HEWLETT
PACKARD

Visual Formalisms as Application Frameworks

Bonnie Nardi; Craig Zarmer
Software and Systems Laboratory
HPL-90-211
December, 1990

visual formalisms; application frameworks; tables; semantic application objects

In order to construct applications that successfully capture task and domain semantics, programmers need support at a higher level than that provided by general programming languages and toolkits such as the X Toolkit. We propose *visual formalisms* as application building blocks that provide higher level support. Visual formalisms are generic visual/semantic computational structures that can be specialized to reflect specific domain semantics. We discuss how visual formalisms serve as application frameworks that effectively structure and present the semantics of users' problems. Our implementation of a Table Visual Formalism is described to show the potential for visual formalisms to serve as reusable computational structures that support the development of specialized applications.

1 Introduction

One of the lessons learned from our study of spreadsheet use (Nardi & Miller, 1990a; 1990b) is that simple graphical formats such as tables provide a great deal of leverage to end users who are designing, programming, presenting and sharing their own applications. The table is an organizing framework within which users lay out and structure their data, and operations on the data. For example, each cell in a spreadsheet table can contain a data value and a formula for deriving the value, or an identifying label or bit of explanatory text. Rows and columns contain relational information. Within the structure of the table's rows, columns and cells, spreadsheet users create surprisingly rich and sophisticated applications.

What struck us about tables – and other simple visual notations such as graphs, plots, panels and maps – is how frequently we see them in everyday life. Tables, for example, turn up in calendars, schedules, hospital flow sheets, decision tables and data displays for scientific, engineering and business data, to name but a few examples.

In software applications that use visual notations, the code to implement them is written afresh each time an application is developed because appropriate software is not available to developers. A goal of our project is to provide software libraries of reusable objects that will enable such applications to be written much faster – and in some cases, much better. Our software objects will provide strong representational, editing and browsing capabilities in objects that can be specialized for specific applications. These computational objects based on everyday visual notations are "visual formalisms."

2 Visual Formalisms

Just what are visual formalisms?

Visual formalisms are diagrammatic notations with well-defined semantics for expressing relations. They are based on simple visual notations such as tables, graphs, plots, panels and maps. Versions of such visual notations that define a precise semantics become truly formal. As we will discuss, visual formalisms are simple but expressive, compact but rich in information. They can form the basis for many kinds of programs because their relational semantics are broadly applicable across many domains. A visual formalism can be specialized to meet the needs of many specific applications; for example, specializations of panels include control panels, menus, forms, and data displays composed of pictures and text.

Visual notations are commonly used for display purposes, but it is less common for users to be able to manipulate their components – to be able to ask about the values behind a point on a plot, for example, or to expand a region on a map to show more detail. It is even less common for these displays and their components to possess any semantic information about their relationships to other displays or components – say, constraints between specific values, or the mapping from one notation to another. Computer-based versions of visual formalisms will provide these capabilities through sophisticated visual/semantic mechanisms, utilizing object-oriented representations and interactive editing and browsing techniques such as filtering and fish-eye views. A library of specializable visual formalism objects can fill a middle ground between the expressivity of general programming languages and the semantics of specific applications, giving application developers higher-level visual/semantic objects with which to develop specialized applications.

In his article "On Visual Formalisms" (1988), David Harel stated,

> The intricate nature of a variety of ... systems and situations can, and in our opinion should, be represented by *visual formalisms*: visual, because they are to be generated, comprehended, and communicated by humans; and formal, because they are to be manipulated, maintained, and analyzed by computers (emphasis in original).

Harel emphasized the need to model complexity and the importance of visual problem solving using semantically rich languages:

> We believe that in the next few years many more of our daily technical and scientific chores will be carried out visually ... The languages and approaches we shall be using in doing so will not be merely iconic in nature (e.g., using the picture of a trash can to denote garbage collection), but inherently diagrammatic in a conceptual way ... They will be designed to encourage visual modes of thinking when tackling systems of ever-increasing complexity, and will exploit and extend the use of our own wonderful visual system in many of our intellectual activities.

