

Building Information Visualizations: A Commonality Analysis

Mark Ardis, Kenneth Cox, Stacie Hibino, Lichan Hong, Audris Mockus, and Graham Wills

Bell Labs/Lucent Technologies
263 Shuman Boulevard
Naperville, IL 60566-7050

{maa, kcc, hibino, lhong, audris, gwills} @ research.bell-labs.com
<http://www.bell-labs.com/org/11359/spr-vis.html>

ABSTRACT

Recently, we applied the software engineering technique of commonality analysis to the domain of interactive information visualization. This produced an analysis of graphical data displays in terms of commonalities (features shared by all such displays) and variabilities (the ways in which displays differ). This analysis has applications in the areas of taxonomy, design and development, and visualization theory. We have used this analysis to develop a Java visualization framework, *mmvz*, which can be used for rapidly and flexibly building a large range of information visualizations. In this paper, we describe the commonality analysis process and its results. We also outline the *mmvz* architecture and present examples of using *mmvz* to create visualizations.

Keywords. Commonality analysis, visualization library, taxonomy, Java, Application Engineering.

1. INTRODUCTION

Many software applications can be classified into *families*, or groups of related entities. Members of a family can be characterized by *commonalities* (their shared properties), and distinguished within the family by variabilities (the ways each member differs from the others). These families can be found at any level of consideration of the software. For example, at the lower end of the scale there are many data structures to store collections of data—heaps, stacks, sets, queues, and so forth. The storage of the collection is the commonality, while the variation is found in the different access mechanisms of the structures.

Commonality analysis is a software engineering technique for identifying commonalities and variabilities of a software family. Such an analysis can play a key role in specifying software requirements and designing a system architecture for the target family of applications. The results of a commonality analysis can be used to produce an Application Engineering Environment in which members of the target software family are created in a consistent and efficient manner. Commonality analyses are also ideal tools for evoking and recording expertise about the domains that are analyzed.

Our own involvement with commonality analysis came about because of several projects that our group was working on. Although these projects involved distinct application domains (software analysis, collaborative work, database mining, and others), they shared many common requirements. In particular, all involved the Web-based interactive visualization of large, heterogeneous data sets. In order to reduce duplication of effort and unify the interfaces and appearance of these tools, we decided to base all these projects on a shared Java-based framework that we chose to call *mmvz* (“vz for the year 2000”).

To create this new framework, we needed to define the scope of the applications to be produced using *mmvz*. We also needed to articulate *mmvz*’s design requirements. Finally, we wanted to capture domain knowledge about interactive information visualization, both our own and that of the larger community. As these goals are exactly the objectives of the commonality analysis process, it was natural to perform such an analysis.

We took as our family the domain of 2D interactive information visualization views, both those used in our projects and those described in the broader literature. We chose to restrict our family in this way for two reasons. First, we wanted to keep the commonality analysis manageable; larger families require correspondingly larger analyses. Second, a general property of commonality analyses is that narrowing the scope of the family allows one to produce a more efficient Application Engineering Environment [We99]. Thus, we expected that our narrow focus would allow us to produce *mmvz* rapidly, and that *mmvz* itself would be an efficient framework for producing our target visualizations. Both of these expectations have been borne out by our preliminary experiences.

Overview. The remainder of this paper is divided into six sections. We begin by discussing related work. In Section 3, we provide background on the commonality analysis process. Section 4 then describes our particular application of this process to information visualization. (A full listing of commonalities and variabilities is provided in an Appendix, for the reader’s reference.) Section 5 describes how the results of the analysis were incorporated into the design of *mmvz*. In Section 6, we provide two example

information visualization views to illustrate how they relate to the analysis and how they are produced with *mmvz*. Finally, we present our conclusions and future work.

2. RELATED WORK

To our knowledge, this is the first time that a formal commonality analysis of the information visualization domain has been conducted. However, it does have similarities to other work in the field.

Our commonality analysis is similar to a taxonomy, in that it characterizes common properties and variations. Several taxonomies for information visualization have been presented (e.g., [Ca99] (introduction), [Sh96], [Ro97]), and we discussed several such taxonomies and took them into consideration during our analysis. However, our analysis differs from previous taxonomies as follows:

- The analysis takes a domain engineering approach to characterizing information visualization.
- The analysis focuses on efficient authoring/creation of information visualization views and components.
- The analysis focuses on a particular family of views. For example, we chose to omit 3D and scientific visualizations from our analysis.

The results of our commonality analysis may not be as wide in scope as previous taxonomies, because we deliberately chose to narrow our focus, but they do cover a large range of information visualization views.*

Our analysis was influenced by Wilkinson's *The Grammar of Graphics* [Wi99]. However, while Wilkinson focuses on a pseudo-language for specifying static statistical plots, we examined the domain of interactive visualization, with the goal of producing an implementable framework for rapidly constructing such visualizations.

Recently, several commercial and free information visualization frameworks have become available (e.g., [Sp00], [Wa94]). However, these packages focus on the use of existing views rather than the creation of new ones. In contrast, our analysis focuses on the creation of a family of views. In addition, we also have taken various user levels into consideration, so that naïve users can easily use existing views within a browser, while developers can create new views based on our analysis.

