CODEN: LUTFD2/(TFRT-5425)/1-54/(1990)

# Evaluation of the Unidraw Framework for Interactive Object Editors

Tomas Szabó

Department of Automatic Control
Lund Institute of Technology
August 1990

**Department of Automatic Control**
**Lund Institute of Technology**
P.O. Box 118
S-221 00 Lund Sweden

*Document name*
MASTER THESIS

*Date of issue*
August 1990

*Document Number*
CODEN: LUTFD2/(TFRT-5425)/1–54/(1990)

*Author(s)*
Tomas Szabó

*Supervisor*
Dag Brück

*Sponsoring organisation*
ABB Corporate Research, KLL

*Title and subtitle*
Evaluation of the Unidraw Framework for Interactive Object Editors

*Abstract*

Unidraw is an object oriented framework facilitating the development of graphical object editors. It was developed by John Vlissides at Stanford university.

The Unidraw architecture and its prototype implementation are examined considering functionality and usefulness. A small editor is implemented, using the Unidraw C++ prototype, revealing its usefulness and some design problems.

*Key words*

*Classification system and/or index terms (if any)*

*Supplementary bibliographical information*

*ISSN and key title*

*ISBN*

*Language*
English

*Number of pages*
54

*Recipient's notes*

*Security classification*

*The report may be ordered from the Department of Automatic Control or borrowed through the University Library 2, Box 1010, S-221 03 Lund, Sweden, Telex: 33248 lubbis lund.*

## 1. ACKNOWLEDGEMENTS

This report is the documentation of the master thesis project carried out at ABB Corporate Research Department KLL, at IDEON research park, Lund. The work concludes my electrical engineering studies at Lund Institute of Technology (LTH), and was done during the late spring and summer of 1990. My supervisors were Dag Brück from the department of Automatic Control LTH, and Bo Johansson here at KLL.

Bo and I had some memorable discussions about whether Unidraw supports graphical object-editors, or graphical-object editors. To the whole staff and other "exarbetare" at KLL: **Thanks for all the pleasant times we spent together.**

## 2. INTRODUCTION

Programming environments and new software packages that facilitate program development are emerging constantly, and the problem is to know how useful they are. In this master thesis, the Unidraw software package is examined, a framework for developing graphical based object editors in different domains.

However, implementing a graphical object editor is not an easy task. And even if there are many packages facilitating the construction of graphical user interfaces, they are general and do not aid the special functionalities of these editors. The Unidraw [25] architecture simplifies the construction of graphical object editors by providing programming abstractions that are common for these editors.

### 2.1 Glossary

The following terms are used in this report:

— **Application objects** are the abstract objects of an application domain. The difference between graphical and non-graphical application objects has to be stressed. While graphical objects already have a well defined appearance, a graphical appearance has to be defined for non-graphical objects. Otherwise they could not be edited naturally in an interactive graphical editor. For example there is a great difference between an editor for music scores and an editor that can edit music! The music scores editor only lets the user manipulate the symbols representing the notes on the sheet, while a music editor has the notes only as one representation of the music. A music editor could have other interfaces to the music than pure music scores. For example views of key boards or other control panels.

  Application objects are often structured in some way and are then called structured application objects.

— **Graphics based object editors** are editors that use interactive graphics to let the user edit application objects. The editing is done by direct manipulation of the application objects' graphical representations. These representations make it easier for the user to make complex operations using manipulations that are more intuitive than the use of often long and complex commands. The user can then concentrate on the object editing instead of the remembering of commands.

— **Abstraction**, is a way of separating and isolating features of reality. It often means simplification and hiding of the details, making generalizations.

### 2.2 Overview of the report

**Project Description** is a short description of why and how the project was done, and the working environment.

**Introduction to Unidraw** tells about the design and prototype implementation of Unidraw, showing the main abstractions and their protocols.

**Specification of an evaluation editor**, "Model Editor", introduces an abstraction of a model and specifies an editor capable of editing such models. This is made as a test case to see how easy it is to use the Unidraw abstraction and prototype implementation.

**The Model Editor prototype implementation** describes how the editor was implemented.

And in the **Concluding remarks**, reflections over Unidraw and the implementation of the model editor prototype is done.

## 3. PROJECT DESCRIPTION

This master's thesis is about understanding how the Unidraw architecture and its prototype implementation works, and how it facilitates the construction of graphical object editors. The work consists of a theoretical part and a practical part.

In the theoretical part, Unidraw was learned by reading documentation and studying the source code of the prototype implementation. Unidraw is a relatively new software package, so there is not so much written about it yet. Even if the Unidraw architecture is ready, its implementation is still under development.

In the practical part of the work, a design and implementation of a small editor is done, using the Unidraw prototype as a framework. By making such an editor implementation, a better understanding of Unidraw is achieved. The editor implementation relies on the Unidraw snapshot we had at hand (December 1989), and parts of the experimental editor applications that followed the snapshot (mainly the Schem schematic editor).

The working environment in the practical part, consisted of the Unidraw prototype C++ class library on top of the InterViews [17] library and the X-Window system (X11). The operating system was unix, run on a HP 9000/300 workstation. The graphics output was presented on a 1280 x 1024 resolution graphics display.

Starting with this work, KLL had a C++ compiler from Oregon Software, Which unfortunately could not compile the Unidraw prototype. In the middle of June, as HP just was finishing its new C++ compiler, we had the opportunity to join the beta-3 test. The HP C++ Beta-3 compiler succeeded with the compilation of the Unidraw library, so the practical part of the work could start.

# 4. INTRODUCTION TO UNIDRAW

This chapter is a summary of the dissertation of John Vlissides [12], where the Unidraw architecture and a prototype implementation of Unidraw is described. I follow the dissertation's division into chapters with minor differences.

## 4.1 Background

Unidraw is developed by John M. Vlissides at Stanford University. It is meant as a framework for creating object-oriented graphical editors in domains such as technical and artistic drawing, music composition, and circuit design. Before he developed the Unidraw architecture and the prototype implementation, he participated in the design and implementation of the InterViews [7,17] user interface toolkit (see chapter 4.3.2). This user interface toolkit offers abstractions that makes it easier for the programmer to create an interactive graphical interface to his programs. Buttons, scroll bars and other interactors are predefined.

## 4.2 The Unidraw architecture

In this section the Unidraw architecture will be described. First comes an overview of the architecture, outlining the major elements and how they can be assembled to form an editor for a particular domain. Then the elements will be considered in more detail, showing their semantics and relationships.

### 4.2.1 Overview

An editor for a particular domain relies on Unidraw for its graphical editing capabilities, on the InterViews toolkit for supporting the "look and feel" of the user interface, and on the window and operating systems for managing workstation resources.

Figure 1 shows the dependences between the layers of software that underlie a domain-specific editor based on Unidraw. Unidraw stands at the highest level of system software, contributing abstractions that are closely matched to the requirements of graphical object editors.

Unidraw has an object-oriented design in which objects encapsulate the common attributes of domain-specific editors. An object-oriented architecture has several advantages over more traditional approaches. (See Appendix I). Objects could mimic the behaviour of real world objects by defining a set of operations or **protocol** that other objects can use. Inheritance and dynamic binding makes extensions easy. A general concept can be embodied in a base class from which classes with particular behaviour are derived. Dynamic binding allows objects from the same class hierarchy to be treated uniformly, independent of their subclass, simply by manipulating them in terms of the protocol defined in the base class.

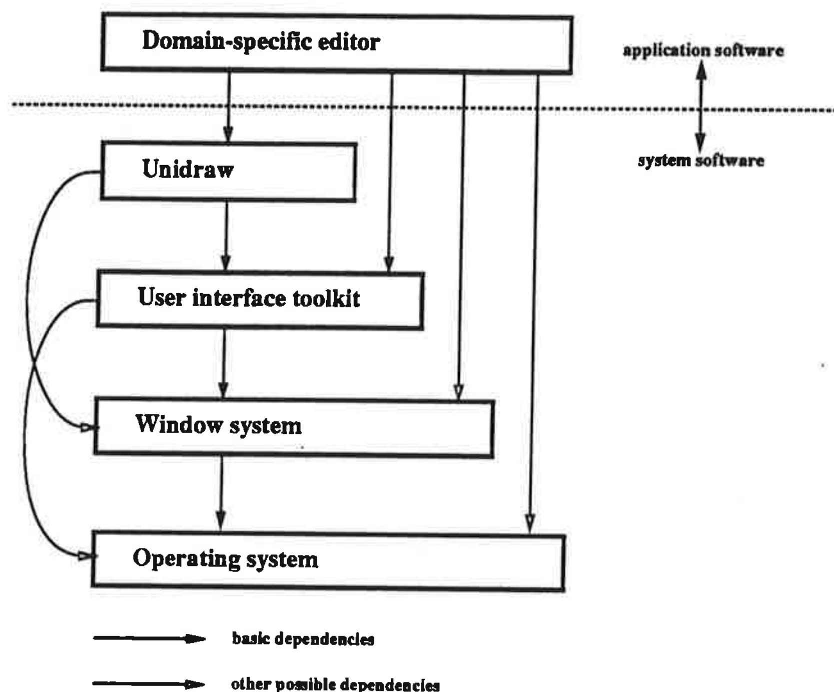



Figure 1. Layers of software underlying a Unidraw based domain-specificeditor.

The common attributes of domain-specific editors are reflected in four main abstractions:

1. **Components** represent the elements in a domain. The Components encapsulate the properties, appearance and semantics of the elements. A domain-specific editor's main objective is to allow the user to arrange components to convey information in the domain of interest.
2. **Tools** support direct manipulation of components. In order to reinforce the user's perception that he is dealing with real objects, tools employ animation and other visual effects.
3. **Commands** define operations on components and other objects. Commands have state and can be executed as well as interpreted. They can also be reverse-executed, allowing rollback to a previous state.
4. **External representations** convey domain-specific information outside the editor. A Component can define one or more external representations of itself.

The partitioning of graphical object editors functionality into components, commands, tools, and external representations is the foundation of the Unidraw architecture.

Components are further separated into two major abstractions, each covering a distinct characteristic of the components:

a. **Component subjects** encapsulate the context-independent state and operations of a component.

b. **Component views** supports a context-dependent presentation of the subject.

A component subject can have one or more views, each offering a different representation of and interface to the subject. Different views can reflect the subject's information in distinctive ways and can provide additional information as well. A view can also define what it means to manipulate a Component with a Tool.

The four main abstractions are supported by a structure of additional objects providing a standard framework for building domain-specific editors. Among these, the **Viewer, Editor, Unidraw** and **Catalog** objects specify how the editor mediates communication between Components, Commands, and Tools. There are a lot more supporting objects in Unidraw making the framework complete.

Figure 2 shows how the Viewer, Editor and Unidraw objects work together with the Component subject, Component view, Tool, and Command objects. The double headed arrows are indicate dependencies between instances, and the white arrows show how commands and tools can influence and change the component subject's state. The dashed fields depict how objects are graphically part of another object.

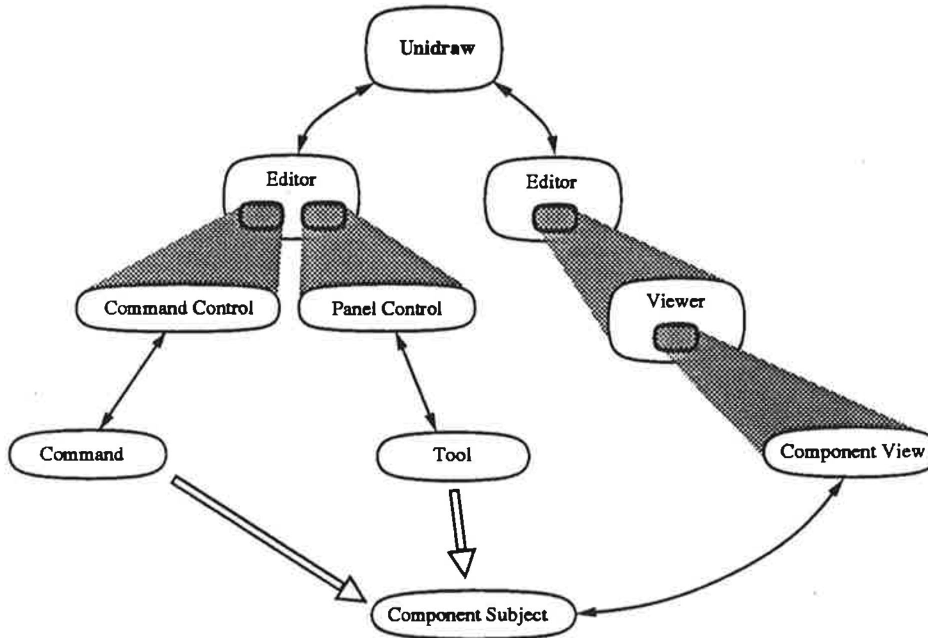



**Figure 2.** The Editor and Viewer interactive objects.

Component views are inserted in viewers, which in turn are inserted in an editor together with Command and Tool interactors. The Unidraw object opens and runs the editors.

#### 4.2.2 Components

Components represent the objects of interest in the editing domain. They define and manage information to mimic the behaviour of real-world objects. They can have one or more graphical representations and can support non-graphical representations as well. Components define how they respond to commands and tool manipulation.

The Unidraw architecture defines separate communication protocols for the Component's Subject and Views. Subjects define **Attach** and **Detach** operations to establish or destroy a connection to a

View. Every time the state of a Subject is changed, the **Notify** operation should be called to alert the Views of the differences. The Views then reconcile any inconsistency between the Subject's state and their own appearance. The **Update** operation could be used to notify the Subject that some of it's state has changed.

The Subject can interpret a Command object via the **Interpret** operation. **Uninterpret** negates the effect. The passed Command is interpreted by causing the operations to make appropriate changes in the subject's state depending on the type of the command, and/or to call the Command's (Un)Execute operation. These operations are highly component-specific, and deriving new components often includes redefinition of the Interpret and Uninterpret operations.