Computer-based versions of visual formalisms define formal semantics for the familiar visual notations we have talked about. These notations have been refined over hundreds, if not thousands of years (Tufte, 1983; Cameron, 1989). Cameron (1989) noted that tables have been in use for 5000 years. Inventory tables, multiplication tables and tables of reciprocal values have been found by archaeologists excavating Middle Eastern cultures. Ptolemy, Copernicus, Kepler, Euler, and Gauss used tables. Tufte (1983) shows and analyzes early graphs and maps.

Visual formalisms provide a logical starting place for a set of objects comprising an application development toolkit. Although we expect to see new visual formalisms emerge over time, creating a successful new visual notation is no easy task. Luckily, the familiar, standard ones still have a lot of fire power in them, and can be specialized to meet the semantic requirements of many domains (Rumelhart, Smolensky, McClelland & Hinton, 1986). For example, Harel's (1988) statecharts, which formally describe a collection of sets and the relationships between them, combine graphs and Venn diagrams. Although Harel's work is quite new, Bear, Coleman and Hayes (1989) have already created a specialization of statecharts called objectcharts, for use in designing object-oriented software systems. And Heydon, Maimone, Tygard, Wing and Zaremski (1989) specialized statecharts to model a language for specifying operating system security configurations.

3 Visual Formalisms in a Nutshell

Visual formalisms have several distinguishing characteristics:

- *Exploitation of human visual skills.* Visual formalisms are based on human visual abilities, such as detecting linear patterns or enclosure, that people perform almost effortlessly. They take advantage of our ability to perceive spatial relationships and to infer structure and meaning from those relationships (Cleveland, 1990). Visual formalisms are capable of showing a large quantity of data in a small space, and of providing unambiguous semantic information about the relations between the data.

- *Strong semantics.* Of the visual formalisms we have identified, graphs have defined for them the most formal semantics, and panels the least. Any implementation of a visual notation as a visual formalism will supply its own formal semantics. In general, visual formalisms excel as expressing and presenting *relational semantics*, such as the relations between items in a column in a table.

- *Manipulability.* Visual formalisms are not static displays, but allow users to access and manipulate the displays and their contents in ways appropriate to the application in which they are used.

- *Specializability.* Visual formalisms provide basic objects that can be specialized to meet the needs of specific applications. They are at the right level of granularity – neither too specific nor too general. Visual formalisms are appropriately positioned between the expressivity of general programming languages, and the particular semantics of applications.

- *Broad applicability.* Visual formalisms are useful because they express a fairly generic set of semantic relations, relevant to a wide range of application domains. Because a large number of applications can be designed around a given formalism, visual formalisms will eliminate a great deal of tedious low-level programming, as well as give developers ideas about editing and browsing techniques with which they may not familiar, such as the use of fish-eye views for large datasets (Furnas, 1986; Ciccarelli & Nardi, 1988).

- *Familiarity.* Because the standard visual notations are so useful, they are found everywhere. Not only do they draw on innate perceptual abilities, but through constant exposure we become very familiar with them. Our schooling explicitly trains us in the use of the basic notations; e.g. using calendars and learning matrix algebra provide experience with tables. Everyday activities provide opportunities to create and use visual notations, such as writing a laundry list or reading a map.

4 Visual Formalisms as Application Frameworks

The purpose of implementing visual formalisms as application objects is not to supply programs to draw tables or lay out graphs. The purpose is to provide strong representational, editing and browsing capabilities in reusable objects that can be specialized for specific applications. The components of a visual formalism, for example, cells in a table or nodes on a graph, must be capable of containing complex semantic information. An implementation of a visual formalism should provide editing and browsing capabilities that

make sense for a particular form, e.g. the ability to select blocks of cells in a table, or to visually filter out the columns in a table which meet some test.

