GECKO/Base
A Basic Graphical Editor Construction Kit based on Object Oriented Technologies

An experience report

Armin Grieshaber, Florian Wohlgemuth, Robert Sakretz
Peter Hofmann, Michael Ryba

University of Stuttgart

Institute of Parallel and Distributed
High-Performance Systems (IPVR)

Breitwiesenstr. 20 - 22

D-70565 Stuttgart

Michael.Ryba@informatik.uni-stuttgart.de

Abstract

GECKO/Base is an object oriented framework for the development of graphical editor. The use of object oriented technologies helps to reduce the development effort. The development of such a framework can be accelerated by using well known design patterns. The usage of design patterns is useful and easy as long as they are used independently. The composition of design pattern may become difficult. The framework can be used to derive several different graphical editors, having the same look and feel and the same base technologies. Since a framework can only provide frequently used functionality, specific features for a certain editor still have to be added by the user of the framework. The presented framework has been used to derive a sample editor containing all the framework functionality as well as to develop a few other specialized graphical applications.

Keywords

object-oriented graphical editors, user interfaces, Eiffel, application framework, design patterns.

Introduction

Today almost no application can be sold without an appropriate graphical user interface, because many users are not computer experts. Especially for CAD applications one of the most important parts of the user interface are graphical editing facilities.

Unfortunately the development of good graphical user interfaces is time consuming, error-prone and thus expensive. This becomes more and more relevant as the user interface's demands diverge from the underlying window systems's capabilities.

One example is the large field of graphical editing. All current window systems provide some capabilities for arbitrary graphical output, but not for graphical input, such as the placement and direct manipulation of graphical objects. These tasks are left over to the application developer.

As a consequence developers cannot concentrate on the core functionality of the application, but have to deal with the burden of designing and implementing graphical editing facilities.

As there are many common features among different graphical editors, the idea of separating the editor component in a reusable way from the rest of the application comes to mind.

Since there are not only common but also some different features among graphical editors the most appropriate way to achieve reusable editor components is an object oriented application framework. Such a framework implements the common features among the different types of editors and lets you derive a specialized editor by refining the according parts of the frameworks components.

We have analysed several different graphical editors and derived a set of independent graphical editing features, which were implemented in such a framework using the Eiffel language.

Existing frameworks such as UniDraw [VL89] or ET++ [WGM89] provide such graphical editing features and have proven the reusability of large parts of graphical applications. Both frameworks are written in C++.

A special interest for the development of the framework was the usability of design patterns in the field of graphical editing. This paper describes the design patterns used in the framework, their specific implementation and the experiences made with them.

Multiple Views by MVC Paradigm and Observer Pattern

One of the most famous design patterns is the *model-view-controller* paradigm, derived from the Smalltalk world [KP88]. For the domain of graphical editing the model-view-controller pattern has been applied on graphical objects as well as their containing documents. One of the results was the ability to display several document views for one document, each containing the corresponding views of the graphical objects. Each document view is capable of displaying a separate clip and zoom factor (Fig. 1).

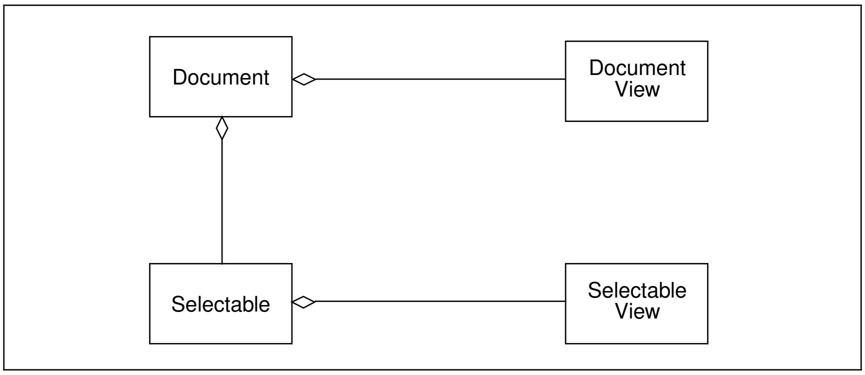

Fig. 1: Double Model and View pair

In our first approach the redraw mechanism was managed as suggested by the observer pattern directly by change/update. Given the situation that there are several overlapping graphical objects selected and you choose a different line colour for them via the inspector, each model receives a `set_line_colour` message which calls the `changed` message which again notifies all its dependants to redraw themselves ..

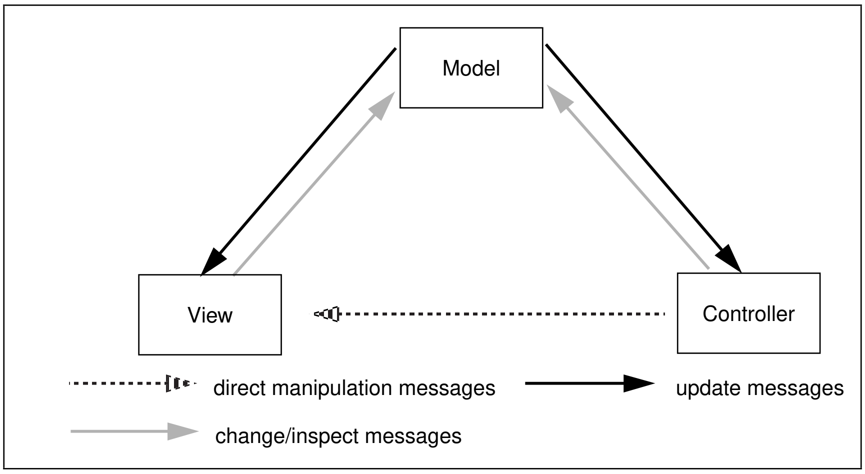

Fig. 2: The classic model-view-controller pattern

This raised several problems concerning the upper/lower relationship of overlapping graphical objects. Since the models do only have information about their own corresponding views, but not of the others they could not take them into account. So overlapping objects suddenly changed their relative position after a change applied to them, which - of course - was not correct.

