

The original MVC reports
Trygve Reenskaug
Dept. of Informatics
University of Oslo

I made the first implementation and wrote the original MVC reports while I was a visiting scientist at Xerox Palo Alto Research Laboratory (PARC) in 1978/79. MVC was created as an obvious solution to the general problem of giving users control over their information as seen from multiple perspectives. MVC has created a surprising amount of interest. Some texts even use perverted variants for the opposite purpose of making the computer control the user.

MVC was conceived as a general solution to the problem of users controlling a large and complex data set. The hardest part was to hit upon good names for the different architectural components. Model-View-Editor was the first set. They are described in my first note of May 12, 1979: *THING-MODEL-VIEW-EDITOR -an Example from a planningsystem*. A scanned version is included below.

After long discussions, particularly with Adele Goldberg, I ended with the terms Model-View-Controller as described in my second note of December 10, 1979: *MODELS - VIEWS – CONTROLLERS*. A scanned version is included at the end of this document.

Jim Althoff and others implemented a version of MVC for the Smalltalk-80 class library after I had left Xerox PARC; I was not involved in this work.

Oslo, February 12, 2007
Trygve Reenskaug

THING-MODEL-VIEW-EDITOR

an Example from a planningssystem

To LRG
From Trygve Reenskaug
Filed on [IVY]<Reenskaug>SMALL> TERMINOLOGY2.DOC
Date 12 MAY 1979

The purpose of this note is to explore the *thing-model-view-editor* metaphors through a coherent set of examples. The examples are all drawn from my planningssystem, and illustrate the above four notions. All examples have been implemented, albeit not within the clean class structure described here. The metaphors correspond to *real world-Model-view-Tool* as proposed in *A note on DynaBook requirements* ([Ivy]<Reenskaug>DynaBook.doc).

THING

DESCRIPTION OF TERM

Something that is of interest to the user. It could be concrete, like a house or an integrated circuit. It could be abstract, like a new idea or opinions about a paper. It could be a whole, like a computer, or a part, like a circuit element.

EXAMPLE: A LARGE PROJECT

The Thing is here a large project. It could be the design and construction of a major bridge, a power station or an off-shore oil production platform.

Such projects represent complex tasks with a very large number of interdependent details. Those responsible for a project have to keep track of all these details and their dependencies in order to understand the consequences of various real or proposed situations that may arise.

A great variety of different abstractions are used as aids in controlling large projects. Each abstraction highlights particular aspects of the total project, and are used alone or together with other abstractions for the control of those aspects. Examples of abstractions are ways of thinking about the requirements for materials; cost estimating; and various budgeting and accounting theories.

One particularly useful abstraction is the notion of *activity networks* or *Pert diagrams*. This abstraction links together *what* should be done with *who* should do it and *when* it is to be done.

In the network abstraction, every task that has to be performed in the project is mapped into a simple element called an *activity*. Basically, an activity is characterised by its *duration* and its *predecessors*. The duration specifies the time needed to perform the corresponding task, and

the predecessors are the activities that have to be completed before the present activity can be started.

This simple abstraction is usually extended in a number of ways. Based upon the activities' predecessors, it is easy to find the successors of each activity. The *start activities* of the network are all activities that have no predecessors, and the *end activities* are the activities that have no successors. Other information may be tacked on to each activity. The activity's resource requirements is one example, the cost and cash flow associated with the activity is another.

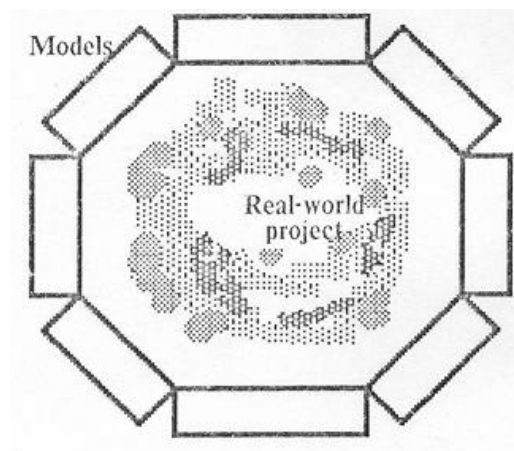
MODEL

DEFINITION

A Model is an active representation of an abstraction in the form of data in a computing system

COMMENTS

As mentioned above, there are in general many different ways of abstracting the same Thing, and it is therefore often useful with several, coexisting Models of a given 'Thing. Alternatively, one may think of the project as having one, large Model that is subdivided into a number of sub-Models.