3. COMMONALITY ANALYSIS

The primary function of a commonality analysis is to identify a family, find the commonalities and variabilities of the family, and describe the members of the family in terms of the variabilities (see, e.g., [Ar97] for a detailed description). The analysis may be done in pursuit of various software design and engineering goals, and is frequently

employed in the design phase of the software life cycle to produce the requirements, system design, and architecture.

A commonality analysis is a part of a more comprehensive process of Domain Engineering [We99]. Domain Engineering approaches the software development problem by facilitating the development of product lines (or families) rather than individual products. This is accomplished by considering all the products together, analyzing their characteristics, and building an environment to support their production. In doing so, development of individual products (henceforth called Application Engineering) can be done rapidly, at the cost of some up-front investment in analyzing the domain and creating the environment. Such Application Engineering Environments can be extremely effective, reducing coding effort by up to a factor of four [Si99].

Besides providing a basis for an Application Engineering Environment, the commonality analysis usually produces several other useful results. The domain experts—the people who produce software in the target family, and who participate in the analysis—frequently have a great deal of knowledge about the domain that is not recorded anywhere. The analysis evokes this knowledge and incorporates it into the analysis documents, making it accessible to others. Often, the problem is that the domain experts view this knowledge as “obvious”, and thus have never seen any point in recording it—or in considering all its implications.

A related useful result of the analysis is the production of an agreed-upon set of terms and concepts for describing the domain and family members. This result may surprise the participants, who typically believe before the process that they all agree on the meanings of the words that they are using. The analysis often reveals that this is not the case, and frequently clarifies long-standing confusion caused by mismatches in the participants' terminology. An example from the visualization field may help illustrate this point: Have you ever seen someone display a bar chart and call it (say) a histogram, and had the immediate reaction, “That's not a histogram”?

The first step in the commonality analysis is the identification of the domain to be analyzed. In our use of commonality analysis to define and create *mmvz*, we chose to focus on interactive data views; this led, during the course of the analysis, to consideration of the interactions themselves and the storage of data as subtopics.

The second step, often performed concurrently with the first, is to identify the participants in the commonality analysis. The participants include domain experts (people with knowledge of the domain), but not every participant needs to be a domain expert. Indeed, it is often useful to have one or more “outsiders” with little knowledge of the domain and no fear of asking questions, particularly questions of the form “What do you mean by that word?” The paper authors performed our analysis, with Mark Ardis acting as the outside moderator.

* At this early stage, we have used our results to implement bar chart, scatterplot, grid, table, parallel axes, graph-and-network, calendar, dynamic list, and hierarchical views, and are working on several others.

The remainder of the commonality analysis process is the iterative production and refinement of an analysis document for the domain. The document is produced through a series of meetings, and is interactively edited at the meetings. Typically, one participant acts as editor, and the document is displayed on a wall screen so that all participants can see it and comment. Between meetings, participants have access to the document-in-progress and are expected to consider discussion points for the next meeting. Individual projects, such as gathering information or preparing material for inclusion, are often assigned. We spent a total of eight calendar weeks on the commonality analysis, with a half-day editing session each week.

The commonality analysis document is the primary artifact of the process. The structure of the commonality analysis document is as follows:

- 1) Introduction
Purpose of the commonality analysis.
- 2) Overview
Summary of the document's contents.
- 3) Definitions
Meanings of common terms, as agreed upon by the participants. Terms may evolve over the course of the analysis, and are italicized throughout the document.
- 4) Acronyms and Abbreviations
List and definitions of acronyms and abbreviations used in the document. These are also italicized.
- 5) Commonalities
List of identified commonalities.
- 6) Variabilities
List of identified variabilities.
- 7) Issues
Pending analysis work, action items to be conducted between sessions, and topics to discuss at a later time.
- 8) Appendices
Any additional material that the participants wish to include. Our appendix included descriptions of five different views in terms of the commonality analysis.

4. RESULTS OF OUR COMMONALITY ANALYSIS

The immediate result of the analysis was the commonality analysis document itself. Our Introduction sets the stage:

The purpose of this document is to define a framework from which one can generate a large range of *interactive information visualizations* more consistently and efficiently than before. We refer to this framework as *mmvz*. Applications that use this framework include InfoStill, CIVE, and ExV.*

Our final document contained 56 definitions and acronyms, 37 commonalities and 22 variabilities. We divided the

commonalities and variabilities sections into sub-sections, for those involving *data*, *views*, *layouts* and *elements*, and *user interactions*. (These terms are described below.)

In this section, we discuss some of the key definitions, commonalities, and variabilities that we identified. Terms are in *italics* and references of the form (C31) or (V16) refer to the commonalities and variabilities we produced. A full list of the commonalities and variabilities is included in the Appendix for the reader's reference.

Data. We use a tabular representation for data, as typically found in relational databases. Each *table* row is a *record*, representing data about a single entity. Each *table column* contains a measurement for all such entities (C31). The *cell* at the intersection of a *row* and a *column* (and thus defined as a particular measurement for a single entity) can have an arbitrary value of the given column data type, or the value can be missing (V16).

