An Extensible System for the Display of Nested Array Data Structures

by

Christopher J. M. Walmsley

A thesis submitted to the Department of Computing and Information Science in conformity with the requirements for the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
July 1998

Copyright © Christopher Walmsley 1998
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-31267-4
Abstract

Q'Nial, an implementation of the Nial language, uses a relatively fixed technique for displaying the nested array data structures that are the core of the Nial language. Viewing and interacting with nested array data is an integral part of developing and using the Nial language. The current display capability of the Q'Nial interpreter is limited by its use of ASCII characters for generating output and by its inability to effectively display array data that is large in size or has large extents.

This thesis presents an architecture and a system design for the development of improved or alternate visualizations for nested array data. The design allows for full user interaction with the visualizations to enable the ability to edit the array data through the visualization or to interact with the visualization to alter its display characteristics.

The architecture and system design support the concept of a Presentation. A Presentation is the encapsulation of a particular visualization and its editing/interaction capabilities. The Extensible Presentation System (EPS) prototype implementation demonstrates the capabilities discussed in this thesis. Using the EPS prototype, a number of different visualizations are developed.
Acknowledgements

I owe great thanks to my supervisor, Mike Jenkins, who managed to keep the process alive over such a long period of time. I have gained much from my work with Mike, through business and school, that will provide me with a great wealth of knowledge for my future endeavours.

I extend my thanks to the other members of my thesis committee, Janice Glasgow, Nick Graham, and Stan Simmons.

I wish to thank the Department of Computer and Information Science for the opportunity pursue my education in such a drawn out manner, and for the support and friendship of all the staff, faculty and students who I have worked with over the years.

I thank my wife, Dianne, for her continuing love, support and encouragement. Thanks go to Andrew and Benjamin for making the effort more challenging and at the same time more rewarding.
# Table of Contents

Abstract ......................................................................................... i

Acknowledgements ......................................................................... ii

Table of Contents ........................................................................ iii

List of Figures ............................................................................... vi

1. Introduction .................................................................................. 1
   1.1. Motivation ........................................................................... 1
   1.2. Presentation Approach ......................................................... 5
   1.3. Thesis Organization ............................................................... 5

2. Related Work ................................................................................ 8
   2.1. Introduction ........................................................................... 8
   2.2. Data Structure Visualization and Algorithm Animation ........... 11
       2.2.1. Array Diagrams and the Nial approach .............................. 11
       2.2.2. An Iconic Description Language ...................................... 11
       2.2.3. Incense: A system for displaying data structures ............... 12
       2.2.4. Conceptual View of Data Structures ................................. 13
       2.2.5. Automated Drawing of Data Structure Diagrams ............... 14
       2.2.6. The Balsa System ............................................................ 16
       2.2.7. The Information Visualizer .............................................. 16
       2.3. Navigation Techniques ........................................................ 18
       2.3.1. Display of large two-dimensional spaces ............................. 18
       2.3.2. Fisheye Technique .......................................................... 20
       2.3.3. Treemaps ........................................................................ 22
       2.4. Other Research .................................................................. 22
       2.4.1. Elision ............................................................................. 23
       2.4.2. Automated Presentation Design ........................................ 24
       2.5. Application to the Nial language .......................................... 25

3. Proposed Presentation Techniques .................................................. 27
   3.1. Introduction ........................................................................... 27
   3.2. Proposed Presentations .......................................................... 27
       3.2.1. Traditional Presentation .................................................. 27
       3.2.2. Flat Text Presentation ..................................................... 28
       3.2.3. Infinitely Diminishing Presentation .................................. 29
       3.2.4. Nested Lists Presentation ............................................... 30
       3.2.5. Information Hiding Presentation ...................................... 30
       3.2.6. Elided Presentation ......................................................... 31
       3.2.7. Composite Presentations ................................................ 32

iii
3.3. Presentation Analysis .............................................. 33
3.3.1. Traditional and Flat Text Presentations ......................... 34
3.3.2. Infinitely Diminishing Presentation ............................ 34
3.3.3. Nested Lists Presentation ....................................... 36
3.3.4. Information Hiding Presentation .............................. 37
3.3.5. Elided Presentation .......................................... 38
3.3.6. Composite Presentations ...................................... 40

