0

ARI Research Note 88-85

DTIC FILE COPY

An Overview of the Architecture for WE 1.0

AD-A199 467

Paulette E. Bush, Gordon J. Ferguson, John B. Smith, Stephen F. Weiss, Jay D. Bolter, and Marcy Lansman
University of North Carolina

for

Contracting Officer's Representative
Judith Orasanu

ARI Scientific Coordination Office, London
Milton S. Katz, Chief

Basic Research Laboratory
Michael Kaplan, Director

U. S. Army
Research Institute for the Behavioral and Social Sciences

September 1988

88 10 1100 9

U. S. ARMY RESEARCH INSTITUTE

FOR THE BEHAVIORAL AND SOCIAL SCIENCES

A Field Operating Agency under the Jurisdiction of the Deputy Chief of Staff for Personnel

EDGAR M. JOHNSON Technical Director	**WM. DARRYL HENDERSON** **COL, IN** **Commanding**

Research accomplished under contract
for the Department of the Army

University of North Carolina

Technical review by

Tracye Julien

Accession For		
NTIS GRA&I	✓	
DTIC TAB	☐	
Unannounced	☐	
Justification		
By		
Distribution/		
Availability Codes		
Dist	Avail and/or Special	
A-1		

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS - -
2a. SECURITY CLASSIFICATION AUTHORITY - -	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE - -	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) TR88-031	5. MONITORING ORGANIZATION REPORT NUMBER(S) ARI Research Note 88-85
6a. NAME OF PERFORMING ORGANIZATION University of North Carolina	6b. OFFICE SYMBOL *(If applicable)* - -
7a. NAME OF MONITORING ORGANIZATION U.S. Army Research Institute for the Behavioral and Social Sciences	
6c. ADDRESS *(City, State, and ZIP Code)* Department of Computer Science CB# 3175, Sitterson Hall Chapel Hill, NC 27599-3175	7b. ADDRESS *(City, State, and ZIP Code)* 5001 Eisenhower Avenue Alexandria, VA 22333-5600
8a. NAME OF FUNDING/SPONSORING ORGANIZATION - -	8b. OFFICE SYMBOL *(If applicable)* - -
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MDA903-86-C-0345	
8c. ADDRESS *(City, State, and ZIP Code)* - -	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
6.11.02	B74F	n/a	n/a

11. TITLE *(Include Security Classification)*
An Overview of the Architecture for WE 1.0

12. PERSONAL AUTHOR(S)
Pauline E. Bush, Gordon J. Ferguson, John B. Smith, Stephen F. Weiss, Jay D. Bolter(see 16)

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Interim Report	FROM 10-86 TO 10-89	1988, September	60

16. SUPPLEMENTARY NOTATION 12. Title (continued) and Marcy Lansman
Judith Orasanu, contracting officer's representative

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	Computers
			Cognition
			Graphics

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This research note discusses WE, a graphics-based Writing Environment. It provides tools to support the entire writing process, from brainstorming to document revision. Users visually transform their ideas from a network to a hierarchy defining document structure.
The prototype system is written in Smalltalk-80, an object-oriented interpreted language. This document presents the architecture of the WE version 1.0 prototype system. Sections cover the high-level component layout, the class hierarchy, the flow of control, the support framework, and the database support. Readers should be familiar with object-oriented programming in general and Smalltalk in particular to understand the note completely.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Judith Orasanu	22b. TELEPHONE *(Include Area Code)* 202/274-5590 — 22c. OFFICE SYMBOL BRO

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Contents

List of Figures

1 Introduction

WE is a graphics-based Writing Environment. Unlike many conventional document-preparation systems, it is more than a word-processor. WE carries the user through the entire writing process; from initial exploratory idea-generaton, through hierarchical organization, to the expression of ideas in text, to editing both structure and expression, and, finally, to producing the linear document, itself. The system is multimodal, providing different working contexts for the different tasks that comprise the overall writing process. It was designed to conform to an underlying theoretical perspective [Smith & Lansman, 1987] and to support a specific writing method [Smith & Smith, 1987]. However, it can be used by writers following a number of different strategies.

A key concept on which the system is based is that writers use different "cognitive modes" to carry out different tasks. A cognitive mode is a particular way of thinking in which different cognitive processes are used to create or to transform different coginitive (intermediate) products in order to accomplish a particular goal in accord with a certain set of constraints. Thus, for example during early exploratory thinking, writers brainstorm, represent ideas, group them into clusters, denote specific relations between pairs of ideas, and build small conceptual structures. They frequently do so in a relaxed frame of mind in order to stress flexibility and originality. By contrast, organization is a much more controlled way of thinking where writers build the actual structure for the document to be written. WE provides four different system modes to support four of the major cognitive modes used for writing: network mode for exploration, tree mode for hierarchical organization, editor mode for writing, and text mode for revision. The system also provides mechanisms for moving intermediate products created in one mode into another. For example, writers can move a small hierarchical relation created in network mode into the hierarchical structure being built in tree mode. Thus, system architecture reflects the underlying cognitive architecture of the user.

WE is being developed using the software engineering methodology of rapid-prototyping. Smalltalk-80 was chosen as the prototyping language because of its power as a rapid-prototyping language: the Smalltalk-80 system code provides a wealth of functionality as an application base and its pure object-oriented construction and code interpreter capabilities give it much more versatility and flexibility than the common procedural programming languages. WE is currently running as a version 1.0 Smalltalk prototype. Design will continue in Smalltalk. Version 1.0 has been translated into Objective-C - a closely related language suited to large, high-performance, end-product systems. For a description of that translation process, see [Shan, Smith, & Ferguson, 1988].

This document de-emphasizes WE user-interface code in favor of discussing the "main-program" classes, the tools, and the database components. These classes are relevant to other applications and will give the new WE programmer a general understanding of the architecture of the system.

The reader of this document should have a general knowledge of Object-Oriented programming and a working knowledge of the Smalltalk-80 programming language in particular. Some exposure as a user to the WE prototype system is also helpful. A reference list of background materials which cover prerequisite areas is included in the Appendices.

Throughout this document, specific smalltalk category names are in sans serif, class names are in SMALL CAPS, and method patterns are in *italics*.

2 Overview of the System

The use of an object-oriented programming language reduces the design discussion to one of class relationships, class knowledge, information hiding, and inheritance – a welcome built-in framework for expressing design issues and one that lends itself nicely to discussion at varying depths. I will, then, exploit this fact and present first an overview of the WE structure.

2.1 Class Hierarchy

The bulk of the prototype code breaks down into five main units: the 'environment code' including the 'controlling' category, 'support' categories, and 'database' categories and the user-interface code including the control panel category (CP-1.0) and mode categories (Network-1.0, Tree-1.0, Revise-1.0, Paragraph-1.0). A major version1.0 design decision was to include only four modes corresponding to four distinct phases of writing: Network Mode which allows a user to construct a directed graph or clustering of ideas, Tree Mode which imposes constraints in allowing the user to translate his ideas into a strict hierarchy (tree), Edit Mode which provides an editor for associating text with single ideas, and Text Mode which allows the user to view and revise his document in its final linear form. These four modes and the system as a whole are managed through a relatively trim (compared to version 0) "control panel" Mode.

The environment code controlling classes WRIST1, AGENT1, MODE1 form the 'main program'. WE is supported graphically by the drawing and toolbox tools and conceptually by small 'glue' classes referred to as utilities (and kept in the Utilities category). A document itself is seen by WE as a 'database' – a large information set whose structure preserves the relationships intended by the user. WE's skeletal structure is shown in Figure 1. The strict hierarchy of the classes discussed in detail in this document (and the four WE mode classes) is shown in Figure 2 where classes are represented as rectangles whose relationships to one another expose subclass relationships.

