace-like integration of EditParts into Eclipse. For a JFace viewer to work it usually needs some content, a factory and some configuration. All standard implementations are provided like drag & drop and other tasks that are time consuming to implement from scratch each and every time [5].

There are two GraphicalViewer implementations: ScrollingGraphicalViewer and GraphicalViewer. The first supports native scrolling and the other don't, besides that they are identical. The creation of the GrapichalViewer is very easy as its constructor has no parameters and provides the method createControl which creates the SWT control of the viewer. When the creation is done the only thing left to do is to attach a RootEditPart and an EditPartFactory to the viewer and set the contents to the root model element.

2.4.7 RootEditPart

The RootEditPart is a special EditPart that has no relation to the model and its only purpose is to provide an environment for the EditParts connected to the model. In essence it is an interface between the GraphicalViewer and the model EditParts. The RootEditPart incorporates layers which are used to separate and/or group figures. Depending on what type the figure is, it is placed in the appropriate layer so that GEF will handle them properly.

3. Design choices

3.1 Choosing data representation scheme

Choosing data representation scheme is one of the most important design decisions for our tool as this is the view that an end user will actually be working with, hence if the scheme is not good enough the tool in its entirety will be essentially useless.

3.1.1 The Data

The data to be visualized is a generic relational database with no bounds on the number of elements in the database. Basically the database will be considered a tree and the focused node will be considered the root to that tree. The focused object and at least its closest children should be displayed with their most important attributes so that a user can get the most important info at a glance.

3.1.2 Hyperbolic Tree browser

The hyperbolic tree browser has been around for a while but it has not made it to any standard applications [3]. As the name suggests it is used to browse tree structures so it is possible that it could be usable in our project. What it does is display a tree in less and less detail the further from the root node a node is. This makes it possible to show trees with several thousand nodes in its entirety on a computer screen.

The earliest work on hyperbolic tree browsers appeared in the mid nineties and there have been several papers about it since then. Peter Pirolli et. al. [3] have done performance tests versus a regular file browser. Their results show that the hyperbolic tree browser was faster in a series of retrieval and comparison tasks. Earlier tests made by Lamping et. al. [1] and by Czerwinski and Larson [2] however does not show any such superiority.

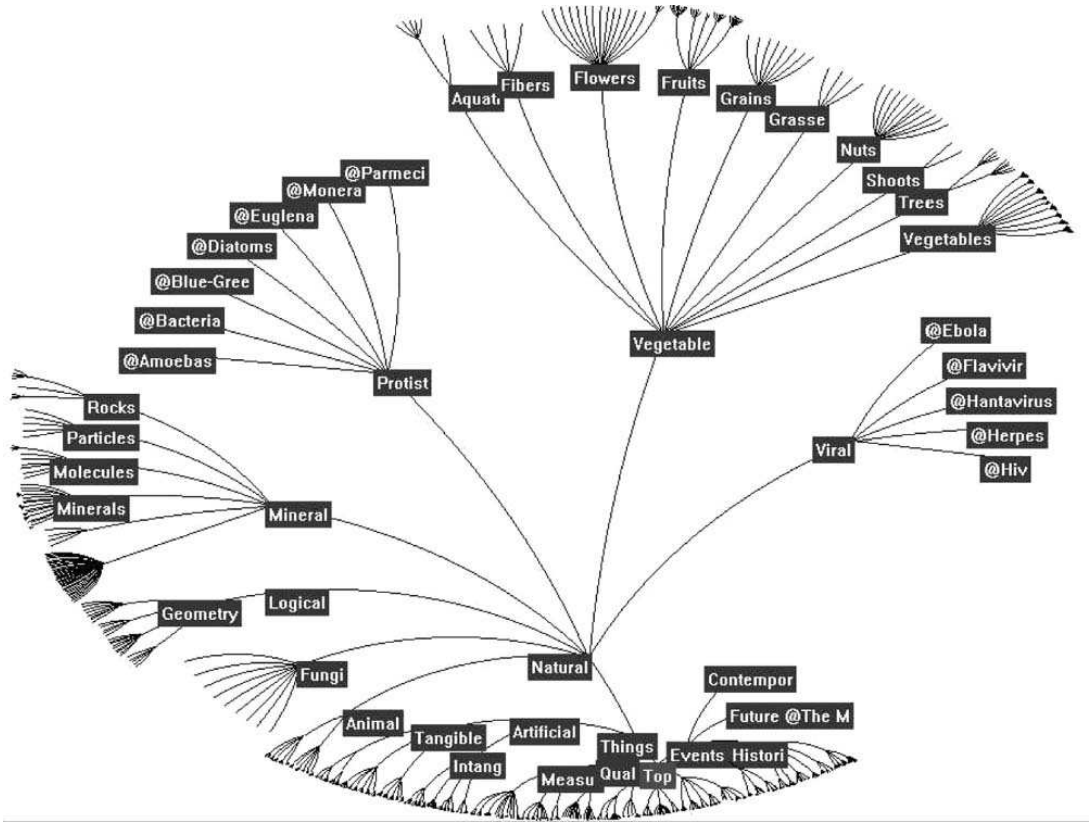

Fig. 3.1 Hyperbolic tree browser

3.1.2.1 Key points

- Combined display and overview: the hyperbolic tree browser always shows the search space in its entirety. It also reduces the amount of information of the nodes gradually the further from the center object they are [3].

- Multiple levels: the division of the search space into a focused part and a context parts is necessary to be able to show the entire search space. And it allows the user to se details about the part currently in center while all info can be browsed at all times.

- Integrated display: as the context and focus parts are displayed in the same display there is less time loss as the user does not need to visually orient him in different displays when he wants to look at different parts of the search space.

- Nondistortion: It is possible to integrate focus and context on a display without disorienting distortions to the user that degrade performance

3.1.2.2 Applicability

The hyperbolic tree browser has its strength in showing very many nodes, even if the majority is shown with very little detail. Using it for this project would be possible even if it has a few drawbacks:

Area: the area used by hyperbolic tree browsers is circular in shape while the area it should be put into is rectangular resulting in unusable space. Also if the hyperbolic tree browser would make use of scrollable panes it would further increase the area that could be used to display the visualization, but as parts of it wouldn't be shown it goes against the original idea that all info should be visible. All this results in a too small area for the focused part of the hyperbolic tree browser, especially with large databases.

Size: a corporation database can be huge, and one of the key points with hyperbolic tree browser is that all nodes are visible. It is simply not possible to show most databases fully on a screen, also there would be to long a delay if the entire database had to be loaded to the local computer at startup. The hyperbolic tree could be used if the set of shown nodes was somehow delimited but that would remove many of the key advantages to it and only using the look of the hyperbolic tree.

3.1.3 Pie Slice

We developed the pie slice scheme as we did not find any applicable existing scheme. The main idea is to have the focused object in the middle and the rest of the objects represented by nodes displayed on circular levels around the focused object as shown in fig. 3.2 below. Each relation type would get a pie slice of the circle to place its objects in. In fig 3.2 there are 3 different relation types, each with its own pie slice. In the figure slices are delimited by radial lines.

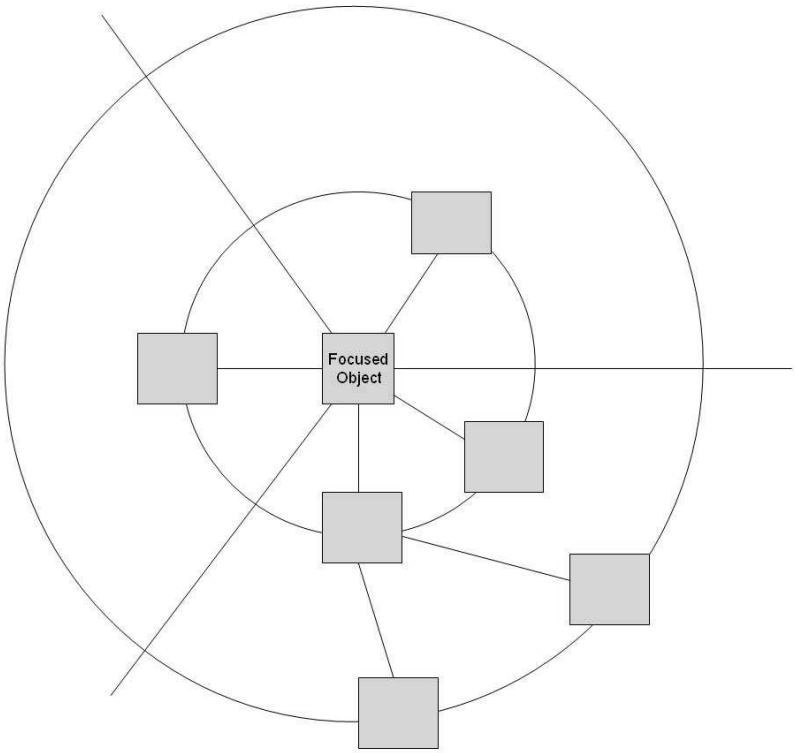

Fig 3.2 Pie slice scheme sample picture, circular and radial support lines are there to help understanding the scheme and would not be part of an implementation.

3.1.3.1 Key points

- Extendable: the pie slice scheme doesn't have any bound on how many nodes that can be shown depth wise but there is only limited space on each level. So when handling objects with many relations there would be need to incorporate a scheme to show only the user selected nodes. It can also make use of scrollable panes to extend the workspace area making it possible to have nodes display some info about the object it represents.

- Flexibility: the pie slice would create a slice for every type of relation to or from the focused object thus always spreading the objects out as evenly as possible.

3.1.3.2 Applicability

As this scheme was thought out by us specifically for this project it has the necessary properties but there are a few drawbacks that made us consider other options.

Why was it good to build the interface in a circular fashion, we gained some flexibility allowing the use of the full 360º around the focused object, but what where the costs? We found that when arranging rectangular objects on a circle it was less area efficient than we had first thought leaving too little room for nodes at least on the inner circles.

Flexibility: could actually be disadvantageous as a user would have no way of knowing in what direction a relation type would be directed thus she would have to orient herself every time she refocused.

The algorithm: granted that the pie slice scheme have a few flaws and that the layout algorithm would be somewhat complicated by all the circles we choose to rework this scheme to lose the most apparent flaws.

3.1.4 Quad tree scheme

This is more or less a reworking and simplification of the pie slice scheme, first the circular theme where dropped and the number of different ways in which a relation type can build its tree where dropped to four. The direction set for every relation type is defined beforehand but can be altered to fit new needs. This allows users to know in what tree direction she is interested in directly rater than having to find out at every refocus. This also means that some relations will have to share directions; this shouldn't be a problem as long as different relations are shown with different kinds of arrows or are visually distinguishable in some other way.

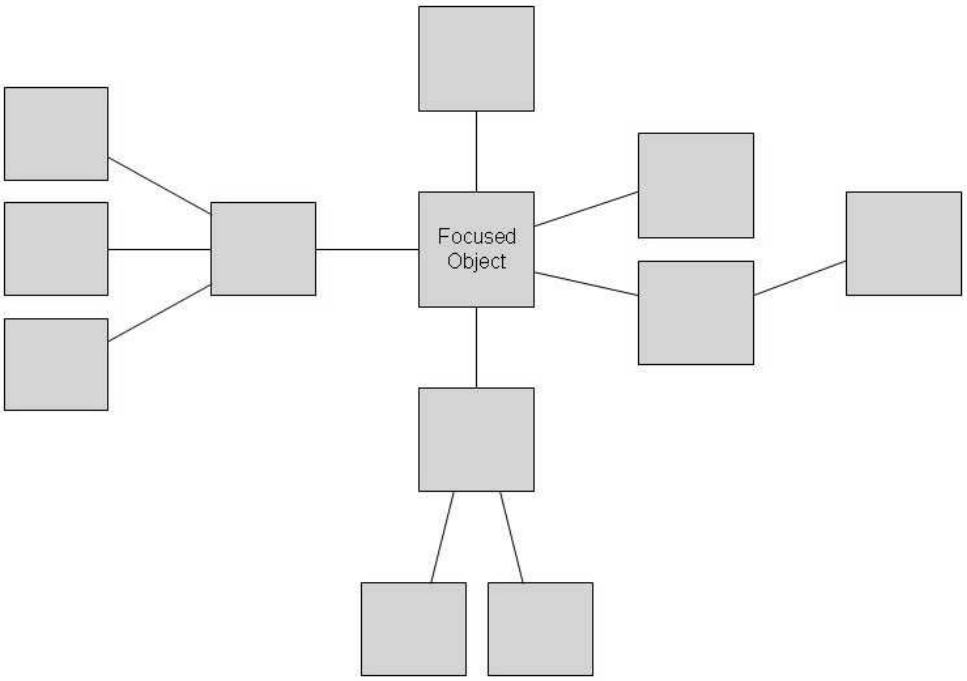

Fig. 3.3 Quad tree scheme. Note that in this picture the relations to the two nodes directly right of the focused object may be of different types. With a more advanced implementation of it one of the lines might be dotted to signal a different relation type.

3.1.4.1 Key points

- Extendable: Quad tree shares the extendable property with the pie slice scheme. See 3.1.3.1
- Predictability: as all relation types always will have their tree built in the same direction a user can get an idea of what is what directly as the picture is drawn.

3.1.4.2 Applicability

The point of this project is to make it easier to find data and understanding the structure of data in databases, with its more rigid direction allocation it is easier to find the structure of the data shown.

There is a risk that with this scheme some side may be under populated while others will be to full to fit all nodes. This mapping problem will have to be dealt with for each individual database that it is connected to as that depends on the allocation of directions, an allocation that is very easy to change.

3.2 Algorithms

Both of the two major algorithms used in this project are based on depth first graph exploration and in the first envision of the solution they where one and the same. Due to the MCV structure of the eclipse/GEF programming paradigm we had to divide the initial algorithm into two, doing the same traversing of nodes but in different MCV layers. The algorithm for choosing nodes works on the model layer and its sole purpose is to select the nodes to be present in the control layer. The second algorithms task is to manage the layout so that the objects are displayed according to the quad tree scheme. As the application presented in this thesis is only a demo it isn't yet capable of selecting info from a database and including it into the model when necessary. To fill this gap there would be need for a third algorithm, this algorithm would be very similar to the node choosing algorithm. The differences are that it wouldn't create the controller objects for the chosen model objects and that it would check for external links for the nodes and fetch any wanted object not already in the model layer.

3.2.1 Choosing nodes

As stated this algorithm only works with the data already in the model layer and uses the data in the DParam component of the model to decide what model objects to create control layer objects for. In the diagram of the algorithm we have chosen not to represent the condition statements instead we have the question they represent. This choice was made to improve the readability of the diagram and a short description of the conditions will be given later.

First we need to know what this algorithm is supposed to do, which nodes are the wanted ones. We firstly consider every relation type two separate types, one type directed away from the focused object and one type directed towards the focused object. This is mainly done to make the quad tree readable at all. With this scheme if a relation between the focused object and a first level object indicated that the first level object is a subpart then all of its children in the quad tree will be subparts of its parents. Had we not done this there would not be any way of knowing what kind of relations its sub tree would contain. Each relation type has a relation constrictor object that specifies how many levels of it will be shown. This constrictor does not know of the direction division it only works on relation type the direction division is achieved by the algorithm. There is also a list of objects considered unimportant and none of these objects children will be fetched but the objects themselves will be visible.

Firstly it should be said that this algorithm consists of to very similar parts. One starts at node 2 and the other at 20, the first of these parts works only on relations directed away from the focused object while the last only works on relations directed towards it. Any list operations are done on the most recently mentioned list.