Because of the hierarchical structure of Components, operations for traversing one level up and down are defined both for the Subject and the View objects. These are the **GetParent**, **First**, **Last**, **Next**, **Prev**, and **Done** operations.

Subjects can also define one or more **StateVariable** objects and one **TransferFunction** object that can be accessed via the **GetState** and **GetTransferFunct** operations. The TransferFunction defines dependencies between different Connector bound StateVariables of a structured Component. It may for example be used for simple simulations.

The Views duplicate some of the operations from the subject protocol (Update, Interpret, Uninterpret, and the parent and child access operations) and adds **SetSubject** and **GetSubject** operations that set and return the view's subject. **Update** is usually called from the subject's Notify. **Interpret** and **Uninterpret** are defined because some objects manipulate Views rather than Subjects, for example the **Viewer** object. A Viewer sends Commands to its Views for (un)interpretation, which may in turn send it to its subject.

An important extension of the basic Component is the **GraphicalComponent**, a Component that has a **Graphic** and which can be displayed in a Viewer. A Graphic is an object that contains graphics state and geometric information. It can draw itself and perform hit detection. Graphical components use Graphic in both their subjects and views to define their appearance. Because of the hierarchical structure of components, graphic also have parent and child access operations.

Graphic Subjects can also define **Mobility**, and the protocol adds **SetMobility** and **GetMobility** operations. The Mobility defines if and how the component can be **connected** to another component. It can have one of three values: **fixed**, **floating**, or **undefined**. In general, a fixed component's position cannot be affected by a connection, while a floating component will move to satisfy the connection's semantics. The concept of mobility is essential to the **Connector** Graphical Component, that allows Graphical Components to be tied together in a specific way. A connector can be connected to one or more connectors. Once connected, two connectors affect each other's position in specific ways as defined by the semantics of the connection.

The Connector subject adds the following operations to the Graphical Component's operations. **Connect** and **Disconnect** operations connects and disconnects the centers of two connectors. The connector's position can be found by the **GetCenter** operation. Connectors can also be bound to a StateVariable and they can have a specific transmission method. These are set and retrieved by the **SetBinding**, **GetBinding**, **SetTransMethod**, and **GetTransMethod** operations. A last operation called **Transmit**, transmits the state variable to all connectors connected to it. (The Transmit operation can be used to make simple simulations for example in schematic editors).

Graphical Views maintain their own Graphic to define their appearance. The View's Graphic is therefore independent of the Subject's Graphic.

The graphical view protocol adds **Highlight** and **Unhighlight** operations to mark component views selected by a **Selection** object. The Highlight operation lets the graphical view turn on its highlighted appearance and the Unhighlight allows reversion to its unhighlighted state.

The Graphical Views define two operations to define how they react when a Tool is manipulating them. This is done through a **Manipulator** object that abstracts and encapsulate the mechanics of direct

manipulation. The **CreateManipulator** operation creates a proper manipulator for a given Tool or event. When manipulation is done, the **InterpretManipulator** operation allows the graphical view to extract any significant information and returns a proper command that encapsulates and defines the effect of the manipulation.

#### 4.2.3 Commands

Commands can be used as messages to components, making them to react depending on the type of the command. They can also be (un)executed by the components or by other objects, by calling the command's (Un)Execute operation. Some commands may be directly accessible to the user through menus, while others are only used by the editor internally. In general, an undoable operation should be carried out by a command object.

The following basic operations are defined by the command protocol: **Execute** performs computation to carry out the command's semantics. **Unexecute** performs computations to reverse the effects of a previous Execute. A command is responsible for maintaining enough state to reverse one Execute operation. The **Store** operation allows a component to store information in the command that it can use later to reverse the effect in case of an Unexecute command. The component can retrieve the stored information with the **Recall** operation.

Commands that operate on selected components, must maintain a record over the component subjects they affect. Commands therefore store a **Clipboard** object that keeps a list of component subjects and provides operations for iterating trough that list. The Clipboard object can be assigned and retrieved with the **SetClipboard** and **GetClipboard** operations.

#### 4.2.4 Tools

Unidraw-based editors use Tool objects to allow the user to manipulate components directly. Conceptually, tools do their work within Viewers, in which graphical component views are displayed and manipulated.

Whenever a viewer receives an input event, it in turn asks the current tool to produce a Manipulator object. The tool's **CreateManipulator** operation either creates and initiates an appropriate manipulator, or the tool delegates the manipulator creation to one or more graphical views, to allow component-specific interaction. The **InterpretManipulator** operation creates a command that carries out the desired effect. Figure 3 shows how a tool is initiated from the viewer (1), and creates a manipulator affecting a view of a component (2). After finished manipulation the tool either interprets the manipulator itself, creating an appropriate command (3), or it asks the component view to interpret the manipulator and to create a command (4). The tool then passes the command to the viewer for execution (5), which often means some state modification of the edited component subject.

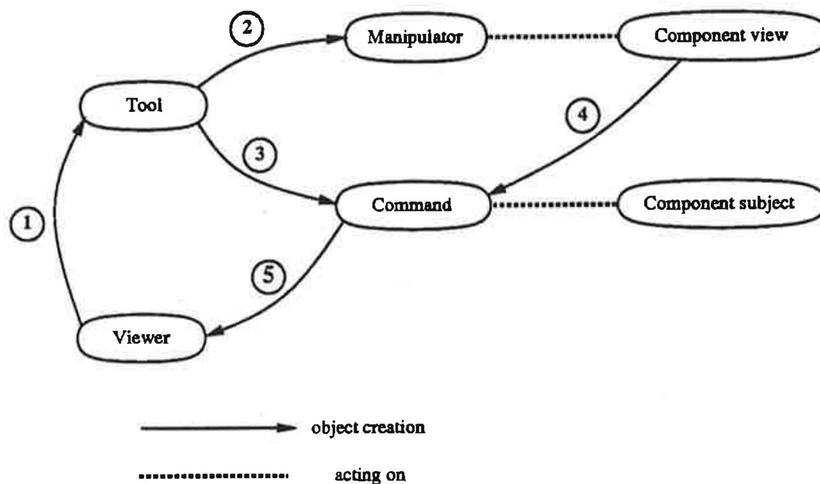



**Figure 3.** The manipulation of a component view

#### 4.2.5 External Representations

An external representation of a component is a non-graphical view of the subject. Domain-specific external representations are derived from the **ExternalView** objects, which in turn are derived from component view objects.

There are two operations defined for the external view protocol. **Emit** initiates external representation generation and calls the **Definition** operation recursively. Emit is used to generate header information that appears only once in the external representation, while Definition produces component-specific, context-independent information.

The architecture predefines **preorder, inorder,** and **postorder** external views each supporting one of three common traversals of the external view hierarchy.

#### 4.2.6 Application Framework

The protocols for Viewer, Editor, Unidraw and Catalog objects will be described here. These are the main objects that mediates communication between the component, tool and command objects.

Viewer

A **Viewer** displays a component view and provides an interface to scrolling and zooming it. Viewers also process user input events and are therefore implemented from window system or toolkit abstractions.

The Viewer's **Update** operation informs the viewer that some part of the graphical view has changed. The viewer can then use incremental techniques to redraw the graphic view. The **Handle** operation provides an interface to the corresponding window system or toolkit software for handling input events. For setting and retrieving the graphical view and editor respectively, the **SetGraphicView, GetGraphicView, SetEditor** and **GetEditor** operations can be used.

Editor

The **Editor** object provides a complete user interface for editing a graphical component subject. It composes one or more viewers with the commands and tools that can act upon the component and its subcomponents. The Editor can compose toolkit objects into a complete user interface. Each window in a specific editor is usually an instance of an Editor subclass designed for the purpose.

The Editor's **Update** operation makes Update on its viewer(s). **Open** and **Close** makes the Editor aware of its appearance and disappearance on the screen. The **GetState** operation provides a standard interface to accessing editor state variables. **SetComponent** and **GetComponent** respectively set and get the component subject that the user edits with the editor. **SetViewer** and **GetViewer** set and get the viewer(s) the editor owns. If there are multiple viewers, they all contains a view of the graphical component subject.

Operations for getting and setting the current tool and selection objects are defined as: **SetCurTool, GetCurTool, SetSelection** and **GetSelection.** How these operations are implemented depends on the application. If each editor has its own palette of tools, then each one must store the current tool. Otherwise they could share a global current tool selection.

Unidraw

A domain-specific editor application must create exactly one instance of a unidraw object. This object implements the main loop of the program: opens and closes the editors, logs all the commands that have been used and stores a reference to the one and only **Catalog.**

The Unidraw objects **Run** and **Quit** operations starts and terminates the main loop of the program. **Open** makes the given editor appear on the display, while **Close** removes the editor. Open and Close calls the corresponding operations on the editor. **Update** is an operation that is called to make internal state changes visible on the display.

The Unidraw object logs commands and can make an arbitrary-level undo and redo through the **Log, Undo,** and **Redo** operations. **SetHistoryLength** and **GetHistoryLength** operations sets and gets how many reversible commands should be logged.

The **SetCatalog** and **GetCatalog** operations set and get the catalogue appropriate for the application. SetCatalog should be called once by the Unidraw object when the application creates it.

Catalog

The Catalog provides independent name-to-object mappings for component subjects, commands, and tools. Name mappings are defined and undefined with the **Register and Unregister** operations, for each of the three base classes respectively. The **GetComponent, GetCommand,** and **GetTool** operations takes a name as an argument and returns the corresponding object, if any. The **GetName** operation makes the reverse mapping.

### 4.3 The Unidraw prototype implementation

There is a prototype Unidraw library developed to test the architecture. This prototype is released, but is still under development.

#### 4.3.1 C++

Because of the object-oriented design of Unidraw, it was natural to chose an object-oriented language for the implementation. The choice of C++ was made because of the earlier experiences with the language from building the user interface toolkit **InterViews.** C++ is an attractive language because it supports object-oriented programming without compromising the execution efficiency of C (see Appendix I, there is a description of object oriented programming and C++).

#### 4.3.2 InterViews

InterViews [7,15,16,17,23,24] was developed by the same research group that supported Vlissides in his development of Unidraw. It is an object-oriented toolkit developed in C++ that provides a library of predefined objects and a set of protocols for composing them. A complex user interface is easily created by composing simple primitives in a hierarchical fashion.

There are two main categories of object, each implemented as a hierarchy of object classes derived from a common base class. The two class hierarchies are:

1. Interactive objects such as buttons and dialogs, derived from the **Interactor** base class. Semantics for composing interactors are defined by the **Scene** subclass. Scene subclasses define specific composition semantics such as tiling or overlapping.
2. Structured graphics objects such as circles and splines, derived from the **Graphic** base class. The **Picture** subclass provides a common coordinate system and graphical context for composite graphical objects.

The Unidraw based domain-specific editors use objects from the InterViews toolkit to implement their basic user interface, and they use Unidraw objects to support graphical object editing.

The Unidraw prototype implements its own graphics library, which is merely a copy of the InterViews graphics library. There are operations added to make it possible to store the graphics state on file in a way that suits the Unidraw implementation. The copying of the whole graphics library could probably be avoided with the use of multiple inheritance.

#### 4.3.3 Structured Graphics

The Unidraw graphics library copies the InterViews graphic library with some extensions and includes abstractions for creating and manipulating graphical objects. The model used in the library is that of **structured graphics**, where geometric primitives can be assembled into hierarchies. Each primitive has some associated state that can be modified.

Structured graphics simplifies the implementation of graphical object editors because it can store the graphics state for the edited objects.

The library is structured with an object-oriented approach, where the graphics functionality is partitioned into a collection of classes that correspond to individual graphical primitives. The prototype Unidraw implementation relies completely on this library for representing graphics states in a domain-specific editor. All graphics maintain graphics state and information. Graphics state parameters are defined in separate base classes, including **Transformer**, **Color**, **Pattern**, **Brush**, and **Font**.

The graphics hierarchy traversal operations resembles the component hierarchy traversal operations in that they are called: **Parent**, **First**, **Last**, **Next**, **Prev**, **Done** and **GetGraphic**, returning the parent or a specific child graphics. All the child manipulating operations, use an object from the **Iterator** class, that iterates through the list of children.

The **Graphic** base class only have three states including transformer and foreground/background colours. Derived classes maintain additional graphics state according to their individual requirements. The **Picture** subclass composes other graphics into a single object. It maintains its own graphics state information, but does not have any geometric information.

In order to minimize the work required to redraw corrupted parts of a graphic, the library includes a **Damage** base class to support incremental update. This object is used to keep the appearance of graphics consistent with their representation.

#### 4.3.4 Components

The implementation of the Unidraw prototype in C++ made it necessary to add some operations to the Component protocols. C++ does not allow sending arbitrary messages to objects. Messages are sent via strongly-typed procedure calls, so a class must declare all acceptable messages at compile-time.

For example, a component operation such as Interpret accepts a message from a command. But in order to carry out the passed command, it has to determine the class of the command. C++ cannot provide this information at run-time. The implementation therefor adds **GetClassId** and **IsA** operations based on programmer-managed class identifiers for message identification.

Another case where the same problem emerges, is when a component subject wants to create a view from a given category, let it be a component view or an external view. The prototype predefines **COMPONENT VIEW** and **POSTSCRIPT VIEW** view categories, and extends the component subject protocol to include a **Create** operation. This operation takes the view category as an argument and returns the proper view.

The mapping between subjects, views, and view categories is defined by the **Creator** class object. Additional view categories are supported by deriving domain-specific creators.

Complex application objects are represented in Unidraw by the **GraphicComps** base class for composite graphical subjects and **GraphicViews** base class for composite graphical views. Each GraphicComps instance stores a picture into which each child's graphic is inserted. When the GraphicComps object interprets a graphical transformation or attribute-modifying command, it will only apply it to its picture, see figure 4. The command will affect the children's graphic automatically via the concatenation mechanism. Only the views associated with the GraphicComps object are notified, the concatenation mechanism takes care of propagation in the views as well.