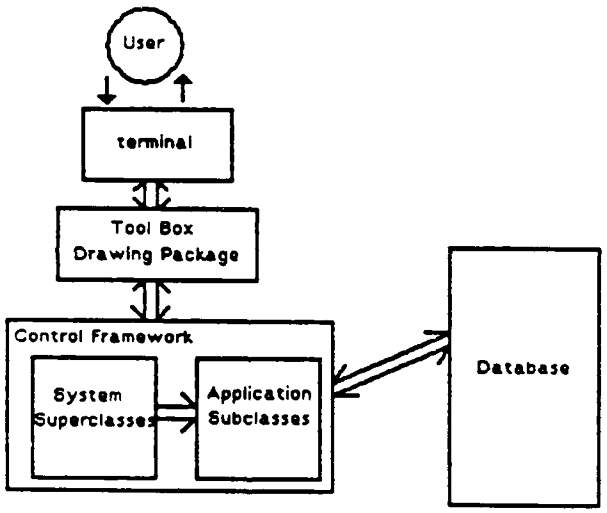


Figure 1: General System Structure

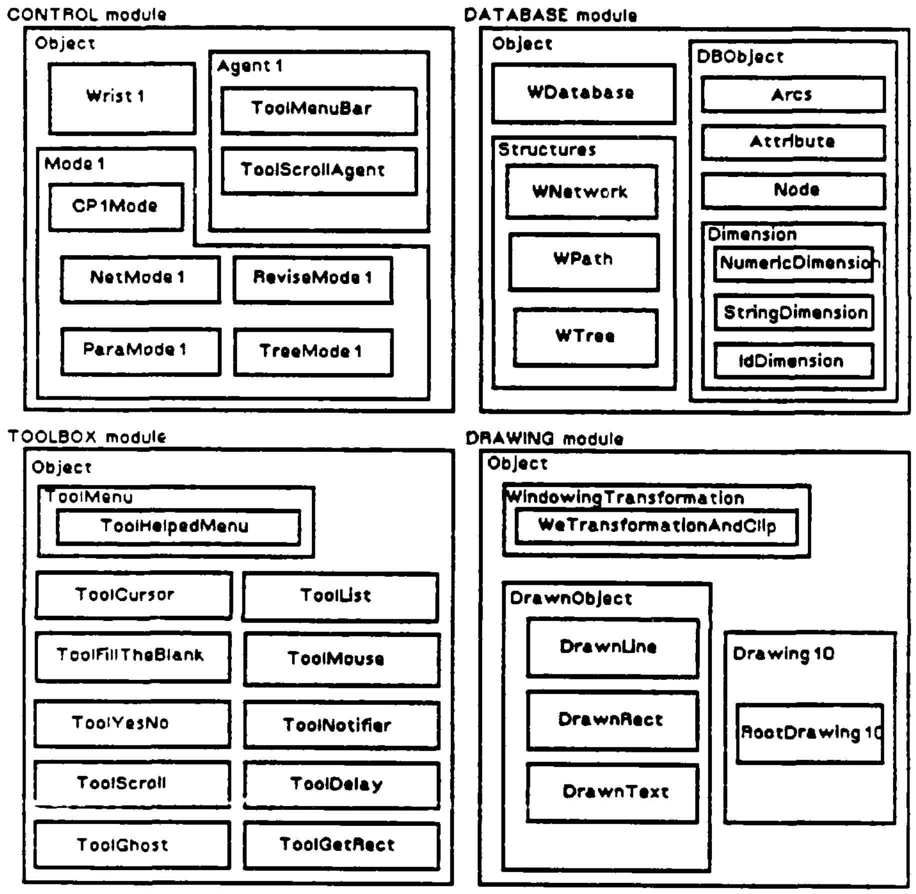


Figure 2: Partial Class Hierarchy

2.2 Flow of Control: the Wrist, Mode, Agent, model

A main goal which motivated the control structure of WE was "vertical integration" - a direct mapping between the concepts on one level and their implementation on the level below. The freedom to represent conceptual objects by actual "objects" in smalltalk is one of the foremost strengths of smalltalk and object-oriented languages in general. Once system design concepts are stable, one can listen to discussion about the behavior of the desired system and pick out the "objects" - the nouns. These nouns can and often should then become objects in the implementation. This mapping of a complete conceptual system into a complete object family (nouns with defined relationships to each other) is an art - a process for which there is yet no algorithm. The real system will only be as good as this original mapping which determines its conceptual integrity.

WE is a system of structured pictures - a directional graph of ideas in one window, a strict hierarchy in another window, a text editor in a third window, and a series of text editors in the forth. The three big nouns are: Wrist, Mode, Agent. They represent three layers of abstraction. An "Agent" is any visual object on the screen (e.g. a directional link in the idea graph, a "node" of the hierarchy picture). A "Mode" is a window. WE has four modes: Network Mode containing the idea graph, Tree Mode containing the hierarchy, Edit Mode containing a text editor, and Text Mode containing a series of text editors. The "Wrist" is the environment controller and manages system-wide communication. These three nouns are mapped directly into superclasses in smalltalk; each specific mode is a subclass of the MODE1 superclass and each specific visual object within the modes is a subclass of the AGENT1 superclass. There is only one Wrist instantiation. This "3 controlling superclasses" structure facilitates code sharing, enforces consistency (tightness of concept), and lays down specific structure skeletons at a high level. Two other advantages of this structure, portability and flexibility, are extremely important to a prototype system.

The high-level structure of WE is shown in Figure 3. The arrows indicate the possible directions of communication.

There are two important features to notice here: 1) Modes cannot directly communicate with each other and Agents cannot directly communicate with each other. 2) The database (the user's document itself) cannot communicate with anything but the external file system - it knows nothing about who is using it. This second point really becomes an advantage when one considers the future

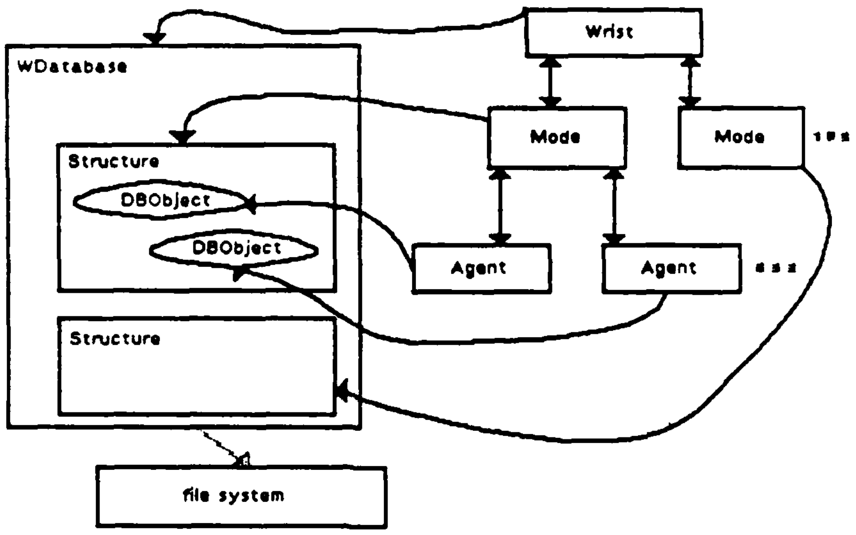


Figure 3: High-level structure of WE: Wrist, Mode, Agent

goal of parallel multiple-user access to the database. A loose exception to the first point is the fact that modes can be coupled together on certain messages in the sense that the message is sent to them both at once.

The main characteristics of the subclasses of the 3 controlling superclasses are as follows:

Wrist - "creates" the environment by initially laying out the modes, does system initialization and termination tasks, drives the WE session, and uses a single database at any given time. Because it coordinates the modes, one can think of it as managing a set of structures (which make up a workspace).

Mode - uses an assigned area of the screen (a window) to present a structure to the user. It is binary (can be either active or inactive at a given time). Only a single mode can be active at any given time. A mode maintains a list of its own agents and responds to the *agentFor: aPoint* message - given a point in space, it answers its corresponding agent.

Agent - represents a single visible object (typically a node or link). It can be "invoked" which means that a relevant operation-choice menu appears for the agent. Agents respond to the *ReactsTo: aPoint* message, giving a boolean answer to the question of whether a given screen coordinate (mouse position) is in its area.

A WE user successively activates modes, invokes agents, and carries out discrete operations (presented as agent menu choices) until the end of the WE session. The pseudo-code "main program" is shown in Figure 4.

```
WRIST
self use the given database and begin;
    self create and display all modes;
    self run
        [while not at the end of the WE session
              poll modes and find one that both contains the mouse and is not hidden;
              activate the mode]
    self terminate;
    (close down each mode and release everything)

MODE (activate)
while not at the end of the session
    [do I contain the current mouse point?;
    if so, poll my agents;
        pass control to my agent containing the mouse point;
        (self agentFor: mousePoint) invoke.
    if not, is user pressing the mouse button in another mode?
        if so, put me to sleep and return control immediately to the wrist
        if not, { no button is being pressed, but the mouse point is in the mode }
              continue looping]

AGENT (invoke)
Is a mouse button pressed?
If so,
    start up my menu on me.
    ToolMouse anyButtonPressed ifTrue: [self class menu startUpOn: self]
```

Figure 4: WE "main program"

3 The WE prototype Environment Code

The "environment code" consists of the controlling classes (WRIST1, MODE1, AGENT1), support classes, and database classes. Each of these classes is briefly discussed below with emphasis on class knowledge and information hiding. The limits of a class's knowledge space are defined by its variables which are its "information holders" and its methods - its capabilities. Thus, each presentation consists of a layout of class variables and instance variables and a short discussion of the important class and instance methods. In the case of some small, simple classes, variables and/or operations are not specifically discussed. The remainder of the WE code - the "user-interface code" - is not covered, although one will be well on his way in understanding the "user-interface" design when he grasps the environment code which is its foundation. For fine details, the programmer should read the well-documented smalltalk code itself. Examples are included in the code for most classes.

3.1 The Controlling Classes

As discussed above, three superclasses *embody the main control loop* of WE: WRIST1, MODE1, and AGENT1. In short, WRIST1 is the driver, instances of MODE1 are the windows, and instances of AGENT1 are the visual objects which the user sees on the screen.

3.1.1 Wrist1.0

The Wrist1.0 category contains WE's three 'main' classes: WRIST1, MODE1 and AGENT1.

WRIST1

The WRIST1 class is the driver. Remember, it has only one instantiation per WE session.
Class Variables:

CurrentDatabase - always points to the current database. It is used in "resuming" work (*self startaNewOn: CurrentDatabase empty: false*). Because it is a class variable, it holds the database even through crashes to smalltalk. (WDatabase)

Running - true if WE is running, false if not (i.e. Running is false when one is programming in the smalltalk environment under WE). The smalltalk Object class uses this variable to know how to handle smalltalk errors: a non-standard error box is used during a WE session. (Boolean)

TexOptions - provides TeX equivalents for WE symbols. (Dictionary)

Instance Variables:

database - the object which holds the user's current document (layout in WE). CurrentDatabase, the class variable mentioned above, is always the same as this instance variable, but is needed because the resume message must be sent to a class. (WDatabase)

activeMode - the currently active mode. (Model)

modes - all modes. (OrderedCollection of Model)

nodeStack - holding area stack for copied nodes. (WeStack1)

structureStack - holding area stack for copied trees. (WeStack1)

validModes - all modes which have been refreshed since the last database change. (Set of Model)

inValidModes - all modes which have not been refreshed since the last database change. (Set of Model)

couples - a record of menu option commands shared by two or more modes. This record facilitates communication between modes. (Dictionary of Symbol (selector), Set of Model)

count - helps the wrist to keep track of new names for text files. These unique file names are of the form w01xxx where xxx is the current value of count. count is incremented to provide the next unused file name. (Integer)

promptForWrite - true when the user wants to be asked before text is saved. (Boolean)

dbChanged - true if database has been changed since the last save. (Boolean)

traceTree - a pointer to a structure which records the WE session for later analysis. (not used in this version). (TrackingTreeShell)

laserWriter - the name of the laserwriter to be used (can be set interactively). (String)

linePrinter - the name of the line printer to be used (can be set interactively). (String)

outFileName - If the autoSend toggle is off, the outFileName is used to indicate the TeX file 'outFileName.tex' or the line printer file 'outFileName.line' into which WE will dump a document sent to a laser or line printer respectively. Its default is 'WEtoPrint'. (String)

autoSend - true if printouts are to be sent directly to the appropriate printer, false if printouts are to be held in a file. (Boolean)

Class Operations:

- instance creation. One instance of Wrist1 exists for an entire WE session (created by *startaNewOn:empty:*). If the user drops back to smalltalk at some point, the wrist must be "restarted" by the *resume* message.