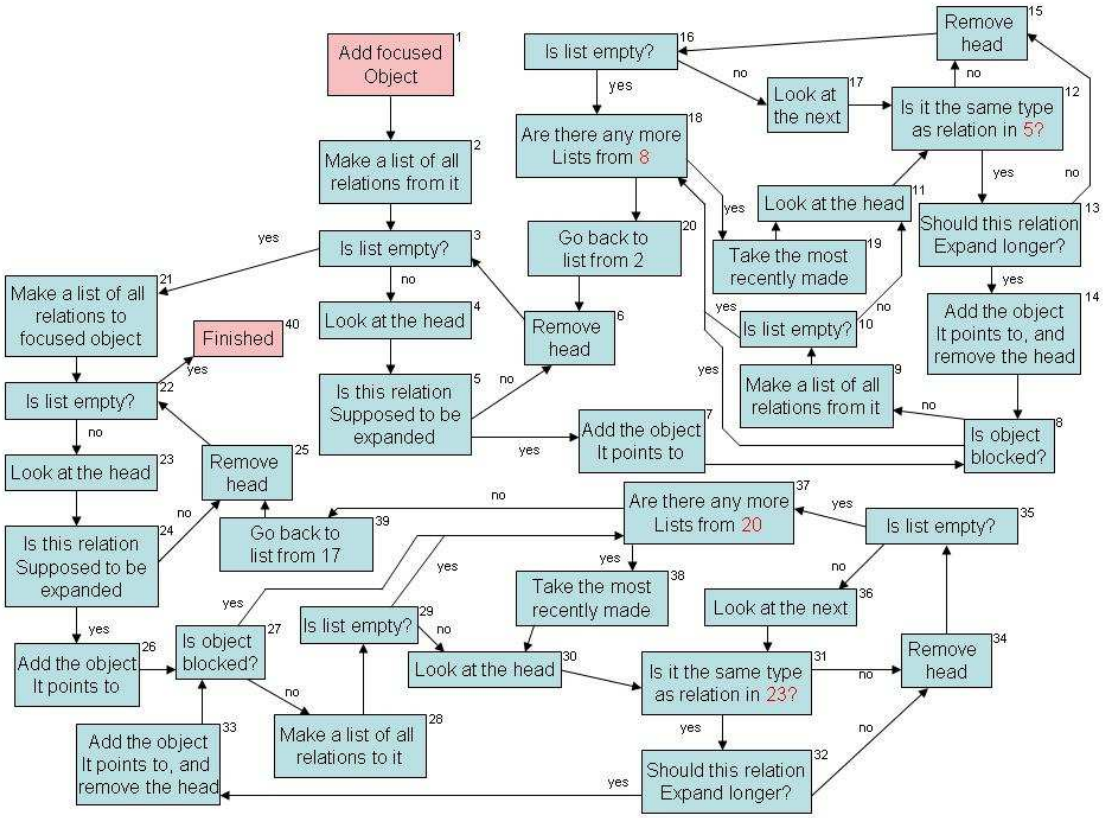

Fig 3.4 Algorithm for finding the wanted nodes.

Conditionals

5, 13, 24 and 32: this checks the constrictor for the relation type to see if there should be another layer.
8 and 27: A check to se if the object is considered unimportant if it is the children is not shown.

3.2.1.1 Realization

The base of this algorithm is recursion as it is a good way of doing depth first. If we look to the diagram nodes 7 through 19 and 26 through 38, they are two different but similar recursive functions.

3.2.2 Placing into zones

When the wanted nodes are found next is deciding how to place them and we have the quad tree scheme to adhere to and to do that the easiest way we have decided on dividing the on screen layout into smaller parts. First we place each node into one zone which is more or less just an area to be placed in. The zones correspond to the layers in the quad tree, each zone is just a list containing zero or more figures but they convey information on where to draw the figures they contain. Each of the four directions has its own zone structure if they all where laid out as their figures should be drawn we would get fig 3.5.

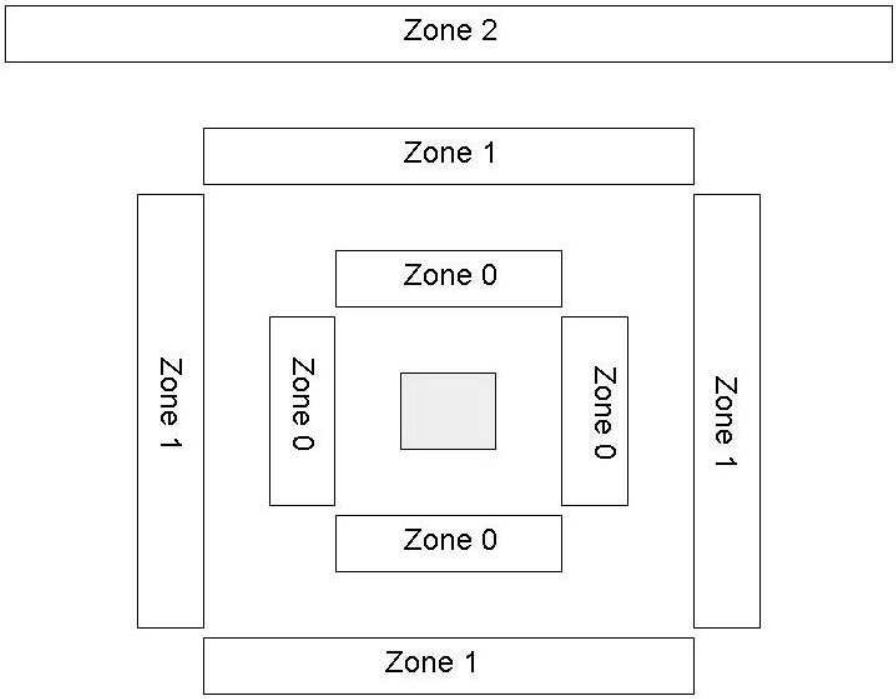

Fig. 3.5 How figures in the zone lists are placed.

Just as with the finding nodes algorithm the zone placing algorithm divided into two parts each doing nearly the same things however there are more differences between the two parts here than in the finding nodes algorithm. Here if a relation type is supposed to point up it only will do so if the relation is directed away from the focused object. If the relation is directed towards the focused object then the direction is reversed and will point down instead. This is done to make the view more easily read. In the figure 3.6 object B is a subpart of A and has the subpart C. If subparts are supposed to point down, we will get them in the same order as we would if A was the focused object making it easier to interpret the data as the order of the figures are the same no matter where along that relation chain the focus is.

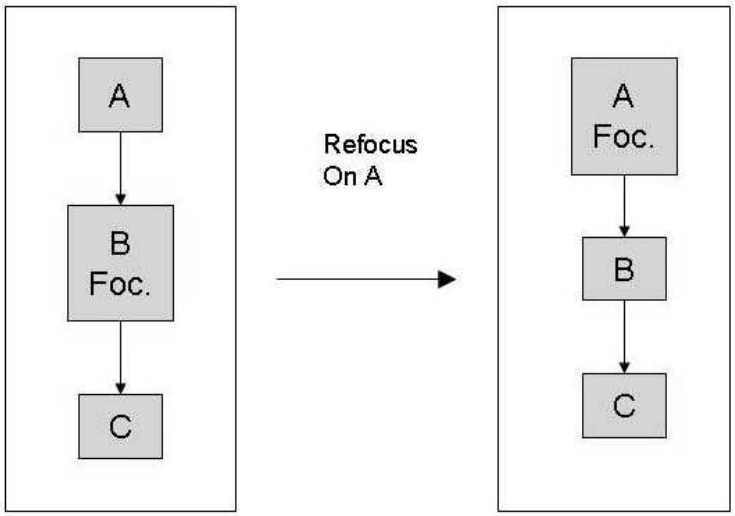

Fig 3.6. Relation direction before and after refocus

As stated each direction has its own zone structure but this is not mentioned much in the diagram mainly due to space issues. However this is how it works: the direction is decided in node 5 and the direction decided there is the one used until node 5 or 22 is reached. Depth is the variable that keeps track of in which level of the zone structure (see fig. 3.5) nodes should be added to.

Here the advantage of the depth first processing of the nodes and figures becomes apparent. Because of the depth first processing the order of the figures in the zones structure have the property that if all of them are drawn from left to right in their respective zones no of the relations will cross. So this is the scheme we use for making sure relations will not cross.

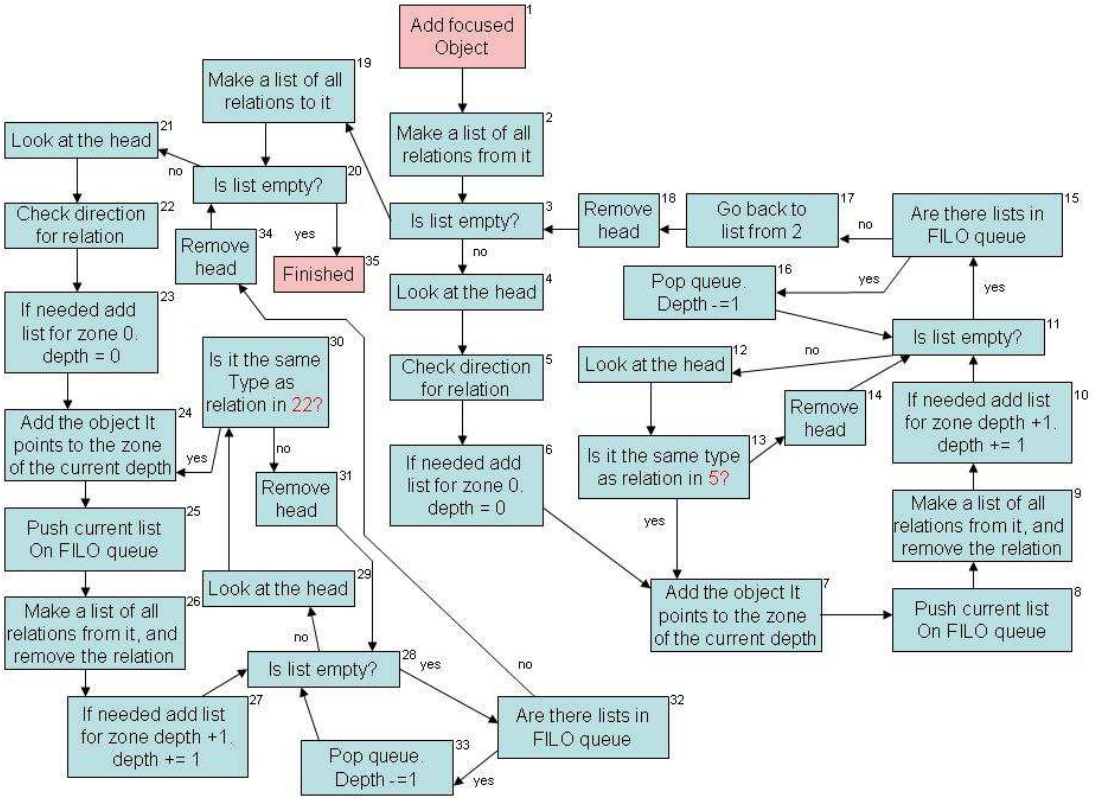

Fig. 3.7 Algorithm for placing figures in zones.

Conditionals

Any list operations are done on the most recently handled list.
5: this checks the constrictor for the relation type to see in what direction it should be shown.
8, 15, 25 and 32: as this was done with recursion these nodes are simply stack manipulations as the recursive function are called or returns.
22 this checks the constrictor for the relation type to see in what direction it should be shown and the reverses the direction.

3.1.2.2 Realization

This was realized in much the same way as the node finding algorithm with the recursion and for-loops. The only real difference is that they work on different levels in the MCV hierarchy.

4. Implementation

4.1 Program structure

4.1.1 The Model

As the GEF/Eclipse approach is highly MCV oriented we needed to build a model capable of containing data objects from any database. So we had to use very general data storage. All data is stored as ArrayLists and Strings so that any amount of data of any kind can be stored.

4.1.1.1 Database emulation

4.1.1.1.1 Database objects

In our modeling of a database we represent any database object as a structure containing a unique identifier string and an ArrayList with an arbitrary amount of attributes. Each of the attributes being represented with a structure containing two strings, one for the attributes name and one for its value. We choose this representation because it is very simple but still very adaptable to different types of data.

4.1.1.1.2 Database Relations

A relation is represented as a class containing two of our database objects and a type string and would be interpreted like this: the relation originates from one of the database objects and points at the other. The relation then has the type specified by the type string. This structure was chosen because GEF relations are built in this way and it is easier to have them work together if they are similar in structure.

4.1.1.1.3 The full database model

The class that constitutes the full database is very simple in structure but a little more complex in its methods. The data it contains are two ArrayLists, one of database objects and one of database relations, there are also a different structure for storing parameters for the drawing of the focus image. The methods are mostly for different sorts of accessing and altering the data in the model.

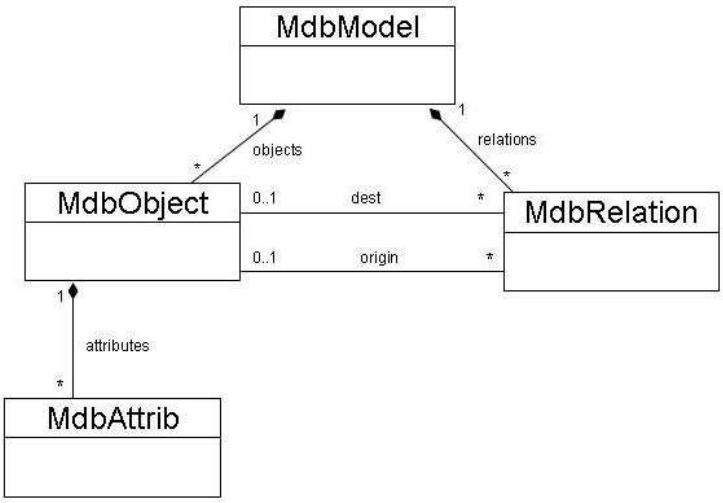

Fig. 4.1 UML diagram showing the model.

4.1.1.2 Parameters for generating quad tree

To solve the problem of knowing which objects to display, we appended a structure that contains all the necessary parameters. The most important parameter when creating a quad tree is the focused object, around which the whole view is centered. There is also one structure for each relation type in the depicted view containing information on how to handle that relation type. It has data on how many levels of objects related to the focused object with this relation type that should be shown or if it should be left out. This structure includes a specification on what direction the relation should be expanded in.

There is also the choice to hide all the children of an object so there is a list of objects that are supposed to be handled in this way.

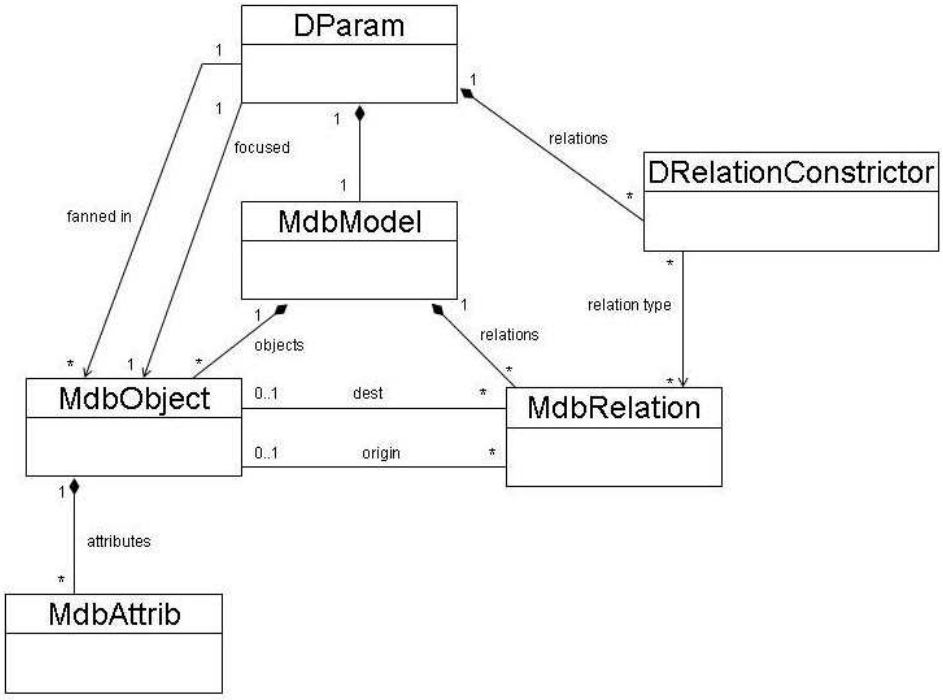

Fig. 4.2. UML diagram showing model and parameter structure.

