Star Graphics: An Object-Oriented Implementation

Dr. Daniel E. Lipkie
Xerox Corporation, El Segundo, California

Steven R. Evans, John K. Newlin, Robert L. Weissman
Xerox Corporation, Palo Alto, California

Abstract : The XEROX Star 8010 Information System features an integrated text and graphics editor. The Star hardware consists of a processor, a large bit-mapped display, a keyboard and a pointing device. Star's basic graphic elements are points, lines, rectangles, triangles, graphics frames, text frames and bar charts. The internal representation is in terms of idealized objects that are displayed or printed at resolutions determined by the output device. This paper describes the design and implementation of a graphics editor using an object-oriented technique based on a Star-wide subclassing method called the Trait Mechanism.

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques - User interfaces; H.4.1 [**Information Systems Applications**]: Office Automation - Word processing; I.3.6 [**Computer Graphics**]: Methodology and Techniques - Interaction techniques; I.7.2 [**Text Processing**]: Document Preparation

General Terms: Design

Key Words: business graphics, subclassing

I. The Star Workstation

In 1975 Xerox started an effort to transfer research from the *Xerox Palo Alto Research Center* (PARC) into mainline office products. Central to this strategy was the development of a top-of-the-line professional workstation, subsequently named Star, that was to provide a major step forward in several different domains of office automation. A retrospective on the development of Star is presented in [2].

A unique aspect of Star is its user interface (UI) and the role it played in the development of Star [5, 6, 7]. About 30 work years of effort were expended in designing the UI *before* the functionality of the system was fully decided and before the computer hardware was even built.

The hardware that supports this UI (figure 1)

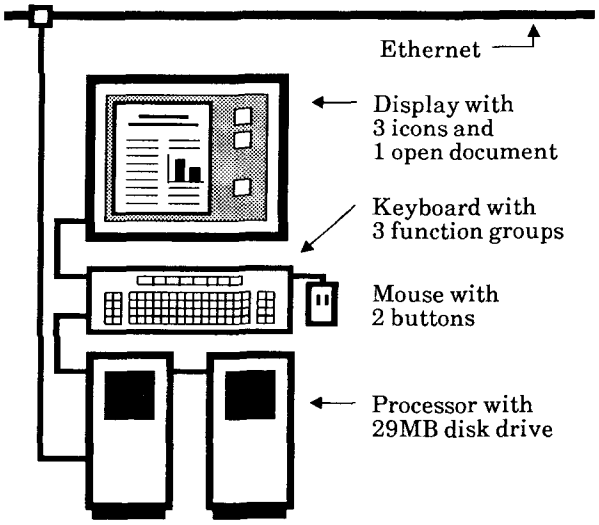


Figure 1
Star Workstation Schematic

consists of a microprogrammable 22-bit virtual, 18-bit real address space processor, an 808 by 1024 pixel (11" x 14") bit-mapped display, a keyboard, a pointing device called a *mouse* and a 10 or 29M byte disk. The workstation may be attached to a 10M bits-per-second Ethernet for access to remote printing, filing, communication and electronic mail services.

The mouse has a ball on the bottom that turns as the mouse slides over a flat surface. Electronics sense the ball rotation and displaces a cursor on the screen in

XEROX®, 8010 and Star are trademarks of XEROX CORP.

corresponding motions. There are two buttons on the mouse, called SELECT and ADJUST, used to make and adjust a selection as described below.

The keyboard has a conventional central part and three groups of function keys.

The left function group contains the *generic commands*: MOVE, COPY, DELETE, SHOW PROPERTIES, COPY PROPERTIES, and AGAIN. Their meaning is defined only in a generic sense; it is up to the currently selected element to further define them as explained below.

The keys in the upper function group are referred to as *soft keys*. Their meanings and use are discussed below.

The right function group includes the command KEYBOARD and others that are not of interest in this paper. When KEYBOARD is pressed, the soft keys allow the user to assign a new interpretation to the central keyboard and to display a window that shows the meaning of each keytop. Keyboards supported are Japanese, various European keyboards, Dvorak and keyboards with useful office and mathematical symbols.

Central concepts to the Star UI are *what you see is what you get*, *visibility* (don't hide things under CODE + key combinations) and a *physical office metaphor*.

One of the functional areas of the office addressed by Star is document creation, which encompasses text editing and formatting, figure editing (graphics), mathematical formula editing and page layout. These are all integrated. As an example of *what you see is what you get*, the Star user edits on the display both text and graphic figures, which appear exactly as they do when the document is printed. This document was prepared using Star; no special step was needed to merge the figures and text. Visibility and the office metaphor are discussed in the next section.

The design of the Star software began in the spring of 1978 and the first release, containing 255,000 lines of code, was completed in Oct. 1981. Over the 3.5 years approximately 93 work years of effort were expended and in excess of 400,000 lines of code were written. This effort was aided by the adoption of an object-oriented style of coding right from the start and by the use later of a *multiple-inheritance subclassing mechanism*, Traits [1], as the basis for defining and implementing objects. An object-oriented implementation was chosen because it corresponded closely to the UI model of interacting screen elements. In this paper we use the term *element* to refer to user perceived entities and reserve the term *object* for the corresponding internal implementations.