- management of the three class variables. There are class methods to create the TexOptions dictionary, to explicitly release the CurrentDatabase, and to maintain the Running boolean and its related error catching directives.

Instance Operations

- handling the details of session control. The wrist initiates, runs, and terminates the WE session (*initiate, run, terminate*). It also controls the details of allowing the user to stop work on one document and begin work on another (*nowUse:, continueWith:*).

- management of and access to the instance variables. This includes maintaining lists of the currently valid and invalid modes, keeping track of the state of the database, managing the holding area stacks, and keeping the printer specifications.

- handling communication between the modes. This includes 'broadcasting' messages to coupled modes and keeping track of the currently active mode.
- redrawing modes when one mode's size is changed or when the user explicitly requests a redraw of the entire screen.
- doing the extensive initialization required upon instance creation (*startaNewOn:empty:*).
- giving out unique file names for text files when requested (*newFileName*).

MODE1

Subclasses of MODE1 are "windows" in WE. Each handles a structure and is responsible for its basic drawing and layout. When modes are "active", their primary function is to find the agent responsible for handling the current situation and then to pass control to that agent. MODE1 also provides the standard mode menu bar for all its subclasses along with the support of the available standard menu options including the management of size switching.

Class Variables:

LargeWindow - a default size specification for a "large" window (one that fills up most of the screen). This variable is set in the class *initialize* method and used as the default when switching sizes. (Rectangle)

MenuBarHeight - the default height (in pixels) of the standard mode menu bar (see class Tool-MenuBar). This variable is set in the class *initialize* method. (Integer)

Instance Variables:

displayArea - the actual complete mode window (including the standard menu bar area) in screen coordinates. (Rectangle)

agents - a collection of all agents in the mode. The mode polls its agents by searching this list (*agentFor: aPoint*). (OrderedCollection)

backgroundAgent - the agent representing the background of a mode. It is a "default" agent which gains control if no other agents in the mode want control. (Agent1)

drawing - the object which represents the visual mode space excluding the standard menu bar area (a subdrawing of modeDrawing). (Drawing10)

status - the flag which is used in the main WE control loop ((Mode1) *activate*, and (Wrist1) *run*). It can take on a value of #active, #sleep, or #quit depending upon the state of the mode. (Symbol)

field - an object which holds the generic characteristics which apply to agents in this mode (i.e. x and y position, title, etc.). Each of these characteristics is itself an object of type Dimension and is duplicated in a dictionary kept by the database in an instance variable called "dimensions". (WField)

prinStructure - the principle structure handled by a mode. For example, the principle structure of Network Mode is a WNETWORK, and of Tree Mode is a WTREE. These structures are subclasses of class STRUCTURE. (Structure)

wrist - the Wrist1 instance which controls the WE session and acts as the mode manager. (Wrist1)

currPos - the screen coordinate of the most recent 'interesting' event (i.e. the position of the last mouse click). (Point)

menuBarAgent - the object representing the standard mode menu bar at the top of the mode's display area. (ModeMenuBar)

modeDrawing - the object which represents the drawing of the entire mode (including the standard mode menu bar area). (Drawing10)

variableTitle - arbitrary string under program control. Currently in WE, it is the user-given name of the structure being viewed in the mode. (String)

windowControl - a utility object which keeps up with large and small window sizes for 'switch-size' toggling, and manages explicit resizing of the mode. (WeWinPack)

viewport - an aid in managing the drawings representing the mode. It represents the entire area taken up on the screen by the mode (including the standard mode menu bar). viewport is only used to inform the mode (re)creation method of the change in mode size due to a switch size or resize command. (Rectangle)

Class Operations:

- instance creation. Model specifies a standard mode creation method (*principleStructure:field:displayArea:*) which is used by all of its subclasses either solely or in addition to a subclass implemented creation method.
- initialization of class variables.

Instance Operations:

- agent polling -passing control along. *activate*, a method vital to the 'main-program' loop, passes control from the mode to its agent that currently contains the cursor.
- managing the mode's visual area on the screen. This includes altering the menu bar appearance when the mode is "invalid" and allowing the user to interactively change the mode's size.
- sending the mode's structure contents to a file in a format suitable for a line printer or a laserwriter.
- access to and management of the instance variables.

AGENT1

Subclasses of AGENT1 represent the visual objects in a structured picture (i.e. nodes in Network Mode, links in the Network Mode, menu bars, etc.). When the user points at a visual object with the cursor, the object is selected and the agent is invoked. Most agents ignore the invocation unless the mouse button is pressed. If the button is pressed, a menu (specified in the particular subclass of AGENT1 and held in a class variable of this subclass) specific to the kind of object (subclass of

AGENT1) is displayed. The user makes a selection which is communicated to the particular agent. That agent executes the associated method, then passes control back to the mode which continues polling (the next agent is invoked - see Figure 4).

Class Variables: none

Instance Variables:

drawing - the object representing the agent graphically form on the screen. (Drawing10)

subject - the database object that the agent represents. (Object)

mode the MODE1 instance which created the agent and now manages it. (Mode1)

Class Operations:

- instance creation. AGENT1 specifies a generic creation method which most subclasses use (*for:mode:drawOn:*).

Instance Operations:

- starting up a menu on the subject to allow the user to perform operations on it (*invoke*). This method is part of the main control loop of WE.
- field requests as to whether the agent is in charge of a certain given screen coordinate (*reactsTo: aPoint*). The mode uses this method (part of the main control loop of WE) in polling its agents.
- access to and maintenance of the instance variables.

3.2 Support Classes

WE must repeat certain small tasks so many times that it becomes advantageous to subdivide, define, and refine these tasks into "tools" which can be used by the system easily, naturally, and quickly. Thus, WE contains a "drawing package" that provides graphics primitives (i.e. lines, rectangles, text) and a toolkit that contains both tools which prompt the user for information and tools which ease and supplement the execution of operations within the system. These "supporting classes" form a kind of storeroom of functionality available to the WE environment code.

3.2.1 Drawing1.0

DRAWING10

DRAWING10 is the base class for graphics in WE. A DRAWING10 has two basic components: 1) a list of DRAWNOBJECTS - the lines, text, filled rectangles etc. to be displayed and 2) a list of sub-drawings. A drawing is really a tree of drawings; the root of this tree is an instance of ROOTDRAWING10 and the other nodes are instances of DRAWING10. The parent of a node is also called its host. Each DRAWING10 has a local coordinate system offset from its host's. All drawings are the same scale as their parent. Actual display is done by tree traversal of the drawings's contents – lines, text, rectangles etc. Drawings are nested and all display is clipped to the host's viewport.

The drawing options (background and foreground color, line style) are handled with special logic - they are bundled into the WEDRAWINGOPTIONS class. There is a default WEDRAWINGOPTIONS instance for the DRAWING10 class. Drawings either inherit the options of their parent or use a local set. If they inherit, then their 'options' variable is nil. Inheritance of options (as specified in the design) saves on space for drawings. Most drawings inherit options and require only the single nil pointer to record this.

Class Variables:

DefaultOptions - an object that holds the default drawing options for parameters like color, line style and text style. This default is set up in the DRAWING10 *initialize* method. (WeDrawingOptions)

Instance Variables:

parent - the host drawing. (Drawing10)

offset - the distance from this particular drawing's origin to its host's origin. offset is expressed in terms of x and y distances. (Point)

extent - the size of the drawing expressed as width(x) and height(y). (Point)

subords - a collection of all subdrawings of this drawing. (OrderedCollection)

contents - the collection of the actual graphical objects which make up this drawing. (Ordered-Collection)

outlined - true if the drawing is outlined. (Boolean)

subject - the database object which this drawing represents. Access to this information is necessary because many times drawings are used as 'indexes' for deciding whether an agent reacts to a given point or not - it does react if its drawing contains the given point.

options - a collection of drawing options which can be set to nil to adopt the default options (see the class variable, DefaultOptions). (WeDrawingOptions)

Class Operations:

- instance creation. Drawing10 has two methods for creating a new drawing (which must by definition be a root drawing): *createAt:*, and *displayBox:viewport:*.
- setting up the default drawing options.

Instance Operations:

- instance creation. The method *subDrawingAt:* creates an instance of DRAWING10 which is to be a subdrawing of the receiver drawing.
- managing the drawing options (e.g. background color, line style, text style).
- switching the outline toggle and creating graphics primitives (lines, texts, and rectangles).
- erasing and deleting the whole drawing tree or erasing and deleting single elements of the drawing tree.
- displaying itself on the screen - clipped or unclipped (see *display* and *displayIn:*).
- access and management of the instance variables having to do with the position of the drawing on the screen.