Another problem were excessive redraw cycles, when performing a change on several objects simultaneously. Each model triggered a new redraw, as it had not enough information to decide, if it was time to redraw or not.

The solution of both problems was to coordinate the redraw mechanism using a combination of the *chain of responsibility* and the *command* pattern.

The solution for the upper/lower relationship was to propagate the need for redrawing a certain area of the screen from the model to its hierarchically higher model, which currently is the document. Since the document contains the whole set of models, it is capable to determine the set of objects to be redrawn. This can be achieved by calculating if they overlap the given area to redraw and then sending them a redraw message in the correct order.

The problem of redraw floods could be solved by dividing the redraw cycle in two phases. One phase for invalidating a set of rectangular areas, which have to be redrawn and another phase to execute the drawing.

The first phase is carried out by the models. Upon a change they notify their corresponding document by invalidating the corresponding rectangular area around them. But the decision, whether the end of the first phase has been reached can not be left over to the models, since they do not have enough information for that.

Each user interaction generates an instance of a command class. These commands then look up the current selection and distribute the action across the selected objects. The necessary information to decide, if the preparation phase can be left and the redraw phase can be entered, can be found in these commands.

Since commands can execute other commands the second phase had to be arranged as a chain of responsibility too.

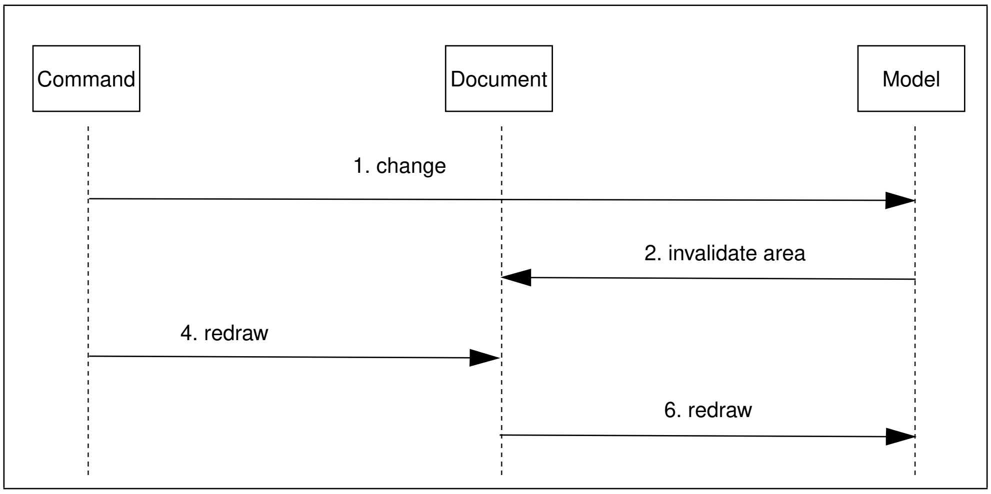

Fig. 3: The two phase redraw mechanism

Central Coordinate Mapping

Multiple views with different magnifications require a sophisticated management of coordinate values. As mentioned before, each view is capable of displaying another area of the drawing with its own zoom factor. Usually the area within a window is measured in pixels while the user of a drawing application thinks metrically or in inches. Additionally, the origin of a window is placed in the upper left corner while the user usually expects it in the lower left corner of his document.

The architecture consists of three different coordinate systems: (Fig. 4)

- the pixel oriented window coordinate system
- the internal application coordinate system with variable granularity
- the coordinate system presented to the user

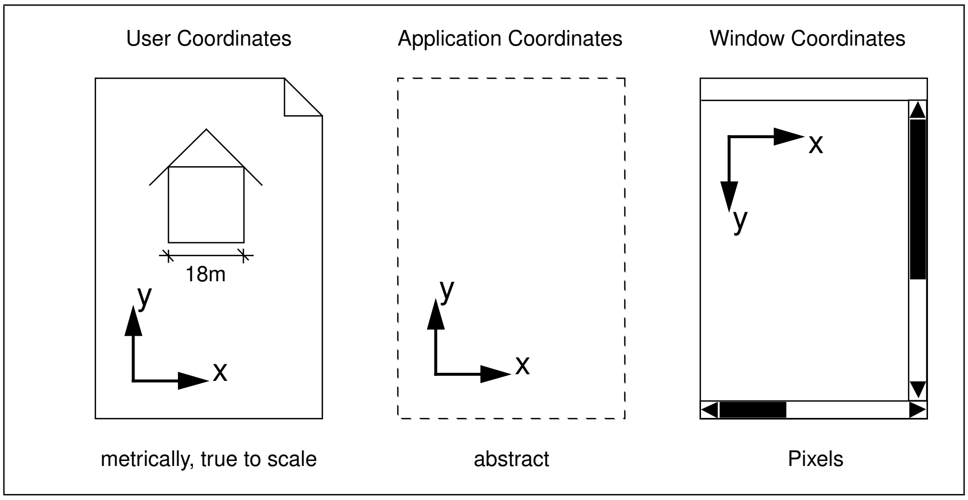

Fig. 4: Multiple Coordinate Systems

The application coordinate system has been made variable to support a wide area of graphical applications. Designers of microprocessors usually work in a nanometer range, while an art director producing a large advertisement is happy with a millimetre resolution.

The conversion routines were placed in a single class called `UNIT_CONVERSION`. Every class representing a document view inherits from this class. Every object needing a coordinate transformation delegates this task to the document view it belongs to. Changing or adding functionality to the conversion routine this way is very easy.

Attribute Manipulation

Different graphical objects have different graphical attributes such as linewidth, colour or fill patterns. But not all kinds of graphical objects have all kinds of graphical attributes. In reality there are many combinations of objects and attributes.