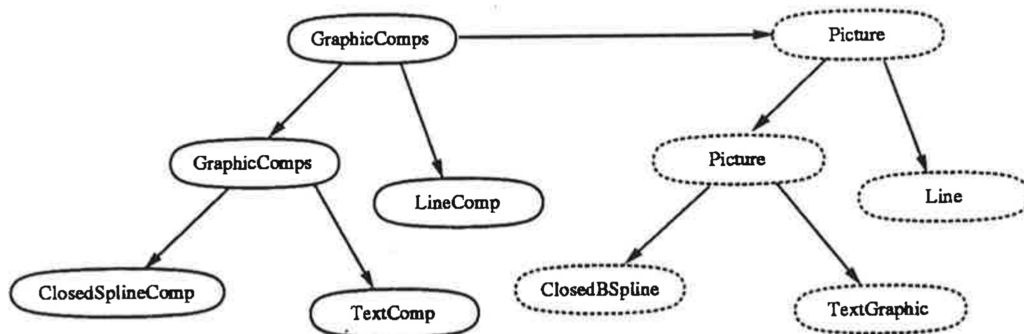



**Figure 4.** A GraphicComps and its Picture.

Both the subject and the view has a graphic, so the GraphicViews also have pictures. It depends on the implementation if the subject's or the view's graphic is used to draw itself. In figure 5 the view's graphic is used to draw the picture of a note.

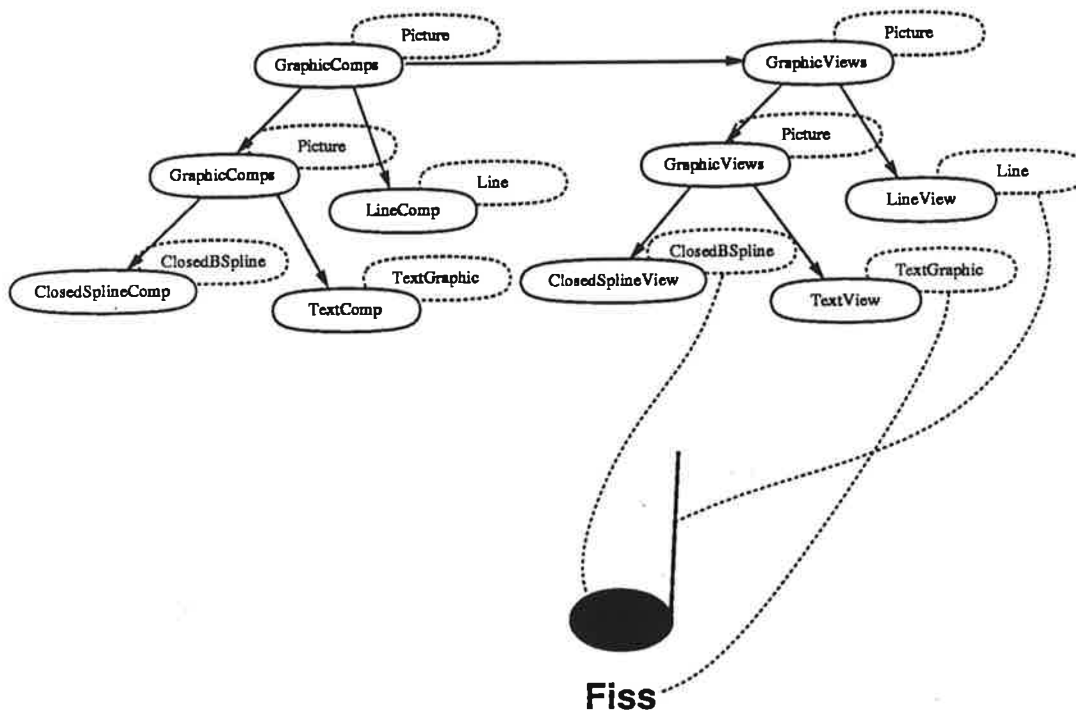



**Figure 5.** A subject and a view with their graphics.

In general, graphical application objects can use the subject's graphic to define their appearance, while more abstract application objects having no or more than one graphical representations, should use their view's graphic to define their appearance.

Implementors of Unidraw based editors should strive to define most domain-specific components through composition [12].

A component view must update its internal state to reflect its subject's state when its Update operation is called. Views with children must be prepared to restructure themselves to conform to their subject's child structure. Usually the subject's structure stays the same or changes only slightly, so an incremental approach in which the view reuses most of its children is preferable. The prototype implementation supports the common case when the view's child structure is identical to the subject's. Components with differing subject and view structures must implement their own Update algorithm.

#### 4.3.5 Data flow

The Unidraw architecture defines Connectors, state variables, and transfer functions in its data flow model. This data flow model allows state variables to be bound to the Connector components, and that the states of bound state variables could be passed between connected Connectors. Data flow is initiated when Transmit is called on a Connector.

The prototype implementation adds another class of object, called **Path**, to detect circularities. A Path maintains a record of connectors that have been **visited**, that is, connectors through which data has passed. The Path class defines two functions: **Visit** registers a connector with the path as having been visited, and **Visited** returns whether or not a connector has been visited.

#### 4.3.6 Commands and Tools

A Component cannot interpret a command without knowing the type of the command, and because of the strict type checking in C++, explicit type information is bound to the Commands in form of unique class identifiers. Tools also need the type information to let component views determine the proper manipulator to create for a given Tool. Therefore tools also have unique class identifiers.

The prototype library defines a **MacroCmd** command subclass that supports composition of command instances. When a MacroCmd is executed it simply executes its children.

While the library does not support compositions of tools themselves, it does allow manipulator composition via the **ManipGroup** subclass of manipulator. It provides a simple form of manipulator composition in which several manipulators can proceed virtually at the same time.

Only four other manipulators are predefined by the library: The **DragManip**, **VertexManip**, **ConnectManip**, and **TextManip** manipulator subclasses. A DragManip supports a downclick-drag-upclick style of interaction. The VertexManip is a DragManip that supports multiple downclick-and-drag interactions terminated by a distinguished downclick. The ConnectManip is also a DragManip and it adds a gravitational bias towards connector views. TextManip provides a text editing interface.

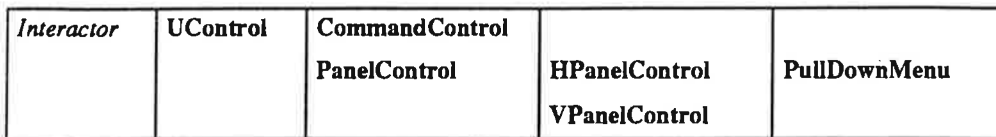
#### 4.3.7 Catalog

The Unidraw prototype maps component, command, and tool names to file names and stores a representation of the object in the corresponding file. Such objects must therefore define what information is written to and read from disk by defining **Read** and **Write** operations. Components can contain state variables and transfer functions, so these classes also define Read and Write operations.

The prototype extends the catalogue protocol to support reading and writing of **EditorInfo** objects, which store a list of strings or string tuples. Domain-specific editors can use this information to store information about what components, commands, and tools they incorporate in their interface. The user could then specify the configuration of the editor by editing the file that contains this information.

#### 4.3.8 User Interface

The prototype relies on InterViews for its user interface, either by letting the user derive Interviews objects directly, or by offering a mechanism for inserting specially derived control interactors in the editor object that gives an interface to the command and tool objects. These control interactors has a sensitive area where the name of the command or a graphical representation of the tool can be displayed. When clicking on the sensitive area, the corresponding command or tool will be activated. Figure 6 shows the control interactors predefined in the Unidraw prototype. The base class for this control mechanism is the **UControl** class, derived from the InterViews Interactor class.



| *Interactor* | **UControl** | **CommandControl** | | |
|---|---|---|---|---|
| | | **PanelControl** | **HPanelControl** | **PullDownMenu** |
| | | | **VPanelControl** | |

**Bold** ——— Unidraw predefined classes

*Italic* ——— InterViews predefined class

**Figure 6.** The Unidraw control interactor objects.

The **PanelControl**, derived from UControl, is the basic class for assembling a number of control interactors. The **HPanelControl** and the **VPanelControl**, derived from PanelControl, assemble control interactors horizontally and vertically respectively. A special control class, derived from HPanelControl, is the **PullDownMenu**, gathering **CommandControl** interactors together. CommandControl is derived from the UControl directly, and offers commands to be accessed through mouse interaction. The PanelControl classes manage tool interaction.

Here follows a piece of C++ code showing how the CommandControl could be used to make a Command interface.

```
void ModelEditor::Include (Command* cmd, PullDownMenu* pdm) {
    ControlInfo* ctrlInfo = cmd->GetControlInfo();
    UControl* ctrl = new CommandControl(ctrlInfo);
    _keymap->Register(ctrl);
    pdm->Include(ctrl);
    cmd->SetEditor(this);
}
```

**Figure 7.** Control interactor example.

The void Include function creates a CommandControl interactor to the command, and includes it in the PullDownMenu interactor. The control interactor is created via a special **ControlInfo** class, mediating information from the command to the Unidraw control interactor.

There is also an instance of the **KeyMap** class. The KeyMap maps key sequences to the control interactors, letting the user initiate commands or tools from keyboard interaction. Finally, the command is linked to the editor owning the Include function.

Figure 8 shows a PullDownMenu that have been activated. The different commands associated with the menu can be selected with the mouse, and are shown in reverse video. The letters on the right side are the corresponding key-codes.

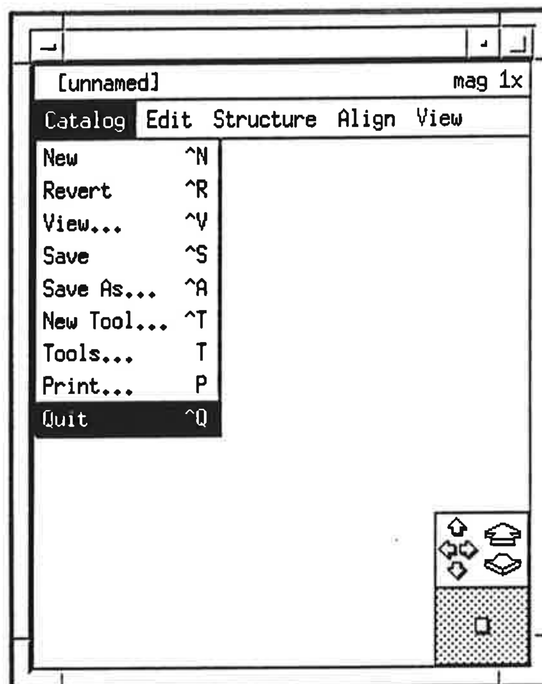



**Figure 8.** A PullDownMenu control interactor.

#### 4.4 Comments on Unidraw

The prototype editors, especially the schematic editor, turned out to be a good help in the understanding of Unidraw and how it can be used to build domain-specific editors. The absence of manuals for the Unidraw toolkit makes it difficult to design programs, because you always have to go back to the source code to find out what the predefined classes do and how their operations could be used.

Another thing is that Unidraw hardly can be used by itself without first learning how to use the Interactor part of the user interface toolkit InterViews.

There is a slight difference between the Unidraw philosophy and the prototype implementation in the division of the components in subjects and views. While the Unidraw architecture stresses the differences of a component's context independent state and its context dependent representation, the prototype implementation lets both the GraphicComp and the GraphicView have a Graphic. This would be all well if the components only represent graphical objects with a well defined appearance, but often object editors have to edit non-graphical objects too. These objects must have a graphical representation to let the editor manipulate them. Implementations of such non-graphical components should have their graphical state in the views.

In the Schem experimental editor (from the Unidraw snapshot), the implementation of a wire is not in accordance with the architecture's component separation into subject and view. The implemented wire defines its appearance in its subject by using the GraphicComp's Graphic instead of the GraphicView's Graphic.

## 5. SPECIFICATION OF AN EVALUATION EDITOR, 'ModelEditor'

The second part of my work consisted of the design and implementation of a small editor using the Unidraw prototype library as a framework. Such an implementation will facilitate the understanding of Unidraw's large number of classes. It would also give valuable information about how easy it was to use the Unidraw prototype.

After discussions with my supervisors, Bo Johansson (ABB/KLL) and Dag Brück (at the Department of Automatic Control at LTH), we found that a prototype of an editor capable of editing hierarchical models or function descriptions would be appropriate.

### 5.1 Related Work

Sven Erik Mattsson and Mats Anderson at the Department of Automatic Control LTH, have worked with a project developing tools for model development and simulation in the period July 1987 to June 1989 [9]. As part of this work they developed a prototype editor for building hierarchical models of control systems using Common Lisp and KEE. This editor creates an OMOLA representation of the model developed. OMOLA is an object-oriented modeling language developed at the Department of Automatic Control. [2]

### 5.2 The Model abstraction

A model abstraction suitable for representing hierarchical models can be structured in different ways. Figure 9 shows a structure where the model's different parts are stored in different variables (lists), making it possible to access the different parts of the model easily.

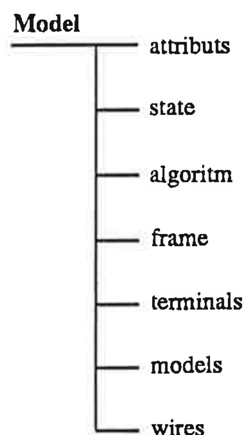



**Figure 9.** The model abstraction.

The **attributes** represents all unique features of a model, for example the name of the model. The **state** refers to model- or terminal-bound variables. The **algorithm** refers to the eventual explicit functions that define the relation between the state variables.

The **frame** represents the abstract border of the model, separating the model's interior from the outside. This frame is also where the model's terminals are bound. **Terminals** are the abstract points where internal states of the model can be accessed from the outside. Terminals can be of different type. In-terminals affect internal states of a model, while out-terminals shows internal states. In-out-type of

terminals both show and affect internal states. The **models** represent the submodels of the hierarchical model. The **wires** finally connects the model's terminals with its submodels' terminals, making a logical link between the connected terminals. The terminals of a model need not necessarily be connected to a submodel's terminal, but can be connected directly to another terminal of the model, or it may be not connected at all. The terminal could instead be bound to an internal state variable defined in the modelVar.