- transforming given points from drawing coordinates to screen coordinates and vice-versa.
- responding to the *containsPoint:* message – essential to the part of the WE main control loop which deals with successive agent invocation.
- moving itself to another screen location.
- accessing the tree of drawings.
- "indexing". association of a database object with the drawing. The database object becomes the subject (see instance variable, subject) of the drawing. A DRAWING10 can also return the appropriate drawing and/or subject for a given screen point.

ROOTDRAWING10

As a subclass of DRAWING10, a ROOTDRAWING10 is simply a special kind of DRAWING10. Specifically, it is the DRAWING10 that is at the root of a tree of drawings. Its capabilities are the same as those of DRAWING10 with the exception of the reimplementation of some methods because of this drawing's unique place in a tree of drawings. Its host, for example, is always nil and it answers in the affirmative when asked if it is a root drawing. It has one instance variable, displayBox, that holds its viewport in screen coordinates. It has no class variables of its own.

DRAWNOBJECT

Each member of the 'contents' orderedCollection of an instance of DRAWING10 is a DRAWNOBJECT. A DRAWING10 is displayed by displaying each one of its 'contents' in turn. (In the same way, a DRAWING10 which represents a tree of drawings is displayed by rendering each content of each DRAWING10 tree node.) Thus, the graphical objects that the user actually sees on the screen are instances of DRAWNOBJECT. All subclasses of DRAWNOBJECT work with a pair of points. These points are used in various ways; typically they define the origin and extent of a rectangle specifying an area for drawing in local coordinates. This rectangle marks the border of a drawn rectangle or the composition rectangle for text. Alternatively, the two points can mark the endpoints for a drawn line (contained in a virtual rectangle). All subclasses of DRAWNOBJECT can actually draw themselves on the screen.

Class Variables: none

Instance Variables:

origin - the origin of the rectangle in the coordinate system of its parent drawing (local coordinates). (Point)

extent - the width and height of the rectangle expressed in terms of x and y relative to the origin point. (Point)

color - the color of the rectangle expressed as a numeric code. (Integer)

lineStyle - the size (thickness expressed in terms of x and y) of the "pen" used to draw the rectangle. (Point)

Class Operations:

- instance creation. *origin:extent:*

Instance Operations:

- management of and access to the instance variables.

DrawnLine

DrawnLine is the subclass of DrawnObject for lines. A DrawnLine keeps track of its size and color and its position in its host drawing. It knows how to draw itself.

Class Variables: none

Instance Variables: none

Class Operations:

- instance creation. Given the two endpoints as parameters, the *from:to:* method creates an instance of DrawnLine which represents a line connecting the two points.

Instance Operations:

- drawing the line on the screen.
- responding to the *direction* message by answering its direction (north, northeast, west, etc.) with respect to a standard cartesian coordinate system.

DrawnRect

DrawnRect is the subclass of DrawnObject for rectangles. A DrawnRect keeps track of its dimensions, color, drawing style (outlined or painted with a color), and its position in its host drawing. It knows how to draw itself. Note that a DrawnRect may be outlined or filled, but not both.

Class Variables: none

Instance Variables:

filled - true if the rectangle is to be filled with color. (Boolean)

DrawnText

DrawnText is the subclass ωι DrawnObject for text. An instance of DrawnText is a rectangular region filled with text. A DrawnText keeps track of its dimensions, color, text style, and its position in its host drawing. Unlike the other DrawnObjects, it cannot be scaled. It knows how to display itself.

Class Variables: none

Instance Variables:

textStyle - the style of text expressed as a numeric code. (Integer)

text - the string of characters which the DrawnText represents. (String)

WeTransformationAndClip

WeTransformationAndClip supports translation and scaling for Drawing10. It is a subclass of the Smalltalk class WindowingTransformation. It adds support for a clipping rectangle.

Class Variables: none

Instance Variables:

clipBox - the rectangle (given in local coordinates) to which the Drawing10 should be clipped after transformation. (Rectangle)

3.2.2 ToolBox

Classes in the ToolBox category are the tools which support the interface between the user and the system. Tools exist for monitoring cusor movement and mouse button action as well as for informing and prompting a user.

ToolCursor

ToolCursor is a class that manages and provides cursors. All cursors used in WE are held in class variables of ToolCursor. Also, a user may define a cursor not already in the system. An instance of ToolCursor is a single cursor. The cursor itself is assigned to the instance variable 'cursor'. The cursor's offset - its sensitive point - is kept in the instance variable 'offset'. ToolCursor is dependent on the smalltalk system class, Cursor.

There are three ways of obtaining a new cursor: 1) by calling one of the system cursors or 2) by explicit definition or 3) by creating one from a 16 by 16 Form. A cursor is (and must ALWAYS be) a 16 by 16 bitmap.

ToolCursors can be used as new "permanent" cursors or they can be shown while a given code block is executed or while a given code block is true by using the *makePermanent, showWhile: aBlock*, and *showWhileTrue: aBlock* methods respectively.

A class method that returns the current cursor (*currentCursor*) is provided to facilitate saving the current cursor so that it can be reinstated later.

ToolMouse

ToolMouse allows access to mouse button status and cursor position. It is really just an interface for WE to the smalltalk system class InputSensor. Thus, ToolMouse has no instance methods of its own. The only difference between ToolMouse and InputSensor is that in ToolMouse the three mouse buttons are referred to as left/middle/right, wheras in InputSensor, they are referred to as red/green/blue.

"Sensor", a system global variable, is an instance of class InputSensor. ToolMouse passes along all its messages to this object, so the real implementations of the methods are in InputSensor instance methods.

ToolMouse capabilities fall into three groups:

1. positioning - *mousePoint, mousePoint:* A user can find out the current cursor position and set the current cursor position.

2. testing - *anyButtonPressed, noButtonPressed, leftButtonPressed, middleButtonPressed, rightButtonPressed.* A user can recieve boolean answers as to the specific state of the mouse buttons at any particular time.

3. waiting - *waitButton, waitClickButton, waitClickLeftButton, waitNoButton.* A user can direct the application to wait for certain mouse events to happen before continuing.

These capabilities allow the user to "program" the mouse buttons to affect his application in prescribed ways. WE, for example, is programmed to make no distinction between the three mouse buttons - it is essentially a "one-button" application.

ToolNotifier

A ToolNotifier can be used to present short pieces of information to an application user. The information appears as a text string in a white rectangle on the screen. One may easily specify the position on the screen where the message is to appear. One may create an instance of ToolNotifier that will be an object to be reused many times (using *message*), or one may request that a certain message be shown only once at the mouse point and then erased (using *show:*). Note: ToolNotifier is dependent on WE class Drawing10.

Class Operations:

- instance creation. as specified in the above paragraph

Instance Operations:

- displaying. The ToolNotifier can be shown at a particular point on the screen.

- erasing. The ToolNotifier can be erased from the screen.

ToolYesNo

A ToolYesNo can be used to ask the application user a yes/no question. It returns a boolean answer. A ToolYesNo appears on the screen as a three-part rectangle. It contains a question, and an actual Yes box and No box. The Yes and No boxes can be activated with a mouse click to indicate user choice. ToolYesNo is dependent on WE class Drawing10.

Class Operations:

- instance creation. The ToolYesNo only requires the intended question, a string of characters, for creation (*question: aString*).

Instance Operations:

- displaying. One may show a ToolYesNo on the screen at a particular screen coordinate location (*displayAt: aPoint*). The ToolYesNo erases itself and returns the appropriate boolean answer when the application user responds.

ToolFillTheBlank

ToolFillTheBlank is used to acquire a string from the application user. It is an interface to the smalltalk system class FillInTheBlank. Thus, it has no instance methods of its own. A ToolFillTheBlank appears on the screen as a two-part rectangle. It contains a question or prompt of some kind and a small editor which accepts an input string. The application user's response is returned from the tool as a string.

Class Operations:

- instance creation. One can specify up to three parameters for a ToolFillTheBlank: a message string, an initial answer, and a screen coordinate at which to display the tool. The message string is required, but defaults will be used for the initial answer and screen coordinate if they are not specified. So, the most specific form of creation is *message: aString initialAnswer: answerString displayAt: aPoint.*

Instance Operations: none

ToolMenu

ToolMenu supports the definition and control of pop-up menus. ToolMenus hold lists of application operations. These operations correspond one-to-one with application messages. When an application user selects an option from the menu, the corresponding message is sent immediately to the object which the menu was "started up on".

Class Operations:

- instance creation. One defines a menu with labels, separation lines, and selectors. Labels are the actual 'choice words' (usually each referring to an application operation) which appear on the visual menu. Separation lines are simply specifications of positions in the menu where black, horizontal lines should be shown between choice words. Selectors, one per label, must be the names of existing application messages. Messages from a single menu must belong to the same class and may not have any arguements. It is not possible to include messages from different classes in a single menu selector list. The basic instance creation message, then, is *label: aString lines: anArray selectors: selArray.*

Instance Operations:

- displaying. *startUpOn: anObject* displays the menu and sends the message associated with the user's selection to anObject. anObject can be anything that understands all the messages in the menu's selector list.

ToolList

ToolList is a general tool that presents a list of objects, allows the user to select one, and then pops up a menu from which the user can select an operation to perform on the selected object. The choice objects are sent to the tool in an ordered collection along with a message that each object understands (usually one which returns the name of the object). The objects are represented in the list by the strings which they return upon receiving the message. ToolList also has a title bar and a menu for operating on the titlebar's associated object.

Class Operations:

- instance creation. One can specify up to eight parameters for a TOOLLIST. The creation message is ***size:* aPoint *onList:* anObjectList *mesg:* aSymbol *menu:* aMenu *titleObject:* anObject *titleMessage:* a2Symbol *titleMenu:* a2Menu**. The parameters are (respectively): a screen coordinate at which to place the tool's upper left-hand corner, a collection of objects to include in the list, a message to send to each object to get its representation string, a list menu (optional), an object for the title bar, a message to send to the title bar object to get its representation string, and a title bar menu (optional).