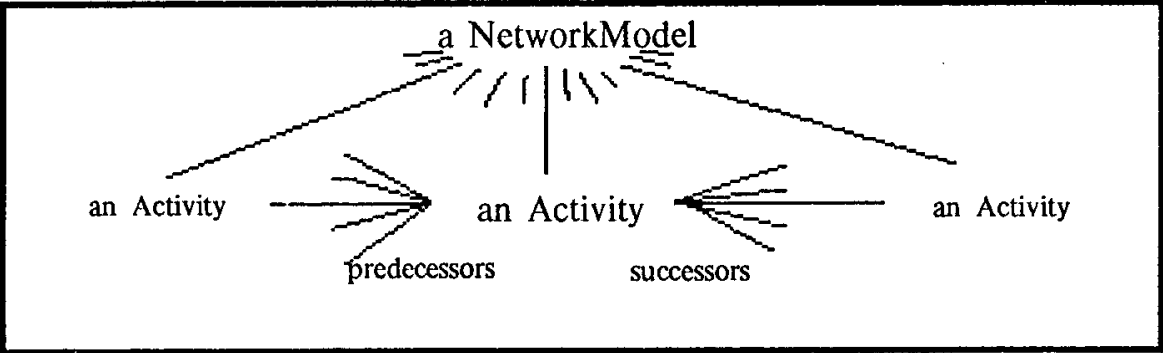
The models are represented in the computer as a collection of data together with the methods necessary to process these data.

Ideally, all models should be totally consistent. This ideal is not attainable in practice because it would require an overwhelming bureaucracy and stifling rigidity. One should therefore accept some inconsistencies, the aim being that the total set of models should be a reasonably accurate representation of the project without spending too much effort on trivialities.

In our example, a given network Model is represented in the Smalltalk system as an instance of class NetworkModel. Each activity in the network Model is represented as an instance of class Activity.

One of the fields defined for class NetworkModel is **activities**, a Dictionary connecting activity names (a UniqueString) and instances of Activity. Three of the fields defined for class Activity are **network**, **predecessors** and **successors**. **network** contains a pointer to the relevant instance of NetworkModel, and **predecessors** and **successors** are Vectors of pointers to instances of Activity.

The overall structure of our Model (= the Smalltalk representation of a network model of a project) is thus as follows: .



Within this general framework, we may now add information about our project that may be connected to the network as a whole, and to each of its activities. To the network, we may for example add the fields **plannedStart** and **plannedFinish**. To the definition of class Activity we may add the fields **duration**, **earlyStart**, **lateStart**, and **resourccRequirements**. We may also add fields for late start and late finish, but decide to compute these quantities whenever needed. (From **duration** and **earlyStart** or **lateStart**).

Note that the network Model with its sub-Models for each activity contains no information about how the information may be shown on the screen, it is a clean representation of the abstract project model.

VIEW

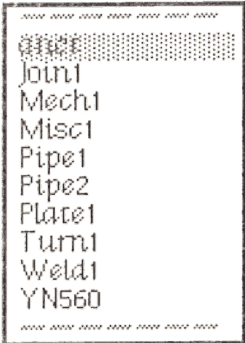
DEFINITION

To any given Model there is attached one or more Views, each View being capable of showing one or more pictorial representations of the Model on the screen and on hardcopy. A View is also able to perform such operations upon the Model that is reasonably associated with that View.

COMMENT

The implementation of all Views would be simplified if they were based upon the Form-Path-Image metaphors.

EXAMPLE 1: List of networks



This View belongs to a super-network, i.e. a collection of networks. It is thus slightly outside the scope of the Models discussed so far, but is included for completeness. The appearance of a list of networks is shown to the left.

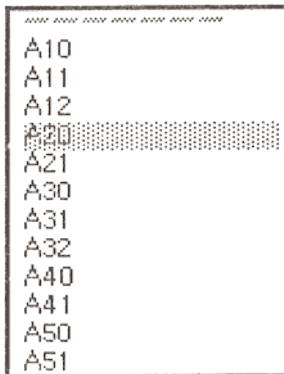
A list of networks is an instance of class NetworkList, which is a subclass of ListView.