### 5.3 Editor specifications

To make the model easy to implement, the modelVar only has to contain the model's name, and the frame of the model is represented by a box. The different transmission directions must have different graphical representations. Unidirectional terminals, (in or out), could be represented by an arrow indicating the direction, and a bidirectional terminal could be represented by a double arrow. See figure 10. The wires connecting the terminals, can be represented by lines.

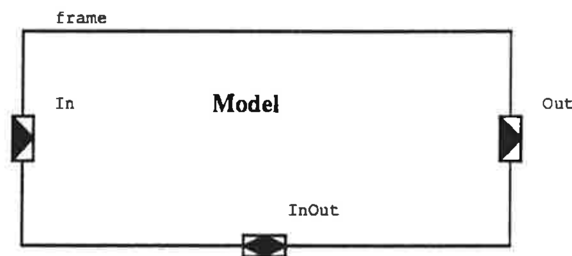



**Figure 10.** The terminal's transmission method.

The **ModelEditor** was meant to be a prototype of a software tool which would make it easier to build hierarchical model descriptions. The model abstraction described in the previous section defines the structure of the model and the graphical representations of the model's different parts. The user should be able to do the following editing on a model:

1. Supporting both top-down and bottom-up building of models creates the need of an empty model without any interior or any terminals.
2. It should be easy to create new terminals on the frame on the edited model, to insert submodels, and to connect the terminals with wires. It should only be possible to connect terminals that have logically corresponding transmission methods.
3. The interior of submodels should be examined when clicking on them with the mouse, by showing a view of the chosen submodel.
4. A second view of the models should be available, showing their hierarchical structure.
5. The user of the editor should be able to define tools from created models, making them easy to create when editing other models.
6. The edited model should be saved on file. Only the names of the included submodels and which of their terminals wires had been connected to, should be stored on file, not the submodels themselves.

7. Moving the terminal should only be possible around the frame, with graphic showing the transmission method. See figure 11.

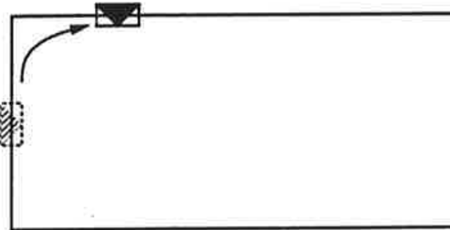


**Figure 11.** Movement of the Terminal with preserved transmission method.

Using the editor, models of quite complex tree structure can be made. The boxes represent models, the terminals externally available data, and the lines the connection between the model and its sub-models. See figure 12.

# 6. THE MODEL EDITOR PROTOTYPE IMPLEMENTATION

## 6.1 Getting started

We were not able to compile the Unidraw prototype library with the Oregon C++ compiler [10] , but had to wait until we got the C++ compiler from HP [5]. With the HP compiler, two of the Unidraw's three prototype editors, the drawing editor and the schematic editor, could be compiled. The User Interface Builder had bugs so we were not able to compile it. The prototype library and the the two compiled editors, seems to function all right except for a few bugs. It is mostly the storing to and retrieving information from files that causes the schem program to collapse.

I decided to first strip the Schem-editor from all those facilities that I did not want, and then build the model-editor by successively implementing the objects needed. The implementation therefore has the same structure as the schem editor, and lots of the classes are common.

As basic tools I decided to have only an empty model, a terminal, and the wire that the implementors of the Schem-editor already had developed. Following Vlissides instructions, I spent a lot of time developing the GraphicComponents representing my model and my terminal. I collected the model abstraction's attributes, state, and algorithm in one **modelVar** where momentarily only the model's name are present.

## 6.2 Prototype environment

The prototype was implemented in the object oriented language C++ using the Unidraw prototype C++ class library. The Unidraw library works on top of the InterViews library which in turn uses the X-Window system.

The operating system was unix, run on a HP 9000/300 workstation. The graphics output was presented on a 1280 x 1024 resolution graphics display.

In the middle of June, as HP just was finishing its new C++ compiler, we had the opportunity to join the beta-3 test program. The HP C++ Beta-3 compiler succeeded with the compilation of the Unidraw library, so the development could start.

For debugging, the HP xdb++ line-oriented debugger was used. The upper half of the terminal window displays the current source code, which is most useful when debugging. The debugger has a host of commands and works together with other languages. It supports most C++ constructs and is nice to work with.

## 6.3 Model Editor Classes

Figure 12 depicts the class hierarchy of the model editor prototype. This section gives a further description of the used classes together with their most important operations.

| | | | |
|---|---|---|---|
| Editor | ModelEditor<br>ToolPalette | | |
| GraphicComp | WireComp<br>**GraphicComps**<br>**Connector** | ModelComp<br>**PinComp** | TerminalComp |
| GraphicView | WireView<br>**GraphicViews**<br>**ConnectorView** | ModelView<br>**PinView** | TerminalView |
| PinGraphic | TerminalGraphic | InGraphic<br>InOutGraphic | |
| NameVar | ModelVar<br>TerminalVar | | |
| StateVarView | ModelVarView<br>TerminalVarview | | |
| Command | InfoCmd<br>NewViewCmd<br>ToolsCmd<br>**ViewCompCmd** | NewToolCmd | |
| BasicDialog | InfoDialog | | |
| Tool | ExamineTool | | |
| Manipulator | PopUpManip<br>**DragManip** | **VertexManip** | WireManip |
| *GrowingVertices* | RubberWire | | |
| Creator | ModelCreator | | |

**Bold** ——— Unidraw predefined classes

*Italic* ——— InterViews predefined class

**Figure 12.** The model editor classes.

### 6.3.1 Components

Model

The prototype implements a **ModelComp** subject, which is derived directly from the GraphicComps class. This object has a rectangle graphic as an argument to its constructor making it possible to make a frame to the model, which is necessary to avoid terminals to be placed anywhere.

The ModelComp has state variables in a variable of the ModelVar class. This ModelVar could be retrieved by the **GetModelVar** operation. The **TerminalCount** and **GetTerminalVar**, operations returns the number of terminals and the terminals' state variables as well. **GetFrame** returns the first

component of the submodel's children, which is a FrameComp if the constructor used had a rectangle as an in parameter. The ModelComp has a component view named **ModelView** having **GetModelComp** operation returning the subject.

Terminal

The **TerminalComp** and **TerminalView** classes, derived from the PinComp and PinView classes respectively, are copies of the Unidraw schem experimental editor's NodeComp and NodeView classes. The only differences are found in the NodeView's manipulator creation, and interpretation operations: **CreateManipulator**, **CreateGraphicCompManip**, **InterpretManipulator**, and **InterpGraphicCompManip**.

These are the operations responsible for defining where and how a terminal could be made and how it can be manipulated. The CreateManipulator defines the tools or events that could create a manipulator. In the case of a terminal, only the component creating tool and the examing tool creates a manipulator. The InterpGraphicCompManip makes it only possible to create a terminal on the model frame.

The **TerminalGraphic** class is derived from the PinGraphic class and defines the appearance of the terminal (for the moment only the uni-directional terminal).

Wire

The wire component, making it possible to connect model terminals, is equal to the Unidraw schem experimental editor's wire component.

### 6.3.2 Commands

The model editor commands is actually a subset of the Unidraw schem experimental editor's command classes, with only a few changes.

All of the command classes have got the **Copy**, **GetClassId**, and **IsA** operations, and at least one constructor. Other important operations and the functionality will be described for each command.

The **NewToolCmd** class makes it possible for the user to insert a new tool in the tools palette. It is an unreversible command making its **Reversible** operation return false. **Execute** inserts a FileChoser (InterViews class) in the editor, letting the user chose a (model) file from which a tool could be made. A model component with the given name is retrieved from the Catalog, (if it does not exist, it is read from file), serving as a template when a ControlInfo object is created using the **CreateCtrlInfo** private operation. In the original (schem) operation, control information from the entire component including its children, was made. This was changed so that only the model component's state, frame and terminals result in control information. The component serving as a template when the new tool is created, is also stripped from the interior wires and models. After creation, the tool is inserted in the tool palette.

The **NewToolCmd** command class was not changed at all when copied from the schem experimental editor. This command's **Execute** operation lets the user chose between already created tools, and install them or remove them from the tool palette via a CatalogChoser object inserted in the editor. The command is unreversible.

**InfoCmd** is a command making it possible to view and modify model or terminal information. Its **Execute** operation calls the private **Modify** operation if the selected component is a model or a terminal. Modify creates an InfoDialog and shows the model's or terminal's current state. This state is currently the model and terminal names, and the terminal transmission method.

#### 6.3.3 Tools

Besides the predefined Unidraw tools, there is only one derived tool class used. It is the **ExamineTool**, making it possible to point at a terminal or a model, and then see its internal state. The ExamineTool's **CreateManipulator** simply creates a PopUpMenu where the user can chose an appropriate command. (For the moment only one, the InfoCmd. The tool is a reduced version of the Unidraw schem editor's ExamineTool which offered the user to chose between more commands.)

#### 6.3.4 Editors

There are two editor classes in the model editor. Both are modified versions of the schem editor's SchemEditor class and ToolPalette class.

**ModelEditor**, is a copy of schem's SchemEditor class with only the class name and internal name changed, and a reduction of the number of commands. The ModelEditor is where the actual editing takes place. It is composed in a common way of InterViews Interactor subclasses like Tray, Boxes, and Borders. Uppermost, the name of the model being edited and the current magnification are shown. Underneath, separated with a border, a PullDownMenuBar interactor object lets the user chose between numerous commands. These commands are accessible through the Catalog, Edit, Structure, Align, and View menu items. Clicking on one of the items, a PullDownMenu interactor with related commands will be shown. Then comes the Viewer in which the editing takes place. It is separated from the PullDownMenuBar with a border. In the right bottom corner, a Panner interactor is inserted making it possible to zoom and pan the viewer. See figure 13.

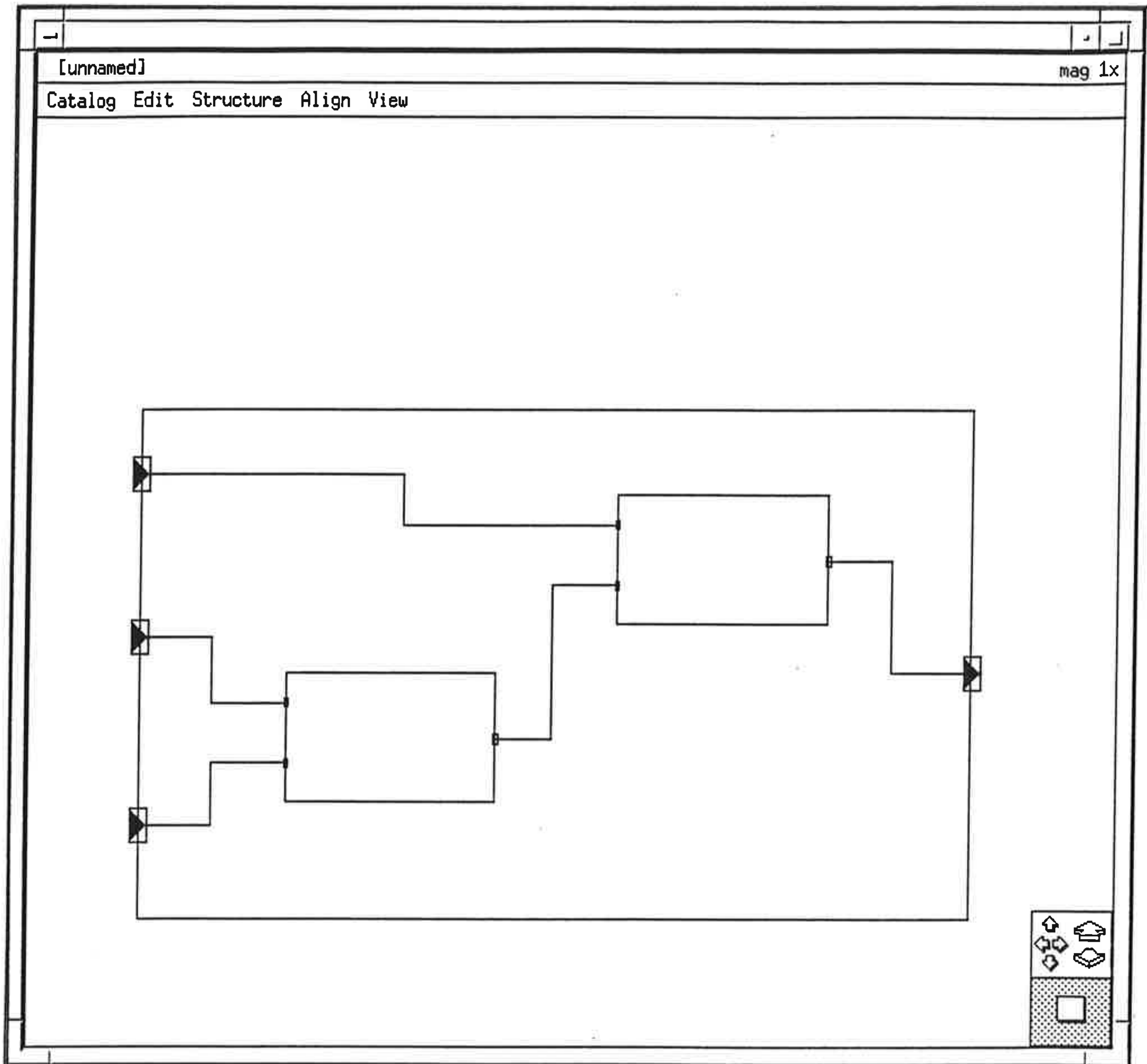



**Figure 13.** The ModelEditor with a created model.

The second editor class, the **ToolPalette**, is a copy of schem's ToolPalette, with the interior changed. There are two categories of tools in the palette. On the left, there are different manipulating tools like move, scale, stretch, and examine. On the right side there are different components serving as templates when a new component tool creates a component in the viewer with the palette component as a template. In Figure 14 a model component have been inserted as a tool. It can be used when hierarchical models are created.