4. Architecture ..................................................................... 42
4.1. Introduction ............................................................ 42
4.2. Objectives of the Architecture ....................................... 42
4.3. Layer Design ........................................................... 45
4.3.1. Array Data Layer ................................................ 47
4.3.2. User Interface Layer .............................................. 47
4.3.3. Presentation Layer ................................................. 48
4.4. Summary ................................................................. 49

5. Detailed System Design ................................................... 50
5.1. Introduction ............................................................ 50
5.1.1. Object Oriented Design .......................................... 50
5.1.2. Classes in the system .......................................... 51
5.2. Relationships between Objects ...................................... 52
5.3. Classes in the Data Layer .......................................... 54
5.3.1. Nial Array Class ................................................ 54
5.3.2. Display Array Class ............................................. 56
5.4. Objects in the User Interface Layer ................................ 58
5.4.1. The VisualObject Class ....................................... 58
5.4.2. ArrayCanvas Class ............................................. 66
5.5. Presentation Layer/Class ........................................... 67
5.6. Important Design Issues ............................................. 69
5.6.1. Recursive Display Generation ................................ 69
5.6.2. Support for Editing and Interaction .......................... 70

6. Presentation Development ................................................ 72
6.1. Introduction ............................................................ 72
6.2. Basic Presentation Development .................................... 72
6.3. Presentation Inheritance ............................................. 74
6.4. Dynamic Presentation Use .......................................... 75
6.5. Composite Presentations ........................................... 76
6.6. Presentation Initialization ........................................... 76
6.7. Rule Based Presentation Assignment .............................. 77
6.8. User Specified Presentations ....................................... 78
6.9. Editing Presentations ................................................ 78
6.10. Presentation Compatibility ........................................ 79
6.11. Presentation Specific State Information ........................ 80
7. Implementation of the EPS Prototype ................................................................. 83
   7.1. Introduction ................................................................................................. 83
   7.2. The ArrayCanvas Class ............................................................................ 83
   7.3. Interaction with the Q'Nial Interpreter ...................................................... 84
   7.4. Sample Presentations ................................................................................ 85
   7.4.1. Traditional Presentation ......................................................................... 86
   7.4.2. Flat Text Presentation ........................................................................... 95
   7.4.3. Infinitely Diminishing Presentation ....................................................... 97
   7.4.4. Nested Lists Presentation ..................................................................... 99
   7.4.5. Information Hiding Presentation ........................................................... 101
   7.4.6. Elided Presentation ................................................................................ 105
   7.5. Composite Presentations .......................................................................... 108

8. Analysis and Conclusions ............................................................................... 113
   8.1. Introduction ............................................................................................... 113
   8.2. Analysis ..................................................................................................... 113
   8.3. The Prototype System .............................................................................. 114
      8.3.1. Application of the Architecture and Design ...................................... 114
      8.3.2. Extending to 3-D ............................................................................... 115
      8.3.3. Composite Presentations .................................................................. 115
      8.3.4. Elision ............................................................................................... 117
      8.3.5. On Screen User Interactions .............................................................. 118
   8.4. Future Work ............................................................................................... 119

References ........................................................................................................... 121