A ListView has fields where it remembers its frame, a list of textual items and a possible selection within the list. It is able to display its list on the screen within its frame, and reacts to messages asking it to scroll itself. It understands a message asking it for an item that is positioned at a given point within its frame, and a message asking it to select a given item.

A ListView is thus somewhat similar to a ListPane in the present system, but it is not a subclass of Window and cannot be scheduled. It must therefore depend upon an Editor to tell it where its frame is and to arrange for scrolling. One possible sequence for selection is that the Editor reacts to redbug and asks the ListView for the item that is pointed to. The Editor then asks the ListView and any other Views to select that item. This separation of user interface and actual command execution provides for great flexibility in the Editor.

The class NetworkList builds on the general facilities of the ListView. It must know that the list in question is a list of networks, and it must know how to hold of such a list. Further, it must be able to execute commands that might be of interest in connection with its View. We will only consider two such commands: give the name of selected network, and tell the Editor to edit a given network.

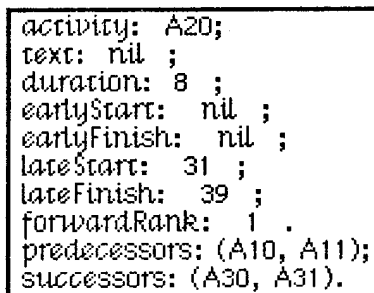
EXAMPLE 2: List of activities within a network



A list of activities within a network is an instance of class ActivityList, which is a subclass of ListView (see above for description of this general class).

An instance of class ActivityList must know which network it belongs to, and how to get hold of a list of all the activities in that network. There are no special commands other than the general selection mechanisms provided by ListView.

EXAMPLE 3: View of activity attributes



This is a textual presentation of all the attributes of an activity. It is an instance of class ActivityText, which is a subclass of TextView.

Class TextView has fields where it remembers its frame and a paragraph of text. It is able to display the text within its frame with wrap-around, and understands messages that ask it to scroll the text within the frame. It is also able to link given coordinates to positions within the text, and to select

text between given positions. It further understands messages asking it to perform various manipulations based upon the selection like *replace*, *cut*, *paste*, *again*, etc.

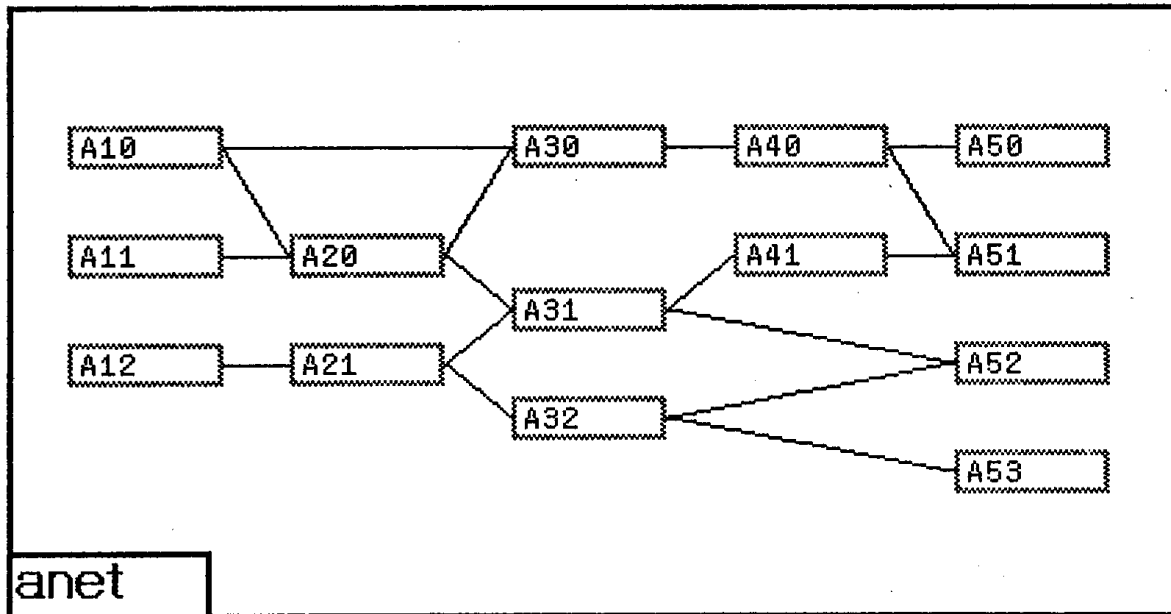
Class TextView is thus similar to the present ParagraphEditor, but each direct user interface is changed into one or more messages so that it may work together with other Views through an Editor, and with a variety of different Editors.