By applying the tool, material, aspect metaphor [RZ95] to graphical objects and their attributes two problems could be solved simultaneously. To achieve the separation of graphical attributes in separate classes and attach them to graphical objects via multiple inheritance. In the context of tool, material, aspect the models of the graphical objects can be interpreted as materials, the attributes as aspects.

In order to provide a set of tools the notion of a separate inspector window has been adopted from the NEXTSTEP user environment. The inspector concept is a separate window which represents a special view on the set of selected objects. The inspector displays the graphic attributes and their corresponding values for an object. Furthermore you can not only inspect attributes as the name would suggest. The inspector also provides some manipulation widgets to change the values of the affected attribute. Fig. 5 shows two simple examples of the inspector window, one to adjust the line colour and one to adjust some line properties, such as width, miter styles and end styles.

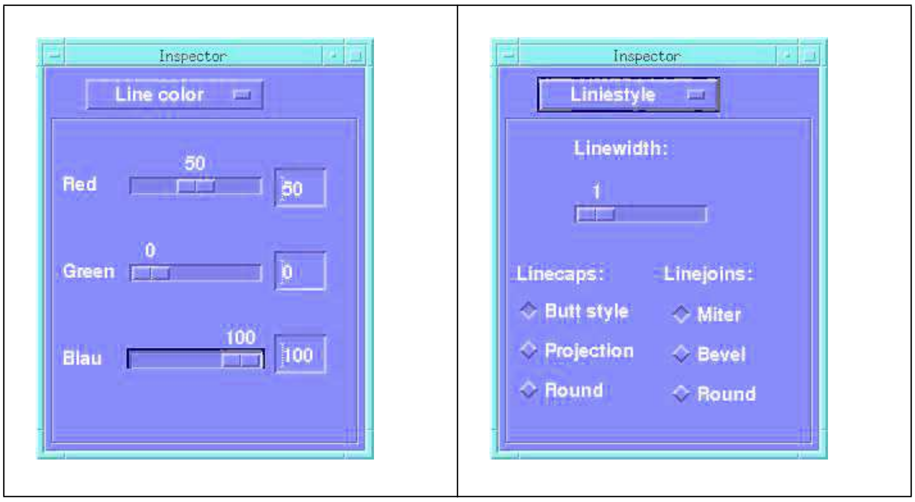

Fig. 5: The inspector window

The Inspector gets notified by the same change/update mechanism as described for the MVC paradigm. It declares itself as dependent on the current selection and whenever the selection changes it gets an update message to change its window contents according to the selected object's attributes.

We will illustrate the mechanisms involved using a simple graphical object, a circle. Of course a circle has an outline, which again has a linewidth and a line colour and maybe a dash pattern. As a circle is a closed object it has a fill colour and a fill pattern. So our circle model will have the aspects `has_color`, `has_linewidth`, `has_dash_pattern`, `has_fill_color` and `has_fill_pattern`. On the other hand a line only has a linewidth and a line colour, so it will only inherit from the classes `has_linestyle` and `has_color` (Fig. 6)

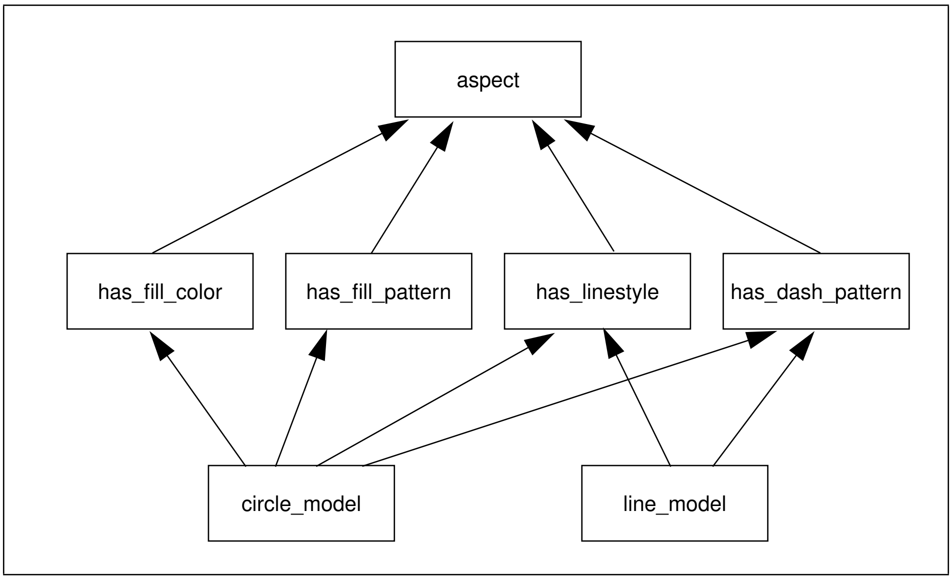

Fig. 6: Usage of the tool, material, aspect metaphor

For the interactive direct manipulation of graphical aspects the MVC paradigm can be applied on graphical attributes, too. For each aspect model there exists a corresponding aspect view, which is used by the inspector for visualizing and controlling the attribute values. In this example you have to provide a colour view, a linewidth view and a pattern view. These are manipulation widgets which reflect and affect the values of the selected graphical objects.

The above combination of MVC and the tool, material, aspect metaphor is an excellent example of extremely high reusability, right in the sense of componentware. You have to implement each desired attribute exactly once and plug it into all corresponding graphical objects, which saves a lot of time and thus money. Additionally you are able to provide the corresponding tools to manipulate their values in the same way too.

On the other hand the definition whether an object posseses an attribute or not is done statically at compile time. You cannot change these definitions dynamically at run time. In our context of manipulating graphcial objects this is no major drawback, since graphical objects have an a priori well defined set of attributes which does not change.

Another problem can be found in combination with the composite pattern. For statically composed objects the applicability of graphical attributes can be defined in the corresponding inheritance statement. Dynamically composed objects, e.g. a group of objects does not fit into this scheme since the members of a group are determined at run time only.