Instance Operations:

- controlling. One activates a TOOLLIST by passing it control (*takeControl*). A TOOLLIST displays itself on the screen, allows the application user to select an object and activate a menu on that object, and then erases itself from the screen.

TOOLLIST is the most adaptable tool in the ToolBox. In WE, for example, it is used both as a selection tool for changing workspaces and as a visual stack and stack manager for the holding area stacks.

TOOLSCROLL

TOOLSCROLL supports creating, displaying, and reading of vertical scrollbars. A TOOLSCROLL knows nothing about the window it is associated with. Window content information must be provided by the application programmer at a higher level. TOOLSCROLL is less powerful than the smalltalk system scrollbar because the application programmer must handle its control sequence and the actual moving of the window contents (a specific message is sent to the class to accomplish this). However, TOOLSCROLL has the advantage of being system independent.

Class Operations:

- instance creation. *create* (no parameters are required).

Instance Operations:

- accessing. Methods exist for getting information from a ToolScroll which may be essential for the application programmer to manage its control. The position of the window contents of the window associated with the ToolScroll can be set (*top:visible:*).
- activating. One can pass control to a ToolScroll.

3.2.3 ToolBox-we

The tools discussed above are generic information-retrieving tools which have wide application in some form in general graphical interactive systems. In contrast, the tools in the ToolBox-we category are more specific to the Writing Environment. They are dependent on other WE classes. They are extras that aid in the fine tuning of WE. Still, they may be helpful in general graphical applications.

TOOLMENUBAR

Visually, a TOOLMENUBAR is a strip of titled areas each of which presents a pull down menu when activated. WE uses subclasses of TOOLMENUBAR both for session control as its Control Panel and for mode control as a standard in each of its four modes. TOOLMENUBAR is a subclass of AGENT1; an instance is created in a drawing and associated with a mode. All specific capability is implemented in subclasses.

Class Operations: There are no class methods - therefore, superclass instance creation methods are used.

Instance Operations:

- activation. Because it is an AGENT1, a TOOLMENUBAR can be invoked (*invoke*). Within the main control flow in WE, this invocation happens when the mouse point is inside the menu bar's area. As with any AGENT1, when a TOOLMENUBAR is invoked its associated menus become available to the user.

TOOLHELPEDMENU

TOOLHELPEDMENU adds help to its superclass, TOOLMENU. When a WE user selects a menu option with the system 'command key' pressed, a help message specific to the option selected appears in lieu of the operation execution. In smalltalk on the SUN, the magic key is actually a combination: both the 'left' and the 'Shift' keys must be pressed together. A TOOLHELPEDMENU holds an appropriate set of help strings. It overrides the TOOLMENU *startUpOn:* message to test for a help request.

Class Operations:

- instance creation. ToolHelpedMenu uses its superclass's creation method with one additional parameter: the array of help strings.

Instance Operations:

- displaying. Like a ToolMenu, a ToolHelpedMenu can be 'started up on' an object (*startUpOn:*).

ToolGhost

ToolGhost draws a 'ghost line' between two given points. A 'ghost line' is a line which is drawn by reversing each pixel that will be covered by the line. This capability is especially useful in WE for drawing links in the Network Mode; as the WE user is moving the mouse to select the end node for the link, the link is successively drawn and erased with ToolGhost. This gives the desired effect of a pulsating, faint guideline.

Class Operations:

- instance creation. ToolGhost requires at least two parameters: the two line endpoints. A clipping rectangle is an optional parameter. Thus, the most specific instance creation method is *from: aPoint to: aPoint clip: aRectangle.* Extra methods which make stars from ghost lines are also included. These methods draw lines from a given point p0 to each of the points in a collection (*starFrom: p0 to: aCollection*). Here again, a clipping rectangle is optional.

Instance Operations: none

ToolGetRect

ToolGetRect is simply an interface between WE and the smalltalk system class Rectangle.

Class Operations:

- return a rectangle specified by the application user. The *get* message gives the application user a chance to delimit a rectangle on the screen. The user is given a corner prompt which he can use to indicate the top left-hand and lower right-hand corners of a rectangle. The TOOLGETRECT returns this defined rectangle.

Instance Operations: none

TOOLDELAY

TOOLDELAY is simply an interface between WE and the smalltalk system class DELAY.

Class Operations:

- freezing the application flow of control. The *seconds:* method takes an integer parameter. Once given this parameter, TOOLDELAY causes all action to freeze for the specified number of seconds.

Instance Operations: none

TOOLSCROLLAGENT

TOOLSCROLLAGENT packages the vertical scrolling function provided by TOOLSCROLL (see section 3.2.2). It is a wrapper which makes TOOLSCROLL easy to use within the Wrist/Mode/Agent paradigm. TOOLSCROLLAGENT is an Agent itself; when it is invoked, it sends a series of *scrollTo:* messages to a specified target object representing the contents of the window to which the scroll bar is attached.

Class Operations:

- instance creation. The creation message is *for: aTarget mode: aMode drawOn: aRectangle imageSize: anInt.* aTarget is the object that will be scrolled. aMode is the associated WE mode. aRectangle defines the size of the window corresponding to aTarget. anInt is the amount (percent) of aTarget's contents which can fit in its associated window at one time.

Instance Operations:

- invocation. Like all AGENT1s, a TOOLSCROLLAGENT responds to an *invoke* message. Upon invocation, a TOOLSCROLLAGENT passes control to its scroll bar (a TOOLSCROLL). A TOOLSCROLLAGENT is invoked only when the mouse point is within 5 pixels of its target's edge (*reactsTo:*).

3.3 Database Construction

Traditionally, the word "database" connotes a large body of information organized in some fashion that allows fast and specific access. WE's concept of a database is much the same at a high level, but its implementation is quite unique and natural to the kind of system that WE is.

WE's database provides an organizational structure for fast and specific access to the information which *is* a document. But further, it is a semantic network - a direct mapping of WE's visual objects into database primitives. WE's database is its internal object-oriented view of its body of information. The fundamental units or 'primitives' are nodes and arcs. These correspond directly to the nodes and links which the user sees in Network Mode, but are not limited to that use.

At the upper level of the database are the structures. They are made up of the lower primitives - nodes and arcs. Structures are similar to 'abstract data types' in procedural languages in that application programmers do not have to concern themselves with the internals of a structure's implementation. A structure is an encapsulation of details. A programmer simply asks (by sending a message) a structure to add a given node, delete a particular arc, etc. and the structure itself maintains its cohesiveness during these adjustments. In WE, structures organize subsets (usually overlapping) of the information in the database in constrained ways. Structures may share nodes, but not arcs. In this sense, structures partition the set of arcs in the database. One works with visual structures in Network Mode and Tree Mode. Typically, structures correspond one-to-one with Modes.

This section presents the classes which comprise the WE database; the lower level (**Database1.0** category) followed by the upper level (**Structures** category). This presentation is couched between a section dealing with terminology clarification and a section covering the external (Unix) representation of WE's database.

3.3.1 Database vs. Workspace

As discussed above, WE's internal view of its body of information is referred to as its 'database'. From a WE user's point of view, the body of information comprising a document is called a 'workspace'. In WE version 1.0, 'database' and 'workspace' are two terms for the same body of information. The distinction between WE's view of a document and a user's view of a document is a necessary one; 'database' refers to the actual interconnections of database objects which represent the document, whereas 'workspace' refers only to the user's conceptual idea of what that document is. In future versions of WE, the database may grow to include many workspaces. Then, the distinction will be more apparent.

3.3.2 Database1.0

The classes in this category are the components of the lower level database. All the objects in the database are subclasses of DBOBJECT; they are Objects that have a unique numerical identifier. The following brief description explains in general how to access these objects and the information associated with each; see the classes involved for more detailed discussion of the full range of functions.

An application wishing to access (read and/or update) the contents of a database begins with an instance of WDATABASE. Such instances are created (typically) by test drivers or by 'changing' to a WDATABASE which is retrieved and reconstructed from an external file. An instantiation of WDATABASE provides access to three high level support dictionaries:

1. A Dictionary of structures. All structures managed by the WE session are kept in this dictionary. The programmer must know the name of a structure to access it. These structures provide access to nodes and arcs.

2. A Dictionary of Dimensions. Dimensions might well have been called attributes. Nodes respond to messages regarding their 'position' in a particular dimension. Entries in the dimension dictionary typically include "x" and "y" - numeric dimensions used to locate the node in the plane, and "title" - a string valued dimension which locates the node in a name space. In fact, DIMENSION is an abstract superclass of these typed dimensions, e.g. STRINGDIMENSION.

3. A Dictionary of Attributes. As nodes have position in various dimensions, so arcs have values on attributes. These too are named.

A programmer can create a database by creating an instance of WDATABASE. Then, he can access the information in the database by looking up database objects in the three high level dictionaries and using the methods from corresponding classes in the Database10 and Structures categories.

DBObject

DBObject is the superclass of all objects in the database. It simply attaches a unique (for all users) numerical identifier to all its instances. This allows DBObjects created in one session to be referred to and used in other sessions without ambiguity. All DBObjects created from information in an external file retain their original identifiers. DBObjects do not know about the database they belong to during a particular WE session.

Class Variables:

Counter - Counter insures the uniqueness of numerical identifiers. It is used and accessed only by the system. (Integer)

Instance Variables:

uniqueId - the unique numerical identifier assigned to a database object.

Class Operations:

- instance creation. All subclasses (and therefore all database objects) inherit and use the creation method implemented here (*new*).

Instance Operations:

- access to the unique identifier. Access is provided by the message *uniqueId*.