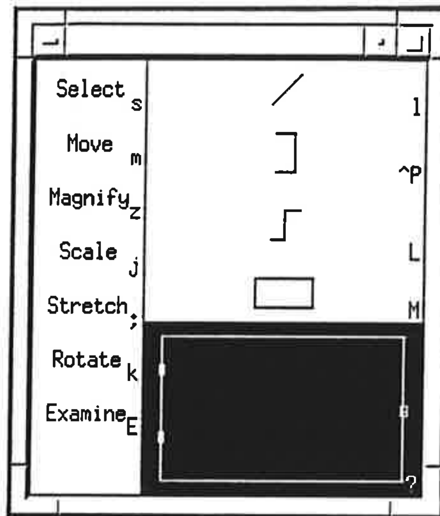



**Figure 14.** The ToolPalette with a selected model tool.

Besides all the operations setting and retrieving information such as key map, button state and selection, there is an operation named **InstallOrRemove** that inserts or removes a tool from the tool palette. The operation takes a tool name as an argument, and retrieves the tool from the Catalog, before inserting it in the palette. The tool in question is actually a NewCompTool (Unidraw) that creates a new component with the component in the palette as a template.

#### 6.3.5 Other Classes

A **PopUpManip** derived from the Manipulator class, shows a tool's PopUpMenu in the viewer. The **InfoDialog** derived from the BasicDialog InterViews class, are used by the InfoCmd command to show the model's or terminal's state.

A **ModelCreator** class is derived from the Unidraw Creator class in order to manage the creation of the unique model editor classes.

The special effects during the creation of the wire, are managed by the **RubberWire** class derived from the Unidraw GrowingVertices class. A **WireManip** derived from the VertexManip class, takes a RubberWire as an argument and manipulates it in the viewer.

The **ModelVar** and **TerminalVar** classes both derived from the NameVar Unidraw class, are copies of the schem editor's ElementVar and NodeVar respectively. They represent the model's and terminal's variables. The corresponding **ModelVarView** and **TerminalVarView** classes make it possible to show the variables, and even change them, in the viewer. They are both derived from the Unidraw StateVarView class.

### 6.4 Running the editor

The model editor must be run under the XWindow system. Before you start the model editor program,

remove the **.modelrc** file, if any, from the current directory. This file contains the names of files with model tools, but a bug in the model editor program does not allow that tools are initiated when starting the program.

After the model editor has been started, two windows appear on the screen. The first one is the tool palette and the second (and larger one) is the model editor itself. There is no model visible in the viewer, so you can start by clicking on the **Catalog** menu above the viewer, and releasing the mouse button at the **New** item. After this, an empty model is visible in the viewer, with only its frame present.

There is a "^N" to the right of the New item in the menu. This is the key code for the New (model) command (control n). Commands and tools having a key-code to the right, can be initiated from the keyboard as well as of the mouse.

At start, the tool palette's **Select** tool is active. By clicking on the desired tool, or by giving the corresponding key code, another tool can be selected. Click on the **Terminal** tool and you can place terminals on the frame of your model.

The name of the model can be changed by selecting the **Examine** tool, and clicking on the model's frame. A popup menu will appear and you can choose the **info** item. In the string editor that shows next, the name of the model can be altered. If the model has any terminals, their name and transmission method can be altered too.

Other commands under the **Edit, Structure, Align** and **View** items on the pull down menu bar, offer different facilities for the editing of a model.

By clicking on the Edit item, different editing commands can be executed. **Undo** and **Redo**, reverse executes the latest undoable operation and reexecutes the operation respectively. The **Cut, Copy, Paste, Duplicate,** and **Delete** commands, are the real cut-and-paste operations of the editor. **Select all**, puts all the present components in the Selection object. The following flip and rotate commands only affect the model components. **Fix** and **Floating** sets the Terminal components' Mobility attribute.

Structure commands should not be used when models are edited, because they collect the selected components in a GraphicComps component.

Under the Align item, commands for different alignment between components are found.

The View item hides the viewing commands **Normal Size, Reduce to Fit,** and **Center Page.The New View** and **Close View** commands, offer views of submodels.

Under the Catalog item, commands that uses the catalogue are found. These are commands for saving, viewing and retrieving a model, and commands that install models in the tool palette. The **Quit** command closes the editor.

### 6.5 Suggestions to improvements

The current model editor prototype is far from complete, and lots of improvements could be done. Besides the fact that the model editor specifications have not been attained, bug fixes and several functionality improvements could be done. Here follows a list of suggested functional improvements:

1. The model editor only implements the "bottom-up" modelling strategy. A "top down" modelling strategy could be added, where empty models could be inserted in a model with only their names and terminals present. These empty models could then be edited at a later stage.
2. An additional component view could show the tree structure of the edited model. This view could also be manipulated, for example with the **Examine** tool, making it possible to chose a submodel to view.

3. External views of the edited model should be implemented. Post-Script views showing the model with its submodels, and the tree structure of the model. Additional external views could produce some form of code suitable for model simulation.
4. Add the possibility to include whole model libraries in possibly more than one tool palette.
5. Make another alias view of the models showing their "natural" appearance. For example the picture of a motor, for a motor model.

## 7. CONCLUDING REMARKS

The Unidraw architecture seems to be a well thought out and even the early snapshot we had was functioning. There has been another improved snapshot released at the end of April from Stanford University, but we still do not have it. It is available by anonymous FTP from: **interviews.stanford.edu in pub/ud-snapshot-0.4.tar.Z.** John Vlissides' Email address is: **<vlis@lurch.Stanford.edu>**.

### 7.1 Understanding Unidraw

The philosophy of the Unidraw architecture is not so hard to understand but when it comes to its prototype implementation it is easy to drown in the large number of predefined classes and all their various operations. The fact that there are no manuals for the prototype, makes it even more difficult.

It was a real relief when we got Vlissides' doctoral dissertation draft, where he describes both the Unidraw philosophy and its prototype implementation in greater detail. Before that I had to extract information from the C++ source code to find out how the objects really worked. I used the awk [1] language to monitor the class hierarchy of the prototype, and to find the dynamic bindings between the classes.

At the end of my master's thesis, my supervisor Dag provided a program that made a graph of the inheritance between the Unidraw classes. The program is called **dag**. The Unidraw prototype class hierarchy is found in appendix II.

The Unidraw architecture is well designed with its division of the objects from the graphical based object editors, into the component, command, tool abstractions. The division of the components into subject and view is also natural. But the external view basic abstraction, could be thought of as an ordinary component view. A view does not necessarily have to be graphical, does it? In the Unidraw prototype the external representations are actually made as component views.

What puzzles me in the Unidraw prototype, is that both the component subject and the component view are supplied with a graphic. Is that really necessary? Pure graphical objects maybe could have their graphic in the subject, but in my opinion they could have it in their view(s) as well. I think it would be better if only the views, or more precisely the graphical views, had a graphic showing their subject's context dependent appearance. It would facilitate the implementation of multiple component views.

Unidraw makes it easier to build graphical object editors, but maybe not graphical based object editors.

### 7.2 The ModelEditor prototype

The development of the model editor prototype was a good help in the understanding of how the different Unidraw prototype classes worked together. The stripping of the Unidraw prototype schem experimental editor was not as easy as I originally expected because of the many dependencies between the Unidraw classes. Some of the classes in the Unidraw prototype are not even mentioned in the existing Unidraw documentation (like the control interactor classes), so these had to be understood from the source code directly.

Maybe it would have been better if the editor had been developed entirely from scratch, and not by using the schem editor as a foundation, but this was a way to understand Unidraw at the same time as the model editor implementation.

### 7.3 The usability of Unidraw

The Unidraw architecture is well designed, by separating the features of a graphical object editor in its main abstractions. The prototype implementation on the contrary, is not so well structured and could be improved. The usability of the Unidraw prototype we had at hand, could be summed up with the following items:

- There is a first big step/obstacle in learning Unidraw, because it is a big library, and there are no manuals available. The underlying user interface toolkit InterViews, has to be learned first because Unidraw relies on InterViews classes for its user interface. This implies that Unidraw can be useful in larger projects, where not only one but groups of editors have to be developed and implemented.
- The adding of a graphics to both the GraphicComp's subject and view, is consequently not following the Unidraw architecture's ideas about separating the context-independent state and the context-dependent representation of a component to its subject and view respectively. This part of the implementation could be rewritten or updated. The Unidraw prototype can be useful as an existing tool, or it can be a source of ideas if a new graphical object editor framework is developed.
- The prototype is based on the InterViews user interface, which makes all implemented editors to have the same look-and-feel. This is good when groups of editors are developed, so the user of the editor can switch between the editors without consulting the editor manuals or any "Help" facility.

## 8. REFERENCES

### BOOKS, DRAFTS, DISSERTATIONS, MANUALS


[1] **Aho A., Kernighan B., Weinberger P. [1988]: The AWK programming language.** Addison-Wesley. The AWK language is well suited to write short manipulation programs. Input, field splitting, storage management and, initiation are all performed automatically.

[2] **Andersson M. [1990]: Omola - An Object-Oriented Language for ModelRepresentation,** Department of Automatic Control, Lund Institute of Technology, Lund, Sweden. Describes the basic requirements on a new, object-oriented language for representing systems occurring in control and process engineering. Sponsored by the National Swedish Board of Technical Development (STU).

[3] **Dewhurst Stephen C., Stark Kathy T. [1989]: Programming in C++** PRENTICE HALL

[4] **Ellis M., Stroustrup B. [1990]: The Annotated C++ Reference Manual** Addison-Wesley

[5] **Hewlett-Packard Company [1990]: HP C++ Beta-3 release documentation**, April 1990. This is a comprehensive set of documents describing the HP implementation of C++. A number of papers written by C++' developer(s) are included, which is a very nice touch.

[6] **Kernighan Brian W., Ritchie Dennis M. [1978]: The C Programming Language** PRENTICE HALL

[7] **Linton M., Calder P., Vlissides J. [1990]: The InterViews UserInterface Toolkit**, draft. This (incomplete) draft is for a book on the subject of programming with InterViews.

[8] **Lippman S. B. [1989]: C++ Primer**, Addison-Wesley. A nice introduction to the C++ language by one of the workers at AT&T Bell Laboratories. The latter chapters need to be read slowly and carefully, since there are a lot of details to digest.

[9] **Mattsson Sven Erik (Editor) [1989]: New Tools for ModelDevelopment and Simulation.** Proceedings of a full-day seminar Stockholm, October 24, 1989. Department of Automatic Control, LTH.

[10] **Oregon Software company [1989]: OREGON C++, Version 1.2 for UNIX,User Manual** Describes the Oregon C++ compiler.

[11] **Stroustrup B. [1987]: The C++ Programming Language.** THE reference book on the C++ language. Published in 1987, it doesn't cover the latest extensions to the language. See Ellis & Stroustrup for a later version.

[12] **Vlissides J. [1990]: Generalized Graphical Object Editing,** Doctoral dissertation. Describes Unidraw, which is a framework for creating object-oriented, domain-specific editors. The Unidraw architecture uses programming abstractions that are common to many applications domains, thus making the programmer's life easier.


### PAPERS


[13] **Coplien J. et. al.: C++: Evolving Towards A More Powerful Language.** AT&T Bell Laboratories. Shows how and why object-oriented programming is a powerful tool in coping with complexity.

[14] **Koenig A., and Stroustrup B. [1989]: C++: as close as possible to C – but no closer.** AT&T Bell Laboratories. Describes the differences between the ANSI C standard and C++.

[15] **Linton M. et. al. [1988]: InterViews-2.6 documentation files.** These libraries are available from the Stanford University. They are also included in the HP C++ compiler distribution.

[16] **Linton M. et. al. [1988]: Composing User Interfaces with InterViews.** Stanford University. A short introduction to the basics of InterViews. Among other things, you learn about things such as HBox, VBox, and Glue...

[17] **Linton M. et. al. [1988]: InterViews: A C++ Graphical Interface Toolkit.** Stanford University. Gives a short overview of the InterViews concepts and comments on the use of X windows in the implementation.

[18] **Lippman S., and Moo B. [1988]: C++: From Research to Practice.** USENIX C++ Conference, October 17-20 1988, Denver, CO, USA. An overview of the factors that make C++ a useful language.

[19] **Pazel Donald P. [1989]: DS-Viewer An interactive graphical datastructure presentation facility.** IBM Systems Journal. vol 28, no 2, 1989.

[20] **Rosengren P. [1988]: Naturliga objekt ar bast.** (swedish) Datateknik 1988:10. Try to find real-world objects as templates for your abstract objects.

[21] **Stroustrup B. [1989]: Should C++ follow C's footsteps?** AT&T Bell Laboratories. The father of C++, Bjarne Stroustrup, gives a few philosophical aspects on the evolution of C++.

[22] **Thomas D. [1989]: In Search of an Object-Oriented Development Process,** Journal of Object-Oriented Programming, May/June 1989. Discusses the differences between the object-oriented and the structured development model.

[23] **Vlissides J. [1990]: A Tutorial for InterViews Programmers, Part I.** Stanford University. Describes how to write a simple "Hello, world" type program using some of the InterViews classes.

[24] **Vlissides J., Linton M. [1988]: Applying Object-Oriented Design toStructured Graphics.** Stanford University. A comparison between the Graphic library in InterViews and an earlier graphics library written in Modula-2. The advantages of using an object based model are stressed by the authors. Recommended.

[25] **Vlissides J., Linton M. [1989]: Unidraw: A Framework for BuildingDomain-Specific Graphical Editors** Stanford University. Makes an introduction to the Unidraw architecture defining its main goals and abstractions.

[26] **Wilson David A. [1990]: Class Diagrams: A tool for design,documentation and teaching.** Journal of Object-Oriented Programming, January/February 1990.