Class ActivityText must know how to get hold of the text for a given activity. This operation may usefully be triggered by a *select* message, usually from an Editor. It must further be able

to react to messages about the activity that are initiated by the user through an Editor. One set of such messages apply to modification of the data presented in the View, this is taken care of by the superclass TextView. The probably most important command to ActivityText itself is *accept*, requesting the View to check its present contents and to pass the information on to the network Model as an update of the attributes of the activity.

This View could be greatly improved if the class was made subclass of a tabular View rather than the present running text

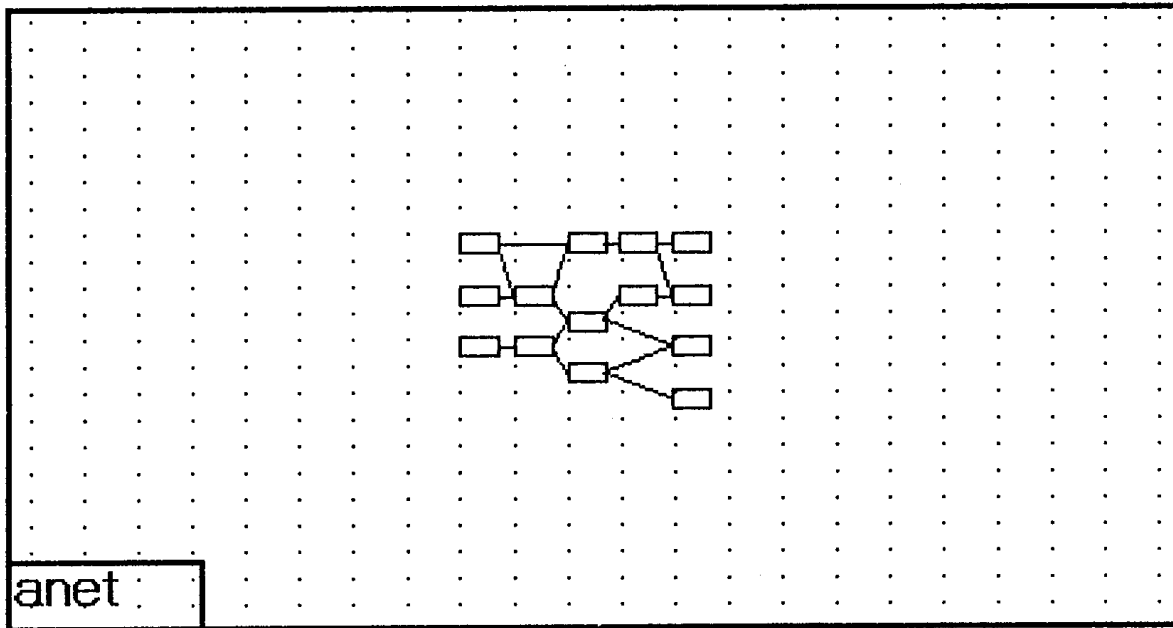
EXAMPLE 4: Network diagram



The diagram above is an instance of class DiagramView. It is the only example where the data in the View cannot be fully deduced from its Model, and it must have fields containing the shape and position of all the symbols, since this information is not part of the Model itself. (Programs exist for the automatic positioning of symbols within a diagram. but we assume that at least some manual editing will be required). The placement of the arrows may be deduced from the dependencies between the activities, this information may be fetched from the NetworkModel whenever needed for the display. Such a procedure will be very slow, however, and we assume that the Diagram View keeps a copy of that information for efficiency reasons.

Just as all the other views, a Diagram View will need to supply the two operations needed to select an activity, and the operations needed for scrolling (this time in two dimensions). In addition, it will need some operations on the View itself, they have to do with the positioning of the symbols in the diagram. Other operations have to do with network Model, for example *modify dependencies* of an activity and *transfer activity* from one network to another.