Dimension

Dimensions associate values with nodes. Nodes are positioned in an information space. The Dimensions in that space are of various kinds, each a subclass of Dimension. So, for instance, the title of a node is, formally, the position of the node in a StringDimension known as "title". A Node has a 'position' dictionary which keeps track of its individual values in the information space and their corresponding Dimensions. Similarly, subclasses of DIMENSION keep a 'nodes' dictionary which keeps track of all the nodes which have a value (position) defined in that particular dimension, and the specific values themselves. These two dictionaries are redundant. Both are needed in version 1.0 only to simplify the process of saving a document to an external file - otherwise, the Dimension 'nodes' dictionary would be unnecessary.

Class Variables: none

Instance Variables:

nodes - a dictionary whose keys are nodes with positions in the dimension and whose values are the specific values that these nodes take on in the dimension.

Class Operations:

- instance creation. All subclasses inherit and use the instance creation method (*new*) implemented here.
- storage to and reconstruction from an external file. DIMENSIONs can be stored onto and retrieved from the external file containing a saved database (see section 3.3.4).

Instance Operations:

- access to the nodes in this dimension. *nodes* returns the nodes dictionary (see above). *nodesBetween: lowValue to: highValue* returns a Set of nodes whose values lie between lowValue and highValue (exclusive) in this dimension. *nodesAt: aValue* returns a Set of nodes whose value is aVal in this dimension.
- insertion into the nodes dictionary. *move: aNode to: aPosition* places aNode into the nodes dictionary with a position of aPosition in the dimension.

IDDIMENSION

An IDDIMENSION is a DIMENSION whose values will be file names. WE uses IDDIMENSIONs to associate text files with particular nodes. The actual name of the external file containing a node's text is its position in an IDDIMENSION.

Class Variables: none

Instance Variables: none

Class Operations: none except those inherited.

Instance Operations:

- comparison. *distanceFrom: value1 to: value2* returns a closeness rating specifying how close the two given values are with respect to this dimension.

NumericDimension

A NumericDimension is a Dimension whose values are numbers. For example, WE stores a nodes's x and y positions in NumericDimensions. There is a NumericDimension for x positions of nodes and one for y positions.

Class Variables: none

Instance Variables: none

Class Operations: none except those inherited.

Instance Operations:

- comparison. *distanceFrom: num1 to: num2* returns a rating of the closeness of num1 and num2 with respect to this dimension. For NumericDimensions, this measurement is simple: the closeness rating is (num1 - num2).

- reconstruction from an external file. NumericDimensions can reconstuct themselves from information listed at a certain point in the database's external file (see section 3.3.4).

StringDimension

A StringDimension is a Dimension whose values are strings. For example, WE uses a StringDimension to hold names of nodes.

Class Variables: none

Instance Variables: none

Class Operations: none except those inherited.

Instance Operations:

- comparison. *distanceFrom: value1 to: value2* returns a rating specifying the closeness of value1 and value2 with respect to this dimension. For STRINGDIMENSIONs, this measurement is binary: The rating is 0 if the value1 and value2 strings are identical, and 1 if not.
- reconstruction from an external file. STRINGDIMENSIONs can reconstruct themselves from information listed in the external file for the database.

NODES

NODEs are the fundamental information unit in the database. They are the direct mapping of the visual nodes which a WE user works with in Network Mode and Tree Mode. NODEs have position in DIMENSIONs which define their location in the information space. A NODE's positions in various DIMENSIONs are kept organized in a 'position' dictionary. See figure 5 below for its structure. NODEs are linked together into structures by arcs. A NODE keeps track of the arcs attached to it through adjacency lists; each node knows the arcs coming into and going out of it.

The basic methods thus involve moving the node about in the information space, attaching and detaching arcs, and answering inquiries about position. Note that positional information is duplicated in the current implementation of the database: each DIMENSION has a 'nodes' dictionary associating nodes with their positions in the particular DIMENSION.

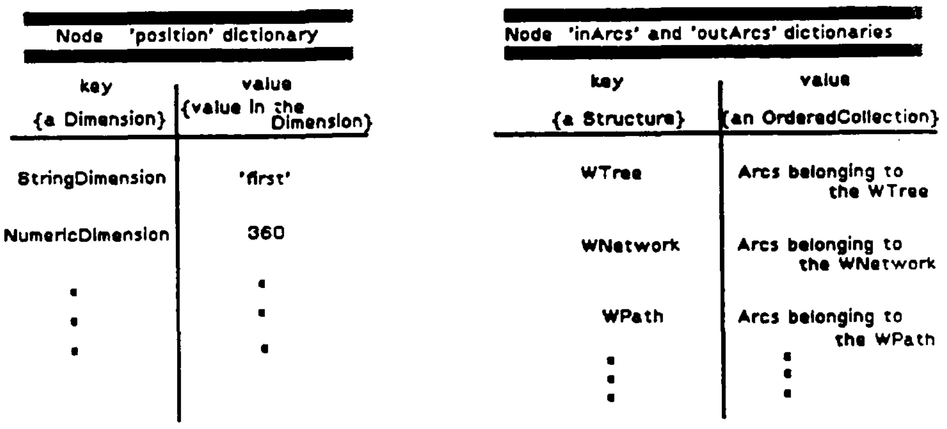
Class Variables: none

Instance Variables: (see figure 5 for structures of these dictionaries.)

position a dictionary whose keys are DIMENSIONs and whose values are the node's positions in these DIMENSIONs. This dictionary only contains DIMENSIONs in which the node has a defined position. (Dictionary)

outArcs a dictionary whose keys are STRUCTUREs and whose values are the node's exiting arcs which are part of a particular structure. (Dictionary)

inArcs a dictionary whose keys are STRUCTUREs and whose values are this node's entering arcs which are part of a particular structure. (Dictionary)



Node 'position' dictionary

key {a Dimension}	value {value in the Dimension}
StringDimension	'first'
NumericDimension	360
.	.
.	.
.	.

Node 'inArcs' and 'outArcs' dictionaries

key {a Structure}	value {an OrderedCollection}
WTree	Arcs belonging to the WTree
WNetwork	Arcs belonging to the WNetwork
WPath	Arcs belonging to the WPath
.	.
.	.
.	.

Figure 5: The Dictionaries kept by a NODE

Class Operations:

- instance creation. NODEs use the creation method inherited from DBOBJECT, but they also do initialization of their instance variables upon creation.
- reconstruction from an external file. NODEs can be reconstructed from information at a certain point in the database's external file.

Instance Operations:

- take on a position in a given DIMENSION. *moveTo: aPosition in: aDimension* enters a new DIMENSION x position pair into the node's 'position' dictionary. The node's new parameter in the information space is cross-referenced in the 'nodes' dictionary for aDimension.

- answer inquiries about position. *positionIn: aDimension* returns the node's position in the given DIMENSION, aDimension.

- addition of entering and/or exiting arcs. *addInArc: anArc* inserts anArc into the 'inArcs' dictionary. anArc knows what structure it belongs to - this information is needed for correct insertion in the dictionary. Similarly, *addOutArc: anArc* inserts anArc into the 'outArcs' dictionary. These methods are private – they are used by Arcs to keep node dictionaries current.

- access to all the node's arcs belonging to a given structure. *inArcsFor: aStructure* returns a collection of the arcs which enter this node and belong to the given STRUCTURE, aStructure. Similarly, *outArcsFor: aStructure* returns a collection of the arcs which exit this node and belong to aStructure.

- access to every arc associated with the node. *inArcs* returns a collection of all the arcs which enter this node. Similarly, *outArcs* returns a collection of all the arcs which exit this node.

- removal of an arc from the node's adjacency lists. *removeInArc: anArc* disconnects anArc (an entering Arc) from the node. The sender must disconnect anArc's other end and mend the structure appropriately. Similarly, *removeOutArc: anArc* disconnects anArc (an exiting Arc) from the node. These methods are private – they are used by Arcs to keep node dictionaries current.

- making a copy of itself. *copy* returns a copy of the node - a node which has the exact same positions in the exact same DIMENSIONs as this node. *copy* does not make copies of text files associated with this node.

- deletion. *delete* removes all trace of this node from the database - permanently.

ARCS

ARCs are database objects which represent relationships between NODEs. An ARC belongs to a single, particular STRUCTURE. Thus, ARCs give the NODEs they connect a place in a particular STRUCTURE. ARCs can have attached ATTRIBUTEs. ATTRIBUTEs are database objects. They are like DIMENSIONs in that ARCs may 'have a position' on an ATTRIBUTE. The values which ARCs have in ATTRIBUTEs are strings. For example, one ATTRIBUTE is the 'label' attribute which represents the name (a string) of the ARC (the WE user can name arcs in Network Mode). An arc's Attribute x y value pairs are kept in its 'attributes' dictionary. See figure 6 below for the structure of this dictionary.

Class Variables: none

Instance Variables:

from - the Node from which the Arc originates. (Node)

to - the Node at which the Arc ends. (Node)

attributes - a dictionary whose keys are the attributes in which the arc has a position and whose values are the positions themselves. (Dictionary)

structure the Structure owning the arc. (Structure)

Arcs 'attributes' dictionary

key {an Attribute}	value {a String}
an Attribute labeled 'tree'	'root'
an Attribute labeled 'tree'	'child'
⋮	⋮

Figure 6: The Single Dictionary Kept by an ARC

Class Operations:

- instance creation. *newIn: aStructure from: fNode to: tNode* creates an arc originating from node tNode and ending at node fNode and belonging to the structure aStructure.
- external file storage and retreival. Methods exist to store an arc on the external database file and to reconstruct an arc from the information at a certain point in the external database file.

Instance Operations:

- access to the arc's knowledge. *from* returns the node from which the arc originates. Similarly, *to* returns the node at which the arc ends. *attribute: anAttribute* returns the value associated with anAttribute for this arc (this arc's position in anAttribute).
- attachment of an attribute. *attribute: anAttribute value: aValue* associates anAttribute with this arc and sets the arc's value in anAttribute to aValue.