The framework includes several subsidiary data classes, including *selections*, *scales* and *axes*. A *selection* is an assignment of a degree of interest to each *record* of a *table*. A *scale* defines a mapping from data to a graphical representation; for example, given a record from a table, a color scale would provide a specific color. An *axis* is a type of *scale* that produces screen coordinates. In most cases, the *scale* determines the value from a *column* (V6).

Views. A *view* is a graphical presentation of data. A *view* consists of a *layout paradigm* (C4) and one or more *view components* (C5). *Views* are displayed within a drawable screen area (e.g., a GUI window or Java applet), and it is possible to display multiple views within a single such area, tiling or even overlapping the views. Bar charts, scatter plots, graphs, and maps are all examples of *views*.

Components. Each *view component* performs a specific logical function, has a graphical representation (a way in which it draws itself), and a mechanism for handling user interactions. *Components* thus implement both the view and the controller portions of the model-view-controller paradigm. The two main types of *components* are *axis drawers* and *layouts*. An *axis drawer* renders an axis, e.g., by drawing a line and labeling tick marks next to the line. A *layout* arranges *elements* (objects capable of rendering *records*) on the screen, using *axes* to obtain the screen coordinates (C13, V2). For example, a *point layout* takes each *table record*, determines from one or more *axes* a screen coordinate, and directs an *element* to draw a representation of that *record* at that coordinate.

Components are permitted to have sub-components, which are arranged within the *component* in a manner similar to the way *components* are arranged within *views*, or *views* within drawable screen areas. Similarly, it is possible to have a *view* nested within a *view*. We can, for example, have an *element* that renders a *record* using a bar chart *view*. When used in a scatter plot, such an *element* would produce a scatter of small bar charts.

* InfoStill is a task-based data analysis. CIVE is a collaborative information visualization environment. ExV is a software expertise browser.

Layout Paradigm. Every view has exactly one *layout paradigm* (C4), which dictates how the *components* are to be arranged within the view (C10). For example, a scatter plot has at least three *components*: a *point layout* for drawing the data and two *axis drawers* for drawing the X- and Y-axes. There are many possible ways to arrange these in a view, but the *layout paradigm* dictates that the screen coordinates indicated by the X-axis drawer must align vertically with the screen X-coordinates used by the point layout, and similarly for the Y drawer and coordinates.

View Parameters. Many of the objects mentioned above, including *axes*, *scales*, and *elements*, are *view parameters*. A *view parameter* affects the graphical representation, and most *view parameters* can be shared. The scatter plot's *point layout* has at least three *parameters*—the *element* used to render the points, the X-axis, and the Y-axis. The latter two *parameters* are shared with the *axis drawers*; the *element* may also be shared, for example so that the scatter plot's glyphs are the same as the glyphs in another *view*.

User Interactions. Most user interactions are handled by the *view* passing the interaction events to the *components* in a given order (V8). Each *component* then determines whether or not it can handle the interaction event. E.g., a *mousing* operation that is interpreted as a request for a *selection* might be processed by several *components*, each evaluating whether the selection encompasses any of the *elements* it is responsible for arranging.

Scripting. Our framework, as described above, composes *views* as hierarchies. We can create a scripting and specification language for this hierarchy, assuming that we have a way of uniquely accessing each *view*, *component*, *parameter*, and so forth (C37). We can also record the user interactions (C30), allowing scripting and playback.

5. MMVZ ARCHITECTURE

Using the commonality analysis to design *mmvz*, our Java framework, involved a number of steps. The nouns defined by the commonality analysis (e.g., *component* or *element*) were captured as classes. In general, the class hierarchy is determined by the definitions; e.g., an *axis* is defined as a type of *scale*, so *Axis* is derived from *Scale**. The methods of these classes were written to enforce commonalities and permit variabilities.

Another important consideration was the binding time of the variabilities. It is necessary to specify when a variability can be changed; whether at compile-time, instantiation time, or while the object is active. In general, we use class fields and (sometimes abstract) methods to implement variabilities that are defined at compile time. We use required constructor parameters to implement variabilities that are defined at instantiation time. Finally, we have a class *Parameter* for dynamic specification of variabilities.

* In implementing *mmvz*, we have kept to the convention that the name of a class is the term as used in the commonality analysis document, capitalized.

Color Figure CF1 shows the class hierarchy for the key classes of the *mmvz* architecture. Space limitations prohibit a complete description of how each commonality and variability is supported, so we will illustrate how this was achieved with only a single class, *Layout*. A *layout* is defined as “A *view component* that arranges *elements*.” *Layout* is thus derived from *Component*, and inherits all commonalities and variabilities of that class. In addition, the analysis requires that a *layout*:

- has at least one *element* (C12), and must be able to use any type of *element* (C15). These commonalities are enforced because the class *Layout* has a field of type *Element* that must be initialized at instantiation time. Any class derived from *Element* can be used in the field.
- uses *axes* to provide coordinates for its *elements* (C13). Only *Axes* can provide screen coordinates from data items, so *Layouts* must use this class.
- represents exactly one data *table* (C14). *Layout* has a field of type *Table* that must be initialized at instantiation time.
- makes at most one call to an *element* draw method for each record (C17). The `draw()` method of *Layout* implements, and thereby enforces, this constraint.
- must be capable of *drill-down*, *identifying* and *selection* (C22-24). *Layout* has abstract methods for each of these interactions, which must be defined by descendants.
- may allow *rotation* (V11). A *Layout* has a method for this user interaction that need not be overridden by descendants (and does nothing if not overridden).