Appendix A Example Presentation ....................................................................... 124
**List of Figures**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>A Sample Q'Nial Session</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Different modes for displaying arrays in Q'Nial</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>A Wide Nested Array</td>
<td>4</td>
</tr>
<tr>
<td>3.1</td>
<td>Traditional Presentation</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Flat Text Presentation</td>
<td>28</td>
</tr>
<tr>
<td>3.3</td>
<td>Infinitely Diminishing Presentation</td>
<td>29</td>
</tr>
<tr>
<td>3.4</td>
<td>Nested Lists Presentation</td>
<td>30</td>
</tr>
<tr>
<td>3.5</td>
<td>Information Hiding Presentation</td>
<td>31</td>
</tr>
<tr>
<td>3.6</td>
<td>Elided Presentation</td>
<td>32</td>
</tr>
<tr>
<td>3.7</td>
<td>Composite Presentation</td>
<td>33</td>
</tr>
<tr>
<td>3.8</td>
<td>Infinitely Diminishing Presentation of a TicTacToe Solution Tree</td>
<td>35</td>
</tr>
<tr>
<td>3.9</td>
<td>Nested List Presentation of a Nial Parse Tree</td>
<td>36</td>
</tr>
<tr>
<td>3.10</td>
<td>Traditional Presentation of a Nial Parse Tree</td>
<td>37</td>
</tr>
<tr>
<td>3.11</td>
<td>Hiding Presentation of a Tic Tac Toe Solution Tree</td>
<td>38</td>
</tr>
<tr>
<td>3.12</td>
<td>Traditional Presentation of a Text Data Structure</td>
<td>39</td>
</tr>
<tr>
<td>3.13</td>
<td>Composite Presentation of a Nial Parse Tree</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>Basic Data Flow</td>
<td>45</td>
</tr>
<tr>
<td>4.2</td>
<td>Enhanced Data Flow</td>
<td>46</td>
</tr>
<tr>
<td>5.1</td>
<td>Basic Objects in the Architecture according to Layer</td>
<td>52</td>
</tr>
<tr>
<td>5.2</td>
<td>UML Class Diagram</td>
<td>53</td>
</tr>
<tr>
<td>5.3</td>
<td>NialArray Class</td>
<td>54</td>
</tr>
<tr>
<td>5.4</td>
<td>Column deletion from a matrix</td>
<td>55</td>
</tr>
<tr>
<td>5.5</td>
<td>DisplayArray Class</td>
<td>56</td>
</tr>
<tr>
<td>5.6</td>
<td>VisualObject Class</td>
<td>58</td>
</tr>
<tr>
<td>5.7</td>
<td>Selection Grouping</td>
<td>59</td>
</tr>
<tr>
<td>5.8</td>
<td>VisualObject Identification</td>
<td>60</td>
</tr>
<tr>
<td>5.9</td>
<td>VisualObjectContainer Class</td>
<td>62</td>
</tr>
<tr>
<td>5.10</td>
<td>Dynamic Offset use</td>
<td>64</td>
</tr>
<tr>
<td>5.11</td>
<td>ArrayCanvas Class</td>
<td>66</td>
</tr>
<tr>
<td>5.12</td>
<td>Presentation Class</td>
<td>67</td>
</tr>
<tr>
<td>7.1</td>
<td>Interface Event Controls</td>
<td>84</td>
</tr>
<tr>
<td>7.2</td>
<td>Emulation of Current Q'Nial Capabilities</td>
<td>86</td>
</tr>
<tr>
<td>7.3</td>
<td>A Copy/Paste Edit</td>
<td>92</td>
</tr>
<tr>
<td>7.4</td>
<td>Layout of Annotations</td>
<td>96</td>
</tr>
<tr>
<td>7.5</td>
<td>Emulation of the display primitive</td>
<td>96</td>
</tr>
<tr>
<td>7.6</td>
<td>Annotations for Higher Dimension Arrays</td>
<td>97</td>
</tr>
<tr>
<td>7.7</td>
<td>Infinitely Diminishing Presentation</td>
<td>98</td>
</tr>
<tr>
<td>7.8</td>
<td>Nested List Presentation</td>
<td>100</td>
</tr>
<tr>
<td>7.9</td>
<td>Independent un-hiding of a Hiding Presentation</td>
<td>102</td>
</tr>
<tr>
<td>7.10</td>
<td>Elided Presentation</td>
<td>106</td>
</tr>
<tr>
<td>7.11</td>
<td>Composite Presentation (sample 1)</td>
<td>109</td>
</tr>
<tr>
<td>7.12</td>
<td>Composite Presentation (sample 2)</td>
<td>109</td>
</tr>
</tbody>
</table>
Chapter 1. Introduction

1.1. Motivation

Nial is a high-level language that combines concepts from APL and LISP [Jenkins 1986]. The central feature of Nial is the nested array data structure. This is based on a mathematical theory of arrays [More 1973][More 1979]. Nial arrays can be of arbitrary dimension and can be arbitrarily nested. The individual elements of arrays can be other arrays (nesting) or atoms. Atoms represent traditional data types such as integers, real numbers, characters or string data, etc. Unlike many traditional programming languages Nial arrays can contain atoms of different types (heterogeneous).

Complex data structures can easily be created and represented using the Nial language. Thus, an important aspect of working with Nial is to allow the user to interactively display arrays with which he or she is working. This ability to see arrays graphically as the user is working with them is a very powerful development tool.