EXAMPLE 5: A variation on the network diagram

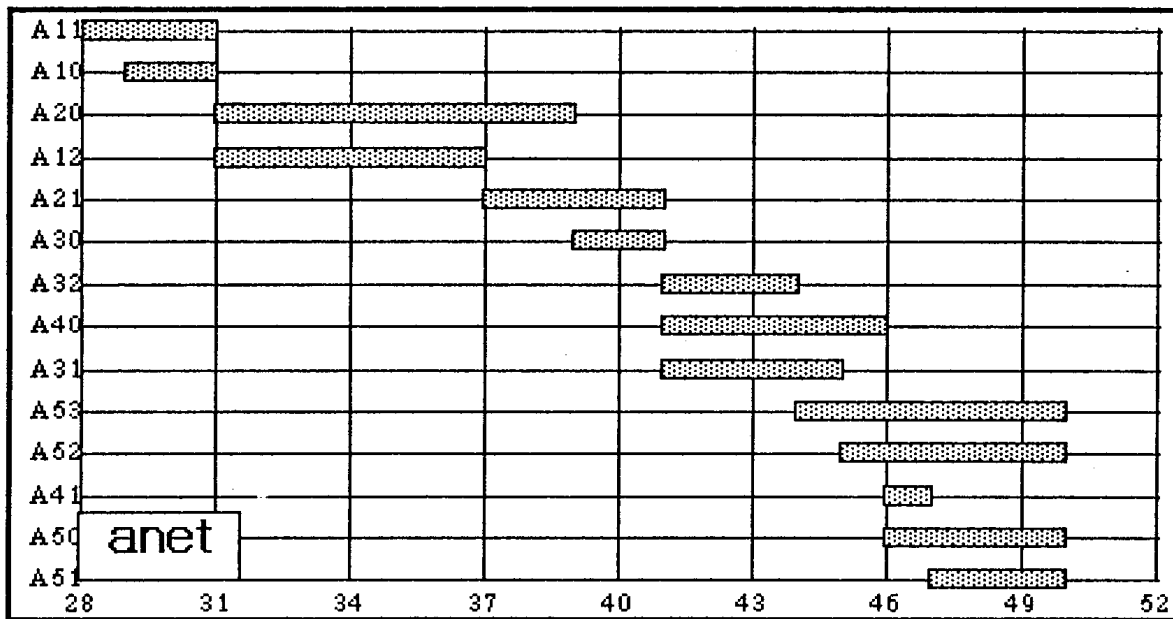


This View provides an alternative to the previous example. The symbols are smaller, and it therefore possible to present a larger part of the network on the screen. The small symbol size makes it impractical to print the activity name in the symbol, and the user must get this information by some other means.

The network diagram is presented on a gridded background. This is used when the user is laying out the diagram, and the dots define the permissible positions of the activity symbols.

This View could be implemented as a subclass of `DiagramView`, or both could be subclasses of some common View. In the present implementation, however, both Views may be generated alternately by the same object: class `Diagram View` has a field `displayType` that may take the values `#large` or `#small`, and the various display methods are controlled by that field. The gridding is displayed in the diagram through a separate method.

EXAMPLE 6: Gantt diagram



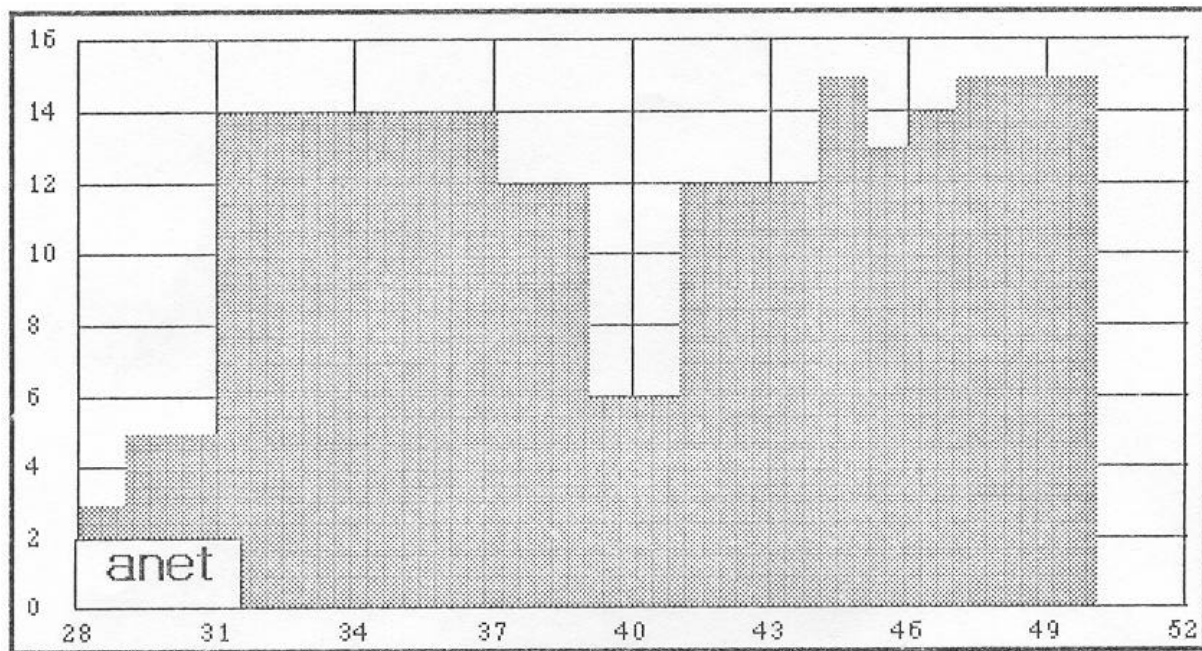
This View shows the activities along the vertical axis and time along the horizontal axis. In this particular view, only one time schedule for each activity is shown. Through varying the shading, it is quite easy to extend the presentation to include how much each activity may *float* in time without upsetting the overall project schedule.