4.1.2 Object package

The object package contains all EditParts, EditPolicies and Figures that is needed for displaying each MdbObject. The package has six classes: ObjectEditPart, ObjectEditPolicy, ObjectSelectionEditPolicy, ObjectNodeEditPolicy, ObjectFigure and ObjectFigureCompartment. The structure of the classes are described in Fig 4.3

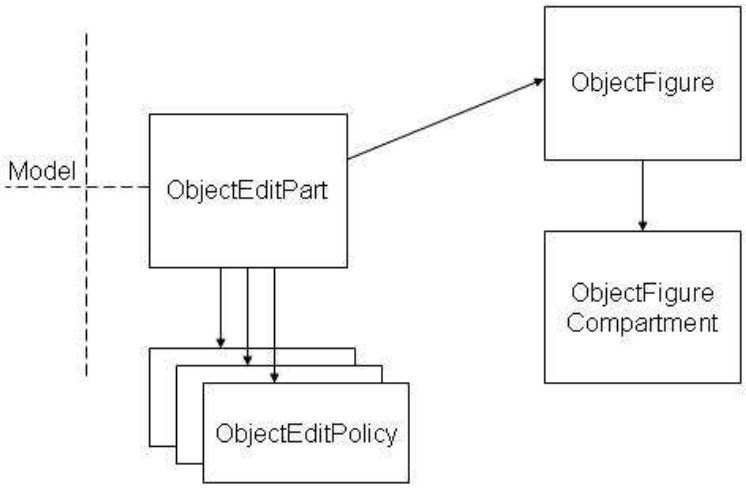

Fig 4.3 The Object package structure.

4.1.2.1 ObjectEditPart

This is the EditPart responsible for displaying an MdbObject in the diagram and is the most central part of the object package. The class extends AbstractGdipEditPart and implements NodeEditPart and IPropertySource [5]. AbstractGdipEditPart is further described in the diagram package. NodeEditPart is a specialized type of GraphicalEditPart that is specially made for handling connections, since our goal is to create a tree-like structure it was the natural choice. IPropertySource is implemented since it provides easy access to the properties view where we want the info on a specific object to be shown. All functions ObjectEditpart contains are needed for these implementations to function correctly. It initializes the EditPolicies and creates the ObjectFigure.

4.1.2.2 EditPolicies

These classes take care of all the requests that are associated with the EditPart. The only requests that are handled are selectionRequests and deleteRequests as the program doesn't support any editing by the user.

4.1.2.3 ObjectFigure

This is the actual figure that the EditPart will draw. It is a very simple figure that in essence is only a label and an ObjectFigureCompartment in a toolbar layout. It is described in Figure 4.4

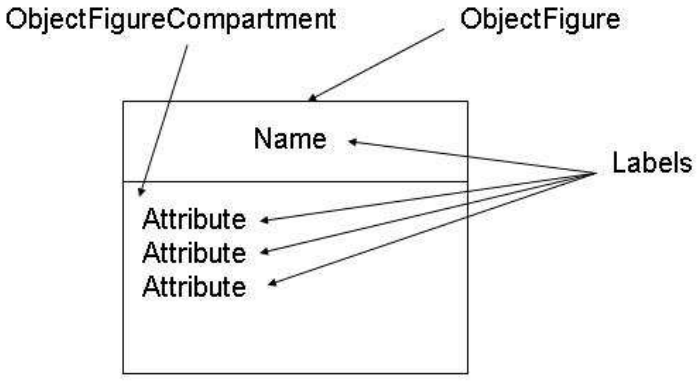

Fig 4.4 Visual structure of ObjectFigure

4.1.2.4 ObjectFigureCompartment

This class is part of the ObjectFigure and it is a figure containing all the attributes of an object. These are in label format and are arranged in a toolbar layout.

4.1.3 Relation package

4.1.3.1 RelationEditPart

It extends AbstractConnectionEditPart which has almost all functionality needed; only the initiations of the EditPolicies are added. We don't need to create our own figure for the relation as the PolylineConnection used by AbstractConnectionEditPart fills all our current needs. It also implements IPropertySource granting easy access to the properties view where we want the info on a specific relation to be shown.

4.1.3.2 HideRelationAction

This is the action for hiding one relation type, this is achieved by setting a Boolean in a DRelationConstrictor which resides in DParam. The focus view would be affected in the same way if the renderdepth for that relation where set to zero. But if it was done that way the previous renderdepth value would be lost.

4.1.3.3 HideRelationDialog

This is the dialog for HideRelationAction. It lists the available relation types in a list and when a type is chosen its current visibility mode is displayed and can be altered. Screenshots of this dialog can be seen in chapter 5.1.5.

4.1.4 Diagram package

The diagram package contains all classes used to represent the full diagram in the different layers, and also a few actions that work on the diagram. The classes are AbstractGdipEditPart, ConstrictorAccessor, DiagramEditPart, DiagramLayout+Manager, Focuser, GdipEditPartFactory, MyXYLayoutEditPolicy, RelatedObject, RelationAccessor, RenderDepthChanger and RenderDepthDialog.

4.1.4.1 DiagramEditPart

This is the EditPart that controls the entire shown diagram, it is here the node selection algorithm (see 3.2.1) is implemented and the rest of the EditParts are created, but this is basically the implementation of that algorithm and a few additional small methods. The figure it uses for graphical representation is a FreeFormLayer which is basically an empty area where other figures can be placed. This area can be extended in any direction if need be so there is always more space to draw figures in no matter in what direction we want to expand the view.

4.1.4.2 DiagramLayoutManager

This class is responsible for the layout of the figures into the FreeFormLayer. Its main algorithm for placing the nodes is more thoroughly explained in 3.2.2. There is more to the layout than just finding the right zone for each figure. Placing the figures in a zone is pretty simple, first a center point for the zone must be found (see fig 3.5). To make sure there is always enough room for the figures, all figures in the zone immediately inside the current one are checked to find out how far the largest figure extends, then the same is done for the current zone. When both these measurements are found a margin is put in between the zones and the safe distance is found. The gradual shading of figures the further from the focused object we get are also a part of this class.

4.1.4.3 MyXYLayoutEditPolicy

The DiagramEditPart has to have a layout policy but we don't need more functionality then is already available in XYLayoutEditPolicy. As XYLayoutEditPolicy is abstract MyXYLayoutEditPolicy extends it and implements "return null" methods for all abstract methods.

4.1.4.4 GdipEditPartFactory

Creates all editparts for the project. The functionality of EditPartFactories are further described in chapter 2.4.

4.1.4.5 RenderDepthChanger

Implements the IEditorActionDelegate interface meaning that it is an action for an editor activated menu or toolbar. This is an action for altering how many levels a relation should be drawn in the focus view. The user only sees the dialog and the action merely starts the dialog and passes the information to the place it should be.

4.1.4.6 RenderDepthDialog

This is the dialog for the RenderDepthChanger. It lists the available relation types in a list and when a type is chosen its current render depth is displayed and can be altered.

4.1.4.7 ConstrictorAccessor, RelationAccessor and RelatedObject

The accessors are used to access info in the DParam and RelatedObject is a temporary storage class.

4.1.5 Editor

This package only contains the class FocusEditor. When all parts of the program are created they have to be displayed. The easiest way to do this is to display the model by extending the GraphicalEditor class. This class creates a good environment for displaying models, but it is not suitable for using directly and should be tailored for specific usage. So therefore the FocusEditor extends GraphicalEditor.

4.1.6 Plugin

This package also only has one class namely the GdipPlugin class. The purpose of this class is to provide a structure for managing UI resources. Therefore GdipPlugin extends AbstractUIPlugin which does exactly that. It uses the generic startup and shutdown methods to manage images, dialog settings, and a preference store during the lifetime of the plug-in.

4.1.7 Data

The purpose of the Data package is to provide an interface for writing and reading files that resides on disk so the main program doesn't have to worry how it is done. It just supplies the necessary information and the Data package will take care of the practical issues of writing and reading from disk. The package consists of three classes: DataIo, DataLoad and DataSave. DataIo is the class that does the real reading and writing the other two classes are just implemented to provide the buttons for loading and saving.

4.1.8 DataIo

This is the central class of the data package the other classes are just for making this class available to the program by creating buttons.
The class contains four functions: convertModel() , convertXML, writeFile() and readFile(). Initially there should only have been two functions but since the XML-Decoder/Encoder didn't accept our classes we had to build a converter that converted our objects into something that it could accept.
The convertModel() takes a MdbModel object and a string. The string is the identifier for the file that is to be written to. The function converts the MdbModel object into a structure consisting of ArrayLists, Strings and Integers. When this is done it calls the writeFile() function with the converted structure and the string as arguments.
The writeFile() function takes an ArrayList and a string as arguments. The string is the identifier for the file. The function then loops through the ArrayList and writes each object to file using the java.beans.XMLEncoder.
The convertXML() takes a String as an argument. This string is used to identify the file that is to be read. The function then calls the readFile() function with this string. ReadFile() then uses java.beans.XMLDecoder to extract an ArrayList from the file. This list is returned to the convertXML() function which converts this structure into a MdbModel object which then is returned.

4.2 Putting it all together

The program is based on the model-controller-view paradigm and with that as a basis we will see how the program works and make use of its parts to produce the final view. It all starts with the model (see chapter 4.1.1 for further details). The model has to be created before the rest of the program starts to execute, and this model is unchanged throughout the programs execution with the exception of the parameters. When the program gets the command to render the view it will first create a controller level this is done by applying the node choosing algorithm (see chapter 3.2.1 for further details) and the nodes that where chosen is passed to the EditPartFactory (4.1.4.3) which creates the ObjectEditParts (4.1.2.1). Each of the ObjectEditParts contain a figure (4.1.2.3), the remaining work to be done is the layout performed by the layout manager (4.1.4.1). It uses the placing into zones algorithm from (3.1.2) and also handles the dynamic placing of the zones so that no two figures may overlap. The gradual shading is also done during this layout.

Now that we have seen what is done when rendering the view we can take a look at the prerequisites for it. As already mentioned there has to be a model with all the objects and relations but it also has to contain the DParam (4.1.1.2) in order to properly render the view this class is very important as it is here that any alterations of how to render the view is saved. Before any rendering can be started there is also the need to create the DiagramEditPart (4.1.4.1) and most of the other classes in the diagram package (4.1.4) as they are mostly used in different ways to create the controller and view layers.

To manipulate the view there are a few tools which in different ways alter the info in DParam so it will change the choice criteria's for the algorithms and a different view will be rendered.

4.3 Working with Eclipse/GEF

Starting this project neither of us had worked with eclipse before, so we thought we should have a chapter regarding our experience working with eclipse for the first time.

4.3.1 The Eclipse paradigm

Although both of us are somewhat experienced Java programmers and have been using Java as our main programming language for the past five years the transition to program for eclipse was a big one. Programming for eclipse has been describes with the phrase "don't call us we'll call you". And though meant humorously it captures the essence of programming for eclipse. This was at first the greatest problem as there is no main class, nowhere where the execution begins. For someone used to write Java programs from scratch this is very perplexing. Through the course of this project there has been many times when we knew how to do some task but not where to put it so that it would be called at the right times. Also one problem that sometimes arose where that since we where programming in a predefined MCV structure there where some abstraction barriers that took some time to get used to, often it was the case that some function or information that we intuitively would place in one class would mean that we would break the abstraction barrier and therefore could not be done that way. It was also the property of this MCV structure that forced our layout algorithm (3.2.1) to be divided into two parts

There are good API:s for booth eclipse and the plug-ins we have used. There are also several tutorials and in depth "how to" guides available on the eclipse homepage [7]. These are of high quality and very helpful.

5 Results

5.1 Screenshots

As the program in it self is the result we have chosen to have an extensive walkthrough of the functionality accompanied by screenshots to show what actually happens and to have a more easily understood explanation of the functionality we have implemented.
First we start with an “untouched” picture of the Focus Editor. The model displayed is generated and has a center object which is surrounded by four binary trees, each with its own relation type and display direction. To give the different trees individual structure we have then fanned in or blocked some nodes from showing their children.

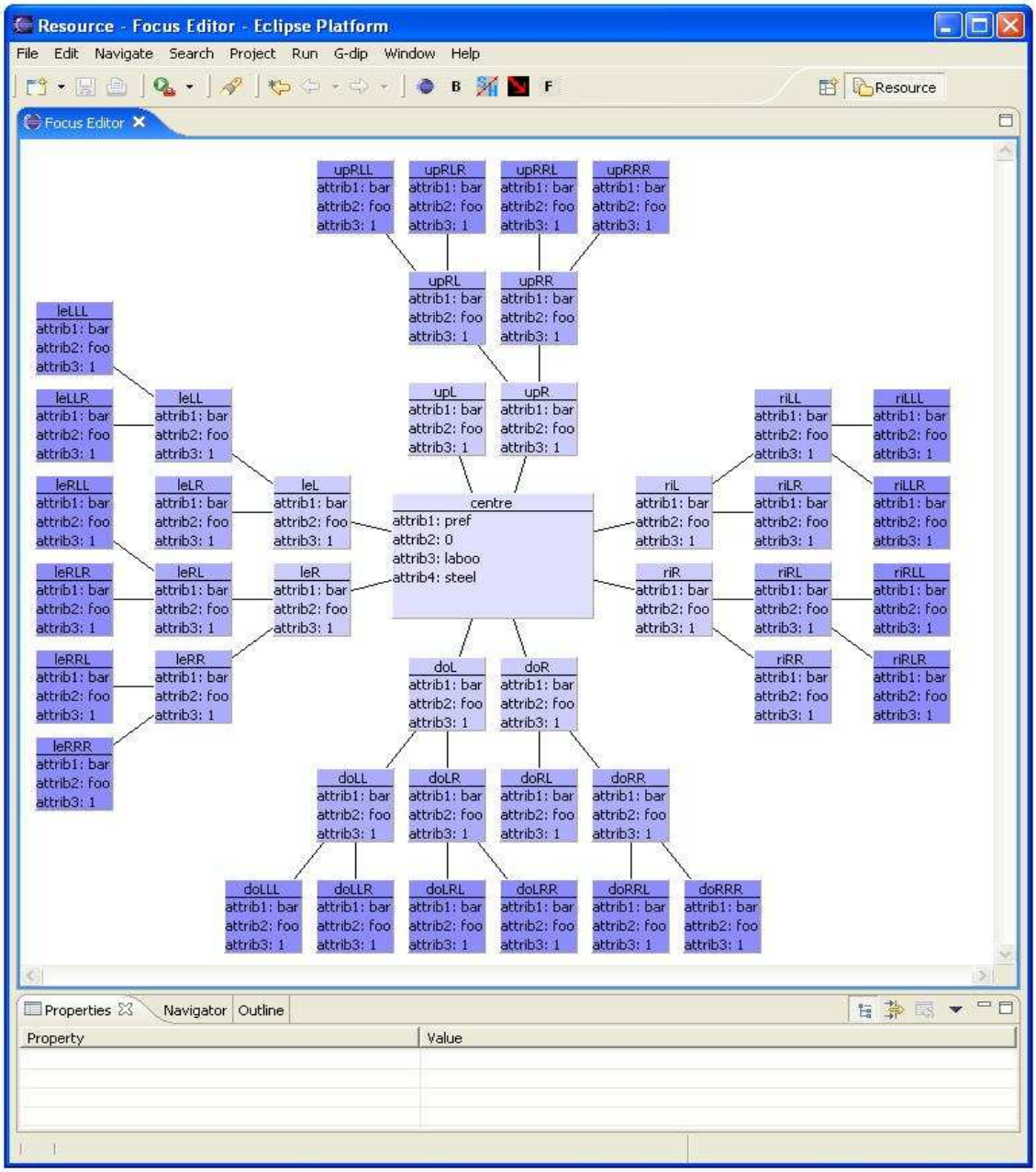

Fig 5.1

5.1.1 Selection

First we show how it a selected object looks, just the standard eclipse selection frame. Down in the properties view the properties of the selected object is shown, in this instance it isn't very helpful as all properties is visible directly in the figure but that won't always be the case.