Visual formalisms are application frameworks. They provide a specific orienting framework in which to cast an entire application. For example, a spreadsheet, a specialized table, allows users to develop applications in which the relations between numerical quantities are laid out and organized within the rows and columns of the spreadsheet table. The spreadsheet provides a structure into which a model is cast. ***Users do not have to invent a structure*** – it is given to them. The initial phase of a modeling problem is reduced to simply recognizing a format into which a problem is framed, rather than being faced with the necessity of inventing a format from scratch (Nardi & Miller, 1990a).

An application framework is very different from a widget. A widget is a simple object whose purpose is to accept user commands or arguments, or to allow for the display and/or editing of simple data values. It exists not to organize whole applications, but to give users access to small but significant pieces of an application. A visual formalism is intended to help developers organize and present an entire complex application, or a large piece of one. A visual formalism will in most cases be used in conjunction with widgets – for example, selectable nodes on a graph might reveal pop-up menus when buttoned. Several visual formalisms may be linked together to comprise a single large application.

5 An Implementation of a Table Visual Formalism

In this section we describe in detail our implementation of a Table Visual Formalism. In the next section we show how the Table Visual Formalism can be specialized to create an application. We prototype the CareVue patient care information system, a Hewlett-Packard software product, as our example application.

We have argued that visual formalisms are especially important to application design because they will provide practical help to developers. In order to be as specific as possible about exactly the kind of practical help we are talking about, in this section we describe our implementation of a Table Visual Formalism. Our implementation suggests how visual formalisms can become reusable computational objects collected into software libraries to form a basis for the development of specialized applications.

Implementing a visual formalism requires defining (1) the semantics of the visual formalism itself, such as correct table structure, and manipulations on individual components of the visual formalism, e.g. cells in a table; (2) the presentation of the visual formalism, including multiple presentations of a given instance of a visual formalism; and (3) the application semantics expressed within the framework defined by the visual formalism.

We have implemented a Table Visual Formalism as part of our object-oriented development environment, ACE (Application Construction Environment) (Zarmer, 1989; Zarmer & Canning, 1990), implemented in C++ and the Interviews user interface toolkit (Linton, Vlissides & Calder, 1989). Our Table Visual Formalism benefits from being a part of ACE which provides abstractions such as dialog management, change notification, and presentation in an advanced user interface architecture modeled after Nephew (Szekely, 1989), with significant extensions (Zarmer, 1989).

The contribution of the Table Visual Formalism is to give developers high-level functionality for developing applications for creating, structuring, modifying, editing, and browsing

tables. Unlike the work of Beach (1985) and Cameron (1989) whose programs *draw* complex tables, we are concerned with providing object-based table structures that can be active application components, not static pictures.

We define a table as a rectangle filled with contiguous, non-overlapping rectangular cells, each of which contains a discrete piece of information (a content entry). A content entry is any object, e.g. an integer, string, bitmap, or instance of an application-specific class. Each content entry has a semantic and presentation component.

Our Table Visual Formalism differs from a spreadsheet in that: a cell can have any type of object for its content; a general programming language can implement cell behavior; it is more structurally flexible, allowing spanning rows and columns; it can be specialized, at a more general level than spreadsheets, into different applications such as a hospital flow sheet, an interactive data display, a decision table.

Both the table as a whole, and each cell within the table, are components with their own presentation and semantics. The presentation and semantic aspects of the table are implemented through 6 classes:

- **Table** represents the semantics of a table as a whole, in particular adjacency information about cells. The operations on tables mainly deal with changing the *structure* of a table – the arrangement of rows and columns.
- **Cell** represents an individual cell of a table. Users will normally be unaware of cell objects – they will only think of the content object (the integer, string, etc.) that is in the cell. The cell can also have names, relational expressions (to express relations within the table), etc. associated with it.
- **TableView** is the visual presentation of a table. It translates the semantic adjacency of cells to a visual arrangement of cell presentations (class CellView). There can be more than one instance of TableView for a given table to provide multiple presentations of the same table.
- **CellView** presents a cell. The generic CellView provides visual feedback (e.g. highlighting) for selected cells (i.e. cells the user selects with mouse or keyboard), and determines how to present the Cell's content object.
- **TableEditor** supports the editing process, providing interface state, such as a currently-selected-cell, which must be maintained on a per-user basis.
- **TableEditorView** is a presentation of a table editor. It provides a graphical interface for the table editor's editing operations, and a presentation of interface state, such as a presentation of the currently selected object.