Although the above information is necessary to the design and development of *Layout* and of classes derived from it, most users of *mmvz* do not need to know it. Ensuring that the library would be accessible at different levels of expertise was an important design consideration. This decision led us to decide explicitly which classes in the library must be understood by users (a) embedding views within HTML pages and scripting simple tasks, (b) using the library views and components within another Java-based application, and (c) writing novel visual components.

This division is illustrated in Color Figure CF1. At a given level of expertise, users only need to know about the behavior of certain specific classes; they can ignore other classes. For example, script-writers need only know about implemented views (such as the bar chart) and how to customize these views with parameters. They do not need to know about the abstract class *View*, nor do they need to know how the *Parameter* mechanism is implemented.

6. EXAMPLES

6.1 Using *mmvz* to create a Parallel Axes View

A parallel-axes view (*ParaView*) [In89] is a multivariate display of data. The display uses a set of parallel axes, each representing the values of some data attribute. Lines are drawn between pairs of adjacent axes connecting associated values.

Figure 1 shows a parallel axes view of Nascar truck racing data. The underlying data table is summary information for drivers who participated in the 1999 racing season. The view displays axes for the manufacturer of each driver's truck (Mfr), the total money earned during the season (Total\$), the number of races each driver won (Wins), and their overall rank for the season (Standings). The lines for selected table rows are colored according to manufacturer; dark gray lines indicate unselected items. We can see that there are selected items associating the manufacturer "Dodge" with earnings of about 300K, 360K, and 450K.

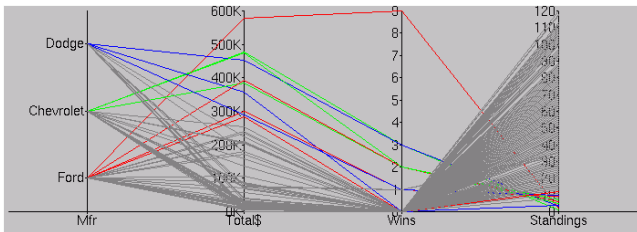


Figure 1. Parallel axes view built with *mmvz*.

According to our commonality analysis, all views have the following properties:

- data to be displayed (C2),
- layout paradigm for the view (C4),
- components of the view (C5),
- view parameters (C8, V1-V7), and
- user interactions (C20-C30).

6.1.1 Data

Each view displays data from one or more *tables* (C2) – or, since we are now talking about the implementation rather than the analysis, one or more *Tables*. The *ParaView* requires two or more *Columns* from a single *Table*; we will call the number of *Columns* (and thus axes in the view) *M*.

6.1.2 Layout Paradigm

The layout paradigm composes the set of *M Columns* into one *Axis*, and each of the *M Columns* onto a separate *Axis*. The layout paradigm states that the column *Axis* is to be displayed within the view. Each data *Axis* is then also to be displayed, with its direction orthogonal to the column *Axis* and at a coordinate determined by that *Axis*, and all are to map to the same extent in the orthogonal dimension.

For example in Figure 1 the column *Axis* is displayed at the bottom of the figure, extending horizontally. The *M* data *Axis* are placed above it, extending vertically, with their *X* coordinates determined by the column *Axis*. Each data *Axis* has the same extent in the *Y* dimension, and maps its respective *Column* into that extent.

6.1.3 View Components

Axis Drawers. The *ParaView* contains *M+1 AxisDrawers*. One *AxisDrawer* displays the column *Axis*. The other *M AxisDrawers* are used to display each of the data *Axes*, one *AxisDrawer* per data *Axis*.

Layouts. The *ParaView* uses *M+1 LineLayouts*, each of which visually connects associated data values from one *Axis* to corresponding data values in the next adjacent *Axis*.

These *Components* are arranged according to the dictates of the layout paradigm, as described above.

6.1.4 View Parameters

Column Axis. The column *Axis* is a *Parameter* to the column *AxisDrawer* and to each *LineLayout*, and is used by the layout paradigm to position the *Components*.

Data Axes. The *Axis* for each *Column* is a *Parameter* to the data *AxisDrawer* for that *Column*, and is also shared by one or more *LineLayouts*.

Elements. The *LineLayouts* use a shared *LineElement*. The *LineLayouts* enumerate over all *records*, and for each *record* use the *LineElement* to draw a line segment.

Degree-of-Interest Scale. This *Scale* is associated with the *Table*. It is used by the *LineElement* when rendering; rows with a degree-of-interest of 0 (i.e., unselected data) are rendered in dark gray.

Color Scale. This optional *Scale*, if provided, is used by the *LineElement* to choose the color of the line. In the figure the color *Scale* is calculated from the “manufacturer” column, but any *Column* of the *Table* (including ones not shown by the *ParaView*) could have been used, as could have an arbitrary *Scale* not computed from any *Column*.