The diagram is an instance of class `GanttView`, which is a subclass of `ChartView`. `ChartView` knows about the diagram background: axis with legend, gridding etc. It does not know anything about the information to be put into the diagram, in this case the horizontal bars. It aids its subclasses in presenting this information, however, through providing methods for conversion from whatever coordinate system is used externally to the frame coordinates of the diagram.

Class `GanttView` understands a number of messages from the user, most commonly given through an Editor. One such message is **viewNetwork:**, this message provides the name of the current `NetworkModel`. It will also be able to answer a question about which activity belongs to the bar under a given point in the diagram, and to select the bar belonging to a given activity. It is also able to pass on operations on the network and its activities that are related to this particular View. Typical operations have to do with modifying the current schedule. Operations should be provided for planning the network as a whole (**backload:** and **frontload:**), for modifying the schedule of a single activity (**plannedStart:** and **plannedFinish:**), and for letting the consequences of such a modification work its way towards the front or back end of the network.

The `NetworkModel` must be able to answer a few questions from a `GanttPane` giving a list of all activities, the schedule of the network as a whole, and the schedule of each individual activity.

EXAMPLE 7: Resource diagram



This diagram shows the sum of the resource requirements for the activities as a function of time. (In general, there will be one such diagram for each resource type.)

The diagram is an instance of class `ResourceView`, which is another subclass of `DiagramView`. The comments about the Gantt-diagram apply to this one as well, but the concept of selection needs elaboration. The views answer to *who:point?* could be the set of all activities that need resources at the time coresponding to the point's x-value. The reaction to the message *select: activity* could be to highlight the resource requirements of that particular activity. For symmetry, it would be nice to combine the two, and open the possibility for multiple selections in all `NetworkModel Views`.

EDITOR

DEFINITION

An Editor is an interface between a user and one or more views. It provides the user with a suitable command system, for example in the form of menus that may change dynamically according to the current context. It provides the Views with the necessary coordination and command messages.

COMMENT

A user task may commonly be better performed with several Views present on the screen at the same time. The user will want to manipulate these Views through pointing at them, through selection in menus, or through some other means. Commands like selection typically apply to several Views at the same time. The purpose of an Editor is to establish and position a given set of Views on the screen, to coordinate them and to provide the user with a suitable command interface.

EXAMPLE 8: UserView workspace

```
XEROX - Learning Research Group  
user restore.  
user screenextent: 640@750 tab: 0@50.  
Changes init.  
NotifyFlag ← true.  
  
to select a default printer  
PrinterName ← 'Clover'.  
  
(dpo file: 'changes.st') filout.  
dpo filin: ↵ ('Changes.st').  
(dpo file: 'text') edit.  
(dpo filesMatching: '*.st') sort  
dpo list. dpo freePages  
dpo delete: 'old'  
dpo rename: 'old' newName: 'new'
```

This is a very general Editor, providing a user interface to any part of the running Smalltalk system that can be reached from a global variable. It can therefore be used as an interface to the Views of the planning system, but it leaves most of the work to the user.