A table application that does not require interactive editing capabilities can be implemented with the first four classes. A table that will be edited will make use of the last two classes as well. Each of these classes can be specialized through subclassing.

Let's look at tables in more detail.

Class Table represents the semantics of tables, that is, the semantics of the table structure itself (not the content entries in cells, which can be anything pertinent to the application).

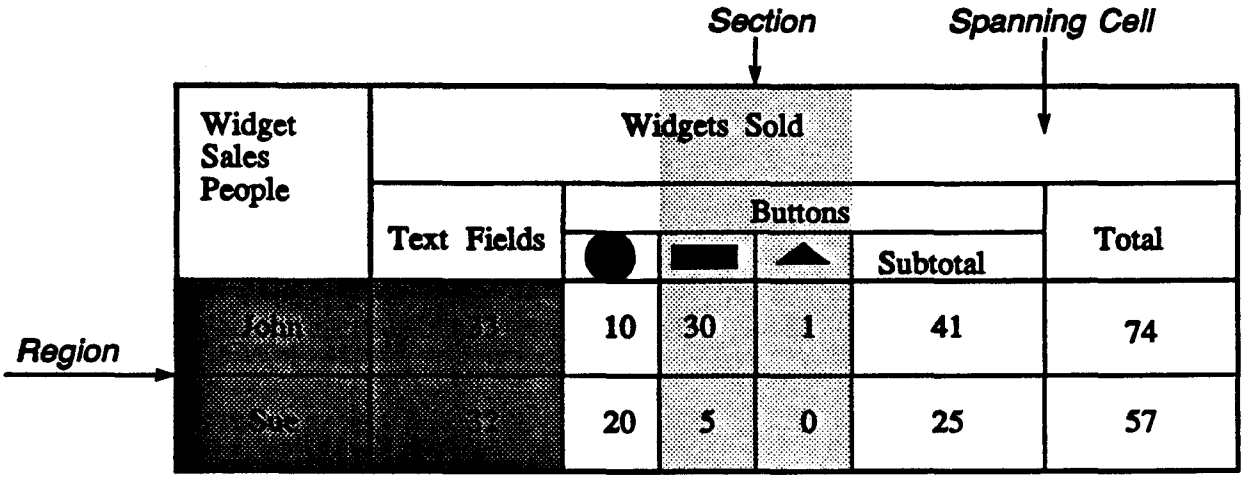

Figure 1: A Table

It implements a "table engine" that manages a set of cells and enforces the structural rules of tables, but provides no user interface. A table is able to grow or shrink to hold any number of cells, but is finite at any given time. Table operations can *add* or *remove* cells from the table, or *change semantic relationships*, expressed as adjacency, between cells. A table has operations to *split* and *join* cells. In the table shown in Figure 1, the cell labeled "Widgets Sold" is a cell that has been joined to span several other cells.

A **section** is a portion of a table that is defined by any two parallel lines that extend completely across the table, either horizontally or vertically. In other words, a section is one or more adjacent full-length rows or columns. A section is the unit for the operations that *add, delete, move*, or *copy* sets of cells into or out of the table, because adding or removing a section is the only unit guaranteed to preserve a rectangular table. In Figure 1, a section is shown in light gray. Note that a section can cut through part of a cell. Deleting this section would eliminate the fully-enclosed cells (like the one labeled "30"), and would shrink the partially-enclosed cells (like the one labeled "Widgets Sold").

A **region** is a rectangular group of contiguous cells within a table. A region could contain an entire table, or only a single cell. A region identifies a collection of cells and associates a name or keyword(s) with it. Names and keywords can be accessed by relational expressions, table viewer operations, etc. A region is shown in dark gray in Figure 1.

To *construct* a table, a user can either (1) ask for a default 1 x 1 table, and then edit it using the re-structuring operations (join, split, add-cell, etc); (2) ask for an "infinite" table, that is, a very large table, such as those found in spreadsheets; or, (3) edit the table properties that specify the number of rows and columns in the table. Tables can be saved with all structure and content intact.