Other Parameters. These include the following:

- *Font and Color Scheme*. These determine label fonts, background color, color of the axes, and so forth. Generally all views will share the same schemes, thus providing a uniform “look-and-feel.”
- *Orientation*. A parameter indicating “horizontal” or “vertical” layout, used by the layout paradigm.
- *Data Label Visibility*. Each data *AxisDrawer* has a parameter indicating whether labels are shown or not.
- *Column Axis Order*. This parameter determines the order in which the *Columns* appear.
- *Data Axes Order*. These parameters (one per data *Axis*) determine the ordering of each *Axis*. For numeric data, this is “ascending” or “descending”.

6.1.5 User Interactions with a *ParaView*

The common user interactions (C20-C30) are automatically provided by the *ParaView*'s parent class *View*. This class has methods that perform *resizing*, *selection*, *identifying*, and so forth. The way in which these user interactions are handled by a specific *view* can vary (V8-V15). This variation is implemented by *Parameters* of the *Components* (for variabilities specified at instantiation or run-time), or by implementing methods that are abstract in the parent class (for variabilities that are specified at compile-time).

As one example, the graphical user interactions of *lassoing* and *brushing* can be used to change the *selection*. We decided that this user interaction was best handled by the

components (V8). When a user lassos or brushes, the *view* passes the user-defined area to those *components* that allow *selection* (V15). The *components* report the *records* that are selected by the area. The *view* merges the *records* reported by the *components* with the previous *selection* (C34) to produce the new *selection*.

All the above operations are common to all *views* and *components*, and thus are built into the *View* and *Component* classes. *ParaView* and its *Components* vary in two ways. First, the *AxisDrawers* do not allow selection, but the *LineLayouts* do allow it (V15); the *Parameter* that indicates this is set by the *ParaView* when it creates each *Component*. Second, the *LineLayout* class implements the *selection()* method (an abstract method of *Component* which performs *selection* as described above) so that it indicates a *record* is selected if the line corresponding to that *record* passes through the user-defined area.

6.1.6 Using a ParaView

Most of the above analysis is necessary only for the implementers of the *ParaView*, or of any new *View* – the “Developers” of Color Figure CF1. Other users, such as script builders, do not need to know all these details. They only need to know how to construct a *ParaView* and how to change *Parameters* from the default settings. Further, they will be able to ignore many *Parameters*, as those *Parameters* normally need not be changed. For example, in the *ParaView* all the *Axis Parameters* are completely determined by the set of *Columns* provided to the *View*. Similarly, the user will rarely need to change the *LineElement* used by the *LineLayout*, although the user may very well want to change some of its *Parameters*, such as the *Scales* that control line color and size.

The operations of construction and parameter setting may be done by writing appropriate Java code to construct the objects, or by writing script commands for interpretation by the *mmvz* scripting facility (C30). The former approach is likely to be taken by the “Simple coders” of Color Figure CF1, while the later is more suited to the “Scripters”.

We also provide a dialog-box interface that allows the user to change *view Parameters*. The interface automatically determines an object’s *Parameters* and their types and provides allowed choices (e.g., true/false for a boolean *Parameter*). When a *Parameter* requires a more complex type, such as a color *Scale*, *mmvz* checks its global registry of shared objects (C8) and presents a list of all appropriately-typed objects.

6.2 Using *mmvz* to create a Calendar View

Figure 2 shows an *mmvz* view designed for an application in collaborative software development. It is part of a suite of tools designed to facilitate developers who need to communicate with remote locations in their jobs.

The screen snapshot shown contains two views. The lower view is a standard *ListView*, used to show individuals’ plans. The upper view is a *CalendarView* showing both

significant dates (as small colored boxes) and individuals’ plans (as series of linked points). The two views are automatically linked, as they share the *selection* associated with the plans *table* (C34). For example, selecting a range of days in the *CalendarView* will cause the *ListView* to show only those plans occurring on those days.

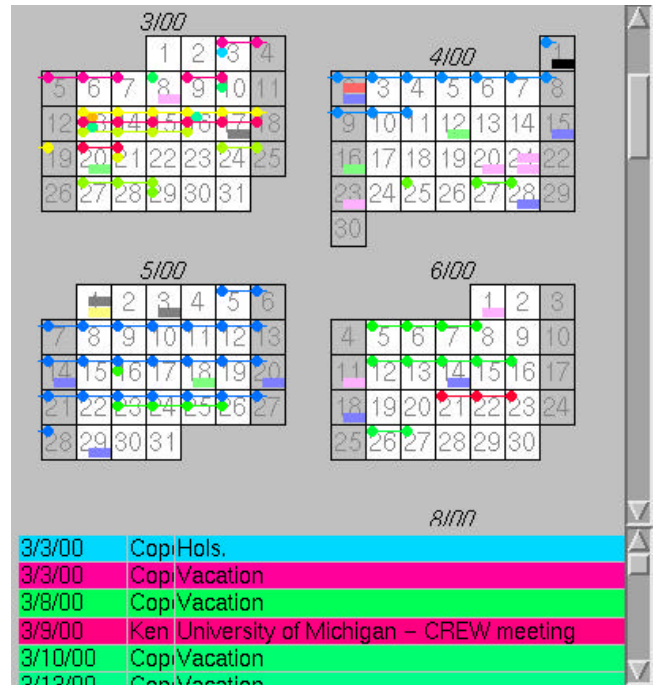


Figure 2. Calendar and List View.