There are two possible solutions. The most easy but also the least satisfying is to avoid attribute manipulations on composite objects by defining a composite object not to have any attributes. The other solution is to define a composite to have all existing attributes. An attribute manipulation on a composite object is being propagated to all its members.

Of course it is not possible to propagate an attribute manipulation to an object which does not possess that attribute. Especially not in a statically typed language, such as Eiffel. Therefore the possession of the corresponding aspect has to be verified using Eiffel‘s reverse assignment attempt. The reverse assignment attempt verifies whether the investigated object inherits from a certain class or not. Using this technique the dynamic composite object is able to propagate attribute manipulation messages to its corresponding members.

Furthermore there is the possibility to provide operations upon graphical objects, such as rotating, mirroring or scaling. And again not each graphical object can be scaled, rotated or mirrored. For example a rotation upon a circle does not make any sense. So the definition whether a certain operation can be performed upon a graphical object must be done individually.

Obviously the definition of operations upon graphical attributes has an affinity to the attachment of graphical attributes. This means that each operation can be implemented as a separated class and inherited according to the tools, material, aspect metaphor.

The tools, material, aspect metaphor provides excellent reusability of all aspect classes by using multiple inheritance. Each aspect is being inherited separately. This again requires a programming language with a flexible inheritance mechanism, such as Eiffel. Although the Eiffel inheritance statement is easy to use, it is flexible enough to express any desired inheritance structure, by using the rename, redefine and select statements.

Direct manipulation

Interaction with graphical editors is typically done by direct manipulation. The user adds a graphical object to the drawing by selecting a tool which "makes" the desired object. Each tool has its own kind of interaction. A simple line is defined by two points, a polyline is defined by multiple points. Furthermore, the user wants to modify already added graphical objects in terms of moving around, resizing, changing constituting points and deletion. These interactions are also direct manipulations as the motion while repositioning a graphical object is directly reflected on the screen.

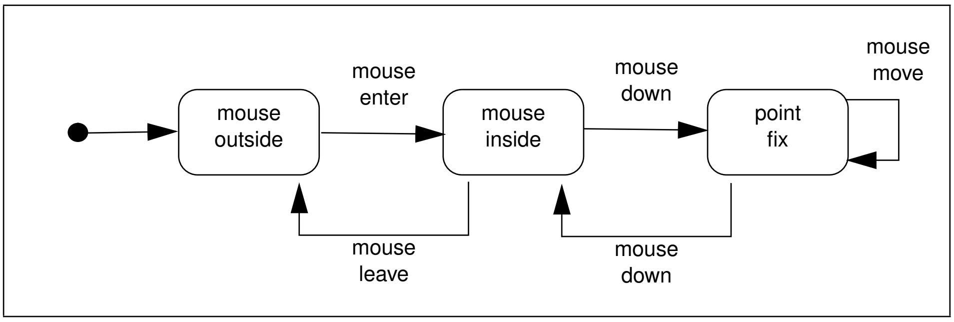

Fig. 7: State Chart for adding a line

The state machine was a convenient way to model these interactions. A characteristic feature of such an interaction is, that the user is expected to perform certain steps in a well defined sequence. If the mouse cursor is not within the window, it is necessary to firstly move the cursor inside a window to perform the interaction there. A typical state-machine of a rather simple interaction is given in Fig. 7. This state machine is performed, when the user adds a line to his drawing. The machine starts in the state labelled "mouse-outside", because the line-tool was selected from another window which is not necessarily within a drawing window. Entering the drawing window with the mouse-pointer you reach the state "mouse inside". Moving the cursor around in this window doesn't change the state. A mouse-click generates a "mouse-down-mouse-up" sequence and defines our first point. Now you enter the "point-fix"-state. In this state, pointer motion causes a line to be drawn from the point defined by the mouse-click to the current position of the cursor. A second mouse-click ends the interaction and adds the new line to the drawing. You are back in state "mouse-inside".

In a first iteration the state machine was simply implemented as a conventional double switch statement. This representation raised several problems. It does not reflect the designed abstract model very well. Either the code of the state or the code of the event (transition) is distributed. The computation of the state transition (i.e the next state) is distributed along every event. Adding a new state or a new event is painful.

In a second iteration the *state* pattern [Gam95] was applied with some enhancements. The information of the state transitions were distributed over the state classes. Each state has a table which defines the next state for a given event. The computation of the succeeding state is accomplished by a single instruction in the context-class: `state:= state.successor(event)`. While the table of the state transitions is distributed, the initialization of the tables is centralized in the context class. This violates the rule, that any attribute of a class should be properly initialized when an object is instantiated. But it seems feasible, because a state object is only meaningful in the context of a state machine. Furthermore you gain the possibility to define the complete state transitions in a single place.

The combination of the table-driven approach and the state-pattern provides the advantages of both worlds. The state transitions are centrally defined and explicitly performed and the state specific behaviour can be implemented in the state class. It is even possible to use one state within several state machines.

Developing a state machine for a circle, some similarities between the circle interaction and the line interaction were discovered. Both are defined by two points. In the third iteration the common code of both automata has been merged in a single "two-point" automaton which provides certain hooks to specify the exact behaviour of the automaton (Fig. 8). Now, line- and circle-maker simply inherit from the abstract two-point automaton. The programmer adds certain routines for displaying the shape of the graphical object while the user searches the second point or for adding the graphical object to the drawing.