The generic Table class can be specialized to add, restrict, or remove operations. For example, a spreadsheet application might remove the split and join operations, to ensure that the table remained a grid. Other applications might add more specific editing operations,

such as inserting a section with a particular format of split and spanning cells.

Class Cell stores table information needed on a per-cell basis. The main component of a cell is a content object. A cell need not have a content object, and the generic class Cell does not automatically create one.

Many applications will be able to use the generic class Cell. Reasons for creating a subclass of Cell might include: supporting multiple values in a cell; restricting content object types; providing specialized content parsing/unparsing; or providing specialized cell relations.

Class TableView arranges presentations of a table's cell views into rows and columns that reflect the relationships between the cells. Class TableView is responsible for deciding how big each row and column should be, and provides operations for end users to override the default sizes.

Class TableView provides several means of managing large tables. First, one or more sections can be *filtered* (removed) from the display. This does not affect the semantic table object at all; it simply leaves more room for other cells to be seen. Using named regions, keywords and constraints, users will be able to request, for example, that a table "Show only columns with budget estimates over $1000." TableView instances can be *scrolled* horizontally and vertically. *Zooming* allows a user to zoom in on a detailed view or zoom out for a larger view.

The major reason to create an application-specific subclass of class TableView is to change the layout policy. One application might require all rows and columns to be exactly the same size, while another might require that row and column size be based on the size of the cell content presenters (e.g. giving a large string extra room). Applications may differ in how they want a table view to respond to a change in the overall size of a table: one may wish more/fewer cells to be shown, while another may want the same cells to be shown, only larger/smaller.

It is important to note what class TableView does not do: it is not involved with the manipulation of a cell's content (the contents' own presenter does that), or feedback about selected cells (the CellViews do that). TableView does not choose the gestures or dialog styles used to invoke Table operations – these are chosen by the application designer using ACE components. Decisions about gestures and dialog styles may be independent of how a table is viewed and hence are not bundled into TableView.

Class CellView provides presentations of a cell. CellView objects know the color, shading, outlining, background pattern and special formatting of cell contents (e.g. numbers shown as dollar amounts) of a cell presentation. This is useful so that tables collected into libraries can be used as templates for other applications. It is also useful when copying regions. Users may want to set up tables with this kind of information even before they have information about cell contents.

CellView has two main jobs: to provide feedback for selected cells, and to choose a presenter for the cell content object. Subclasses might change either the type of feedback given or the policy for choosing presenters.

A cell can have more than one presentation, including larger pop-up views that show more detail than can be shown in the cell within the table (for cells with e.g. a large bitmap, or a lot of text). These "BigCellViews" can be popped up on request, moved to any place

on the screen, and left on-screen if desired, so that several may be viewed at once. Having BigCellViews combines the advantages of the table's presentation of relational information for many cells, with the ability to see more detail on selected cells.

In summary, the ACE Table classes described here show how an abstract visual formalism can be captured as concrete classes for use by application developers. The classes encapsulate the essential aspects of tables while making it easy for programmers to specialize Tables to meet the task-specific requirements of individual applications.

6 Using Tables to Build an Application: An Example

As an example of how visual formalisms can be used to produce sophisticated applications, we'll sketch the process of building an application similar to an existing Hewlett-Packard product, CareVue. CareVue is an automated patient record based on the tabular flowsheet used in hospital intensive care units to record patient data. Each row in the flowsheet represents a different measurement, and each column represents a time when a measurement was taken. (See Figure 2, next page.)

To build CareVue we start with the Table visual formalism described in the previous section. The major extension to class Table is to add coordination with a database in order to provide the high data reliability required for a hospital application.

CareVue allows users to record annotations with data values. To support annotation, we construct a specialized version of class Cell that provides for annotation objects as well as a primary data object. Class CellView would then be similarly specialized to indicate whether an annotation was present, and show it on demand. Class Cell can also be specialized to create intermediate values when it does not have a reported value.