As explained below, there is no graphics editor per se, but of the 255,000 lines of code in the release about 28,000 are associated with editing figures in documents.

In Section II we describe the Star user interface. The Trait mechanism is presented in Section III. Its application in the Star implementation is discussed in Sections IV and V.

II. The Star User Interface

The Star UI differs from that of other computer systems through its heavy use of the graphics capabilities of a bitmap display, its adherence to a physical office metaphor, and its rigorous application of a small set of design principles [3]. The graphics capabilites reduce the amount of typing and remembering required to operate the system; the office metaphor makes the system familiar and friendly; the design principles unify the nearly two dozen functional areas of Star.

One important principle is to make objects and actions in the system visible. The system should not hide things under obscure CODE + key combinations or force the user to remember a lot of conventions. When a choice had to be made between easy novice use and efficient expert use, Alan Kay's maxim was followed: "Simple things should be simple; complex things should be possible".

As you make everything visible, the display becomes reality, and the user model becomes identical with what is on the screen.

Using the physical office metaphor Star creates electronic counterparts to the physical elements in an office: paper, folders, file cabinets, mail boxes and so on. The Star screen represents a *desktop* on which are placed small (~1" x 1") pictograms or *icons* that represent these elements, e.g. the document (paper) and file drawer (file cabinet) icons in figure 2.

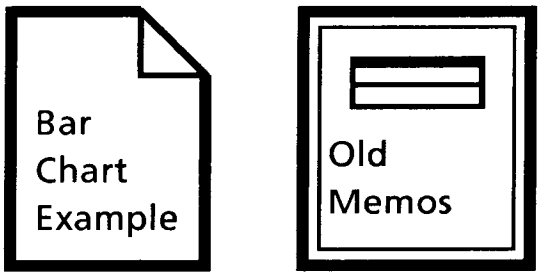


Figure 2
Document and File Drawer Icons

Within a illustration the currently implemented graphics elements are points, lines, rectangles, triangles, graphics frames, text frames and bar charts. Examples are shown in figure 3. A graphics frame is a

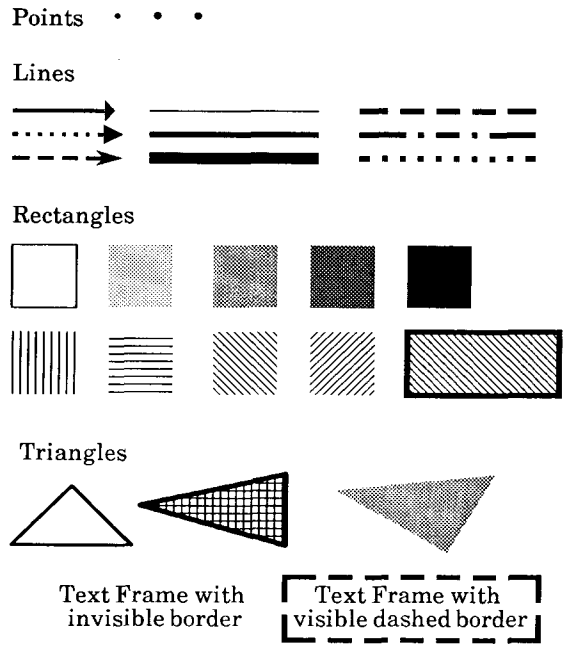


Figure 3
Examples of Graphic Elements

rectangular area reserved for figures. Text frames allow the user to put labels in figures.

Iconic, text and graphic elements are selected by pointing at them with the mouse and clicking (pressing and releasing) one of the buttons on the mouse.

Icons show that they are selected by *highlighting* (video reversing) their image. Character selections highlight themselves by inverting a rectangular region around the characters.

The user selects a graphics element by pointing anywhere along one of its lines or edges. When a graphic element is selected, it inverts a small square region around each of its *control points*. Lines have control points at each end, rectangles (figure 4) and

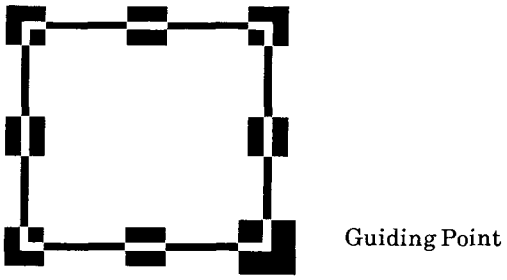


Figure 4
Rectangle with Inverted Control Points
(Expanded Scale)

frames at each corner and midpoint of each side and triangles at each vertex. The inverted region around the control point closest to the cursor is slightly larger. This control point is called the *guiding point* and becomes attached to the cursor when the element is moved, copied or stretched.

An element highlights as if selected when the mouse button is depressed, but it is selected only when the button is released. The user may change the candidate selection by moving the mouse with the button still depressed until the desired element is highlighted.