In this section, we sketch how *mmvz* was used to build the *CalendarView*, following the outline used in section 6.1.

6.2.1 Data

Two *Tables* are used by the view. One (“globaldays”) provides holidays and other significant national dates, and the second (“plans”) has people’s schedules.

6.2.2 Layout Paradigm

The *CalendarView* layout paradigm is dictated by the common conventions for displaying calendars, e.g., seven-day weeks. A *CalendarAxis*, derived from *Axis*, is used to provide the screen coordinates for dates.

6.2.3 View Components of a CalendarView

The *CalendarView* contains several *Components* that draw into the same area of the screen under control of the shared *CalendarAxis*. We list the *CalendarView Components* in the order they are drawn (C11).

The *CalendarAxisDrawer* is an *AxisDrawer* that displays an empty calendar – just the boxes, with month titles and day numbers. The areas in which these are to be drawn are obtained from the shared *CalendarAxis*.

A standard *AreaLayout* is used to draw the “globaldays” *Table*. The *CalendarAxis* maps the date of each record to a

screen area, and the *AreaLayout* directs its *Element* to draw the record there. The version of *Element* used renders the record as a colored box or, if the available area is large enough, as an icon. In the *CalendarView*, the icons are national flags, as determined by a scale based on the “country” *Column* of the “globaldays” *Table*.

Finally, another *AreaLayout* is used to draw the “plans” *Table*. This *AreaLayout* uses a *SeriesElement*, which renders each plan as a series of linked points. The *SeriesElement* also handles the complication of stacking icon locations within each day so that icons do not overlap.

The only new classes implemented for this view were the *CalendarAxis*, *CalendarAxisDrawer*, and *SeriesElement*. The implementations were generally quite small, as the commonalities are built into the base classes and only the variabilities need to be defined. For example, the *CalendarAxisDrawer* only had to define its `draw()` method, requiring just 40 lines of code. Further, once the classes were defined, we added them to the library for use by other views. The *SeriesElement*, for example, can be used in a map view to show movement patterns over time.

6.2.4 View Parameters

New *Parameters* for this view are:

CalendarAxis Start/End. The *CalendarAxis* allows the user to specify the start and end month of the data to display.

CalendarAxis Columns. The user can choose the number of columns of months in the *CalendarAxis* (e.g., 2 columns of months are shown in Figure 2).

CalendarAxis Numbers. The numbering of days can be turned on or off.

6.2.5 User Interactions with a CalendarView

The common user interactions (C20-C30) are automatically provided by inheritance. The only additional interaction supported is double-clicking on a month label to select the whole month.

The *CalendarView* is, like the *ParaView*, usable both as a Java class and through the *mmvz* scripting facility.

7. CONCLUSIONS AND FUTURE WORK

Our commonality analysis of the domain of 2D interactive information visualizations resulted in definitions of terms, common building blocks for information visualization applications, and parameterizations of these constituents in the form of variabilities. Furthermore, it encapsulated the experiences of a team that has been building information visualization applications for over a decade. We expect to expand our commonality analysis to the domains of 3D visualization and data retrieval in the future.

The visualization Application Engineering Environment, *mmvz*, was based on the results of the commonality analysis. Our early experiences, illustrated by the examples provided in this paper, show how we can easily and efficiently use *mmvz* to create a variety of very different

visualizations. In the future, we plan to further evaluate the applicability and efficiency of our *mmvz* framework through several avenues, including: conducting user studies; implementing more sophisticated and more specialized views; building higher-level Application Engineering Environments; and inviting contributions and extensions from the wider visualization community through the distribution of documentation, applications, and code.

We believe that a commonality analysis is an invaluable tool for recording and codifying expert knowledge throughout information visualization. The resulting efficient visualization engineering environments minimize replication of effort and focus the research on new and important unresolved questions, while providing practitioners with the latest tools and knowledge.

REFERENCES

- [Ar97] Defining Families: The Commonality Analysis. Mark Ardis and David Weiss. Tutorial given at *International Conference on Software Engineering*, May 19, 1997. Available on the Internet: <http://www.bell-labs.com/user/maa/icse97/index.html>
- [Ca99] *Readings in Information Visualization: Using Vision to Think*. Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman. Morgan Kaufman, 1999.
- [In89] Visualizing Multi-Variate Relations with Parallel Coordinates. Inselberg, A. and Dimsdale, B., in *Proceedings of the Third International Conference on Human-Computer Interaction, Work with Computers: Organizational, Management, Stress and Health Aspects; Interface - Displays and Controls, 1*, pp. 460-467, 1989
- [Ro97] Towards an Information Visualization Workspace: Combining Multiple Means of Expression. S.F. Roth, M. Chuah, S. Kerpedjiev, J. Kolojechick, and P. Lucas. *Human-Computer Interaction Journal*, Vol. 12, No. 1 & 2, 1997, pp. 131-185.
- [Sh96] The eyes have it: A task by data type taxonomy of information visualizations. Ben Shneiderman. *Proc. IEEE Symposium on Visual Languages '96*, IEEE, Los Alamos, CA (September 1996), pp. 336-343
- [Si99] Measuring Domain Engineering Effects on Software Change Cost. Harvey P. Siy and Audris Mockus. *International Symposium on Software Metrics*. 1999.
- [Sp00] Spotfire home page, <http://www.spotfire.com>
- [We99] *Software Product-Line Engineering: A Family-Based Software Development Process*. David M. Weiss and Chi Tau Robert Lai. Addison Wesley, 1999.
- [Wi99] *The Grammar of Graphics*. Leland Wilkinson. Springer. 1999.
- [Wa94] *XmdvTool: Integrating Multiple Methods for Visualizing Multivariate Data*. Matthew Ward. *Proceedings of IEEE Visualization '94* (October, 1994), pp. 326-333.