Since this Editor is always available, it is used to start up more specialized Editors and to perform commands that are not used sufficiently frequently to warrant such editor. One example of the latter is a command for the fileout of the complete data base.

EXAMPLE 9: An imaginary planning Editor

```
XEROX  
Learning  
Research Group  
Demonstration of a  
Smalltalk planning  
system
```

This Editor is very similar to the previous one, but it has been created in the environment of a demonstration network. Messages to that network and all its activities may therefore be typed in and executed directly through the *doit* command. The complete protocols of the network and the activities are therefore available through this Editor.

The Editor has not been implemented in this form, but as a part of a larger Editor shown in the next example.

EXAMPLE 10: A sub-Editor belonging to the

demonstration Editor.

```
INQUIRE AND
COMMAND:
NETWORK
  anet startActivities
  anet endActivities
  anet backlog: 50.
  anet forwardRank.

INQUIRE AND EDIT:
ACTIVITIES
  activity duration
  activity persons
  activity predecessors
  activity successors
  activity lateStart
  activity lateFinish
```

This Editor is part of the larger *demonstration Editor* (see next example), and is designed to work within its context. (The illustration of example 9 was actually the same Editor scrolled to its top.)

The Editor is an instance of class *DemoEditor*, which is a subclass of *TextEditor*. General texts may therefore be typed in, stored and edited in this Editor. Further, since *DemoEditor* knows about the demonstration network *anet* and all its activities, any message understood by these objects may be typed in and executed through a *doit* command.

Since this Editor is part of the larger *demonstration Editor* (see next example), we may execute the more general expressions like *activity duration*, where *activity* is interpreted as the currently selected activity.

A variation on this sub-Editor could talk to a an instance of a subclass of *ListView*, limiting the user's possibilities to executing predefined commands through prohibiting any editing of the text shown.

EXAMPLE 11: A demonstration Editor.

The appearance of this Editor on the screen is shown on the next page. The Editor is used to demonstrate that one given Model may be displayed through many different Views.

The demonstration Editor is implemented as an instance of class *DemoEditor*, which is a subclass of *PanedWindow*. Each of its panes is a sub-Editor that communicates with its particular View. It also gives commands to the demonstration Editor and receives commands from it.

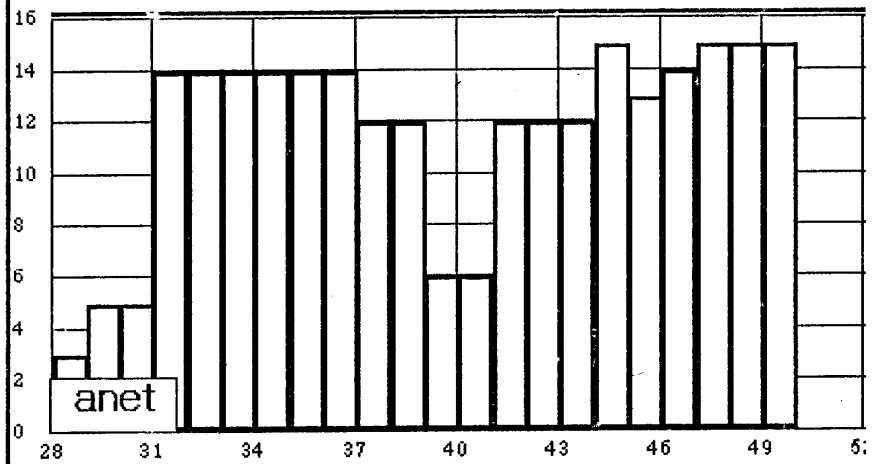
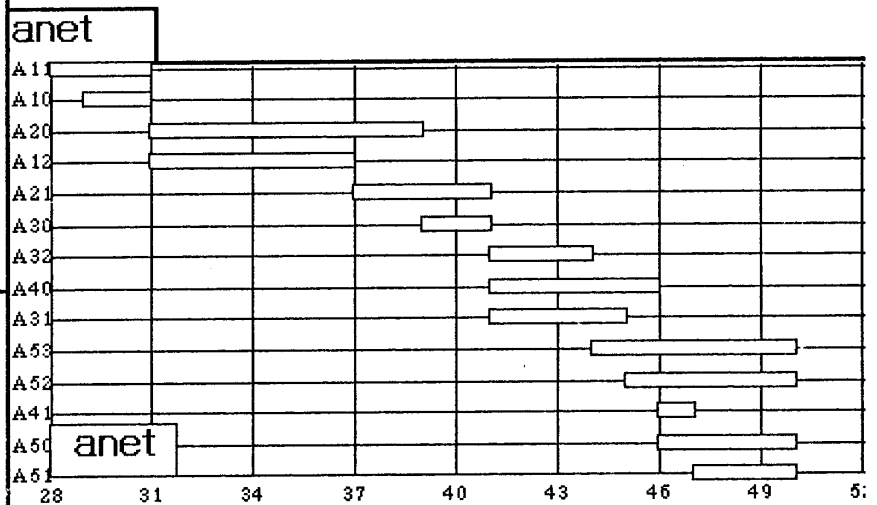
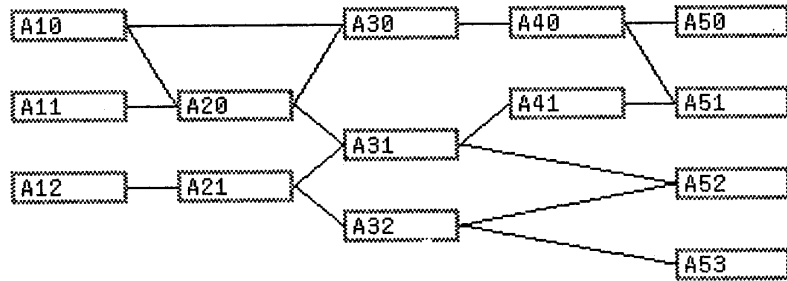
**INQUIRE AND EDIT:
ACTIVITIES**