After an icon, character or graphics element has been selected, it may be manipulated by one of the generic operations. To move a document to a file drawer, select the document icon, press MOVE, point at the file drawer icon and click the SELECT button. Elements may also be manipulated in other ways described below.

The meaning of the operation is determined by the selected element. Moving a document icon to a file drawer icon sends the document over the Ethernet and stores it on a file server; moving it to an out-basket icon sends the document via electronic mail; moving it to a printer icon makes a hardcopy of it.

Copying and deleting have similar straightforward semantics.

The OPEN command in the left function group may only be applied to an icon and creates a window through which the icon's contents are displayed and edited. Star has a *modeless* editing style; there are no *start edit* or *end edit* commands. The user merely selects a character in a window displaying a document and begins typing and the text is appended to the selected character. The page content is reformatted as the user edits. The generic operations also may be applied inside a window; e.g. text may be moved, copied or deleted by merely selecting, pressing the function key and pointing to the destination.

Before discussing the other editing actions, we will explain how graphics elements are entered into text.

To enter a figure into a document the user selects a character in the document and types a character that represents a graphics frame. (The character is found on a keyboard accessible through the KEYBOARD key.) This non-printing, but screen-displayable, character is inserted after the selected character. It looks like a boat anchor and represents an anchoring point for the graphics frame. The frame appears between two lines of text at the same time the anchor character is typed. As the textual content of the document is reformatted during subsequent editing this anchor character is shifted as any other character and in addition its associated graphics frame is also repositioned, e.g. the anchor character acts like a footnote reference mark,

and the graphics frame moves from page to page as its reference mark is moved.

Once the user has a graphics frame in a document, other graphics elements may be moved or copied into the frame. Star graphics has only two *creation* actions, inserting a graphics frame as described above and MAKE LINE described below. All other graphics elements are made by copying. Every desktop has a *directory* icon that contains a blank document and a graphics document that has all the graphics elements. The directory's documents can only be copied, not moved or deleted, so the user always has a source of documents and graphics elements.

Pressing the SHOW PROPERTIES key opens a small window in which a *property sheet* appropriate to the current selection is displayed. The property sheet displays the property values for the currently selected element. The properties are changed by setting them to the desired values and clicking at the Done command which applies the new properties and closes the property sheet window. For each property the property sheet either displays an enumeratation of all possible values or provides a box into which the value is typed. The property sheet for a graphics line is shown in figure 5.

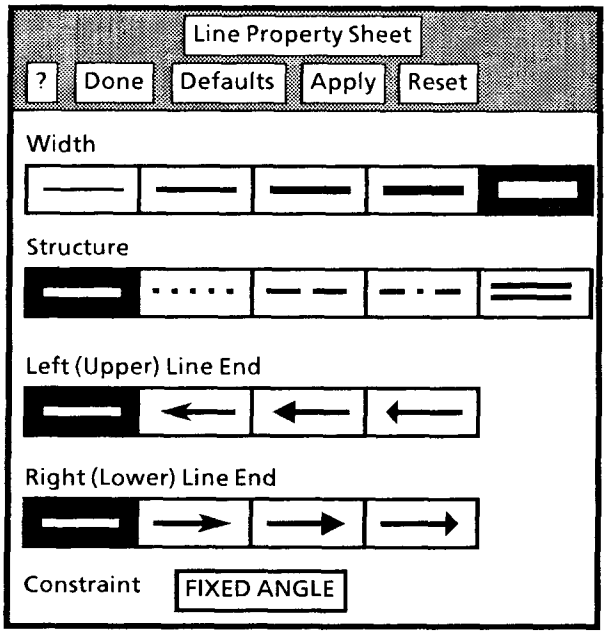


Figure 5
Line Property Sheet

The ? command provides access to online documentation about the line property sheet; Defaults sets the properties to system defined values; Apply applies the properties but does not close the property sheet; Reset sets the properties to the values they had when Show Properties was invoked.

There are three kinds of properties: *choice*, *state* and *text*.

A choice-type property displays a set of mutually exclusive values for the property which are shown immediately adjacent to each other; e.g. a line's width, structure and line endings are each choice properties. Exactly one is on at any one time and is video reversed. To change it the user points at the desired value and clicks a mouse button.

A state-type property may be either on or off. Pointing at and clicking a mouse button toggles its setting. A line may be constrained to be at a fixed angle so that its length but not its direction may be changed during the stretching action described below. An unconstrained line may have its length and/or direction changed.

A text-type property displays a box into which a text value is typed. None of the properties on the line property sheet are text-type. But an example is the text-type property on the document property sheet which determines the name of the document.

The properties of a rectangle are the width and style of its bounding box, its interior shading and a fixed shape constraint.