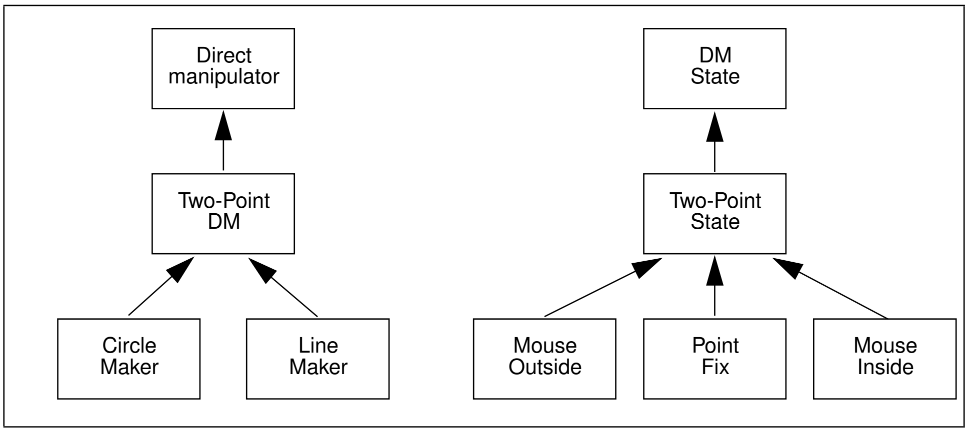

Fig. 8: Final class structure

Other interactions such as selection and resizing are combined in a complex automaton (Fig. 9). It covers the area select and the moving as well. The resizing of graphical elements is performed in sub-automata. The complete state machine takes about 12% of the lines of code of the whole framework.

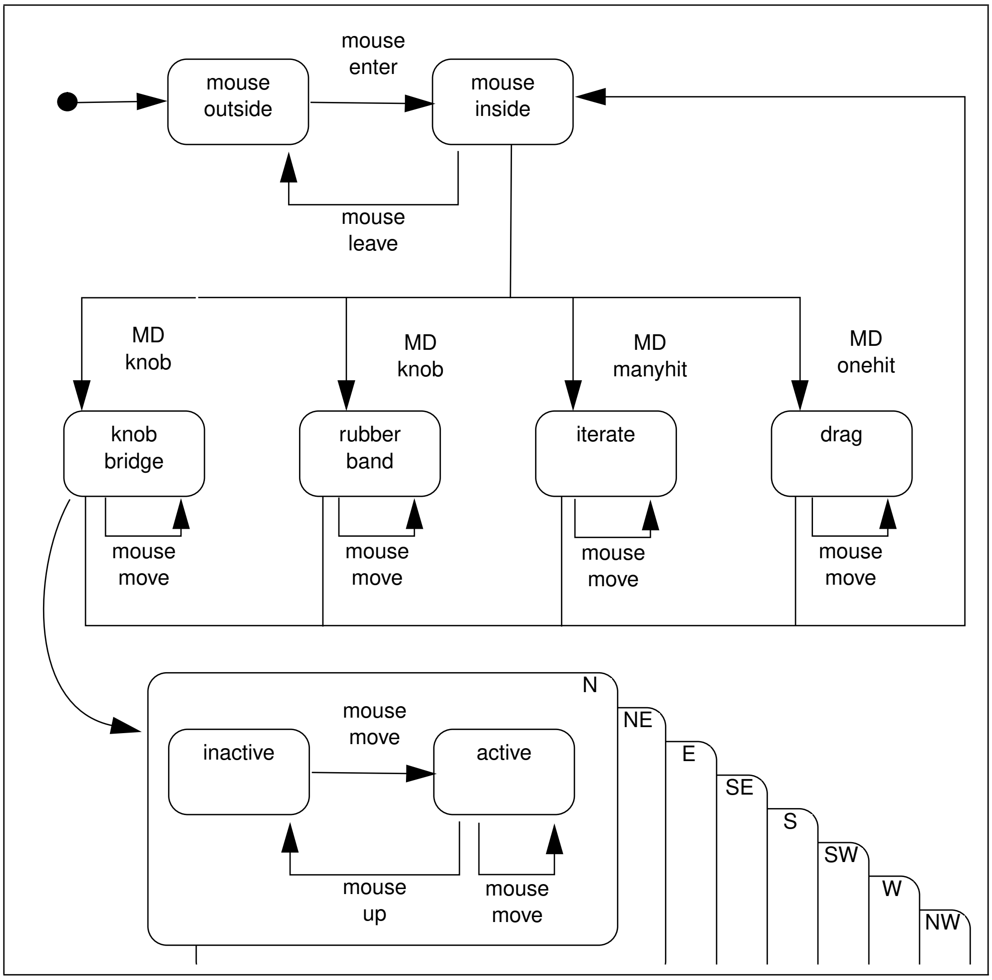

Fig. 9: State Chart for selection and resizing automaton

Store and Retrieve Mechanism

To preserve the work on a drawing or to exchange it with a co-worker, a way to represent the drawing external of the application is needed. The save-functionality of the `STORABLE` class provided by the ISE base library [Mey94] is easy to use. Let a class inherit from `STORABLE` and every object which can be reached from the inheriting class will be put in a file when the "write" method of `STORABLE` is called. The problem is, in fact, every object reached is stored. Even objects are stored, which represent resources of the operating system and are valid only within a specific program run.

The decision which objects should be saved must be made flexible and more fine grained.

To achieve a flexible store and retrieve mechanism the *coder-decoder* pattern of NeXT's appkit-framework [NCI94] was implemented in Eiffel.

The pattern suggests two classes, an encoder and a decoder-class. The encoder (or archiver) is responsible of storing objects, decoder (or unarchiver) retrieves objects. Both inherit from `CODER`, where common routines and the abstract interface are defined. Classes which should be stored and retrieved must inherit from class `CODING`. `CODING` is an abstract class and enforces the implementation of two methods:

encode_with_coder, and init_with_coder. The first method is called when the object is stored. Init_with_coder is called, when the object is restored. Each class can implement these methods individually and therefore flexibly store the attributes which are needed later for a proper restoration of the whole object.