activity duration
 activity persons
 activity predecessors
 activity successors
 activity lateStart
 activity lateFinish

**INQUIRE AND
COMMAND:
NETWORK**

anet startActivities
 anet endActivities
 anet backload: 50.
 anet forwardRank.

- A10
- A11
- A12
- A20
- A21
- A30
- A31
- A32
- A40
- A41
- A50
- A51
- A52
- A53



From: Trygve Reenskaug

Date: 10 December 1979

MODELS - VIEWS - CONTROLLERS

MODELS

Models represent knowledge. A model could be a single object (rather uninteresting), or it could be some structure of objects. ~~The proposed implementation supports knowledge represented in something resembling semantic nets (If I understand Laura correctly)~~

There should be a one-to-one correspondence between the model and its parts on the one hand, and the represented world as perceived by the owner of the model on the other hand. The nodes of a model should therefore represent an identifiable part of the problem.

The nodes of a model should all be on the same problem level, it is confusing and considered bad form to mix problem-oriented nodes (e.g. calendar appointments) with implementation details (e.g. paragraphs).

VIEWS

A view is a (visual) representation of its model. It would ordinarily highlight certain attributes of the model and suppress others. It is thus acting as a *presentation filter*.

A view is attached to its model (or model part) and gets the data necessary for the presentation from the model by asking questions. It may also update the model by sending appropriate messages. All these questions and messages have to be in the terminology of the model, the view will therefore have to know the semantics of the attributes of the model it represents. (It may, for example, ask for the model's identifier and expect an instance of Text, it may not assume that the model is of class Text.)

CONTROLLERS

A controller is the link between a user and the system. It provides the user with input by arranging for relevant views to present themselves in appropriate places on the screen. It provides means for user output by presenting the user with menus or other means of giving commands and data. The controller receives such user output, translates it into the appropriate messages and pass these messages on .to one or more of the views.

A controller should never supplement the views, it should for example never connect the views of nodes by drawing arrows between them.

Conversely, a view should never know about user input, such as mouse operations and keystrokes. It should always be possible to write a method in a controller that sends messages to views which exactly reproduce any sequence of user commands.

EDITORS

A controller is connected to all its views, they are called the parts of the controller. Some views provide a special controller, an *editor*, that permits the user to modify the information that is presented by the view. Such editors may be spliced into the path between the controller and its view, and will act as an extension of the controller. Once the editing process is completed, the editor is removed from the path and discarded.

Note that an editor communicates with the user through the metaphors of the connected view, the editor is therefore closely associated with the view. A controller will get hold of an editor by asking the view for it - there is no other appropriate source.