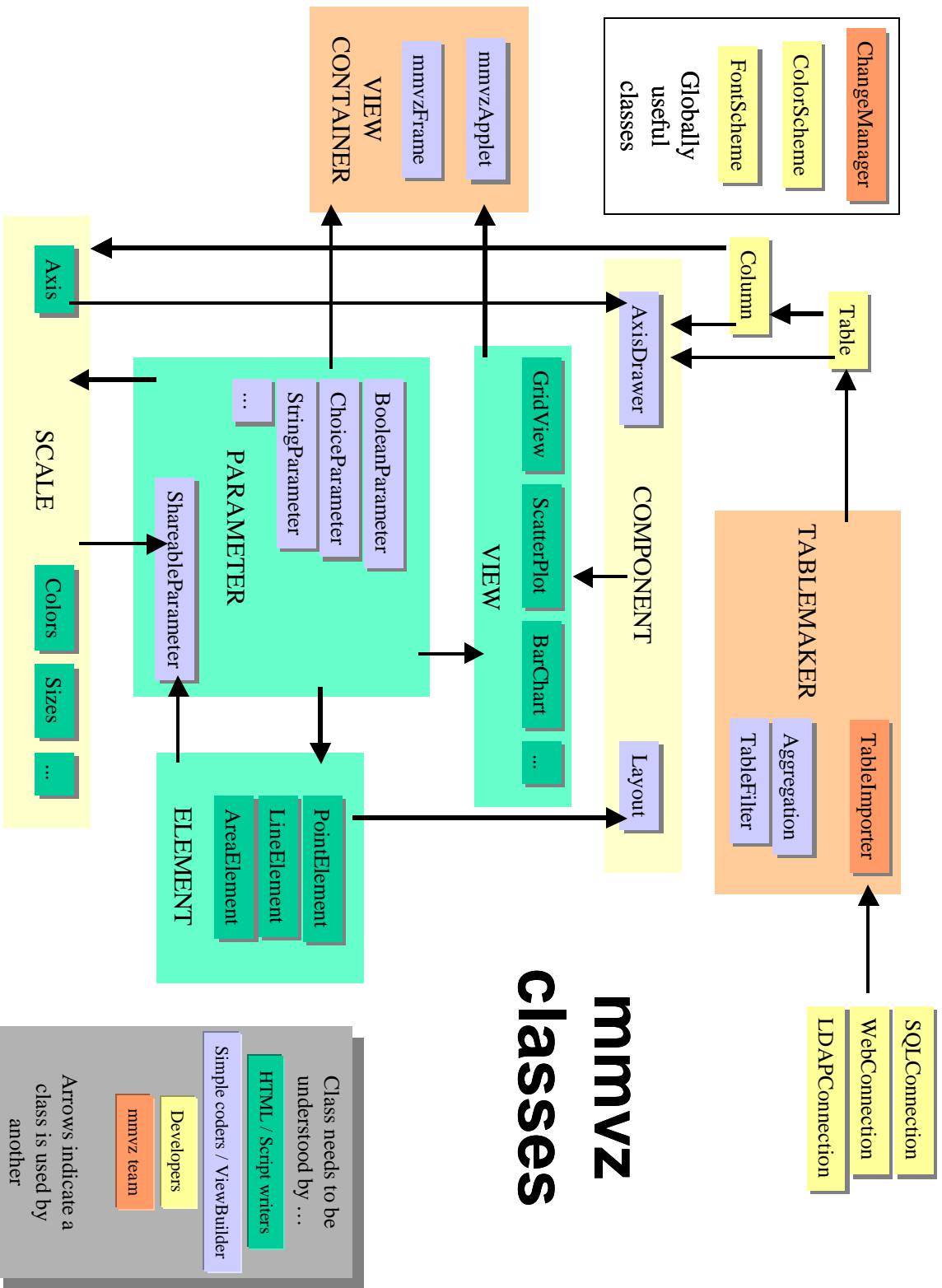
APPENDIX: COMMONALITIES AND VARIABILITIES

Table A-1. Commonalities

No.	Object	Commonality
VIEWS		
C1	<i>view</i>	appears on the screen
C2	<i>view</i>	displays data from one or more data <i>tables</i>
C3	<i>view</i>	must deal with the case when data is not available
C4	<i>view</i>	has exactly one <i>layout paradigm</i>
C5	<i>view</i>	has at least one <i>view component</i>
C6	<i>component</i>	can not be shared between <i>views</i>
C7	<i>component</i>	must be capable of being hidden
C8	<i>view parameter</i>	must be capable of being shared
C9	<i>component</i>	has a defined area with the <i>view</i> where it will be drawn
C10	<i>view</i>	defines where its <i>components</i> are drawn through the <i>layout paradigm</i>
C11	<i>view</i>	draws itself by telling all of its <i>components</i> to draw themselves in a given order
LAYOUTS & ELEMENTS		
C12	<i>layout</i>	has at least one <i>element</i>
C13	<i>layout</i>	uses <i>axes</i> to provide coordinates for its <i>elements</i>
C14	<i>layout</i>	represents exactly one data <i>table</i>
C15	<i>layout</i>	must be able to use any type of <i>element</i>
C16	<i>element</i>	must be able to render <i>records</i> at a point, between two points, and within an area
C17	<i>layout</i>	makes at most one call to an <i>element</i> draw method for each <i>record</i>
C18	<i>element</i>	must have <i>color</i> and <i>selection</i>
C19	<i>element</i>	must use <i>scales</i> to map <i>records</i> to graphical representations
USER INTERACTION		
C20	<i>view</i>	must be capable of allowing <i>rearrangement of components</i>
C21	<i>view</i>	must be capable of being <i>resized</i>
C22	<i>layout</i>	must be capable of <i>drill-down</i>
C23	<i>layout</i>	must be capable of <i>identifying</i>
C24	<i>layout</i>	must be capable of <i>selection</i>
C25	<i>axis drawer</i>	must be capable of <i>identifying</i>
C26	<i>axis drawer</i>	must be capable of <i>panning</i>
C27	<i>axis drawer</i>	must be capable of <i>zooming</i>
C28	<i>element</i>	must be capable of <i>identifying</i>
C29	<i>mechanisms for user interaction</i>	must be consistent for all <i>views, layouts, etc</i>
C30	<i>user interaction</i>	must be scriptable and recordable
DATA		
C31	<i>column of table</i>	has a length equal to the number of <i>records</i> in the <i>table</i>
C32	<i>record of table</i>	has a number of <i>cells</i> , which is equal to the number of <i>columns</i> in the <i>table</i>
C33	<i>column</i>	must have name <i>metadata</i>
C34	<i>table used in linked view</i>	must have a method for obtaining the <i>selection</i> information for each <i>record</i>