The properties of text and graphics frame include the width and style of the border. Text frames may be constrained to be *fixed* or *flexible*. A flexible text frame will change shape as its text contents are edited. Graphics frames may be positioned horizontally within a column (flush left, centered, flush right) and vertically within a column (top, centered, bottom or floating). Graphics frames also have a grid that may be displayed as dots or plus marks at each grid point or as ticks around the edge of the frame. Grid spacing is also variable.

Another way to change a selected element's properties is to press COPY PROPERTIES and then point at an element that is the source of the desired properties.

Associated with every text selection is a multi-click level: character, word, sentence or paragraph. Clicking at an unhighlighted character with the SELECT mouse button selects the character at the character level; clicking again with the SELECT mouse button at the same character selects the enclosing word; clicking at any character in a selected word selects the enclosing sentence; clicking at any character in a selected sentence selects the enclosing paragraph; clicking at a character in a selected paragraph brings the selection back to the character level and selects that character. Clicking at a character with the ADJUST mouse button expands or shrinks the selection at the current level to minimally span the pre-existing selection and the character pointed at.

There is no selection level associated with a graphics selection, but the ADJUST button has a graphics interpretation that is used to extend the selection to include multiple elements. Clicking the ADJUST button at a graphics element toggles it in/out of the current selection. The ADJUST button may also be used to extend the selection by adding all elements properly contained in a bounding box. The user presses the ADJUST button, which fixes one corner of the bounding box, and moves the mouse with the button depressed. The current mouse position defines the opposite corner of the bounding box. As long as the ADJUST button is depressed, a box is drawn on the screen from the fixed point to the current mouse postion and all elements properly contained are highlighted. When the button is released (at *button up*) these elements are added to the selection. An extended selection may be moved, copied, deleted, joined, stretched or the elements may have their properties changed.

The elements of an extended selection may be of different types, e.g. lines, rectangles and text frames.

Whenever there is a graphics selection the soft keys at the top of the keyboard take on graphics meanings: STRETCH, MAGNIFY, GRID, MAKE LINE, JOIN/SPLIT and TOP/BOTTOM. When the current selection is textual the soft keys take on meanings that allow the appearance of the charaters to be changed, e.g. bold, italic, underlined, superscript, subscript, larger font size and smaller font size.

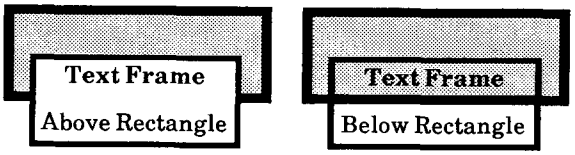
When STRETCH is pressed the selection is de-highlighted and the control point furthest from the guiding point is replaced by an X and is considered pinned. The guiding point becomes attached to the mouse when a button is pressed. As the mouse is moved the selection is horizontally and vertically scaled to conform to the pinned and guiding points and redisplayed. On button up the element retains the new size, the X is removed, and the selection is rehighlighted. MAGNIFY is simlar to STRETCH except that the same scaling factor is applied in the horizontal and vertical directions, i.e. aspect is maintained.

The GRID soft key toggles the grid on/off for the frame containing the selection. If the grid is active, it controls the placement of the guiding point during move, copy, stretch and magnify.

MAKE LINE creates a line between two successive mouse click positions.

JOIN combines an extended selection of graphics elements into a *cluster* element. Once joined, all of the original elements behave as a single element for purposes of selection and editing. This allows users to define their own graphics symbols. The SPLIT function acts on a cluster and reverses the effect of JOIN.

Graphics and text frames are *opaque*, that is they obscure elements that are *under* them. In figure 6a the



(6a) (6b)

Figure 6
Overlapping Elements

text frame is above the rectangle, while in figure 6b it is below. The soft keys TOP and BOTTOM allow the user to move the current selection to the top or bottom level in a frame.

In keeping with Star's style of modeless editing, the graphics editor is not invoked in the traditional sense. In fact, as we shall see later, there is no graphics editor in the traditional sense. All graphics editing capabilities are available whenever there is a document open. The Star user may pause during document editing and read incoming mail or use the records processing feature or any of the other Star functions. The responsibility for making the transition between these *editing environments* resides with the elements on the desktop, not the user. This is a major difference between Star and other information systems, including the Alto system [9] where the user explicitly invokes BRAVO for text editing, SIL or DRAW for figure editing, and LAUREL for electronic mail.

The full text editing capabilities are available for editing the contents of text frames within graphics frames, e.g. text frames may contain anchor characters and graphics frames. This means that Star must support the *virtual nested invocation* of editors.

III. Traits - The Star Subclassing Mechanism

Object-orientation is a method for organizing software such that, at any time, computation is performed under the aegis of a particular object, not a centralized program that handles every case from one place. The nature of the Star UI and the user model it fosters led to the adoption of an object-oriented method from the beginning of the software development.

Subclassing is a refinement of the basic object-oriented methodology that constructs objects out of more primitive behaviors. Initial Star subclassing efforts were in the SIMULA-67 and Smalltalk [8] style where the specialization relations form a tree. We found it necessary to generalize this concept to allow specialization relations that are represented by directed acyclic graphs. A full description of the Trait

mechanism and the generalized concept of multiple-inheritance subclassing is beyond the scope of this discussion but may be found in [1].