The ENCODER and DECODER classes provide mechanisms to store basic objects (like BOOLEAN or INTEGER) and complex objects. An object requested to encode_with_coder receives an instance of CODER as a parameter and directs this archiver to take care of the attributes to be stored. Complex objects must con-

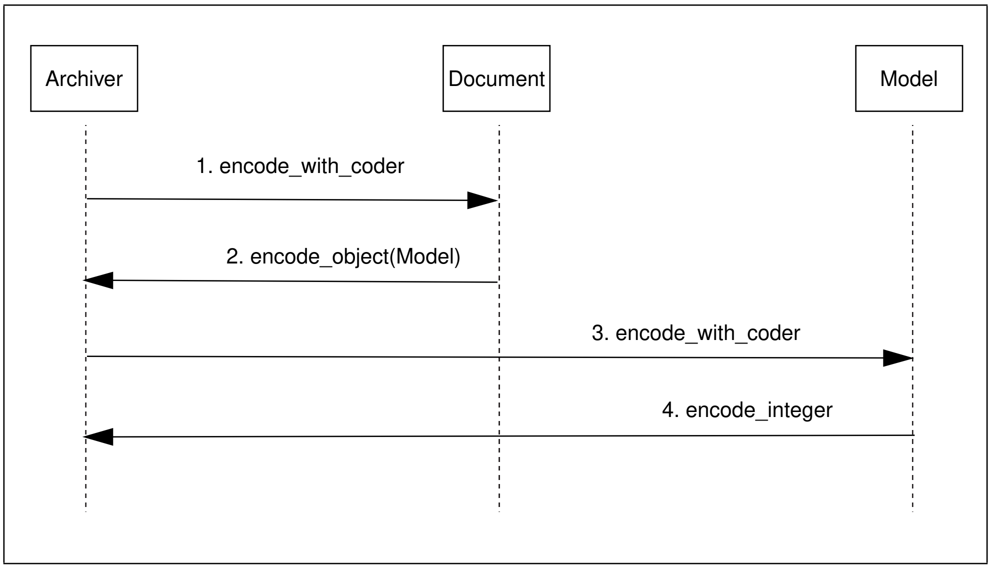

Fig. 10: Encoding objects

form to CODING and are simply called with object.encode_with_coder(Current). Some bookkeeping tasks must also be performed such as serializing the object structure and storing the class name. If the object has been stored earlier, only the reference number of previously stored object is saved. This way infinite iteration on a circular object structure could be avoided. During restoring (or deserializing) every object is kept in a table. If an object is requested again, the reference to the object can easily be found and returned to the caller.

Compared to the visitor pattern there are differences and similarities. Here the code for externalizing is distributed along the classes to avoid the otherwise necessary exhibition of class internals. The archiver-class has per se no influence on the sequence in which the objects are stored. On the other side there is a parametrically provided archiver which is used by the encoding routines of the object to be stored.

The external representation can easily be changed by inheriting from coder and decoder and redefining the appropriate methods. It is not limited to a file representation, an object structure can also be transported by a network connection to an application of a co-worker. Future developments will be using the same technique in order to achieve printing and graphical file format exports.

Current state of the implementation

Currently the general arc hitecture of the framework is stable although its complete implementation has not been finished yet. The current implementation concerning the document management encompasses

- zooming ,
- displa ying metrical rulers ,
- la yer management,
- grid snap mec hanisms ,
- sa ving and loading
- an efficient automatic redra w mec hanism.

Furthermore not all graphical objects ha ve been implemented yet, but some of them, suc h as

- lines , polylines , polygones , circ les , llipses , rectangles , parallelograms ,
- text labels
- statically composite objects , dynamical composite objects (groups)

Graphical attributes need to be completed too . Currently you can set

- line styles (width, caps and joins)
- line colour
- fill colour
- font

upon the corresponding graphical objects .

Fig. 11 shows a screenshot of a sample application. Y ou can see two different documents , eac h of them represented by two document views at different zoom levels .

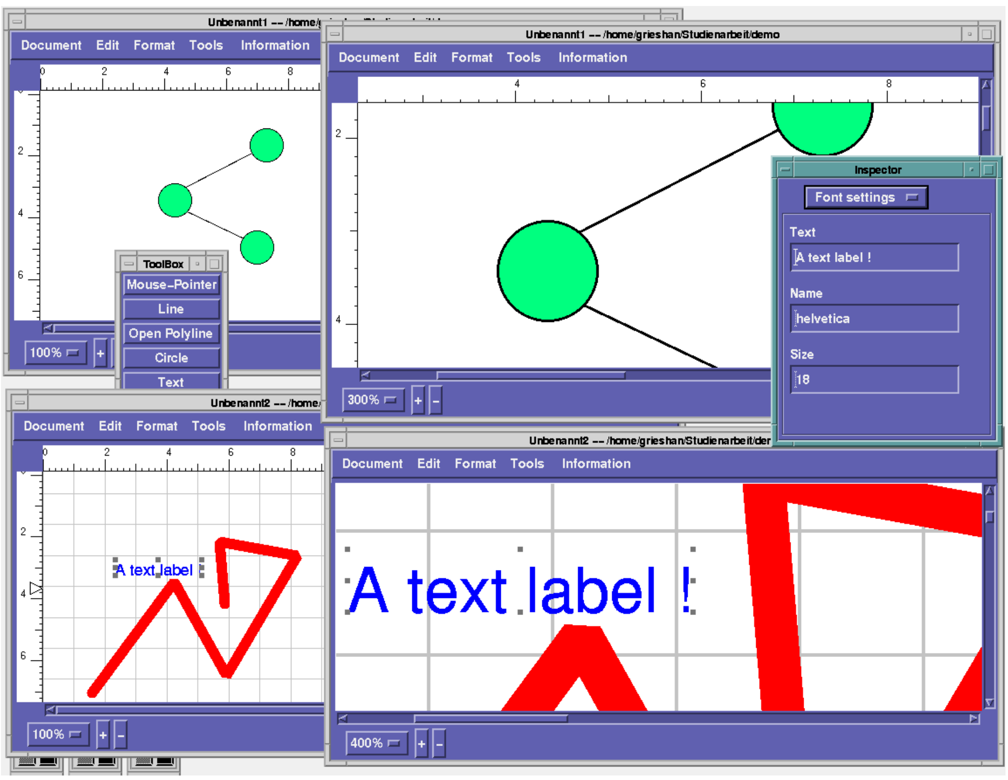

Fig. 11: The GECKO sample application

The framework was implemented using ISE Eiffel (Version 3.3.7) with its base libraries and EiffelVision libraries [MVB95] for Motif applications on SUN SPARC machines running Solaris 2.5.