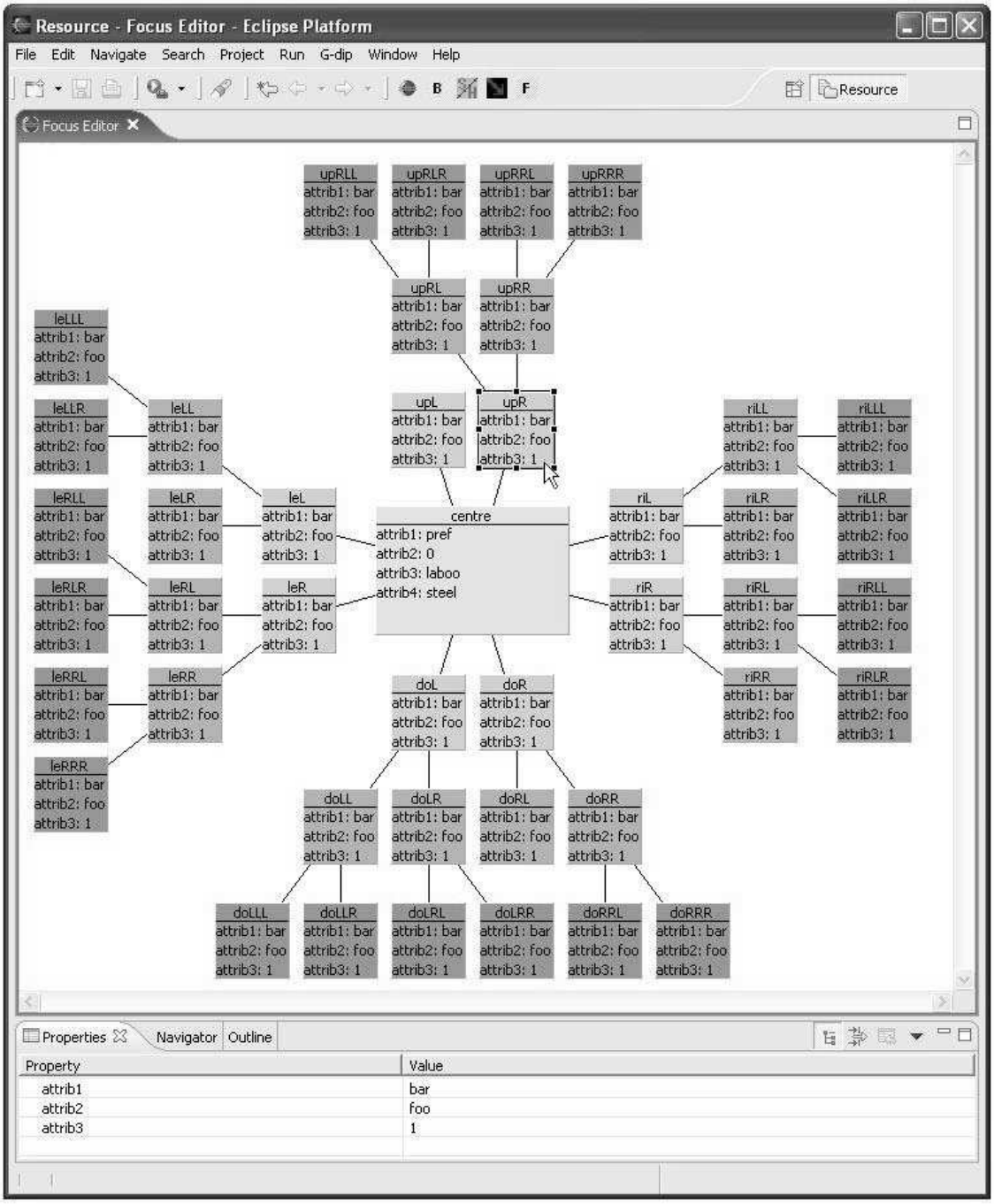

Fig 5.2 Object Selected

Below we have a picture with a selected relation again it uses the standard eclipse selection look, and just as before the properties is shown in the properties view.

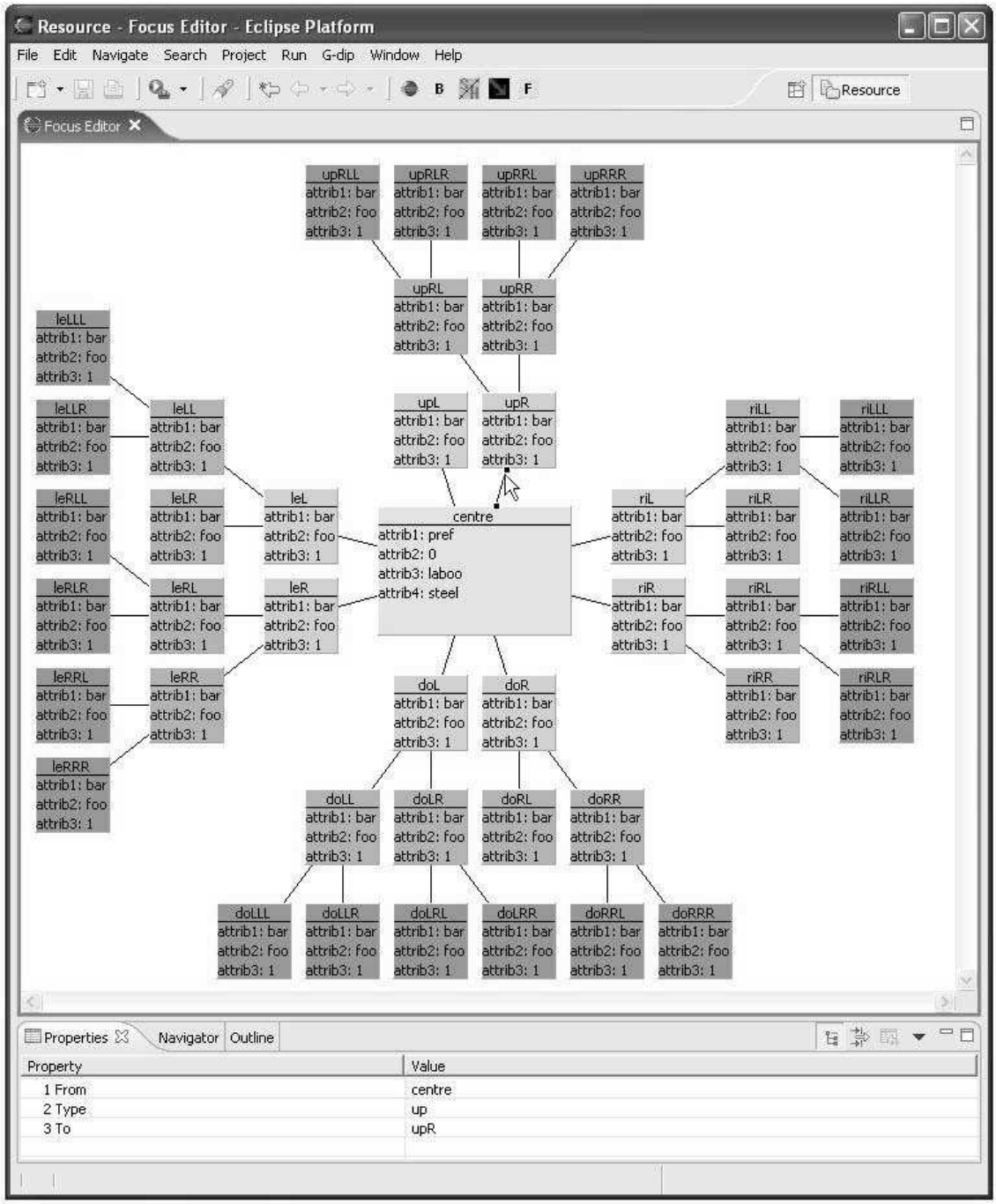

Fig 5.3 Relation Selected

5.1.2 Refocusing

Refocusing is the most central of the functionality we have implemented and we have chosen to show a refocus to object leR the selected one in fig. 5.4.

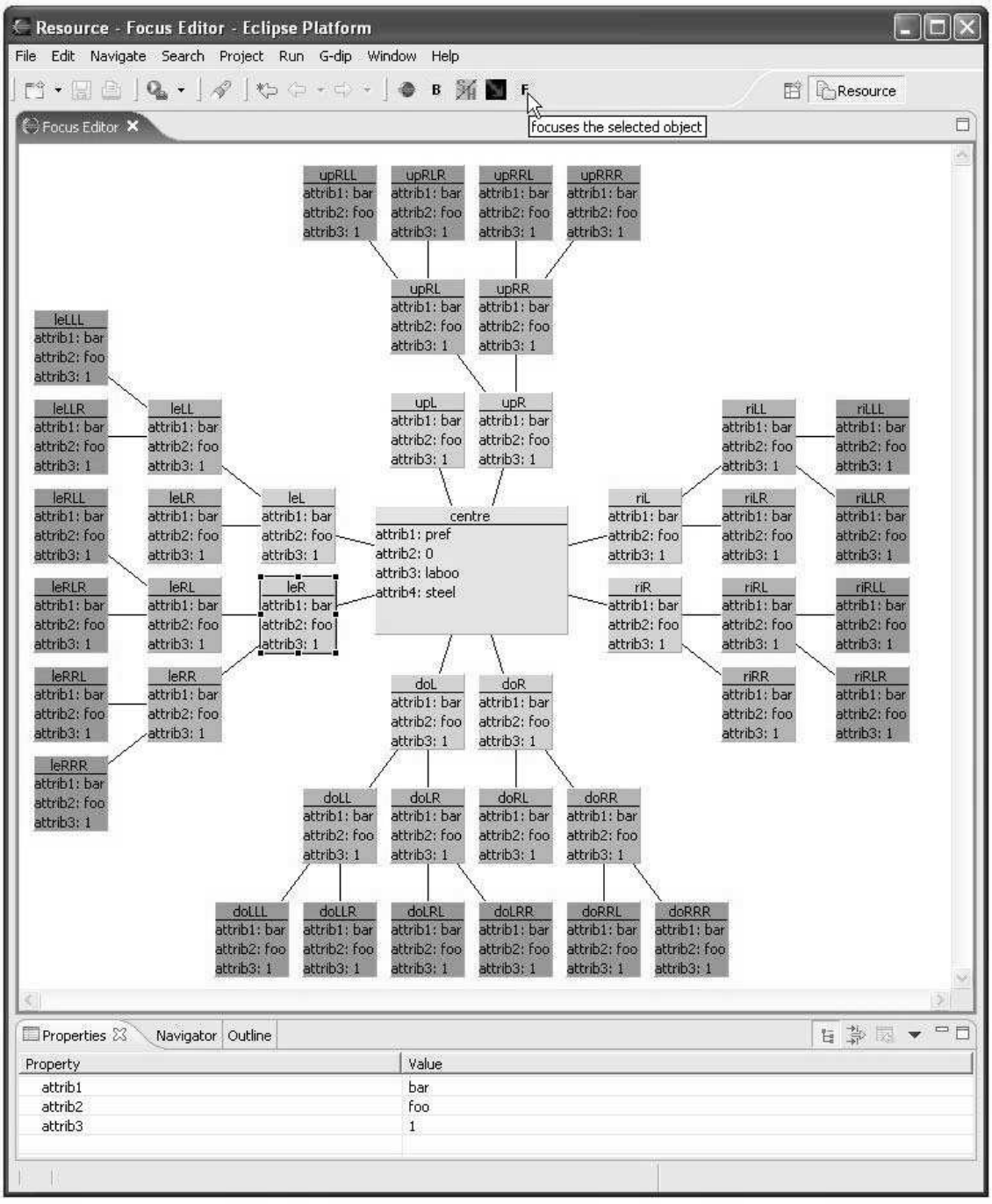

Fig. 5.4 before refocus object selected and mouse pointer posed.

As can be seen in fig. 5.5 leR is now the focused object. Centre is placed to the right even though it has the same relation type as all the other objects in the fig5.5. It is done this way as the relation to centre has another direction then the other objects relations. If we compare fig 5.4 to 5.5 we can se that the objects that are in 5.5 keeps the same structure after the refocus as before, making it easier to recognize objects that appears in a refocus.

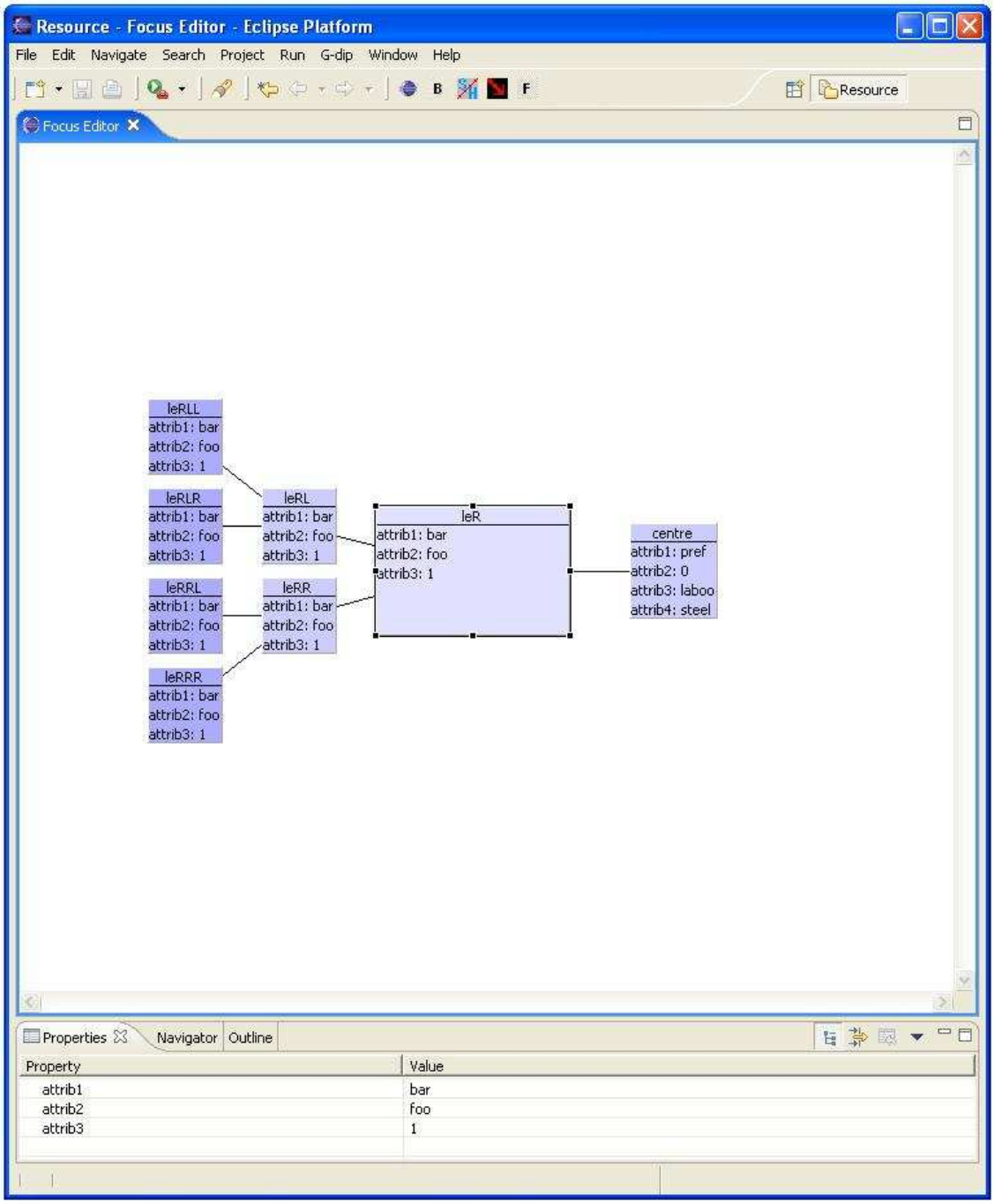

Fig. 5.5 after refocus

5.1.3 Fanning

Fanning in or out sets a state in the DParam to make the node choice algorithm ignore all of one nodes children when selecting nodes for the focus view. In fig. 5.6 we can see the program ready for a fan in with an object selected and the mouse pointer hovering over the fan/block button.