## 9. APPENDIX I Object-oriented languages

This appendix was written by Nick Hoggard here at KLL, and it describes the fundamentals of object oriented programming and compares different OOP languages:

The section starts with a summary of the concepts of object-oriented languages, followed by an explanation of their benefits, and a survey of languages. Finally there is an in-depth look at C++ based on KLL's experiences under 1989.

### 9.1 Concepts in object-oriented languages

— Object

An object is a combination of data and procedures that operate on that data.

— Data Encapsulation

This means that an object's data can only be accessed by the object's own procedures.

— Methods and Messages

Methods and Messages are roughly equivalent to Procedures and Procedure Calls in conventional languages. A procedure that belongs to an object is called a Method. One sends a Message to a Method instead of calling it.

— Classes and Instances

Classes and Instances are roughly equivalent to Types and Variables in conventional languages. The Class definition of a object defines the data and procedures that belong to the object. To use an object, one has to create Instances of it, each with an area of memory allocated to it.

— Inheritance

One can define new classes by taking a class that is already defined and adding new data and procedures to it. The new class automatically inherits the data and procedures of the original class.

— Dynamic Binding (also called Late or Delayed Binding, or Polymorphism)

This means that you can define a new class based on an existing class and not only add new data and procedures but actually replace some of the existing procedures with different procedures having the same name and parameters.

— Multiple Inheritance

Multiple inheritance means that a new class may inherit from more than one existing class.

Of the above concepts, the three that are considered essential for a language to be called object-oriented are Data Abstraction, Inheritance, and Dynamic Binding.

Data Abstraction and Inheritance mean that it is easier to predict the effects of changing an object without having to look at the code that uses the object. This makes code easier to change.

Dynamic Binding allows code to be re-used in a completely new way. Take, for example, list handling, which can be implemented quite easily in normal languages, but has to be re-implemented for each type of data you want to put in a list. Dynamic binding lets you write a single list-handling package which can then be re-used for many different classes of object, even mixing different classes of object in the same list, and allowing you to treat the list as if it contained only one class of object. For example, you can print a list of mixed objects by simply writing the equivalent of "REPEAT print

object UNTIL end of list", provided only that you have defined a print procedure for each object. The dynamic binding feature sorts out at run-time which print procedure is to be called for each object.

### 9.2 The benefits of object-oriented languages

Object-oriented languages produce code that is easier to debug, easier to modify, and much more reliable than conventional languages. Experience of using object-oriented languages on real projects has shown that the design phase takes longer, but that coding is quicker, and software integration time is drastically reduced. A lot of the design phase is concerned with defining objects and their interfaces. At the end of the design phase, one has a lot of empty objects, like empty shells, ready to be filled with code. In this sense, object-oriented languages work both as a program design language and an implementation language. Using the same language, relatively experienced staff can design the objects, and relatively inexperienced people can implement them.

Some critics say that object-oriented languages are slower than conventional languages. In fact the situation is much the same as when block-structured languages (i.e. with BEGIN...END) such as Pascal appeared: the loss in performance is insignificant and the benefits are enormous.

### 9.3 Languages designed to be Object-Oriented

These languages are designed from scratch as object-oriented languages. None of them are likely to become widely used because the industry is very conservative about adopting new languages.

— Smalltalk

Smalltalk has a good interactive programming environment and is sometimes used for prototyping. However, it is generally thought unsuited to the "real world" of commercial products. It is difficult to become proficient in Smalltalk because it only allows pure object-oriented programming - it does not allow one to program "normally" or use "quick and dirty" solutions. One has to think out the program thoroughly first.

— Eiffel

This is a new object-oriented language that has been highly acclaimed. Eiffel emphasizes program correctness and encourages the programmer to check values of parameters when calling and returning from procedures.

### 9.4 Object-oriented versions of conventional languages

These languages are modified versions of conventional languages. They are already so successful that they will almost certain replace today's languages. They have two major advantages:

1. They can be linked to the well-developed and tested libraries of code that already exist in the language, such as interfaces to operating systems, window systems, etc.
2. They exploit the existing knowledge of the language among programmers. The programmers can use as much or as little of the object-oriented features as they want, so that they can gradually learn the new features of the language but still be productive during that period.

The most important languages are as follows:

— Object-oriented Lisp (various dialects)

Object-oriented Lisp is of interest to people who want to program in Lisp, but not to anyone else.

— Object-oriented Pascal (various dialects)

Object-oriented Pascal is likely to become widely used in industry because of the number of people who use Pascal. As yet there is no single dominating product or standard, but a number of variants are starting to appear on the market, such as Apple Object Pascal for Macintosh, and Borland Turbo Pascal 5.5 and Microsoft Quick Pascal 1.0 for DOS.

— Object-oriented C (Objective-C, C++)

Technically, object-oriented C has a major disadvantage - it is based on C, which is medium-level language, somewhere in between assembler and Pascal. However C is widely used in industry, and for this reason object-oriented C is likely to become a major language.

### 9.5 Other languages

No completely new object-oriented language is likely to become widely-used. Object-oriented versions of existing languages will. It is only a matter of time before object-oriented Fortran, Basic and Cobol appear. The example of C++ shows that object-oriented versions of languages can be designed to be nearly 100% compatible with the original language and just as fast.

### 9.6 KLL's experiences of C++

KLL has under 1989 had experience of C++ in two projects, and the impression is very favourable. C++ is particularly ingenious because it is designed in such a way that C++ programs can always be made as fast as, or even faster than, C programs, and so no project manager is likely to lose his job by deciding to use it.

C++ can be faster than C because of the possibility of inline-expansion of functions. The speed of most programs is limited by the speed of a very small percentage of the functions in the program. Once these bottlenecks are identified they can usually be speeded up considerably by inline expansion.

Most C++ compilers are based on the C pre-processor from AT&T which produces C code, which in turn has to be compiled by a C compiler. The C source code is often up to 7 times larger than the C++ source code. The C compiler can have problems compiling the source code if memory is limited.

There are some C++ compilers available which compile to binary code directly and these have smaller memory requirements and compile faster. The resulting code is likely to be better too because the possibilities for compiler optimizing are better.

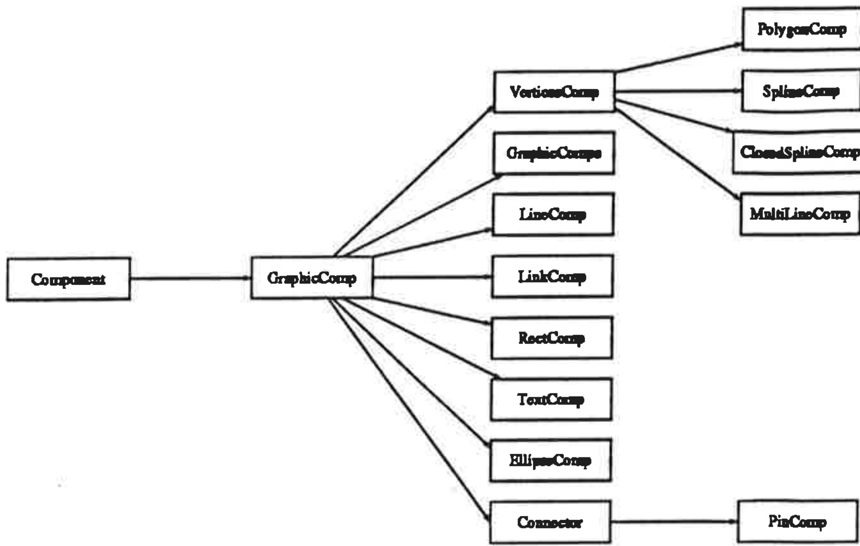
Up until now there has been no source-code debugger available for C++, but some are starting to appear now.

AT&T have now released the source code to C++ version 2.0 and various companies are starting to sell binary versions. The most important new feature in version 2.0 is multiple inheritance, which means that a class can now be derived from two or more base classes, inheriting all of the variables and methods of the base classes. This is a significant improvement.

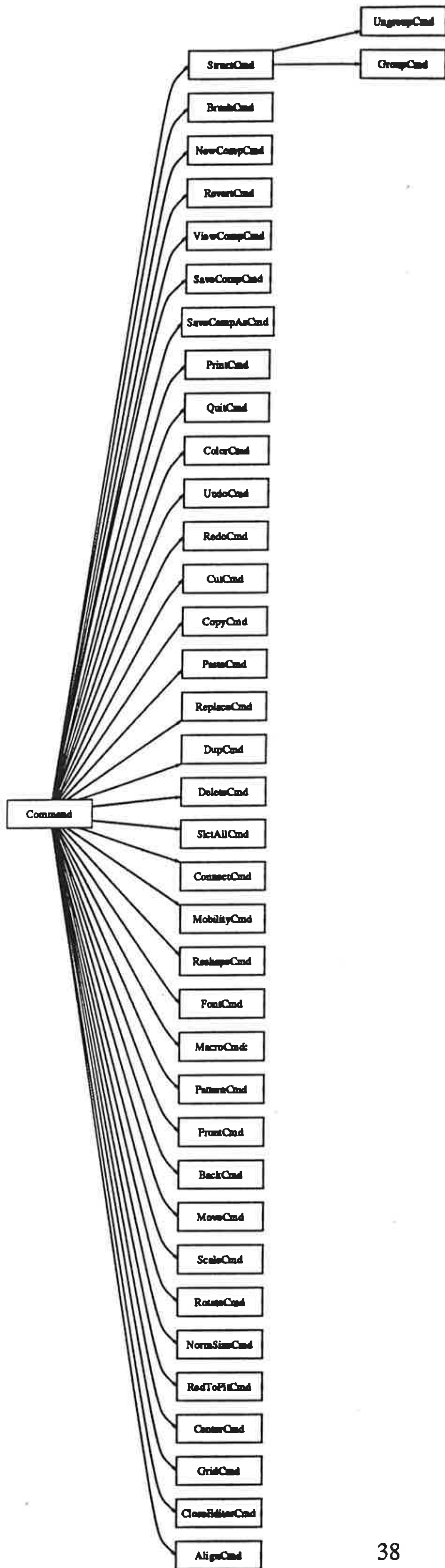
C++ is now the dominant C-based object-oriented language on the market and is likely to be the first object-oriented language to be widely accepted in the industrial world.

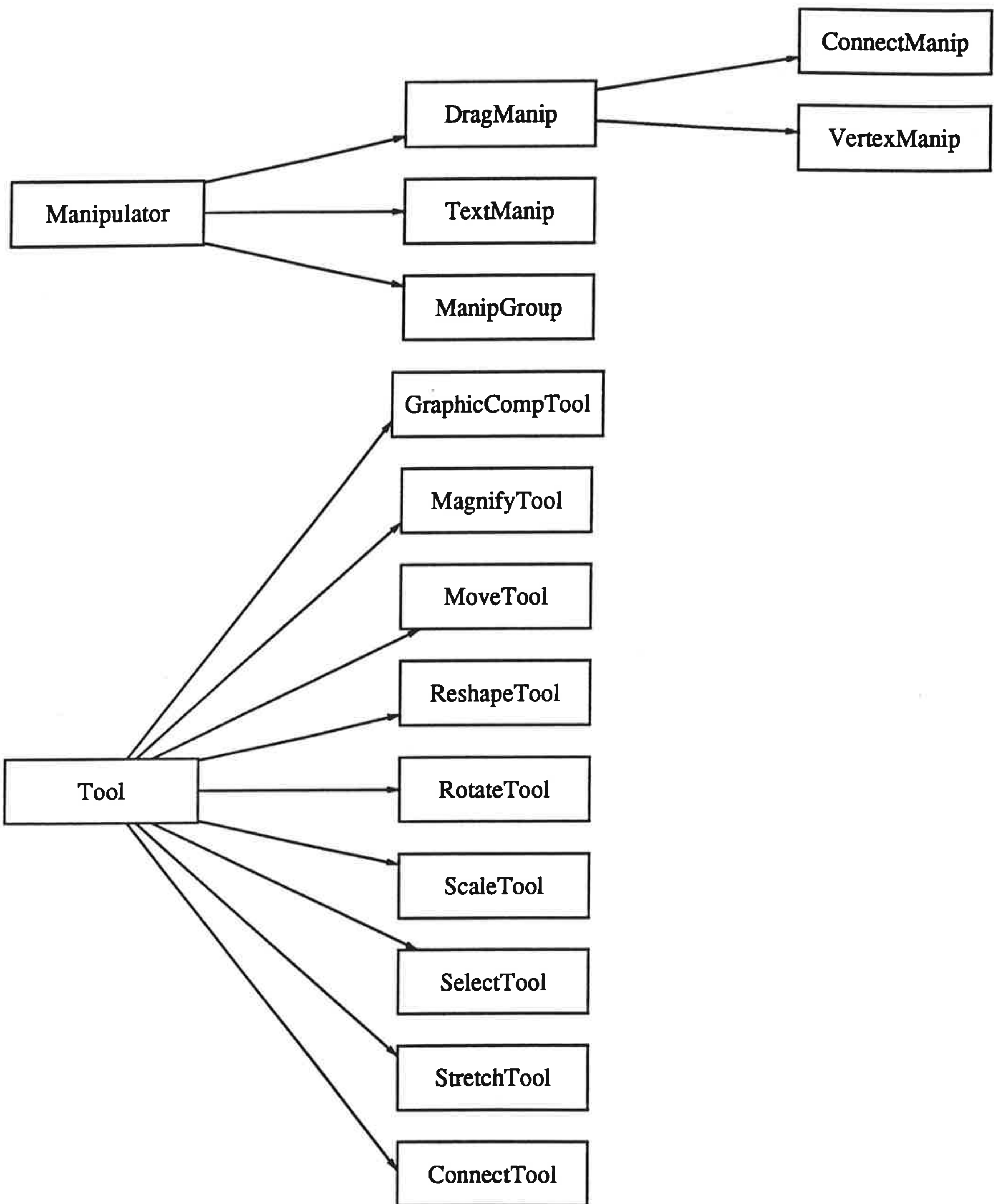
### 10. APPENDIX II The Unidraw prototype class hierarchy