C35	<i>table used in linked view</i>	must have a method for obtaining the <i>weighting</i> information for each <i>record</i>
C36	<i>table used in linked view</i>	must have a method for uniquely <i>identifying a record</i>
C37	<i>view, component, element, scale, table, column</i>	must have a unique name within a specific <i>visualization</i>

Table A-2. Variabilities

No.	Object	Variability	Values
VIEWS, LAYOUTS & ELEMENTS			
V1	<i>view</i>	the number of <i>view parameters</i> associated with a <i>view</i>	varies
V2	<i>view</i>	the number of <i>axes</i> in a <i>view</i>	0+
V3	<i>view</i>	the number of <i>axes</i> in a <i>view</i>	fixed or variable
V4	<i>element</i>	whether an <i>element</i> requires <i>parameters</i> to accommodate the situations when they don't have enough information to draw their <i>records</i>	yes/no
V5	<i>element</i>	type of <i>parameters</i> to an <i>element</i>	<i>scales</i> or constants
V6	<i>scale</i>	value produce by a <i>scale</i>	constant or <i>record-dependent</i>
V7	<i>view</i>	the number of <i>layouts</i> in a <i>view</i>	1+
USER INTERACTION			
V8	<i>user interactions</i>	whether <i>user interactions</i> are handled by the <i>view</i> passing the interaction events to the <i>components</i> in a given order, or directly by the <i>view</i>	either
V9	<i>view</i>	whether a <i>view</i> allows <i>rotation</i>	yes/no
V10	<i>view</i>	whether a <i>view</i> allows <i>semantic zooming</i>	yes/no
V11	<i>layout</i>	whether a <i>layout</i> allows <i>rotation</i>	yes/no
V12	<i>axis drawer</i>	whether an <i>axis drawer</i> allows <i>selection</i>	yes/no
V13	<i>axis drawer</i>	whether an <i>axis drawer</i> allows <i>rotation</i>	yes/no
V14	<i>component</i>	whether a <i>component</i> allows <i>identifying</i>	yes/no
V15	<i>component</i>	whether a <i>component</i> allows <i>selection</i>	yes/no
DATA			
V16	<i>cell</i>	value of a <i>cell</i>	missing or measured
V17	<i>column</i>	data status of a <i>column</i>	read-write or read-only
V18	<i>cell</i>	whether the content of a <i>cell</i> can be changed while it is being represented in a <i>view</i>	yes/no
V19	<i>column</i>	whether the length of a <i>column</i> can be changed while it is being represented in a <i>view</i>	yes/no
V20	<i>cell</i>	whether the number of <i>cells</i> in a <i>record</i> can be changed while it is being represented in a <i>view</i>	yes/no
V21	<i>column</i>	whether a <i>table</i> has a <i>column</i> representing the color of <i>records</i> in the <i>table</i>	yes/no
V22	<i>column</i>	whether a <i>table</i> has a <i>column</i> representing the <i>selection of records</i> in the <i>table</i>	yes/no



mmvz classes

Color Figure CF1. mmvz Classes