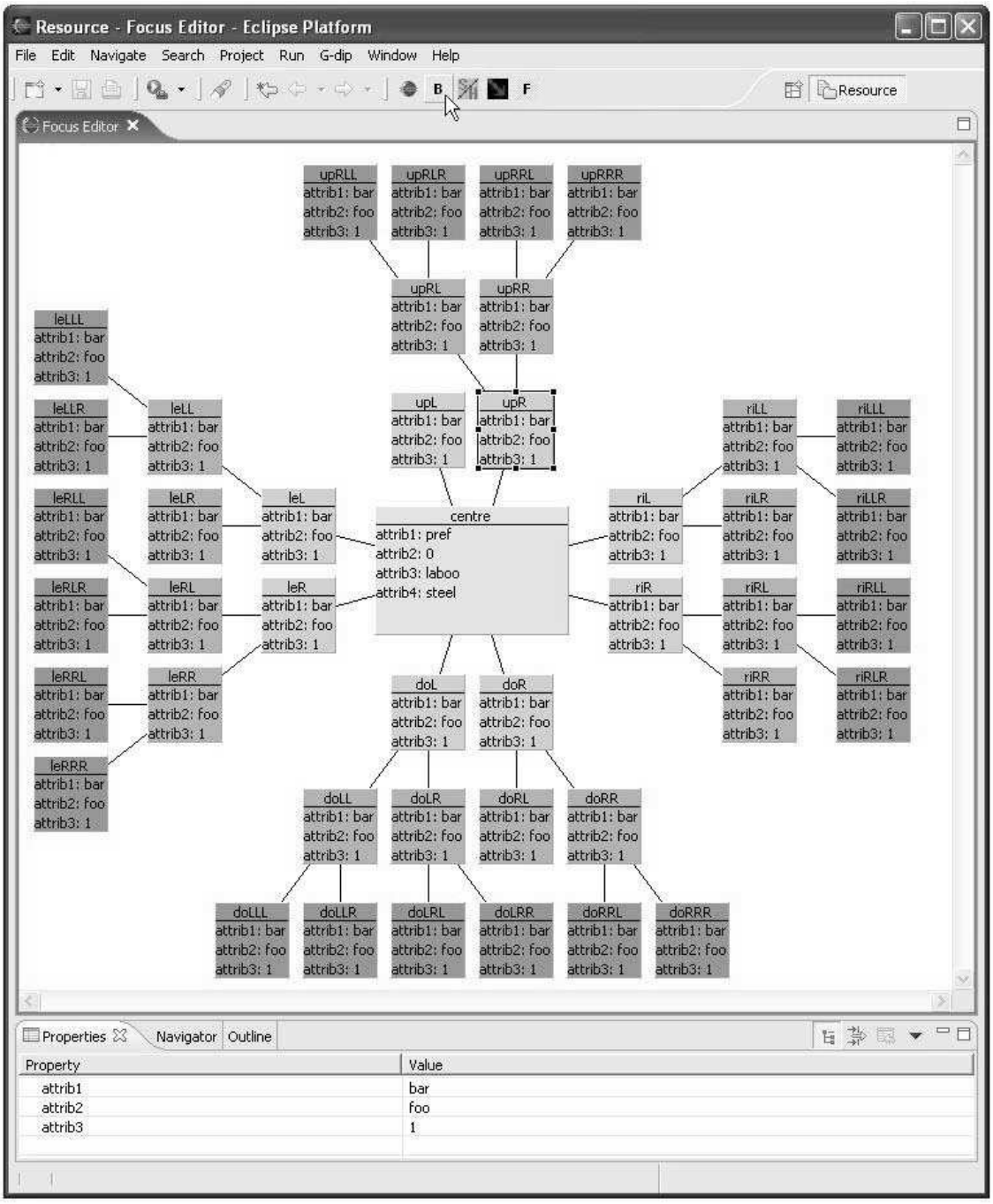

Fig 5.6 before fan in/block, object upR selected and mouse pointer posed for click

After the fan in the view displayed is exactly the same except that the children of object upR is no longer displayed. The program is set up to do a fan out as upR still is selected and the mouse pointer still hover the fan/block button.

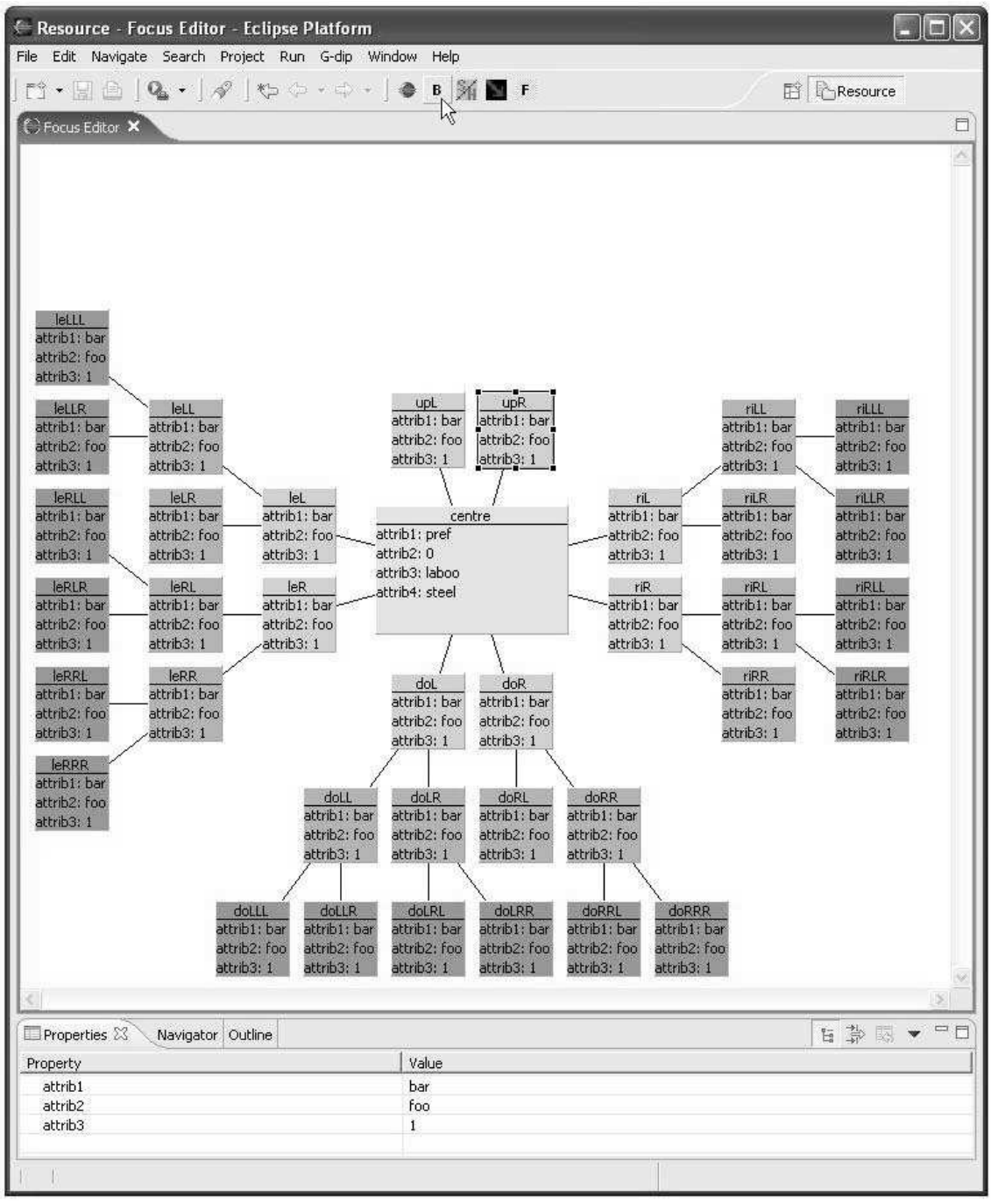

Fig. 5.7 After fan in, object still selected and mouse pointer hovering fan/block button.

When the fan out is done the objects is rearranged and the children of upR is again included in the view. This shows that the action doesn't change the model but simply alters the criteria of the choice algorithms.

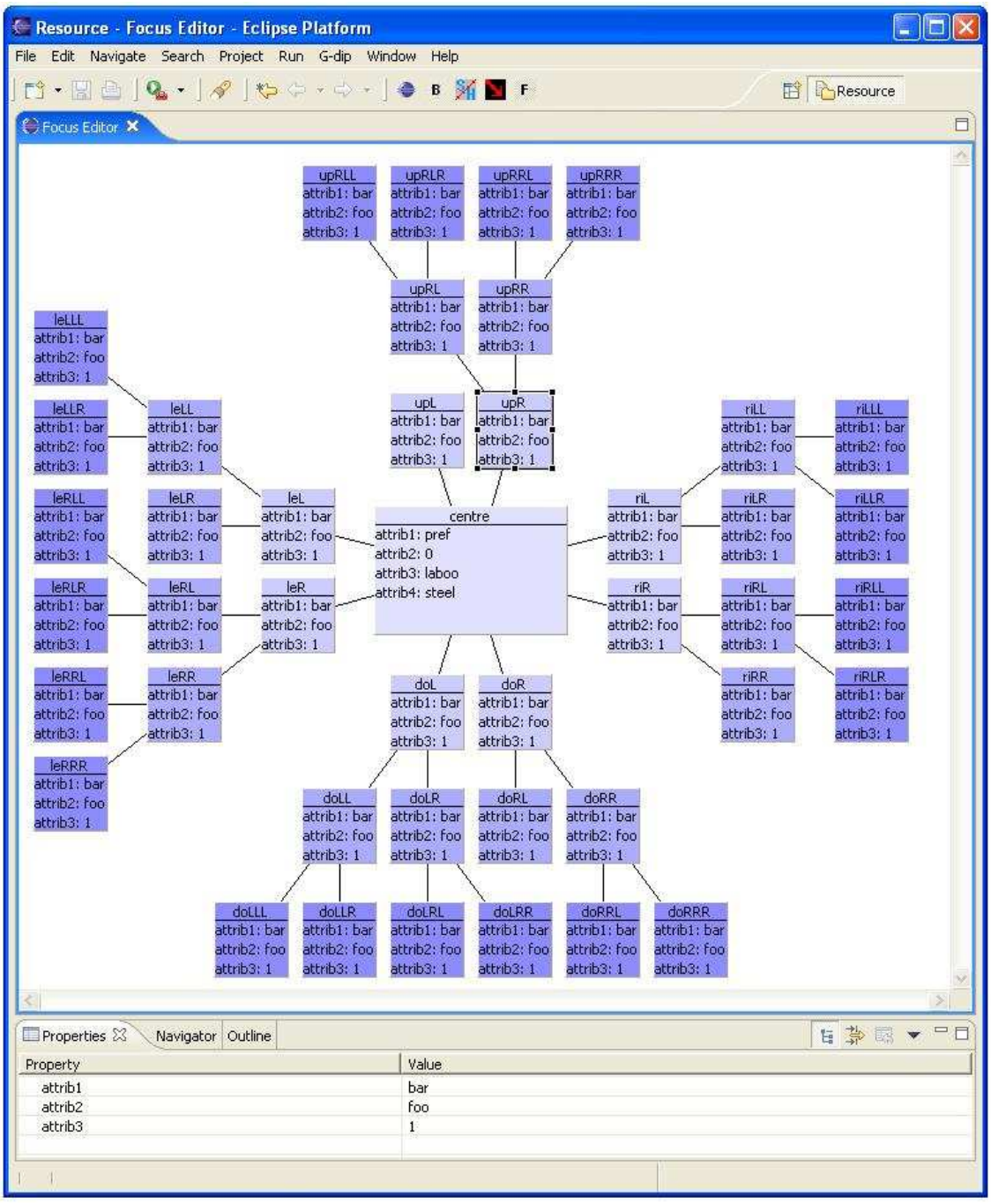

Fig 5.8 after fan out

5.1.4 Changing Render depth

Setting the number of steps of each relation type makes for easy manipulation of the focus view to show only the data the user is interested in. In Fig. 5.9 we can see the starting view and the mouse pointer hovering over the render depth button.

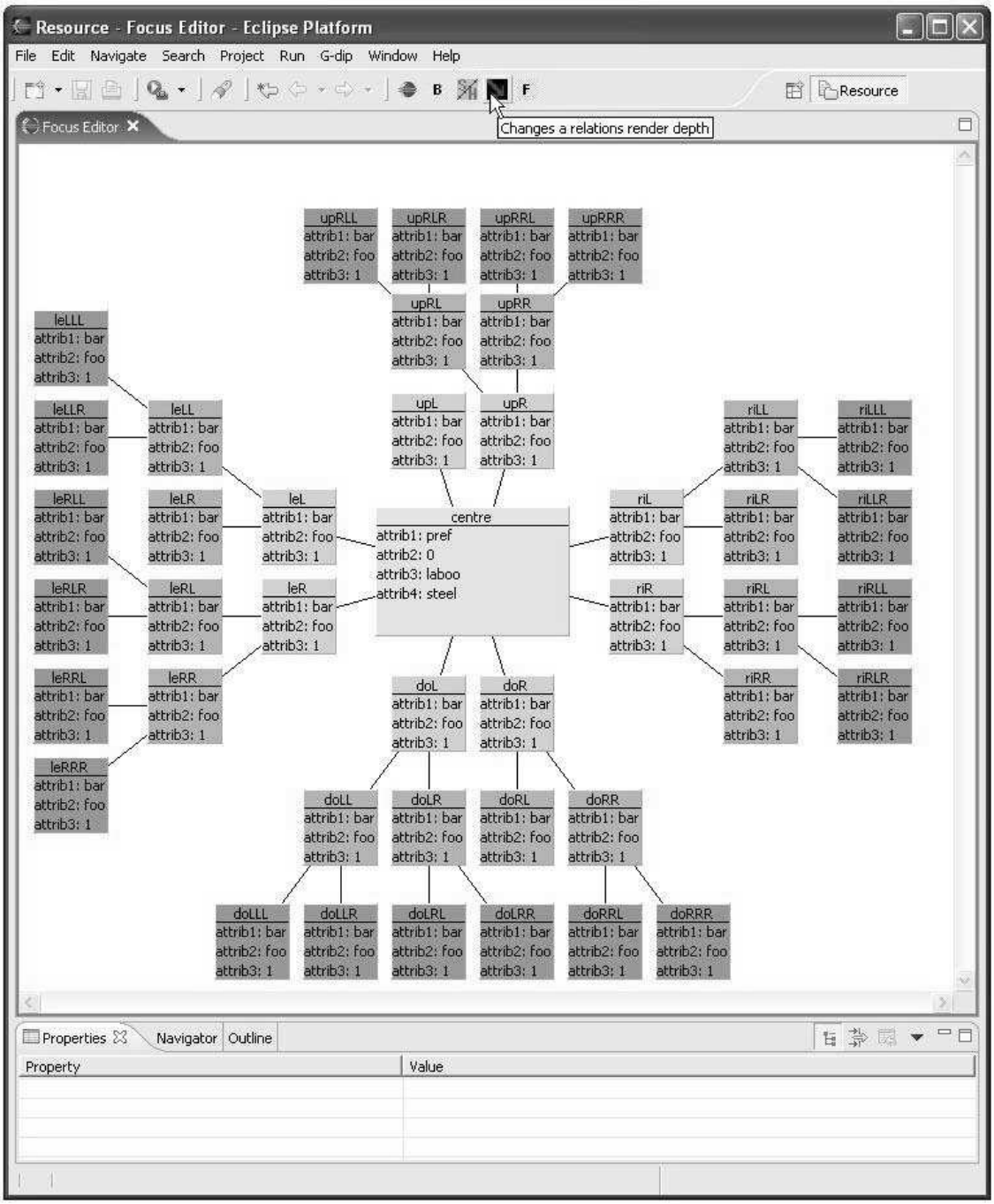

Fig. 5.9 Mouse pointer hovering the change render depth button

When the render depth button is pressed the render depth dialog appears, it gives a list of available relation types. These relation types don't have to be visible in the view or even represented in the model, they are the types the program is capable of recognizing and make alterations to.