Subclassing as a way of implementing objects was not used during initial development of Star. This was partly because the designers had had little experience with subclassing as a methodology for large production software systems where performance is a primary consideration. It was also believed, incorrectly, that an extensible design based on subclassing would necessitate a violation of the typing mechansim of the implementing language, Mesa [4]. But as implementation progressed, it became clear that significant code-sharing was possible since we were dealing with objects that were more similar than different and we re-examined the subclassng problem.

We first present some of the central concepts of the trait mechanism and then describe how it has been applied in graphics. The initial graphics implementation was about 17,000 lines of code and space does not permit a full presentation of the graphics traits and their interaction. During this description we will refer to trait definitions summarized in Section VI.

A *trait* is a characterization of a behavior and is the primitive abstraction used to define objects. A trait used to define an object is said to be *carried* by the object, e.g. the trait TreeElement is carried by objects

Trait definition:
trait name
state
component traits
fixed operations
replaceable operations

that live in tree-like data structures. To implement a behavior, an object carrying a trait remembers information in a *state* defined by the trait. For example objects carrying TreeElement have 3 state variables, next, parent and eldest, that are pointers to the corresponding objects or are the special value *objectNil*, pointer to no object.

In a departure from SIMULA-67, traits may carry other *component traits* where the carry relationship is represented as an acyclic directed graph. This permits behaviors to be built on multiple lower-level abstractions. The basic imaging trait, Schema, carries TreeElement because all imaging objects are part of an *imaging tree* rooted at a Desktop Object that manages the Star display (see figure 9 below).

A trait defines *operations* as a means of presenting information to or extracting information from an object, e.g. the operations GetParent and SetParent for TreeElement. Operations also may be invoked for effect, e.g. the Schema operation Paint is a request to an object to paint its image on the display.

An operation is invoked on an object by specifying a trait carried by the object, an operation defined by the trait and the object. In Mesa, an operation invocation is implemented as a procedure call with the object handle as the first parameter and other parameters as needed, e.g.

Schema.Paint[object, ...]

Operations that extract information are implemented as procedures that return values.

A trait operation has a *specification* (name, parameters, return type) and a *realization* (an implementing piece of code).

Fixed operations are those for which the trait supplies the realization, e.g. the implementing code for GetParent in TreeElement is the same for all objects carrying the trait, it merely accesses the state variable parent and returns its value.

Replaceable operations are analogous to SIMULA-67 VIRTUAL procedures. The trait defines the specification and each class supplies its own realization that is used by all objects in the class, e.g. the Schema trait provides the specification for Paint but the classes Line and Rectangle each provide their own realizations that access the object's state to display the appropriate image.

A *class trait* is a trait that provides fixed operations for creating and destroying objects in the class. Associated with each class is a *replaceable operations vector* that is the composition of its own and its component trait's replaceable operations. The realizations of replaceable operations are assigned to the vector elements. The vector for the Line class is shown in figure 7.

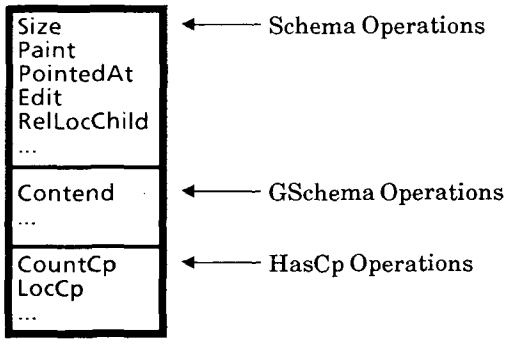


Figure 7
Replaceable Operations Vector for Line Class

Each object created by a class trait has an *object state vector* that is the composition of the class's state and the class's component trait's states. The vector for

a Line object is shown in figure 8.

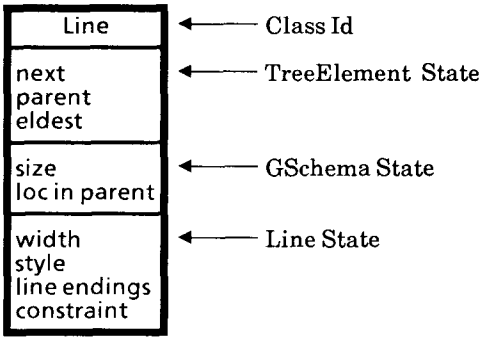


Figure 8
Object State Vector for a Line Object

IV. Applying the Trait Mechanism to Star

The first release of Star defined 169 traits, 129 of which were class traits. 99 traits required state variables and 39 had replaceable operations.

Non-class traits we will discuss are: TreeElement, Schema, GSchema, ListSchema and HasCp. Class traits we will discuss are AnchoredFrame, Line and TextBlock. Traits definitions for these traits are summarized in Section VI.