The whole project is being analysed by some metrics: The framework currently consists of about 240 classes with 23000 lines of code. The relatively small size was achieved by a well structured design and a lot of code inheritance as described for the graphical attributes, for instance.

Among these 23000 lines of code 48% of the classes have a size less than 50 lines of code, 25% are between 50 and 100 lines of code. That means most of the code is split into relatively small pieces which can be managed easily.

Using and extending the framework

Basically, GECKO/Base consists of a document management cluster and a graphics cluster. The document management cluster furthermore provides the abstract coordinate system, arbitrary zooming and scrolling mechanisms, multiple layers and other related functionality. This cluster is organized of loosely coupled components. The construction of a concrete editor is based on plugging in the required components. Usually, writing special extensions is not necessary.

On the other hand, to support specialized graphical notations, the graphics cluster is intended to be more extensible. The specialized graphical components, defining the visual representation and behavioural aspects, have to be developed by the framework user.

Adding a new graphical object requires to write a model, a view and a direct manipulator to insert the new object. Since writing direct manipulators is time consuming, complex and error-prone the framework provides a set of predefined abstract direct manipulators. For example there is a direct manipulator which lets you define two points in the drawing area. This two point direct manipulator lets you define a lot of different graphical objects, such as a line, a circle or a rectangle. In order to use the abstract two point direct manipulator you simply have to inherit the corresponding class and have to effect the corresponding routines to instantiate the desired object.

In addition, this also applies to graphical attributes and operations to perform upon graphical objects. Currently the system supports a set of graphical attributes but is not yet complete. So the set of attributes can be extended by the framework user. In order to provide a new graphical attribute you have to write the corresponding aspect class and if needed an aspect view class, which provides a set of manipulation widgets for the inspector.

Likewise, graphical operations can be added to the system. Therefore you have to write the corresponding aspect class (e.g. `is_rotateable`), and either a direct manipulator, an inspector plugin or an operation panel.

Future Development

Application frameworks need continuous improvement in order to fit the users' needs. So it is necessary to derive some more editors from the framework in order to gain some more experiences.