The visual appearance of CareVue is largely tabular, but it provides extra filtering and scrolling options to allow different pre-defined subsets of the table to be viewed at the press of a button. (See Figure 2.) Class TableView already provides the necessary filtering and scrolling services.

A number of specialized data types are created to represent values such as blood pressure. Integers could be used, but it is better to produce a class that models blood pressure (e.g. a blood pressure has two components; reasonable values are between 0 and 300; and one value should be lower than the other). This also allows for specialized presenters to show the values. Finally, the table is organized into a panel, along with the required ACEKit components such as buttons and menus, to produce the finished application.

7 Summary

We believe that the next generation of software programs will emphasize helping users solve complex problems in scientific, engineering, and business applications. Visual formalisms can be the basis for many of these applications. Visual formalisms encode precise semantics, take advantage of human visual abilities, provide macro-level frameworks for application design, and can be offered as reusable computational structures to support the development of specialized applications.

Because visual formalisms already exist as recognizable visual notations, they map easily

into computational objects that can be collected into reusable, extensible software libraries. We described our implementation of a Table Visual Formalism to suggest how the semantic and presentational aspects of visual formalisms can be managed in an object-oriented table structure that is a high-level building block for designing and implementing specialized applications.

8 References

Bear, S., Coleman, D. & Hayes, F. (1989). Introducing objectcharts, or how to use statecharts in object-oriented design. HPL-Report-ISC-TM-89-167. Bristol, England: Hewlett-Packard Laboratories.

Beach, R. (1985). Setting tables and illustrations with style. Technical report CSL-85-3. Palo Alto: Xerox Palo Alto Research Center. May, 1985.

Cameron, J. (1989). A cognitive model for tabular editing. OSU-CISRC Research Report, June, 1989. Ohio State University.

Ciccarelli, E. & Nardi, B. (1988). Browsing schematics: Query-filtered graphs with context nodes. *Proceedings of the Second Annual Workshop on Space Operations, Automation and Robotics (SOAR '88).* pp. 193-204. July 20-23, 1988. Dayton, Ohio.

Cleveland, W. (1990). A model for graphical perception. AT&T Statistics Research Report. Murray Hill, NJ: AT&T Bell Labs.

Furnas, G. (1986). Generalized fisheye views. *Proceedings of CHI'86, Conference on Human Factors in Computing Systems.* pp. 16-23. April 13-17, 1986. Boston.

Harel, D. (1988). On visual formalisms. *Communications of the ACM* 31, 514-520.

Heydon, A., Maimone, M., Tygar, J., Wing, J. & Zaremski, A. (1989). Constraining pictures with pictures. In *Proceedings of IFIPS '89.* August 1989. pp. 157-162. San Francisco.

Linton, M., Vlissides, J. & Calder, P. (1989). Composing user interfaces with InterViews. *IEEE Computer* 22, 8-22.

Nardi, B. & Miller, J. (1990a). The spreadsheet interface: A basis for end user programming. *Proceedings of Interact'90.* pp. 977-983. 27-31 August, 1990. Cambridge, England.

Nardi, B. & Miller, J. (1990b). Twinkling lights and nested loops: Distributed problem solving and spreadsheet development. Forthcoming in *International Journal of Man Machine Studies.*

Rumelhart, D., Smolensky, J., McClelland, J. & Hinton, G. (1986). Sequential thought processes in PDP models. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* Volume 2. pp. 7-57. Cambridge: MIT Press.

Szekely, P. (1989). Standardizing the interface between applications and UIMSs. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology.* pp. 34-42. Williamsburg, Virginia.

Tufte, E. (1983). *The Visual Display of Quantitative Information.* Cheshire, Connecticut: Graphics Press.

Zarmer, C. (1989). ACEKit overview. STL-Report-89-29. Palo Alto: Hewlett-Packard Laboratories.

Zarmer, C. & Canning, P. (1990). Using C++ to implement an advanced user-interface architecture. HPL-Report-90-21. Palo Alto: Hewlett-Packard Laboratories.