The **TreeElement** trait allows objects to be organized into tree data structures. The tree structure corresponding to a 3 page, 3 column document containing graphics and text is shown in figure 9. This

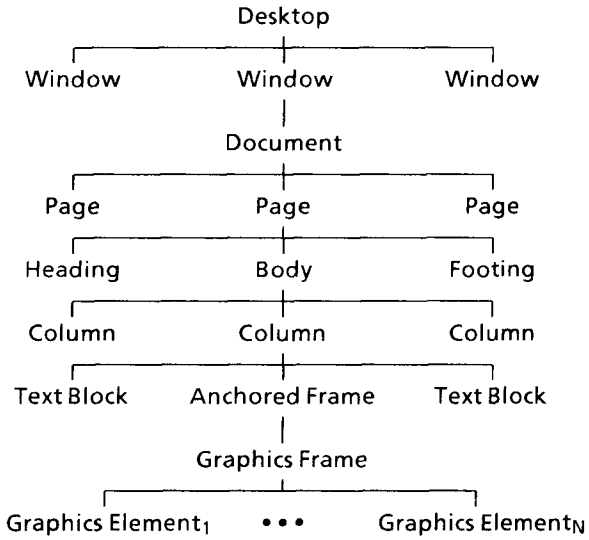


Figure 9
Desktop Imaging Tree

structure will be explained more fully after we have introduced the Schema and ListSchema traits.

The **Schema** trait forms the basis for imaging, pointing resolution, selecting and editing within Star. It defines 22 replaceable operations but for the purposes of this discussion we are only be concerned with those shown in Section VI. These operations allow an object to be asked its size (Size), to honor a request that it paint its image (Paint), to handle a pointing action by the mouse (PointedAt), to respond to an editing action when it is selected (Edit), to return the location of a child relative to itself (RelLocChild) or relative to the upper left corner of the screen (ScreenLocChild).

GSchema is an extension to Schema to meet the needs of graphics objects and is the basic trait carried by all graphics objects. It provides state variables for its size and location within its parent. Of the 39 replaceable operations it defines we are concerned only with Contend which is used during pointing.

ListSchema is a trait carried by an object that has non-overlapping children that are arranged either vertically with left edges aligned (pages in a document) or horizontally with top edges aligned (columns on a page), see figure 10. These two

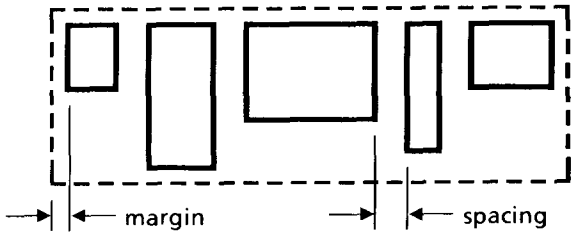


Figure 10
Horizontal ListSchema

arrangements are embodied in the ListSchema trait that is carried by an object that wishes to arrange its children in this manner. The state defines the inter-child spacing, the margin between the children and the parent carrying this trait and the color for the areas not covered by children. A list with color black and non-zero spacing and margin values is a common method for drawing lines around objects.

HasCp is a trait carried by all graphics objects that have control points. The only graphics object that does not have control points is the cluster object created and destroyed by the JOIN and SPLIT functions. For a given class the number of control points is the same so a replaceable operation, CountCp, is defined to return this value, e.g. 2 for line objects, 8 for rectangles. The replaceable operation LocCp returns the object-relative location of a control point. The fixed operation HighlightCp highlights a control point in one of the styles: default (small square), guiding (larger square) or pinned (an X). ClosestCp and FurthestCp are fixed

operations that enumerate the control points for an object and use LocCp to determine the control point closest and furthest from a particular coordinate. They are used to determine guiding and pinned control points. Rectangles, graphics frames and text frames have the same number and arrangement of control points and so use the same realizations for CountCp and LocCp. This increases code sharing so that only 2 realizations for 2 separate replaceable operations are needed to implement all the control point behaviors for 3 classes of objects. This is typical of the code sharing benefits of the trait mechanism.

AnchoredFrame is the class trait for the graphics frame that is associated with the anchor character. There is no screen-visible element for this object. The keyboard insert of an graphics frame actually creates two objects, an anchored frame with a graphics frame inside it. It is the graphics frame that is the visible box. Anchored frame objects are also used to anchor equations in text.

An anchored frame object forms the boundary between the non-graphics and graphics domains. A page column is a vertical list of left edge aligned text blocks and anchored frames. The one and only child of the anchored frame is a graphics frame that may be aligned flush left, centered or flush right within the anchored frame as determined by a property on the graphics frame property sheet. Within the graphics frame there are no restrictions on object arrangement.

Line is the class trait for graphics lines. Its state retains the properties shown in figure 5.

TextBlock is the class trait for objects that have textual content. Further details about this trait are beyond the scope of this discussion. Text blocks and anchored frames are the only objects that exists in a document column.