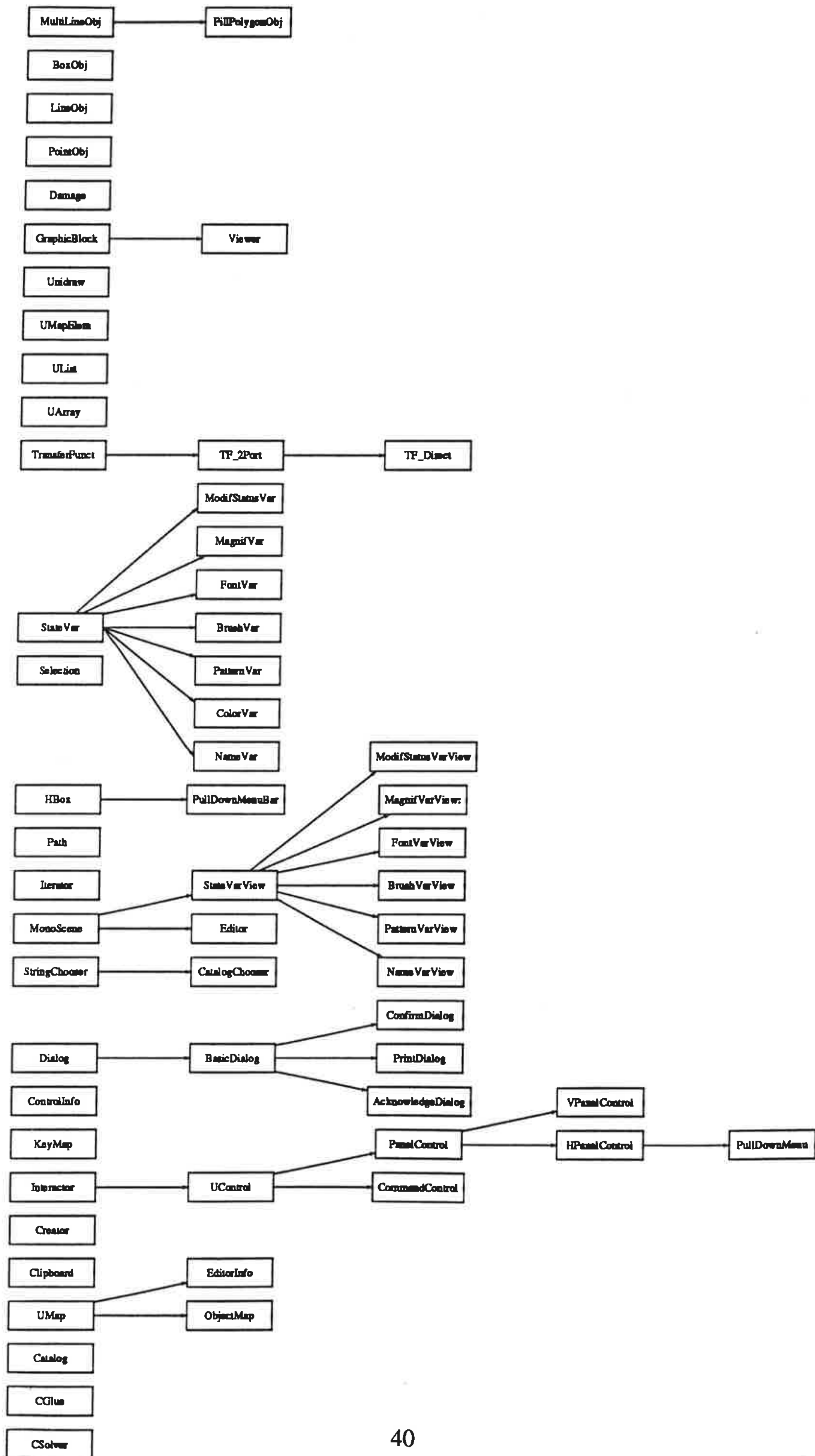
Here comes a graph of the unidraw prototype class hierarchy, showing predefined InterViews classes with ellipses. Then follows a list of the same class hierarchy but in alphabetic order. Both in normal order, the main class at the left, and in reverse order, the subclasses being at the left.

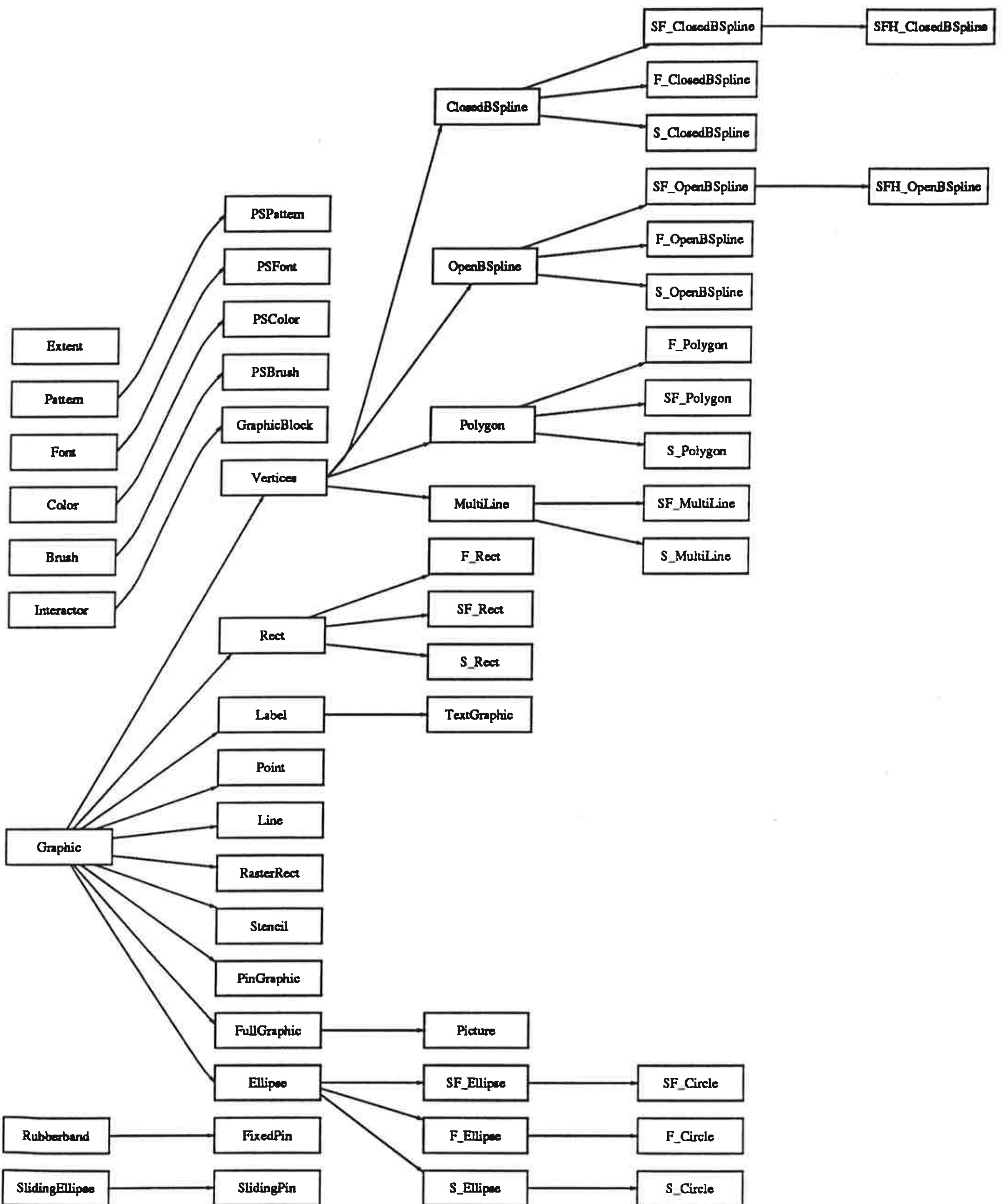



PolygonView
VerticesView
SplineView
GraphicViews
ClosedSplineView
LineView
MultiLineView
LinkView
GraphicView
RectView
TextView
EllipseView
PSPolygon
ConnectorView
PinView
PSVertices
PSSpline
PostorderView
PSLine
PSClosedSpline
ComponentView
ExternView
InorderView
PSLink
PSMultiLine
PreorderView
PostScriptView
PSPin
PostScriptViews
PSRect
PSText
PSEllipse


UngroupCmd
StructCmd
GroupCmd
BrushCmd
NewCompCmd
RevertCmd
ViewCompCmd
SaveCompCmd
SaveCompAsCmd
PrintCmd
QuitCmd
ColorCmd
UndoCmd
RedoCmd
CutCmd
CopyCmd
PasteCmd
ReplaceCmd
DupCmd
DeleteCmd
Command
SlctAllCmd
ConnectCmd
MobilityCmd
ReshapeCmd
FontCmd
MacroCmd
PatternCmd
FrontCmd
BackCmd
MoveCmd
ScaleCmd
RotateCmd
NormSizeCmd
RedToFitCmd
CenterCmd
GridCmd
CloseEditorCmd
AlignCmd


ConnectManip
DragManip
VertexManip
Manipulator
TextManip
ManipGroup
GraphicCompTool
MagnifyTool
MoveTool
ReshapeTool
Tool
RotateTool
ScaleTool
SelectTool
StretchTool
ConnectTool


MultiLineObj
FillPolygonObj
BoxObj
LineObj
PointObj
Damage
GraphicBlock
Viewer
Unidraw
UMapElem
UList
UArray
TransferFunct
TF_2Port
TF_Direct
ModifStatusVar
MagnifVar
FontVar
StateVar
BrushVar
Selection
PatternVar
ColorVar
NameVar
ModifStatusVarView
HBox
PullDownMenuBar
MagnifVarView
Path
FontVarView
Iterator
StateVarView
BrushVarView
MonoScene
Editor
PatternVarView
StringChooser
CatalogChooser
NameVarView
ConfirmDialog
Dialog
BasicDialog
PrintDialog
ControlInfo
AcknowledgeDialog
VPanelControl
KeyMap
PanelControl
HPanelControl
PullDownMenu
Interactor
UControl
CommandControl
Creator
Clipboard
EditorInfo
UMap
ObjectMap
Catalog
CGlue
CSolver


SF_ClosedBSpline
SFH_ClosedBSpline
F_ClosedBSpline
ClosedBSpline
S_ClosedBSpline
SF_OpenBSpline
SFH_OpenBSpline
PSPattern
F_OpenBSpline
PSFont
OpenBSpline
S_OpenBSpline
PSColor
F_Polygon
Extent
PSBrush
SF_Polygon
Pattern
GraphicBlock
Polygon
Font
S_Polygon
Vertices
Color
MultiLine
SF_MultiLine
Brush
S_MultiLine
F_Rect
Interactor
SF_Rect
Rect
S_Rect
Label
TextGraphic
Point
Line
Graphic
RasterRect
Stencil
PinGraphic
FullGraphic
Picture
Ellipse
SF_Ellipse
SF_Circle
Rubberband
FixedPin
F_Ellipse
F_Circle
SlidingEllipse
SlidingPin
S_Ellipse
S_Circle