In Fig 5.10 we can se the dialog as it pops up and the mouse pointer placed at the right relation type. With a mouse click we move to fig 5.11. When the relation type is selected its current render depth limit is displayed and selected, this shows the current value and allows the user to directly type in the new value without having to remove the old one.

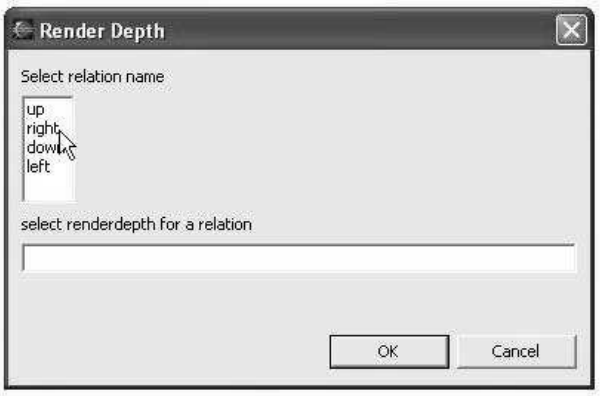

Fig. 5.10

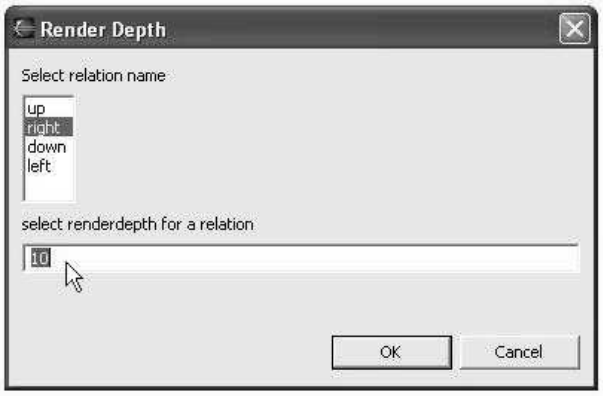

Fig. 5.11

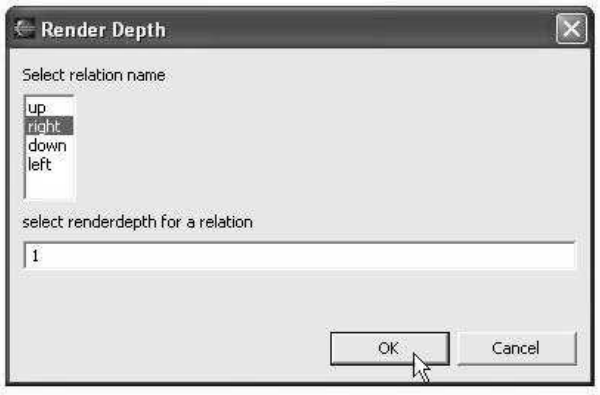

Fig. 5.12

As seen in fig. 5.12 we chose 1 to be the new render depth and the mouse pointer now hovers the ok button. Again a mouse click takes us to the next figure.

Fig. 5.13 shows us that the view is the same but for the right relation type which now is only shown one step removed from the focused object.

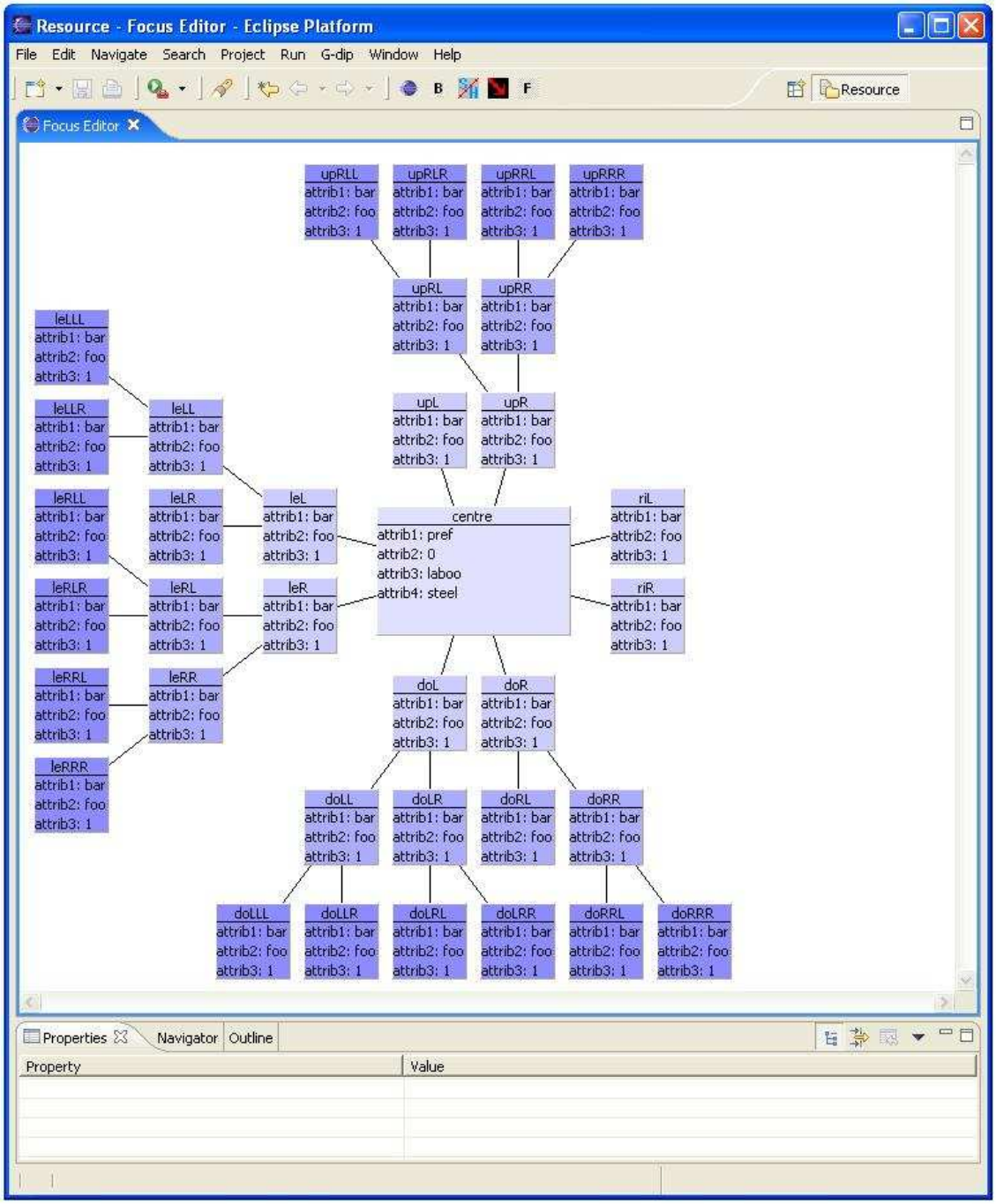

Fig. 5.13

5.1.5 Hide relation

On the same idea as the render depth changer above, we got the hide relation action. The difference is that hide relation is an on or off state: either it is hidden or not, the hidden state would look exactly the same as if we set the render depth to 0. However if we use the hide option the render dept will be remembered when we chose to show the relation again. . In Fig. 5.14 we can see the starting view and the mouse pointer hovering over the hide relation button.

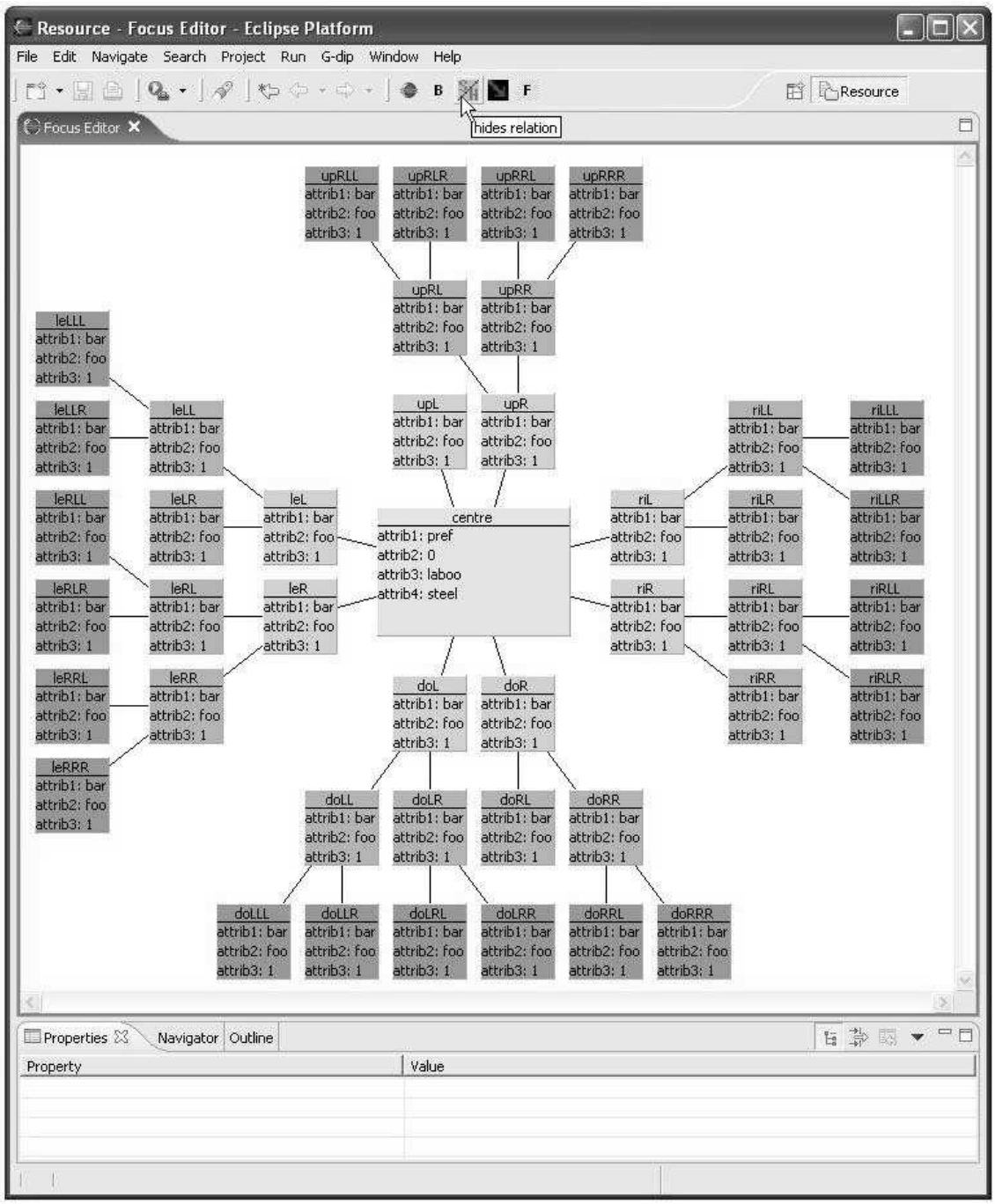

Fig. 5.14

Pressing the hide relation button we open the hide relation dialog (fig. 5.15) just as in the render depth dialog there is a list of the available relation types and when one is selected we get the current state displayed, in this case relation type down has “show“ as its state.

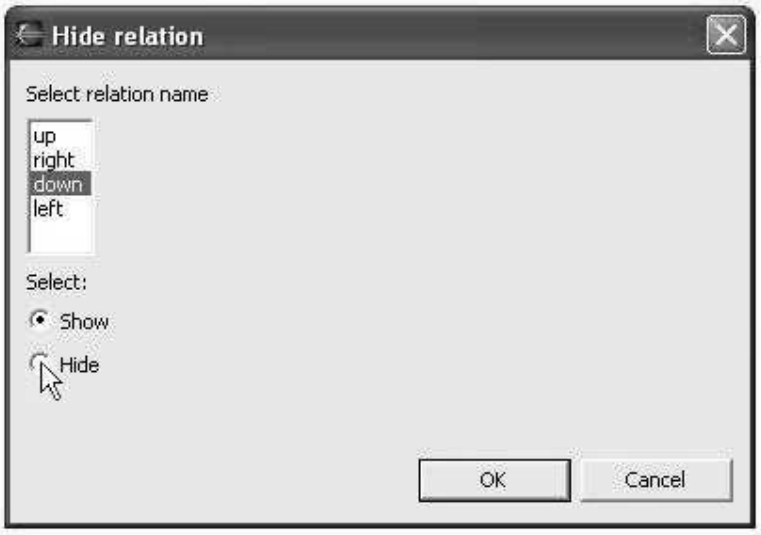

Fig. 5.15

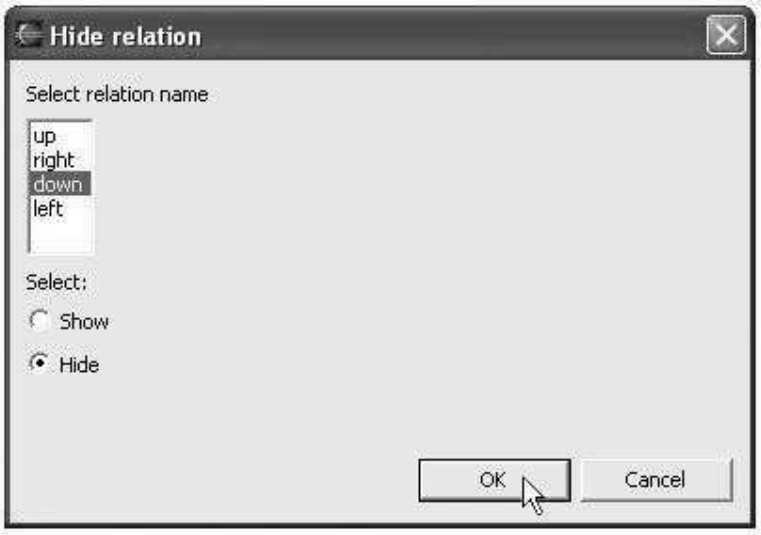

Fig 5.16

The state for relation type down is set to hide, the ok button is pressed.

The resulting focus view is seen in fig 5.17 where there are no object displayed that are related to the focused object by the down relation type.