Note that the Schema trait defines operations for asking an object its size and location but does not define corresponding state variables. Also note that the replaceable location query is a request to a parent object for the location *within the parent* of a child object, i.e. a parent-relative location. This is done, as we discuss below, for flexibility and economy of storage and for performance.

It was felt best not to force all classes to store their size in the same manner at the Schema level because the trait is used as a component trait for a large number of classes each with possibly quite different behavior, e.g. a horizontal list-like object may determine its size by summing the widths of its children and use the height of its tallest child as its own height while a graphics object may store this information in its state. This judgement has been shown correct by the diversity of methods for determing size that now exist as the Star software has matured and new features, objects and behaviors have been implemented. It is quite common for a trait to define a behavior, such as Schema Size, that requires the cooperation of all objects that carry it in order to complete the behavior.

For performance reasons the fundamental location query is in terms of location within parent. Displaying an object or changing its location on the screen should not require changing its state.

For example, the Star workstation processor has instructions that support moving bits from one part of the screen to another. Scrolling a page upward is merely a matter of moving existing screen bits and painting new bits into the vacated portion of the window; none of the scrolled objects needs to be told to modify any of their state. This processor support also aids performance because it is not necessary to invoke the Paint operation for objects that already have their image on the screen.

Also, changing the size of an object or deleting an object near the front of a document does not require changing the state of all following objects in the document.

When the screen location of an object is needed for an operation the object is passed its screen location as a parameter or it invokes the operation ScreenLocChild on it parent.

V. Two Examples

Star graphics was the first major piece of Star software designed in terms of traits and that used the full generality of the mechanism. Pieces of software designed or implemented prior to graphics have subsequently been converted to the traits mechanism. In this section we will describe two interactions between the traits presented in section III. We first show how the GSchema trait *completes* the Schema size and location behaviors and second show how it *extends* the Schema trait for pointing behavior.

The GSchema state records a size and parent-relative location. Fixed GSchema operations allow this information to be accessed and changed. All GSchema objects use the same realization for the Schema replaceable operation RelLocChild which invokes the fixed GSchema operation GetRelLocSelf on the child object. Note that for a graphics object

```
GSchema.GetRelLocSelf[ object ]
```

returns the same value as

```
Schema.RelLocChild[
    TreeElement.GetParent[ object ], object ]
```

Objects that carry the Schema trait are responsible for a rectangular patch of the screen. Among sibling objects this may be sole responsibility, as is the case between page objects, or may be a shared

responsibility, as is the case between overlapping graphics objects.

Sole vs shared responsibility has interesting implications for the implementation of imaging behaviors. We will look at the pointing and selecting behavior of objects.

The document object carries the ListSchema trait and is the parent of page objects. It is quite easy for a document to determine which page contains the cursor and then pass the buck for processing the pointing action to that page. As long as the cursor remains within the bounds of the window displaying the document and within the bounds of the page, the page object has sole responsibility for tracking the movement of the cursor, for providing user feedback in the form of highlighting and for making a selection when the user releases the mouse button. If the cursor leaves these bounds with the button still depressed the page passes responsibility back to the document object for continued processing. A page satisfies its obligation by passing the buck to the column containing the cursor, etc.

When the user releases the button, the currently pointed-at object registers itself as the current selection with a central mechanism. Subsequent user editing actions are sent to it via the Schema Edit operation. It is up to the object to decide how to respond to the editing action, e.g. graphics lines ignore typing.

This method for button processing is embodied in the Schema replaceable operation PointedAt. Parameters for the operations are the object being asked to process the pointing action, its screen location, a tracking region, the current cursor location and the state of the mouse buttons. The return value for the operation is an updated cursor location and an updated mouse button state. The object must track the cursor as long as it is in the tracking region and must return control when the cursor leaves the region.

The semantics of PointedAt were designed for the non-graphics domain where nested list-like arrangements predominate, e.g. pages in documents, columns in pages. List-like arrangements also predominate outside windows but a description of their uses there is beyond the scope of this discussion.

The ListSchema trait provides a buck-passing realization for PointedAt that is used by almost all objects carrying the ListSchema trait.

Sibling graphics objects must share responsibility for pointing, eg. pointing *inside* the boundary of a rectangle may really lead to the selection of some other object. If the user points at the letter "x" in "Text" in figure 6a the text frame does not allow the user to point through to the rectangle under it. If the user points at the upper left corner of the text frame in figure 6b the user is pointing *through* the rectangle. This sharing behavior is implemented by the GSchema operation Contend described below.

If the cursor is positioned inside a graphics frame and a mouse button is pushed, the list-like PointedAt realization behavior described above resolves the cursor to the window containing the cursor, the document within the window, the proper page, the proper column and finally to the anchored frame within the column.

The anchored frame's realization for PointedAt is to enumerate it descendants and ask each how much interest it has in the current cursor position. The child with the greatest interest is passed the buck for processing the pointing action by invoking its PointedAt realization. The tracking region it is passed is *very* small, a box about 1/8" square. This allows the anchored frame to regain control and re-poll its descendants if the user moves the cursor any significant amount. The GSchema operation Contend is the operation used to ask a graphics object how much interest it has in the current cursor position. The descendants are enumerated top-down and enumeration stops when all have been enumerated or one of the descendants says stop, e.g. the text frame in figure 6a when the cursor is pointing at the letter "x".