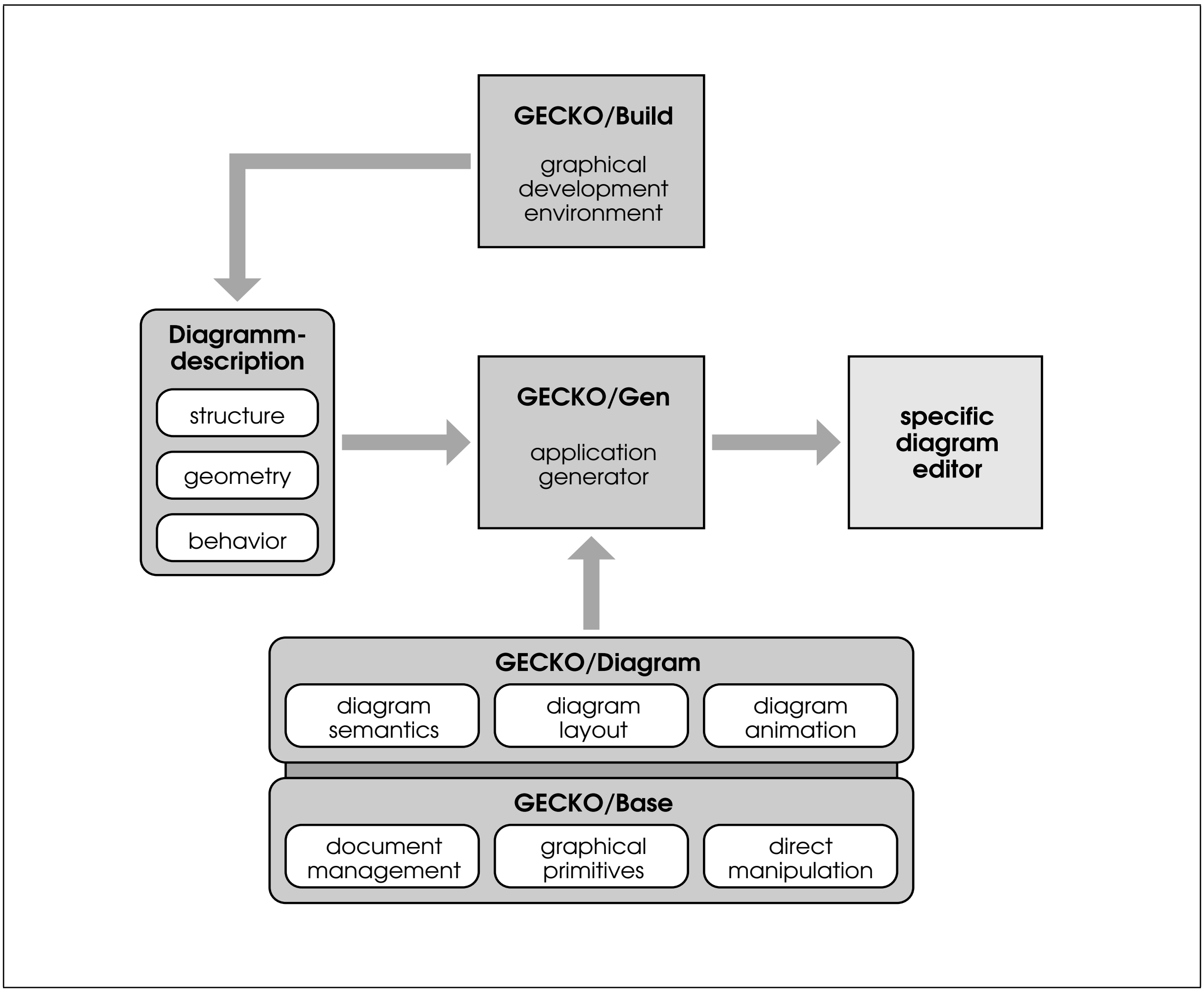

Fig. 12: The GECKO/Diagram structure

For example the application framework has been used to derive another editor for a data set visualisation system. It is not yet complete, but one of the experiences already made is the need for a hierarchical document concept like FrameMakers frame concept.

Another branch of the project will also work on the integration of the OASIS animation system and the editor framework. The result will be an integrated environment for interactive animation definition and runtime environment.

Another domain of our interest is generating specialized diagram editors. Each kind of diagram has a set of common and a set of specific properties and rules. These have to be identified, the common features have to be extracted and provided independently as an additional layer on top of the current base level. Then a diagram description language will be developed. This language will describe the structural, geometrical and behavioural aspects of a diagram as a high level programming language.

With the help of the description language it should be possible to establish the desired generating system for diagram editors (GECKO/Diagram). This system will be able to generate a specific diagram editor for each desired type of diagram, including structural, semantic and layout support. Fig. 12 reflects the overall structure of the desired system.

Conclusions

The development of an application framework for domain specific graphical editors has tought us several lessons. An application framework for graphical editors can be established on top of the current window system as an additional layer according to Fig. 13.

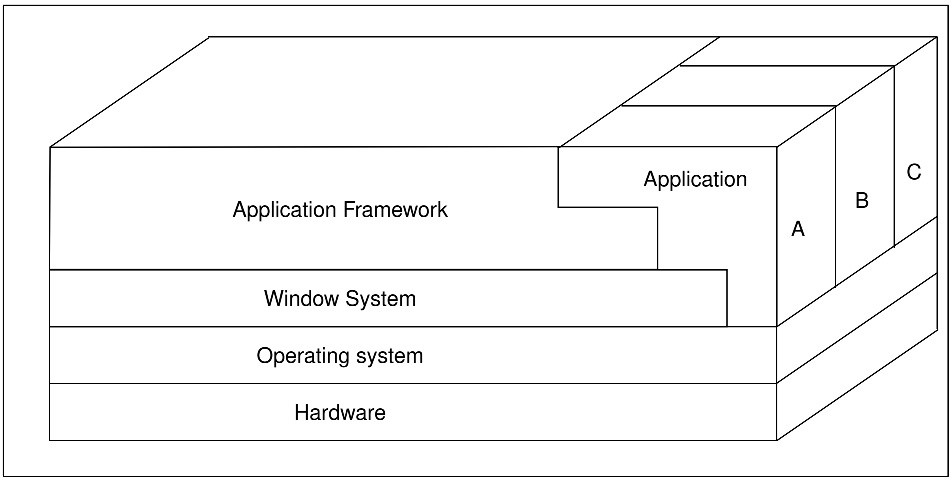

Fig. 13: Architecture Diagram of the GECKO Framework

The development can be simplified by using well known low level design patterns. These low level patterns can be composed to achieve a high level pattern for the development of graphical editors. The composition is easy as long as it is a structural composition. You simply transpose them into the framework, each of them for its dedicated purpose. But if two or more patterns have to be combined functionally, such as the redraw mechanism described above, then the patterns have to be modified. This can become a hard job and results in a completely new pattern, which again can be reused. The usage of design patterns is one step to a more efficient software development method. Once they are identified, verified and documented, they can be easily used in many different contexts and provide a solution to recurrent problems. As they can be combined to higher level patterns they are the building blocks of tomorrows componentware development.

Design patterns help to increase the comprehension of a set of classes and their relationships amongst them. Together with a comprehensible and powerful programming language, such as Eiffel, they play an increasingly important role in cost effective software development.

References

BMR96 Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal
A System of Patterns
John Wiley & Sons, Chichester, England, 1996

Boo97 Grady Booch
Patterns
http://www.rational.com/pst/tech_papers/patterns.html

Gam95 Erich Gamma et.al.
Design Patterns: elements of reusable object-oriented software
Addison Wesley, Reading, Massachusetts, 1995

KP88 Glenn E. Krasner, Stephen T. Pope
A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80
ParcPlace Systems, 1988

Mey92 Bertrand Meyer
Eiffel: the language
Prentice Hall International (UK), Hertfordshire, 1992

Mey94 Bertrand Meyer
Reusable softwarre: the base object oriented compound libraries
Prentice Hall International (UK), Hertfordshire, 1994

MVB95 Bertrand Meyer, Dino Valente, Eric Bezault
Building graphical Applications with EiffelBuild
ISE Technical Report TR-EI-43/UL, Interactive Software Engineering 1993-1995

NCI94 NeXT Computer, Inc.
The Foundation Kit
1994

RZ95 Dirk Riehle, Heinz Züllighofen
A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor
in:
Coplien, Schmidt
Pattern Languages of Program Design
Addison Wesley, Reading, Massachusetts, 1995, pp. 9-42

VL89 John M. Vlissides, Mark A. Linton
Unidraw: A Framework for Building Domain-Specific Graphical Editors
Proceedings fo the ACM SIGGRAPH/SIGCHI User Interface and Technologies ´89 Conference, Williamsburg, Virginia, 1989

WGM89 André Weinand, Erich Gamma, Rudolf Marty
Design and Implementation of ET++, a Seamless Object-Oriented Application Framework
Structured Programming, Vol. 10, No. 2, Springer Verlag, 1989