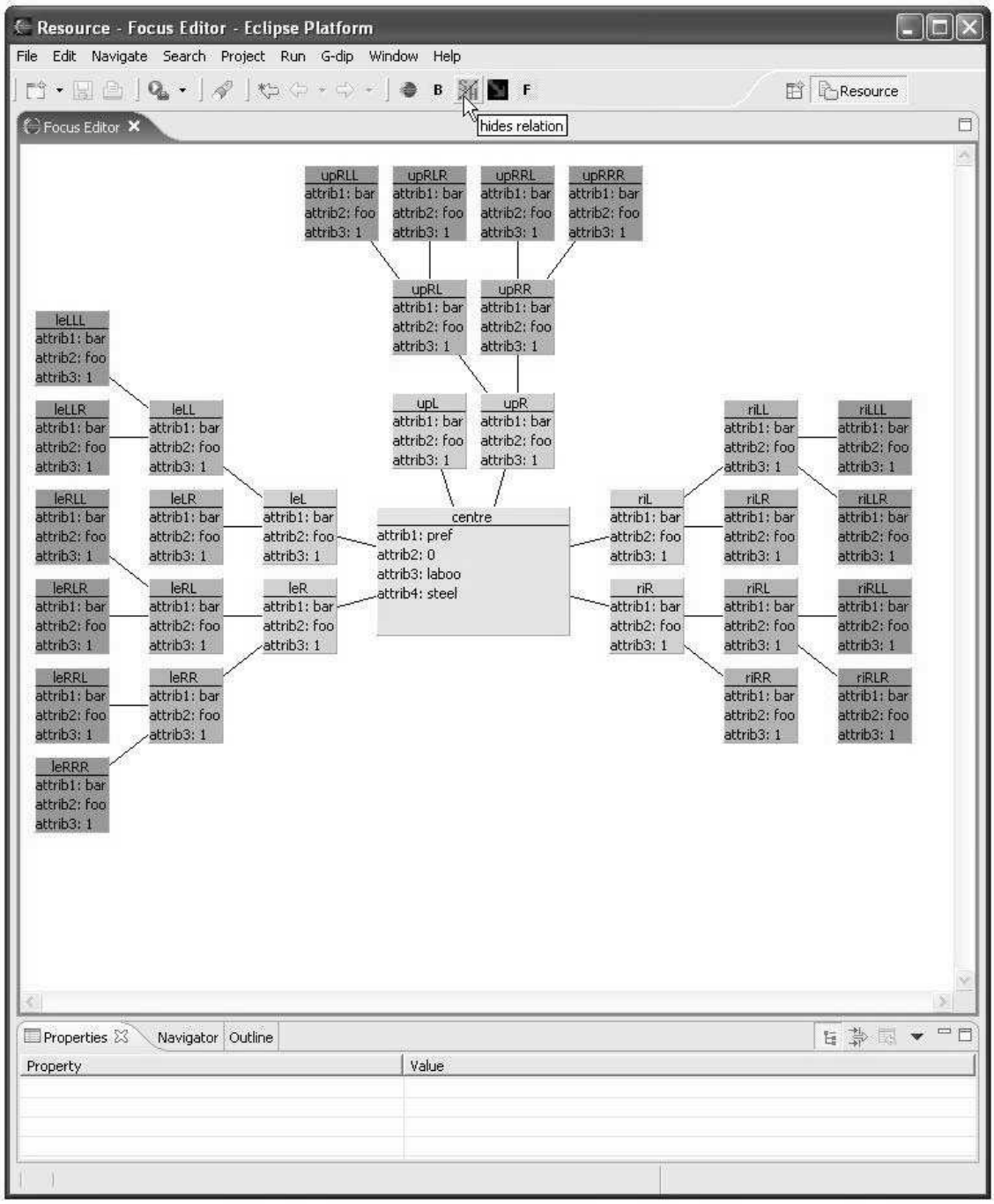

Fig. 5.17

5.2 Performance tests

5.2.1 Platform

AMD athlon 3000+, 512 MB ram, Windowx XP, java 2 SE 5.0, eclipse 3.0

5.2.2 Set up

Tests where run on a standard personal computer with a few open though inactive programs as to simulate standard running conditions for the program. The data where generated in four binary tree structures where one tree where completely filled to a set depth before the next where used so the last of the four where not always as large as the other three. When the tests where done all of the objects in the model where shown and thus there where no relations that was to be ignored, if there where many such relations the refocus times should be a little longer but only marginally.

5.2.3 Tests

For each model size a refocus from displaying the minimum possible number of nodes to displaying the maximum number. For all tests several different measurements where made, first of all the total time and also the times for choosing the visible nodes and for the layout. The results of these tests are done with all data already in the model, if it where necessary to retrieve data from a database that would slow things down considerably especially in the beginning when all data needs to be retrieved.

5.2.4 Results

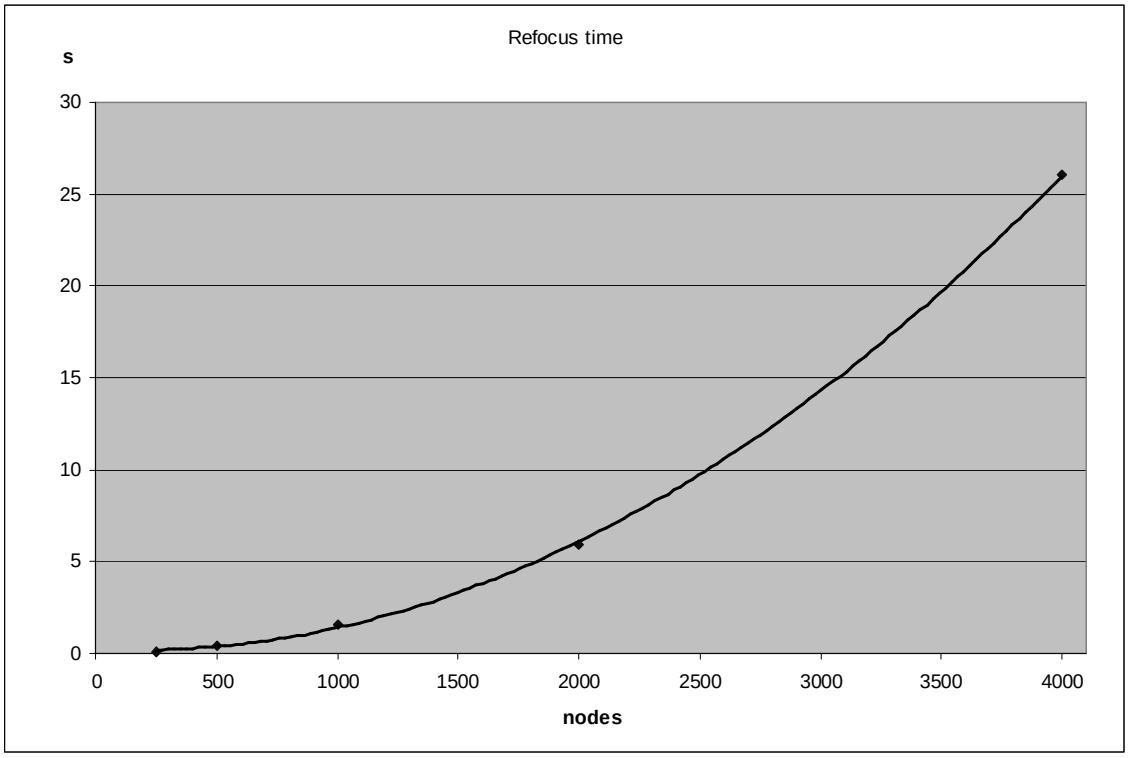

Fig. 5.18 Refocus time graph.

The runtime of a refocus grows in a polynomial manner

Formula 5.1 $T = 2*10^{-6} n^2 - 9*10^{-4} n + 0{,}3512$

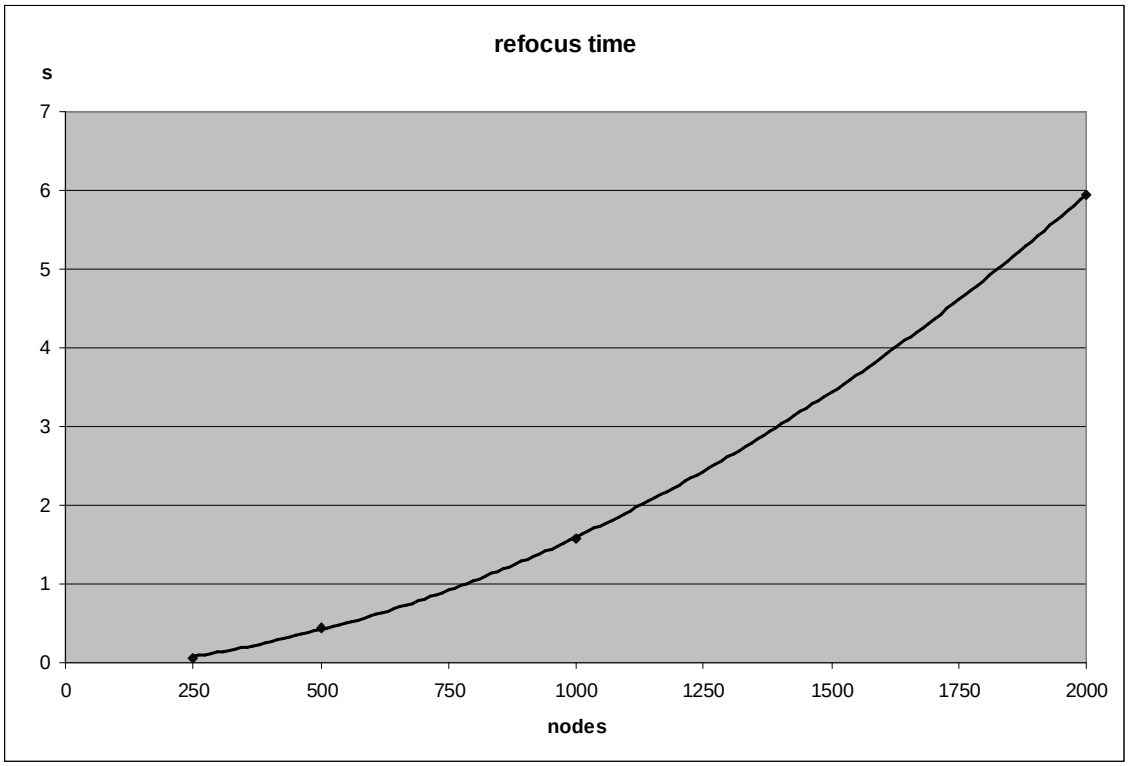

Fig. 5.19 Refocus time graph, enlargement

The more relevant part of the refocus time diagram as waiting times of more then a few seconds is way too long to be useful.

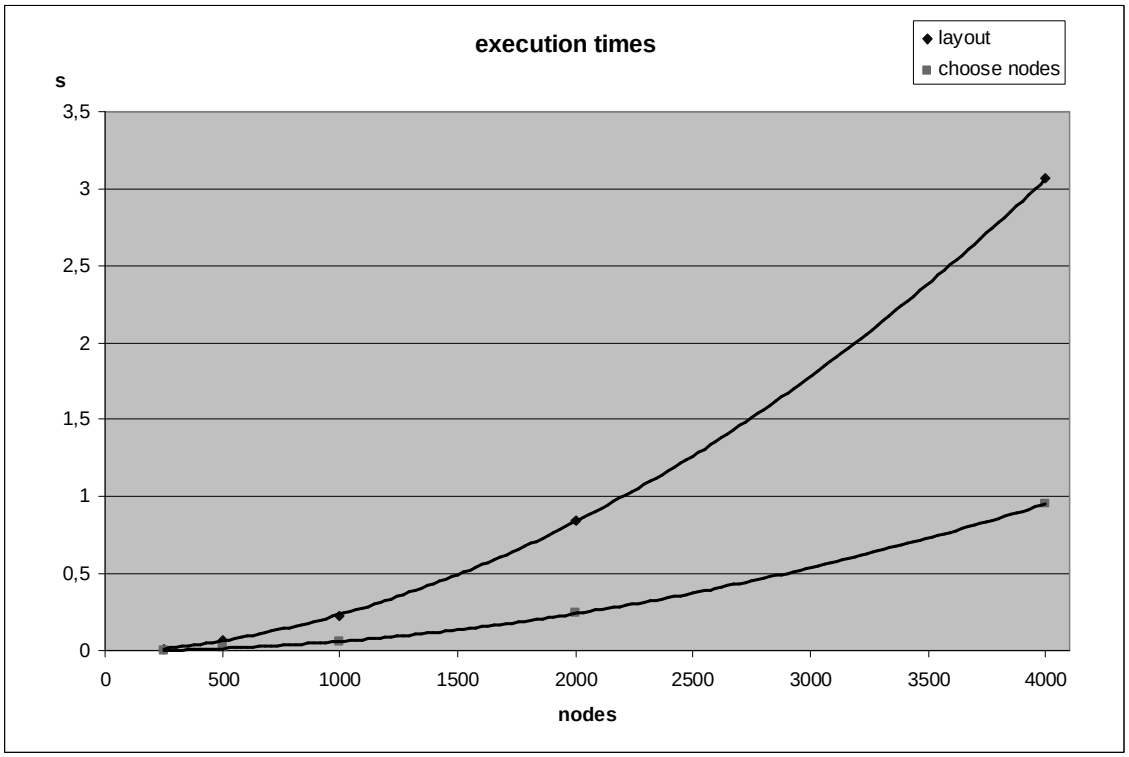

Fig. 5.20 Refocus time graph for layout and node choosing algorithms

The execution times for the two major algorithms in our program both have execution time curves similar in shape to that of the full refocus Fig 5.18 and Fig 5.20. As shown in the percentage diagram (fig 5.21) the node choice algorithm keeps at ~3 - 4 % of the total runtime. The layout algorithms execution time grows at a slower rate than the total time and takes up ~ 14 – 16% of the runtime.

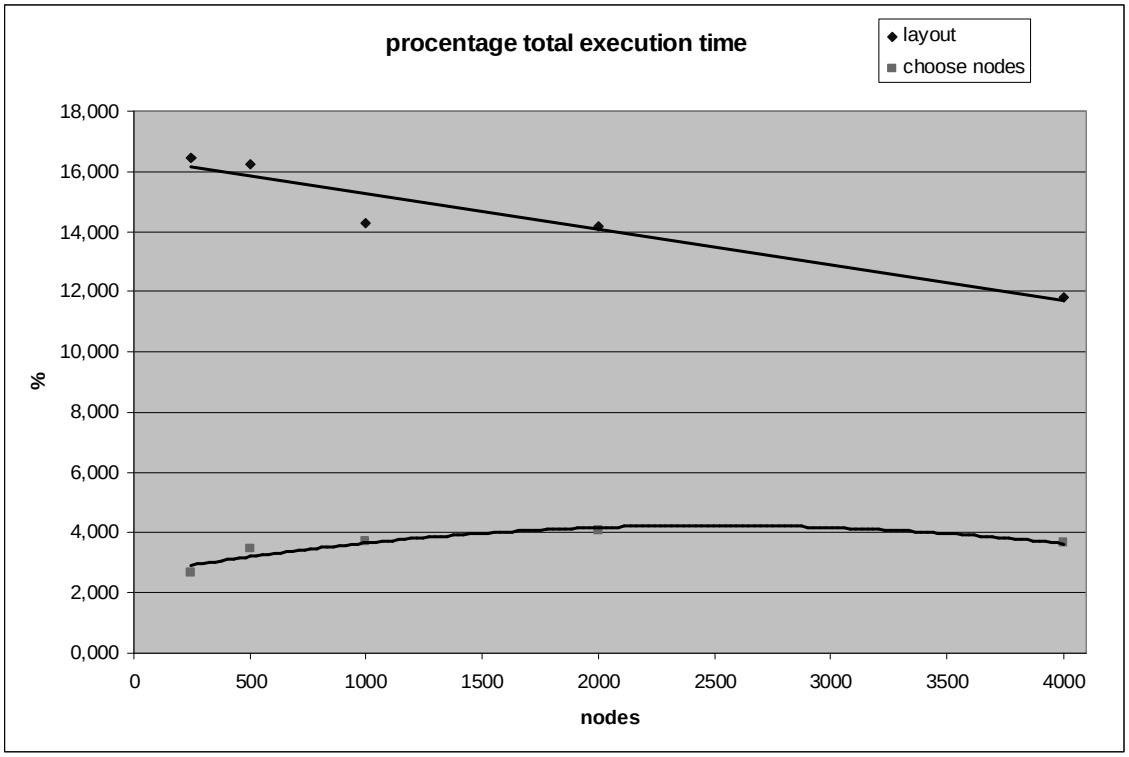

Fig. 5.21 Layout and node choosing algorithms runtime in percent of total refocus time

5.2.5 Discussion

The total refocus time is the most interesting as it shows us how many nodes the program can handle while keeping the delays within reasonable bounds. As can be seen in fig 5.18 the delay for a refocus grows rapidly when there are many nodes. From this diagram we can see that node counts over 2000 nodes starts taking over 6 seconds which is defiantly too long to be considered useful. So the useful frame is from 0 to about 1000 nodes. 1000 nodes are however plenty of information, even if the user only needs to browse a tenth of that the focus time won't be much of a bother. If the user is stepping around in the data with many fast refocuses the node count should be kept to at most 500 where the refocus time is at about half a second.