Rather than change the semantics of PointedAt for graphics objects, or replacing it completely with a new set of operations to do pointing resolution, we merely added a pre-processing phase by adding Contend. The extending of behaviors by addition, not replacement, of operations is a capability offered by the traits mechanism and used widely throughout Star.

Note also that the user is allowed to button down near a graphics element and see it highlight, move the mouse with the button still down out of the graphics frame and point to a letter in the main document text and see the graphics element de-highlight and the letter highlight, continue dragging the mouse out of the document window and point to an icon and see the letter de-highlight and see the icon highlight and then select the icon by releasing the button.

All this is possible as a single user action. In the traditional sense this may be thought of as automatically invoking three editors in succession, the *graphics editor*, the *text editor* and the *desktop editor*, and passing control between them when in reality we are traversing a tree of objects and asking each to exhibit its own behavior. The implementation corresponds to this model and for this reason there is no *graphics editor* per se that is invoked by the Star user, there are only graphics behaviors that are exhibited in response to user actions and these behaviors are available at all times.

VI. Trait Summary

The following trait summary is in the order they were introduced above. Additional state variables and operations beyond the scope of this discussion are represented as "...".

trait name: **TreeElement**
state: next, parent, eldest
component traits: none
fixed operations: GetNext, SetNext, GetParent, SetParent, GetEldest, SetEldest, ...
replaceable operations: none

trait name: **Schema**
state: none
component traits: TreeElement
fixed operations: ScreenLocChild, ...
replaceable operations: Size, Paint, PointedAt, Edit, RelLocChild, ...

trait name: **GSchema**
state: size, location in parent, ...
component traits: Schema
fixed operations: GetSize, SetSize, GetRelLocSelf, SetRelLocSelf, ...
replaceable operations: Contend, ...

trait name: **ListSchema**
state: margin, spacing, color
component traits: Schema
fixed operations: ...
replaceable operations: ...

trait name: **HasCp**
state: none
component traits: none
fixed operations: HighlightCp, ClosestCp, FurthestCp, ...
replaceable operations: CountCp, LocCp, ...

trait name: **Line**
state: width, style, line endings, constraint
component traits: GSchema, HasCp
fixed operations: none
replaceable operations: none

trait name: **TextBlock**
state: text contents, ...
component traits: Schema, ...
fixed operations: ...
replaceable operations: ...

VII. Conclusions

Adopting an object-oriented implementation and the traits mechanism has been a success.

The initial graphics design and implementation (without bar charts and text frames) was done in one work year by a new hire who knew nothing about the Mesa language or the Star object-oriented methodology or the traits mechansim. This was in large part due to the building block nature of the methodology. Also the graphics functional specification had already been written, and one of the authors had validated the graphics user interface by prototyping on the Xerox Alto using a different implementation technique.

Subsequent to graphics, most of Star has been converted to this methodology, and three other major pieces of software have been undertaken: an equations editing capability, a 3720 emulations window, and a table editing capability. All are having equally good results.

The trait mechanism has allowed a rather straightforward mapping of Star UI elements to internal implementing objects.

REFERENCES

1. G. Curry, L. Baer, D. Lipkie and B. Lee, "Traits - An Approach to Multiple-Inheritance Subclassing," *Conference on Office Automation Systems*, Philadelphia, Penn., June 1982.

2. E. Harslem and L.E. Nelson, "A Retrospective on the Development of Star," submitted to *6th International Conference on Software Engineering*; Tokyo, Japan, Sept, 1982.

3. C. Irby, L. Berginsteinsson, T. Moran, W. Newman and L. Tesler, "A Methodology for User Interface Design", Systems Development Division, Xerox Corporation, January 1977.

4. J. Mitchell, W. Maybury and R. Sweet, "Mesa Language manual," *Technical Report CSL-79-3*, Xerox Corp., Palo Alto Research Center, Palo Alto, CA, April 1979.

5. J. Seybold, "Xerox's 'Star'" in *The Seybold Report*, Media, Pennsylvania: Seybold Publications, v. 10, no. 16, 1981..

6. D. Smith, C. Irby, R. Kimball and E. Harslem, "The 7tar User Interface: An Overview," *NCC '82*.

7. D. Smith, C. Irby, R. Kimball, B. Verplank, and E. Harslem, "Designing the Star User Interface," *Byte*, v. 7, no. 4, 1982.

8. L. Tesler, "The Smalltalk Environment", *Byte*, V. 6, no. 8, 1981.

9. C. Thacker, E. McCreight, B. Lampson, R. Sproull and D. Boggs, "Alto: A Personal Computer," *Computer Structures: Principles and Examples*, D. Siewiorek, C. Bell and A. Newell, editors, McGraw-Hill, 1982.