ATTRIBUTE

ATTRIBUTEs are database objects which only have meaning when associated with an arc. Arcs have values in attributes. Attributes are named and respond to messages about the name. An example of an attribute is the 'label' that links (arcs) have in the Network Mode – the arc's position in this attribute is what a user thinks of as the link (arc) name.

Class Variables: none

Instance Variables:

label the name of the attribute. (String) (Note to WE programmers - it is EXTREMELY confusing to call the name of the attribute its label and to have a 'label' attribute - an attribute whose name is 'label')

Class Operations:

- instance creation. *labeled: aString in: aDatabase* creates a new Attribute named aString belonging to the database aDatabase.

- storage to and retrieval from a file. Methods exist for storing an Attribute on the external database file and for reconstructing an Attribute from information at a certain point in this file.

Instance Operations:

- access to the attribute's knowledge. *label* returns the name of the attribute.

- changing the name. *label: aString* changes the name of the attribute to aString.

WDatabase

WDatabase represents the database (WE's internal view of the document) and is also the central point in the mapping between the database and the external database. A WDatabase keeps three dictionaries which in total contain all database objects in the database. WE uses these dictionaries as lookup tables when doing database operations. WDatabase also provides access to a template 'empty database' which WE gives to its users as a 'clean slate' to start a document.

Class Variables: none

Instance Variables: (see figure 7 for the format of the three main dictionaries.)

structures - a dictionary which keeps track of all Structures in the database and their names. (Dictionary)

dimensions - a dictionary which keeps track of all Dimensions in the database and their names. (Dictionary)

attributes - a dictionary which keeps track of all Attributes in the database and their names. (Dictionary)

dbTitle - the name of the database itself (user-given). (String)

WDatabase 'structures' dictionary

key {a String}	value {a Structure}
'net 29 1paper'	WNetwork
'tree one'	WTree
⋮	⋮

WDatabase 'dimensions' dictionary

key {a String}	value {a Dimension}
'x'	NumericDimension
'y'	NumericDimension
'title'	StringDimension
⋮	⋮

WDatabase 'attributes' dictionary

key {a String}	value {an Attribute}
'tree'	an Attribute labeled 'tree'
'label'	an Attribute labeled 'label'
⋮	⋮

Figure 7: The Three Dictionaries Kept by a WDATABASE

Class Operations:

- instance creation. *emptyWS* creates a database for an empty workspace. This empty workspace is the clean slate with which a WE user begins a document (workspace). Its database has an empty tree and an empty network. *newNamed: title* creates a new WDatabase named title. This creation method is only used when reconstructing a database from its external file.

Instance Operations:

- access to the database's knowledge. *attributes* returns the dictionary containing all the attributes in the database. *dimensions* returns the dictionary containing all the dimensions in the database. *structures* returns the dictionary containing all the structures in the database. *dbTitle* returns the name of the database. *rename: dbName* changes the database's name to dbName.

- generation of an empty workspace. *emptyWS* sets up and returns the framework for an empty database which the WE user needs to begin a new workspace.

- storage on and reconstruction from a file. A WDatabase has the ability to store itself on an external file and reconstruct itself from an external file (see also section 3.3.4). *storeOn: filename* causes the database to store itself onto an external file by successively telling all its objects (ATTRIBUTEs, DIMENSIONs, and STRUCTUREs) to store themselves. *scanFrom: aFileStream* causes the database to reconstruct itself by successively reconstructing its objects and telling them to reconstruct themselves.

3.3.3 Structures

Classes in the Structures category form the framework for the upper level of the database. STRUCTURE is the superclass for all structures; it is a template on which structures are modeled. Access to the information contained in a structure is almost entirely provided in the subclasses. This allows the subclasses to enforce the constraints which give them their character. Structures are simply non-primitive database objects (still subclasses of DBOBJECT). If one thinks of the database as a single graph, then a structure is a specific subgraph. Structures consist of a set of nodes and a set of arcs which link elements of the node set. The default implementation keeps an explicit list of nodes, but some subclasses do not. A structure knows the database it belongs to and its own title. The standard set of subclasses used by WE consists of WTREE, WPATH, and WNETWORK.

Most subclasses either access nodes by following arcs from a head arc or by maintaining an explicit node list. Every structure responds to a *do: aBlock* message which performs the *aBlock* code on every node in the structure. This way of gaining access to each individual node of a structure is crucial for display: recall that in WE, every mode has an associated STRUCTURE which is kept in its **prinStructure** instance variable. When a mode wants to derive its structure from the database, it sends the *do: aBlock* message to its principle structure. The *aBlock* code creates an appropriate AGENT1 (really a customized subclass of AGENT1) for each node or other DBOBJECT in the structure. It assigns the particular DBOBJECT (node, arc, etc.) as the agent's subject, the mode that the structure belongs to as the agent's mode, and a created DRAWING10 (a subdrawing of the mode's ROOTDRAWING10) as the agent's drawing. Once the collection of agents corresponding to a structure is built up, the mode's drawing (ROOTDRAWING10) is sent the *display* message which causes the drawing and all its subdrawings (corresponding to individual agents) to be displayed on the screen.

For example, the WE Network Mode has a WNETWORK as its principle structure. To derive this network from the database, each node is accessed in turn via the *do: aBlock* message sent to the WNETWORK. For each node, a NETNODEAGENT1 (subclass of AGENT1) is created and for each arc in each node's **outArcs** dictionary a NETLINKAGENT1 (subclass of AGENT1) is created. In this way, agents are built up for each DBOBJECT in the WNETWORK structure. These agent's drawings form a tree of drawings which are displayed by sending the *display* message to the Network Mode's ROOTDRAWING10 kept in its **drawing** instance variable.

STRUCTURE

STRUCTUREs are abstract data types made up of nodes and arcs. All structures in the database are subclasses of STRUCTURE. WE provides three kinds of structures: networks (WNETWORK), paths (WPATH), and trees (WTREE). Refer to the following subsections and Figure 8 for specifics on these subclasses. Basic transformations (from one structure to another) are provided among the standard subclasses. Copying methods are also provided.

STRUCTURE lays out the capabilities for its subclasses.

Class Variables: none

Instance Variables:

nodes - the set of nodes contained in the structure. Only WNetworks keep this explicit node list (in the other subclasses, it is left empty). WNetworks need an explicit node list because their nodes are not guaranteed to be all connected in some way by arcs; some (or all) nodes can be loners that are only known about from the explicit node list. (Set)

title - the name of the structure. (String)

database - the database which the structure belongs to. (WDatabase)

Class Operations:

- instance creation. *in: aDatabase* creates a new Structure belonging to aDatabase.
- storage to and reconstruction from a file. Structures can store themselves on the external database file and thus can also reconstruct themselves from the information at a certain point in this file.

Instance Operations:

- access to the structure's knowledge. *head* returns a 'special' node (i.e. the 'root' node of a tree) which is different depending on the specific structure. Most subclasses reimplement this method. *contains: aNode* returns a boolean answer indicating whether or not aNode is in the structure. *title* returns the name of the structure. *do: aBlock* performs the aBlock code on each node in the structure.

- transformation. Default methods exist to transform any subclass of STRUCTURE into a network, a tree, or a path. WNETWORK, WTREE, and WPATH themselves usually reimplement the transformation methods to take advantage of their specific knowledge to make these methods more efficient.

WPATH

A WPATH is a 'linked list' of nodes. A path is either empty or it has a first node; every node but the last has a unique successor. By convention, the last node's successor is nil. The list is implemented by keeping track of a 'head' arc which points from the last node to the first. If the path is empty, there is no head arc - it is nil; if there is only one node in the path, then head is a cyclic arc on this node. A path does not keep an explicit node list (see discussion of STRUCTURE above). Instead, it depends on the fact that all nodes are connected – each node in a patn knows the arcs leading to its adjacent nodes. For speed, a path keeps an explicit count of the number of nodes it contains. See figure 8 fcr a picture of a generic WPath.

Class Variables: none

Instance Variables:

head - the special arc pointing from the last node back to the first node. (Arcs)

size - the number of nodes in the path. (Integer)

Class Operations:

- instance creation. *newFrom: aNode in: aDatabase* creates a new WPath (for the database aDatabase) with aNode as its single Node.

Instance Operations:

- access to the path's knowledge. *head* returns the first node in the path while *tail* returns the last node. *contains: aNode* returns a boolean indicating whether or not the path contains aNode. *succOf: aNode* returns the node that follows aNode on the path while *predOf: aNode* returns the node that precedes aNode on the path. *isEmpty* returns a boolean indicating whether or not the path is empty (contains no nodes). *nodeCount* returns the number of nodes in the path.

- growth and shrinkage. *add: aNode* adds aNode to the end of the path while *detach: aNode* removes aNode from the path. *insert: newNode after: predNode* puts newNode into the path following predNode while *insert: newNcde before: succNode* puts newNode into the path before succNode.
- destruction. *delete* permanently removes the entire stucture from the database.

WNetwork

A WNetwork is a generic directed graph. It is a direct representation of the directed graph a WE user creates and manipulates in Network Mode. Like all Structures, it consists of nodes connected with arcs. A network node can remain unattached - a loner which has no entering or exiting arcs. Also, network nodes can be connected in a very general fashion: cyclic links are allowed (though not from a single node back to itself) and there is not a limit on the number of arcs which can enter or exit a particular node. A WNetwork supports basic graph operations like adding and removing nodes and arcs. See figure 8 for a picture of a WNetwork.

Class Variables: none

Instance Variables:

label - an attribute in which all network arcs have a value. This specific attribute is defined to hold the name of a network arc. (Attribute)

Class Operations:

- instance creation. *in: aDatabase* creates a new empty network in the database, aDatabase.