As most of the execution time, about 80%, is performed in the eclipse framework there is very little that we can do there to improve the performance. The parts that can be improved are the layout and the node choice algorithms, the node choice algorithm takes up to 4% of the execution time, so improvement

of that wouldn’t make much of a difference. The layout is more time consuming at about 16% and there might be worth it to try to improve its performance if the current performance would prove too bad in the end but at the moment there is no need to do that.

6 Conclusion

We have produced the working skeleton for a graphical database program for use in Eclipse. This result doesn't quite fulfill the initial goals to actually produce a full working database browser. As we got some insight into the subject it was clear that it would not be possible to do all this. Looking at the revised goals this is exactly what we set out to do.

The database is displayed in Quad tree fashion and the tree can be navigated through selecting different objects and changing the focused object. Databases can be massive and therefore we have implemented limits on how many steps a relation is allowed to proceed, also we have fanning and blocking to further prune the tree to a more readable form. The Program also has the ability to read a database from and save it to disc via an XML- interface, this will also save all the settings such as fanned relations.

All these functions where tested and timed showing that the program preformed within reasonable time limits. This proves that our choice of algorithms where sound and good enough.

Most of our insights are in the Eclipse/working with new paradigms area, which individually is too insignificant and too numerous to mention here, but we have some general thoughts about Eclipse. One of Eclipses greatest strengths is also one of its greatest weaknesses: It is all there; almost everything you need to do is made for you just waiting to be used. For someone with a lot of experience with eclipse this is very convenient and time saving. But finding all these interfaces and classes for the first time is very time consuming and when found three quarters of the work is still to be done in understanding how and where to use it correctly. Most of all we learned that it is invaluable to have some one with this experience to ask about this sort of things.

In retrospect there are very few things we would like to have done differently, most of all we read some books on subjects that later became out of scope, mostly books about databases as that part was left out of the project. That time would have been better used to study books about Eclipse/GEF.

7 Further Works

If we look at the use cases described in appendix A many of those are as of yet not implemented and also much of the infrastructure these use cases need to function is not implemented yet.

7.1 Infrastructure

The first and most obvious thing that needs to be done is to create the connection to a database as that part is totally left out in this project. To do this we need to firstly create a library to read the database and translate it to our model classes so that it can be used by our plug-in.

Then we need to rework the node choice algorithm (chapter 3.2.1) so that it notices if there are any nodes it is supposed to choose that are not yet read into the model then invoke some method of the library described in the previous paragraph to add this node to the model and then resume its work. To make this work there will be need to store some additional data in the DParam to allow fast checks to see whether there are any relations out of the model from whichever node is being processed at the moment.

7.2 Functionality

There are several bits of functionality that we have in the vision of the finished product that we haven't begun to implement simply because of the time frame of this project.

7.2.1 Favorites

Favorites is supposed to be similar to the favorites found in most modern internet browsers. Granting the possibility to have several regular starting points and thus allowing an easy and fast way of restarting something the user has worked with earlier.

Use cases
Add/ remove favorite
Focus on favorite

Add and remove is quite self explanatory, and the focus on favorite simply does a refocus with the favorite object in focus, for further info see appendix A

7.2.2 Object notes

Object notes are an addition to the information already shown when an object is selected, more or less it is a notepad for the user where he can add his or her own thoughts, clarifications and/or notes.

Use cases
Save object specific note
Also hide/show object specific notes

Only the save object specific note is taken up among the use cases but as hide/show will be taken care of by the eclipse framework there is no real need to describe them.

7.2.3 Messages

With some large database systems it is possible to subscribe to the changes of objects in the database. This takes the form of an update message whenever an object the user subscribes to is changed, these messages will be collected and displayed.

Use cases
Focus on ping
Delete ping

7.2.4 Stepping

Most web browsers have buttons for undoing one or more actions and also to redo the ones undone. This functionality would be useful for our plug-in as well.

Use cases
Back
Back several steps
Forward
Forward several steps

7.2.5 Visible objects

Our current program takes no heed to how many objects is fitted into a zone. If there are too many objects they may be drawn outside the area designated to that zone. The planed solution to this is to make a check and only show the amount that can be fitted in the zone and add scrolling so that the figures in the zone can be scrolled pretty much like a list with a scrollbar.

Use cases
Scroll visible objects

7.2.6 Search

The plug-in is meant to handle the standard query-response searches as well as the graphical representation.

Use cases

Search
Refine search
New focus

7.2.7 Database

Logging in and out of databases are also left to be done.

References

[1]LAMPING, J., RAO, R., and PIROLLI, P. 1995. A focus C context technique based on hyperbolic geometry
for visualizing large hierarchies. In Proceedings of CHI ’95, ACM Conference on Human Factors in Computing Systems. ACM, New York.

[2]CZERWINSKI, M. and LARSON, K. 1997. The new Web browsers: They’re cool but are they useful?
In People and Computers XII: Proceedings of the HCI ’97 Conference, H. Thimbleby, B. O’Conaill,
and P. Thomas, Eds. Springer Verlag, Berlin.

[3] PETER PIROLLI, STUART K. CARD, and MIJA M. VAN DER WEGE. 2003. The Effects of Information Scent on Visual Search in the Hyperbolic Tree Browser
Palo Alto Research Center. In Transactions on Computer-Human Interaction. ACM, New York.

[4] Object Technology International, Inc.
Eclipse Platform Technical Overview July 2001 (updated for Eclipse 2.1 in Feb. 2003)
www.eclipse.org

[5] Chris Aniszczyk Using GEF with EMF June 8, 2005
www.eclipse.org

[6] Bill Moore, David Dean, Anna Gerber,Gunnar Wagenknecht and
Philippe Vanderheyden. Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework February 2004.
www.ibm.com/redbooks

[7] Eclipse Homepage as of 2005-10-01
www.eclipse.org

[8] Alan Dix, Janet E. Finlay, Gregory D. Abowd, Xristine Faulkner. 2003. Human Computer Interaction. ISBN 0130461091

[9] Donald A. Norman. 2002. The Design of Everyday Things. ISBN 0-465-06710-7

Appendix A Use Cases

Start with a desktop favorite

-Prerequisites

There is a favorite file available and the plug-in is installed.

-Sequence of interactions

User “runs” the favorite file and the plug-in is started and a login dialog run, when user have logged in the plug-in shows the favorite object in focus and selected.

-Post requisites

Plug-in running and logged in to the database and an favorite object is in focus and selected

Load a desktop favorite

-Prerequisites

There is a favorite file available and the plug-in started.

-Sequence of interactions

User “runs” the favorite file and the plug-in is started and a login dialog run it the user has not already logged in, when user are logged in the plug-in shows the favorite object in focus and selected.

-Post requisites

Plug-in running and logged in to the database and an favorite object is in focus and selected

Login to database

-Prerequisites

Plug-in running and not already logged in.

-Sequence of interactions

User starts by selecting “login” from a scroll down menu and types username and password in a dialog. Plug-in sends information to the database and logs itself in. User is informed if the login was successful or unsuccessful. If it was unsuccessful user gets to retry.

-Post requisites

Logged in if user has access to database.

Logout from database

-Prerequisites

Plug-in running and logged in.

-Sequence of interactions

User starts by selecting “logout” from a scroll down menu. An affirmation popup is displayed if “yes” is selected Plug-in sends information to the database and logs itself out.

-Post requisites

User is logged out from database

Search

-Prerequisites

Plug-in running and logged in.

-Sequence of interactions

User types the keys to search for in the search view and presses the search button. The information is then sent to the database where the search is executed and the result sent back to the Plug-in which displays the result in the search view.

-Post requisites

The result of the search is displayed in the search view.

Refine search

-Prerequisites

Plug-in running and logged in, the result of a search is displayed in the search view.

-Sequence of interactions

User presses the refine search button in the search view. The information used in the previous search is then displayed instead, and the user may alter them.

-Post requisites

The keys from the previous search is displayed in the search view.

New focus

-Prerequisites

Plug-in running and logged in, the result of a search is displayed in the search view.

-Sequence of interactions

User selects the appropriate object in the search view. The Focus view then renders the object and it is then selected.

-Post requisites

The Focus view has the object focused and selected.

Focus on ping

-Prerequisites

Plug-in running and logged in, the result of a change ping is displayed in the update view.

-Sequence of interactions

User selects the appropriate object in the update view. The Focus view then renders the object and it is then selected.

-Post requisites

The Focus view has the object focused and selected.

Back

-Prerequisites

Plug-in running and logged in, at least two objects has been in focus

-Sequence of interactions

User presses the back button and the object in focus is changed to the previously focused object which is also selected.

-Post requisites

The Focus view has the previously focused object focused and selected and the previous list and forward list is updated.

Back several steps

-Prerequisites

Plug-in running and logged in, at least three objects has been in focus

-Sequence of interactions

User presses the back several steps button and picks one object from the list of previously focused objects. That object will be focused on and selected.

-Post requisites

The Focus view has the previously focused object focused and selected and the previous list and forward list is updated.

Forward

-Prerequisites

Plug-in running and logged in, forward list is not empty

-Sequence of interactions

User presses the forward button and the object in focus is changed to the object first in the forward list object which is also selected.

-Post requisites

The Focus view has the object focused and selected and the previous list and forward list is updated.

Forward several steps

-Prerequisites

Plug-in running and logged in, at least two objects is in forward list

-Sequence of interactions

User presses the forward several steps button and picks one object from the list of previously focused objects in the forward list, that object will be focused on and selected.

-Post requisites

The Focus view has the previously focused object focused and selected and the previous list and forward list is updated.

Refocus focus view

-Prerequisites

Plug-in running and logged in, at least one not focused object is shown in the focus view.

-Sequence of interactions

User selects one of the not focused objects in the focus view selects to focus on that object instead. The focus view then changes focus to that object and selects it.

-Post requisites

The Focus view has the new development aspect of the object focused and selected.

Refocus V view

-Prerequisites

Plug-in running and logged in, an object is shown in the V view

-Sequence of interactions

User selects one of the development steps from the V view selects to focus on another development aspect of the object currently in focus. The focus view then changes focus to that development aspect and selects it.

-Post requisites

The Focus view has the new development aspect of the object focused and selected.

Read document

-Prerequisites

Plug-in running and logged in, a document object is shown in the focus view.

-Sequence of interactions

User selects a document object and chooses to read it. The appropriate program is then started and the document opened in it.

-Post requisites

Document is showed in an appropriate program.

Change visible objects

-Prerequisites

Plug-in running and logged in, an object and some of the objects that is related to it is shown in the focus view.

-Sequence of interactions

The user selects a group of objects that is related in the same way to the object in focus. Right click on them and select “choose visible” this will launch a popup where the user can select which objects to show. If the number of selected objects are greater then the maximum allowed objects of that type the exceeding objects will not be shown.

-Post requisites

The focus view will show the selected objects.

Change number of visible objects

-Prerequisites

Plug-in running.

-Sequence of interactions

Select “visible objects” from a scroll down menu and a popup will be launched where the user can specify the maximum number of objects that can be shown for each relation and levels from the object in focus. When the done button the focus view will be drawn according to the new rules.

-Post requisites

The focus view will be drawn in accordance with the new rules.

Change render depth

-Prerequisites:

Plug-in running.

-Sequence of interactions:

User selects “Change render depth” in a scroll down menu. The plug-in displays the “Change render depth” window. The user then changes how many levels of a specific relation will be rendered. The user then closes the window and the plug-in will register the changes.

-Post requisites:

The Focus view now shows the appropriate number of levels of all relations.

Focus on favorite:

-Prerequisites:

Plug-in running, user logged in.

-Sequence of interactions:

User selects the appropriate object in the Favorite scroll down menu. The Focus view then focuses on the object and selects it.

-Post requisites:

The Focus view has the object focused and selected.

Add to Favorites:

-Prerequisites:

Plug-in running, user logged in, object focused.

-Sequence of interactions:

User selects the “Add to Favorite” option in the Favorites scroll down menu. The plug-in then displays an affirmation pop up window. The user then selects “OK“ and the Plug in will add the focused object to the Favorites list.

-Post requisites:

The added object will now be available from the Favorites scroll down menu.

Remove favorite:

-Prerequisites:

Plug-in running, Favorite list not empty.

-Sequence of interactions:

User selects the “Remove Favorite” option in the Favorites scroll down menu. The plug-in then displays a list of the objects in the Favorites list. The user then selects the object he wishes to remove from the Favorites list and selects “OK”. The plug-in then displays an affirmation pop up window. The user selects “OK” and the Plug in will remove the object from the Favorites list.

-Post requisites:

The removed object will not be available from the Favorites scroll down menu.

Delete ping

-Prerequisites:

Plug-in running, ping inbox not empty.

-Sequence of interactions:

User selects a ping in the update view right click on it and choose delete.

-Post requisites:

The removed message will not be available.

Show relation:

-Prerequisites:

Plug-in running, object selected, A specific relation is not shown in the Focus view.

-Sequence of interactions:

User Right clicks on the focused object and selects the appropriate relation. The plug in will then draw all relations of that kind in the Focusview.

-Post requisites:

The focus view will show the appropriate relations.

Hide relation:

-Prerequisites:

Plug-in running, object selected, relation visible.

-Sequence of interactions:

The user Right clicks on a relation of the type he wants to hide and selects Hide relation from the scroll down menu. The Plug-in then hides all Relations of that kind.

-Post requisites:

All the relations of the specific kind is not shown in the Focus view

Fan out unfocused object:

-Prerequisites:

Plug-in running, focused object selected, a unfocused object that isn't Fanned out.

-Sequence of interactions:

The user Right clicks on the object he wants to fan out and selects Fan out from the scroll down menu the plug in then displays the relations of the object.

-Post requisites:

The relations of the Fanned out object is shown in the Focus view

Fan in unfocused object:

-Prerequisites:

Plug-in running, focused object selected, a unfocused object that is Fanned out.

-Sequence of interactions:

The user Right clicks on the object he wants to fan in and selects Fan in from the scroll down menu. The plug in then removes the objects visible relations from the Focus view

-Post requisites:

The relations of the Fanned out object is not shown in the Focus view

Scroll visible objects:

-Prerequisites:

Plug-in running, focused object selected, There are more relations of a kind than can be shown.

-Sequence of interactions:

The user presses the next button in the Focus window next to the relations he wants to scroll. The Plug-in then shows the next set of relations of that type.

-Post requisites:

The Focus view now shows the next set of relations of that kind

Save object specific note:

-Prerequisites:

Plug-in running

-Sequence of interactions:

User focuses an object and the Plug in then displays the object. The user then proceeds to select an object and then selects the "object note" view and writes down a note and presses save. The Plug in then saves the note and associates it with the object.

-Post requisites:

The Plug in now displays the notes about the object in the "object note" view when that object is selected.

Select object:

-Prerequisites:

Plug in running, object focused.

-Sequence of interactions:

The user Left-clicks on the object he wishes to select and the Plug in then displays information on the object in the “object info” view.

-Post requisites:

The object is selected and its info is presented in the “object info view.