## 11. APPENDIX III Parts of the model editor source code

Here comes selected parts from the Model Editor source code with comments. (The code is not compilable because of missing parts).

```
/*
 * Unique schematic capture system class identifiers
 * $Header: modelclasses.h  Tomas Szabo
 */

#ifndef modelclasses_h          //Every .h file should only be defined once
#define modelclasses_h

#include <Unidraw/classes.h>    //All the Unidraw unique class identifiers

#define NEWVIEW_CMD             2001  //All the new
#define MODEL_COMP              2006  //components,
#define EXAMINE_TOOL            2016  //tools,
#define NEWTOOL_CMD             2008  //commands and
#define MODEL_VAR               2007  //state variables
#define TOOLS_CMD               2012  //have to have
#define TERMINAL_VAR            2013  //unique identifiers
#define WIRE_COMP               2014
#define TERMINAL_COMP           2015

/* Composite ids associating subjects with their views */

#define MODEL_VIEW          Combine(MODEL_COMP, COMPONENT_VIEW)
#define TERMINAL_VIEW           Combine(TERMINAL_COMP, COMPONENT_VIEW)
#define WIRE_VIEW           Combine(WIRE_COMP, COMPONENT_VIEW)

//There are presently only one view implemented in the model editor,
//the component view.

#endif
```

```
/*
 * Model Editor main program.
 * $Header: main.c  Tomas Szabo
 */

int main (int argc, char** argv) {
    ModelCreator creator;               //A model creator have to be passed
                                        //to the Catalog, so it can create
                                        //the model editor specific classes
                                        //when they are read from file.
    Unidraw* unidraw = new Unidraw(
        new Catalog("model_editor", &creator), properties, options, argc, argv
    );

    ToolPalette* tp = new ToolPalette;

    unidraw->Open(tp);
    unidraw->Open(new ModelEditor(tp)); //The ToolPalette are linked to the
                                        //ModelEditor when it is created.

    unidraw->Run();                     //The main loop!
    delete unidraw;
    return 0;
}
```

```
/*
 * Implementation of model editor specific commands.
 * $Header: modelcmds.c  Tomas Szabo
 */

/*****************************************************************************/

static void Warning (Editor* ed, const char* warning) {
    AcknowledgeDialog dialog(warning);
    ed->InsertDialog(&dialog);           //A warning dialouge
    dialog.Acknowledge();
    ed->RemoveDialog(&dialog);
}

/*****************************************************************************/

void NewToolCmd::Execute () {
    Editor* ed = GetEditor();

    if (_dialog == nil) {
        _dialog = new FileChooser(
            "", "Create model tool for:", "~", 15, 24, " Create "
        );
    }
    ed->InsertDialog(_dialog);

    if (_dialog->Accept()) {
        ed->RemoveDialog(_dialog);
        const char* name = _dialog->Choice();
        InitNewTool(name);    //Initiate a tool for the chosen model

    } else {
        ed->RemoveDialog(_dialog);
    }
}

void NewToolCmd::InitNewTool (const char* protoName) {
    GraphicComp* comp;
    Catalog* catalog = unidraw->GetCatalog();

    if (catalog->Retrieve(protoName, (Component*&) comp)) { //The model is in
        ModelEditor* modelEd = (ModelEditor*) GetEditor();  //the Catalog !

        ToolPalette* tools = modelEd->GetToolPalette();
        ControlInfo* ctrlInfo = CreateCtrlInfo(comp);       //Create the
        Tool* tool = new GraphicCompTool(ctrlInfo, comp);   //model's control-
                                                            //information and
        char toolName[CHARBUFSIZE];                         //create the tool
        strcpy(toolName, protoName);
        strcat(toolName, ".tool");

        tools->InstallOrRemove(toolName, tool);             //Install the tool
    }
}

static ControlInfo* CreateCtrlInfo (GraphicComp*& origComp) {
    GraphicComp* comp = (GraphicComp*) origComp->Copy();
    Iterator i;
    for (comp->First(i); !comp->Done(i);) {         //Take only the frame
        if (comp->GetComp(i)->IsA(MODEL_COMP) ||   //and the terminals as
```

```
            comp->GetComp(i)->IsA(WIRE_COMP)) {    //children
            comp->Remove(i);
        } else {
            comp->Next(i);
        }
    }
    GraphicView* view = (GraphicView*) comp->Create(COMPONENT_VIEW);
    comp->Attach(view);
    Graphic* g = view->GetGraphic()->Copy();
    delete view;

    float l, b, r, t, scale;
    const float size = 3*cm;    //The size of the included tool

    g->GetBounds(l, b, r, t);
    scale = min(size/(r - l), size/(t - b));
    ScaleCmd scaleCmd((Editor*) nil, scale, scale);
    comp->Interpret(&scaleCmd);

    origComp = comp;
    return new ControlInfo(comp, "?");
}
```

```
/*
 * Model editor specific component definitions.
 * $Header: modelcomps.c  Tomas Szabo
 */

/*****************************************************************************/

ModelComp::ModelComp (SF_Rect* rect) {
    _modelVar = new ModelVar("model");

    RectComp* frame = new RectComp(rect);
    Append(frame);                          //Insert the Frame in the list
}                                           //of children components

GraphicComp* ModelComp::GetFrame() {
    Iterator i;
    First(i);                               //The Frame is the first
    return GetComp(i);                      //child component
}

Component* ModelComp::Copy () {             //This is a standard way
    ModelComp* copy = new ModelComp();      //of making copys of components

    copy->_modelVar = (ModelVar*) _modelVar->Copy();

    *copy->GetGraphic() = *GetGraphic();
    *copy->_modelVar = *_modelVar;

    Iterator i;
    for (First(i); !Done(i); Next(i)) {
        copy->Append((GraphicComp*) GetComp(i)->Copy());
    }
    CopyInternConnections(copy);

    return copy;
}

/*****************************************************************************/

TerminalComp::TerminalComp (PinGraphic* pingr) : (pingr) {
    TerminalComp::SetBinding(new TerminalVar("terminal"));
    SetTransMethod(In);
}

/*****************************************************************************/

TerminalView::TerminalView (TerminalComp* subj) : (subj) { }

Command* TerminalView::InterpretManipulator (Manipulator* m) {
    DragManip* dm = (DragManip*) m;
    Tool* tool = dm->GetTool();
    Command* cmd = nil;

    if (tool->IsA(GRAPHIC_COMP_TOOL)) {
        cmd = InterpGraphicCompManip(m);
    } else if (tool->IsA(MOVE_TOOL)) {
        cmd = GraphicView::InterpretManipulator(m);
//          cmd =InterpMoveManip(m);                    Not implemented!
    } else if (tool->IsA(CONNECT_TOOL)) {
        cmd = InterpConnectManip(m);
```

```
    }
    return cmd;
}

Command* TerminalView::InterpGraphicCompManip (Manipulator* m) {
    DragManip* dm = (DragManip*) m;
    Viewer* v = dm->GetViewer();
    Editor* ed = v->GetEditor();
    GraphicView* view = v->GetGraphicView();
    BrushVar* brVar = (BrushVar*) ed->GetState("Brush");
    SlidingPin* sp = (SlidingPin*) dm->GetRubberband();
    Transformer* rel = dm->GetTransformer();
    Coord px, py, dum;
    float dx, dy;
    TerminalGraphic* terminalGraphic;
    Command* cmd = nil;

    sp->GetCurrent(px, py, dum, dum);

    Selection* target = view->ViewIntersecting(
        px-SLOP, py-SLOP, px+SLOP, py+SLOP
    );

    if (!target->IsEmpty()) {   //Make it only possible to create a
       Iterator i;              //Terminal on the frame
        target->First(i);
        GraphicView* view = target->GetView(i);

        if (view->IsA(RECT_VIEW)) {      //The Frame can be a RECT_VIEW
            GraphicView* p = (GraphicView*) view->GetParent();
            if (p!=nil) {
                view = p;
            }
        }

        if (view->IsA(MODEL_VIEW)) {    //Or it can be a MODEL_VIEW !!
            Coord l[2], r[2];
            RectView* rect = (RectView*) view;
            rect->GetCorners(l, r);
            if (abs(px-l[1]) < SLOP) {
                px = l[1];
            } else if (abs(px-r[1]) < SLOP) {
                px = r[1];
            }
            if (abs(py-l[2]) < SLOP) {
                py = l[2];
            } else if (abs(py-l[2]) < SLOP) {
                py = r[2];
            }
            if (rel != nil) {
                GetOffset(rel, px, py, dx, dy);
                rel = new Transformer;
                rel->Translate(dx, dy);
            }
            terminalGraphic = new TerminalGraphic(px, py, stdgraphic);
            if (brVar != nil) {
                terminalGraphic->SetBrush(brVar->GetBrush());
            }
            terminalGraphic->SetTransformer(rel);
            cmd = new PasteCmd(ed, new Clipboard(NewSubject(terminalGraphic)));
```

```
        }
    }
    return cmd;
}

/*****************************************************************************/

TerminalGraphic::TerminalGraphic (Coord x, Coord y, Graphic* gr) : (x, y, gr) {

void TerminalGraphic::draw (Canvas* c, Graphic* gs) {
    if (!gs->GetBrush()->None()) {
        Coord x[3], y[3];
        x[0] = x[2] = _x-PIN_RAD;
        x[1] = _x+PIN_RAD;
        y[0] = _y-2*PIN_RAD;
        y[1] = _y;
        y[2] = _y+2*PIN_RAD;

        update(gs);
        _p->Rect(c, x[0], y[0], x[1], y[2]);  //The Terminal with its
        _p->FillPolygon(c, x, y, 3);          //arrow
    }
}
```

```
/*
 * Model editor main class implementation.
 * $Header: modeled.c  Tomas Szabo
 */

/******************************************************************************/

ModelEditor::ModelEditor (
    ToolPalette* tp, GraphicComp* comp
) {
    SetClassName("ModelEditor");
    SetName("model_editor");

    SF_Rect* rect = new SF_Rect(xModel[0], yModel[0], xModel[1], yModel[1],\
                                                        stdgraphic);
    rect->SetPattern(psnonepat);
    ModelComp* model = new ModelComp(rect);    //Create a frame
    if (comp==nil) {
        NewCompCmd* cmd = new NewCompCmd(this,model);
        cmd->Execute();                         //Initiate the model
    } else {
        SetComponent(comp);
    }

    _toolPal = tp;

    _keymap = new KeyMap;
    _keymap->Register(tp->GetKeyMap());

    InitViewer();
    InitStateVars();

    Insert(new Frame(Interior()));      //Create the interior of the editor
    Update();
    GetKeyMap()->Execute(CODE_SELECT);
}

Interactor* ModelEditor::Interior () {

    _tray = new Tray(
        new VBox(
            new HBox(
                new ModifStatusVarView(_modifStatus),
                new NameVarView(_name, Left),
                new MagnifVarView(_magnif, Right)
            ),
            new HBorder,
            Commands(),
            new HBorder,
            _viewer
        )
    );
    _tray->Align(BottomRight, new Frame(new Panner(_viewer)));
    _tray->Propagate(false);

    return _tray;
}

Interactor* ModelEditor::Commands () {
    PullDownMenuBar* commands = new PullDownMenuBar;
```

```
    commands->Include(CatalogMenu());

            ...

    return commands;
}

void ModelEditor::InitViewer () {
    const int sw = round(PAGE_WIDTH * inches);
    const int sh = round(PAGE_HEIGHT * inches);
    const int vw = round(VIEWER_WIDTH * inches);
    const int vh = round(VIEWER_HEIGHT * inches);

    GraphicView* view = (GraphicView*) _comp->Create(COMPONENT_VIEW);
    _comp->Attach(view);

    Graphic* pageBoundary = new S_Rect(0, 0, sw, sh, stdgraphic);

    _viewer = new Viewer(this, view, pageBoundary, vw, vh);
    _viewer->Update();    //Update the viewer
}

void ModelEditor::Include (Command* cmd, PullDownMenu* pdm) {
    ControlInfo* ctrlInfo = cmd->GetControlInfo();
    UControl* ctrl = new CommandControl(ctrlInfo);
    _keymap->Register(ctrl);
    pdm->Include(ctrl);
    cmd->SetEditor(this);
}

PullDownMenu* ModelEditor::CatalogMenu () {
    const char* menuName = "Catalog";
    PullDownMenu* pdm = new PullDownMenu(new ControlInfo(menuName));

    SF_Rect* rect = new SF_Rect(xModel[0], yModel[0], xModel[1], yModel[1], \
                                                               stdgraphic);
    rect->SetPattern(psnonepat);
    ModelComp* model = new ModelComp(rect);

    Include(
        new NewCompCmd(
            new ControlInfo("New Model", KLBL_NEWCOMP, CODE_NEWCOMP), model
        ),
        pdm
    );


      ...

    return pdm;
}
```

```
/*
 * ToolsPalette implementation.
 * $Header: modelpal.c  Tomas Szabo
 */

/*****************************************************************************/

ToolPalette::ToolPalette () {
    SetClassName("ToolPalette");
    SetName("model tools");
    _curCtrl = new ButtonState;
    _selection = new Selection;
    _keymap = new KeyMap;

    InitStateVars();
    InitEditorInfo();
    Insert(new Frame(Interior()));

    if (_initEdInfo) {
        Catalog* catalog = unidraw->GetCatalog();
        const char* attrib = catalog->GetAttribute("setup");
        catalog->Save(_edInfo, attrib);
    }
}

void ToolPalette::InstallOrRemove (const char* toolName, Tool* tool) {
    UControl* ctrl = FindControl(toolName);
    Catalog* catalog = unidraw->GetCatalog();
    const char* attrib = catalog->GetAttribute("setup");

    if (ctrl == nil) {
        if (tool == nil) {
            catalog->Retrieve(toolName, tool);
        }
        ctrl = new VPanelControl(tool->GetControlInfo(), _curCtrl);
        _compTools->Insert(ctrl); //Insert a new tool

        if (_edInfo != nil) {
            _edInfo->Register(toolName);
            catalog->Save(tool, toolName);
            catalog->Save(_edInfo, attrib);
        }
        _compTools->Change(ctrl);

    } else {
        if (_edInfo != nil) {
            _edInfo->UnregisterName(toolName);
            catalog->Save(_edInfo, attrib);
        }
        _compTools->Remove(ctrl);
        _compTools->Change(ctrl);

        tool = (Tool*) ctrl->GetControlInfo()->GetOwner();

        if (tool == GetCurTool()) {
            GetKeyMap()->Execute(CODE_SELECT);
        }
        _keymap->Unregister(ctrl);
        delete ctrl;
    }
```

```
}

Interactor* ToolPalette::Interior () { return Tools(); }

void ToolPalette::InitEditorInfo () {
     Catalog* catalog = unidraw->GetCatalog();
     const char* edInfoFile = catalog->GetAttribute("setup");
//See what tools are in the .modelrc file
     if (edInfoFile == nil) {
          _edInfo = nil;
          _initEdInfo = false;

     } else if (catalog->Retrieve(edInfoFile, _edInfo)) {
          _initEdInfo = false;

     } else {
          _edInfo = new EditorInfo;
          _initEdInfo = true;
     }
}

Interactor* ToolPalette::Tools () {
     _compTools = new VBox;
     _tools = new VBox;

     InitTools();

     if (_edInfo == nil || _initEdInfo) {
          InitCompTools();
     } else {
          ReadCompTools();
     }
     return new HBox(
          new VBox(_tools, new VGlue),
          new VBorder,
          new VBox(_compTools, new VGlue)
     );
}

void ToolPalette::InitTools () {    //Default tools
     Include(
          new SelectTool(new ControlInfo("Select", KLBL_SELECT, CODE_SELECT))
     );
     Include(
          new MoveTool(new ControlInfo("Move", KLBL_MOVE, CODE_MOVE))
     );
     );

     ...

}
void ToolPalette::InitCompTools () {    /Default component tools

     TerminalGraphic* term = new TerminalGraphic(0, 0, stdgraphic);
     term->SetPattern(pssolid);

     TerminalComp* terminalComp = new TerminalComp(term);
     Include(
          new GraphicCompTool(
               new ControlInfo(terminalComp, KLBL_PIN, CODE_PIN), terminalComp
```

```
        ), "Terminal.tool"  //A Terminal!
    );

    WireComp* wireComp = new WireComp(xWire, yWire, nWire, stdgraphic);
    Include(
        new GraphicCompTool(
            new ControlInfo(wireComp, KLBL_LINK, CODE_LINK), wireComp
        ), "Wire.tool"     //A Wire!
    );

    SF_Rect* rect = new SF_Rect(xModel[0], yModel[0], xModel[1], yModel[1],\
                                stdgraphic);
    rect->SetPattern(psnonepat);
    ModelComp* modelComp = new ModelComp(rect);
    Include(
        new GraphicCompTool(
            new ControlInfo(modelComp, KLBL_MODEL, CODE_MODEL), modelComp
        ), "Model.tool"     //An empty Model
    );
}
```