Instance Operations:

- access to the WNetwork's knowledge. *labelOn: anArc* returns the name of anArc (its position in the network attribute, label). *reLabel: oldArc as: newName* changes the name of oldArc to newName. *linksFrom: aNode* returns a collection of the arcs exiting aNode while *linksTo: aNode* returns a collection of the arcs entering aNode.

- operations on network nodes. *add: aNode* adds aNode to the network as an isolated node. *link: aNode to: bNode andCall: aString* creates a new arc from aNode to bNode named aString. *detach: aNode* removes all the arcs entering or exiting aNode (aNode remains in the network as an isolated node).

- removal of arcs and nodes. *remove: aNode* removes aNode from the network and by implication removes all its entering and exiting Arcs. *removeLink: anArc* removes anArc from the network and *removeLinkFrom: fromNode to: toNode called: aString* removes any Arcs named aString which connect fromNode and toNode.

WTREE

A WTREE is a structure that imposes a strict node hierarchy. Visually, tree restrictions mean that all nodes are involved in the structure; unlike WNETWORKS, there can be no 'loners'.

A tree is represented internally as shown in figure 8. Each arc has a position in a tree attribute named 'tree'. The root arc is a cyclic arc on the root node. There is only one root arc. Arcs with a tree attribute named 'child' link a parent to its first child. A 'sibling' arc links a node to its next sibling. Each node, thus, has a single entering arc. It may have zero, one, or two exiting arcs.

The methods provide a full set of tree operations - insertions and deletions of various kinds as well as subtree rearrangement operations.

Class Variables:

Tree - the Attribute in which tree arcs have positions. These positions are strings - names indicative of the arc's representative relationship (e.g. 'child', 'sibling', 'rootArc'). (Attribute)

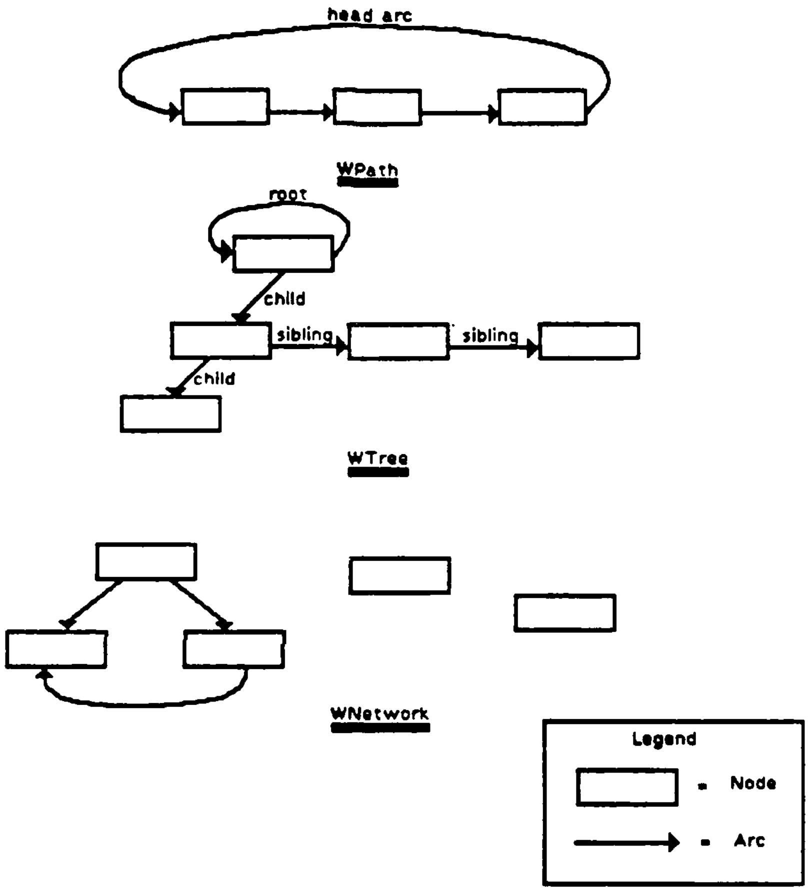


Figure 8: WE STRUCTURES

Instance Variables:

rootArc - the special arc which is cyclic on the root node - the head arc of the tree structure. (Arc)

Class Operations:

- instance creation. *create: aNode in: aDatabase* creates a new tree rooted at aNode in the database aDatabase.

Instance Operations:

- access to the tree's knowledge. *contains: aNode* returns a boolean indicating whether or not aNode is in the tree. *isEmpty* returns a boolean indicating whether or not the tree has any nodes. *node: relNode isAncestorOf: aNode* returns a boolean indicating whether or not aNode is a descendant of relNode. *childrenOf: aNode* returns a collection of all aNode's children. *nextChild: aNode* returns aNode's next sibling while *prevChild: aNode* returns aNode's previous sibling. *parentOf: aNode* returns the parent of aNode. *root* returns the root node. Finally, *do: aBlock* evaluates the code in aBlock for each node in the tree.
- insertion, deletion, and rearrangement of Nodes. *insert: newNode asLastChildOf: dad* puts newNode into the tree as the last child of dad. *nodeMove: aNode after: predNode* moves newNode into the tree with predNode as its immediate younger sibling. *subsume: younger to: elder under: newParent* inserts newParent into the tree at the place specified by considering younger and elder as its end children. *treeMove: aNode after: predNode* moves aNode and all its progeny to become predNode's elder sibling. *deleteNode: aNode* removes aNode from the tree. *deleteTree: relRoot* removes relRoot and all its descendants from the tree.
- grafting. Methods exist for grafting a given subtree onto the tree in a specified position.

WFIELD

A WFIELD is a special kind of structure (not a subclass of STRUCTURE) which is very different from it's counterparts in the Structures category. A WFIELD is a utility for collecting DIMENSIONS into a single structure. WE uses a WFIELD to hold the set of dimensions it deals with. MODEs access this (single) subclass of WFIELD when they need quick access to a certain dimension. In the current implementation, WFIELD has no variables or methods; it is simply an abstract superclass.

3.3.4 File Support

As discussed above, a WDATABASE can store itself into an external file. Databases save themselves when WE users explicitly command a save. From a WE user's point of view, his workspace can be saved as one big chunk under a single name. The user can use this name in the future to recall the database into WE. In reality, however, a complete workspace is saved in a set of files. In the UNIX directory where the workspace belongs, there is a directory xxxx.ws (where xxxx is the user-given name of the workspace) which represents the workspace as a whole. This xxxx.ws directory contains a SINGLE file xxxx.db which is the database. The xxxx.ws directory also contains a set of .txt files each corresponding to a single node's text. A .txt file is created and saved for a node when text is associated with it. Thus, there is no .txt file for a node which has no text.

A database is reconstructed from its external .db file when a WE user calls for it. .txt files are kept permanently in external files and managed by WE through its Edit and Revise Modes - .txt files are NOT part of the database in the most restrictive sense because they do not all reside internally during a session.

WE's external file support of workspaces is essential to the usefulness of the system.

4 Acknowledgments

Portions of this research were supported by the National Science Foundation (Grant # IRI-8519517), the Army Research Institute (Contract # MDA903-86-C-0345), and the International Business Machines Corporation (Contract # SUR-423). The project was led by John B. Smith and included faculty researchers Stephen F. Weiss, Jay D. Bolter, and Marcy Lansman. Gordon J. Ferguson was chief architect of the system. Graduate students who contributed to the coding effort included Paulette Bush, Yen-Ping Shan, Valerie Kierulf, and Katie Clapp. Greg Berg and Irene Jenkins contributed to the cognitive science aspects of this project. Oliver Steele developed the protocol tracker and replay program. Paulette Bush was the principal author of this report.

5 Appendices

5.1 Roaming: Traversing an Infinite Drawing Space

The Roaming capability in WE allows a WE user to navigate through visual structures that are too large to fit entirely within a Mode's window boundaries. When Roaming is used, a small additional window appears in the current Mode. This window represents an 'infinite' drawing surface. A structure's placement in this world is represented by a small replica inside the additional window. A WE user can specify the section of a structure which he wants to appear in the current Mode window through simple, natural operations. The design details of the Roaming capability are presented in the forthcoming documentation: *A Roaming Package for the Writing Project*, by Valerie Kierulf and Gordon Ferguson.

5.2 Tracking: Recording and Replaying a WE session

The Tracking capability in WE enables the system to collect a transcript of a WE session which can be later analyzed and parsed using a WE grammar. Information gained from parses of WE transcripts for different kinds of writers can support and/or point out inconsistencies in the theory behind WE. These transcripts can also be used to visually 'play back' the WE session in WE. For design details of this tracking capability, see the forthcoming documentation: *The WE Transcript and Replay Mechanisms: A Programmer's Guide*, by Oliver Steele.

6 Reference List

Clapp, K. N., & von Borries, H. (1987). *Programming in Smalltalk.* Chapel Hill, NC: UNC Department of Computer Science Technical Report TR87-023.

Cox, B. J. (1986). *Object Oriented Programming.* Addison Wesley.

Goldberg, A., & Robson, D. (1983). *Smalltalk-80: The Language and its Implementation.* Addison Wesley.

Shan, Y., Smith, J. B., & Ferguson, G. J. *Software Development in an Academic Research Setting: To Prototype or to Reprototype?.* Chapel Hill, NC: UNC Department of Computer Science Technical Report TR88-033.

Smith, J. B., & Lansman, M. (1988). *A Cognitive Basis for a Computer Writing Environment.* Chapel Hill, NC: UNC Department of Computer Science Technical Report TR87-032.

Smith, J. B. et al. (1987). WE - A writing environment for professionals. *National Computer Conference*, 725-736.