Current Q’Nial [Jenkins 1993] implementations provide a traditional command line interface for interacting with the Nial language. Much development with Q’Nial is accomplished through interactive program development, particularly for prototyping. Users tend to develop code experimentally, by trying different Nial primitive operations on arrays to see the result. This makes the display of Nial arrays a critical aspect of the language’s usefulness. Figure 1.1 illustrates a typical session.
The sequence of commands issued is typical of program development in Nial. Nial primitives are combined and applied to arrays experimentally until the desired result is achieved. The example in figure 1.1 uses rather simple array data, but the basic display capability of Q'Nial is illustrated in the boxed diagrams that depict the array values.

The current display capabilities available in Q'Nial provide a clear and intuitive depiction of most Nial arrays. Q'Nial does offer some control over how arrays are displayed on screen. A “diagram” mode can control whether simple arrays are drawn with boxes or not, and “sketch” mode can control the addition of extra decorations to assist in clarifying the types of atomic array data. Figure 1.2 shows a simple heterogeneous array displayed using the different modes.

While the current display method is clearly usable and effective, it does have a number of limitations.
Array displays are created using ASCII characters for output. Array pictures are created in their entirety within the Q'Nial interpreter and then are cut horizontally into sections (the width of the screen) to be paged out to the screen. For arrays that have a depiction width that is less than or equal to the width of the screen, the output may be quite readable. For arrays that have greater widths, the output will be broken into pages and displayed. This paging can lead to very discontinuous output. This is especially so when the vertical size of the depiction is large as well.

Figure 1.3 shows the output of a sample arrays where the width of the depiction exceeds the screen width and becomes paged. It is easy to see how the user can become disoriented when trying to interpret the output. The user must visually try to connect array boxes at the far right of the screen with their corresponding continued boxes on the left of the screen.
Figure 1.3 is a relatively simple example the general problem where arrays have large extents or deep nesting.

Q'Nial provides no capability to interact with the array diagrams once they are displayed. They are a uni-directional method of interacting with the Q'Nial interpreter.
1.2. Presentation Approach

This thesis focuses on the idea of a Presentation as an embodiment of how array data can be visualized and how it can be interacted with. An architecture and system design is presented that can provide a programming environment where Presentations can be developed with relative ease.

The system design provides support capabilities to allow the generation of greatly varied visualizations while at the same time providing general support facilities to allow complex user interaction with these visualizations.

The focus of a Presentation is to be able to encapsulate the visualization technique and the user interaction capabilities into a unified Presentation object. Different presentations can be used interchangeably to provided different visualizations and interaction techniques for the same array data.

A prototype implementation of the system architecture and design is discussed in detail in Chapter 6. The Extensible Presentation System (EPS) is a Java based implementation that provides most of the capabilities discussed in this thesis.

1.3. Thesis Organization

The next chapter examines research into two main areas that relate to this thesis. First is the area of visualization techniques. This is the study of effectiveness and usability of different visualization techniques. The second area of study is system designs and architectures to support visualizations. This thesis falls into the second
category but relies on the first for examples of how a visualization support system might be used.

Chapter 3 introduces some Presentation techniques that were chosen to provide potentially improved visualization techniques for Nial arrays, and to provide a test bed for the EPS prototype system. The Presentation techniques are evaluated to access their ability to address the issues raised in Chapter 1, and for their general applicability to particular array data.

Chapters 4 and 5 present an architecture and a system design to support the concept of a Presentation as a model for producing visualizations for array data. The architecture presents a layered approach that is based on division of responsibility and on information flow.

Chapter 6 discusses some the important issues involved in developing Presentations within the system design and architecture.

Chapter 7 presents the EPS prototype system and described in detail how a number of sample visualizations were created with the system architecture and design. Sample code is included to illustrate the relative simplicity of the code necessary to create fully functional Presentations.

A sequence of sample Presentations are presented. The sample Presentations start by modelling the current display capabilities of Q'Nial but then show how the system
design and architecture can be used to created more capable and useable Presentations.

Chapter 8 summarizes the work presented in this thesis and then discusses some of the interesting issues that arose from the development of the EPS prototype. The EPS system provides a prototype foundation for exploring many different visualization techniques for Nial arrays. Issues involved in developing a complete implementation that is integrated into a Q'Nial environment are also discussed in this chapter.
Chapter 2. Related Work

2.1. Introduction

This chapter discusses research that explores designs and architectures to support visualizations and visualization techniques.

Examining visualization techniques is of value in order to establish what potential techniques may be applicable to viewing Nial array data, and what capabilities the design and architecture proposed in this thesis should be able to support.

Exploring research into other architectures and designs for developing or supporting visualizations is important in order establish if there are design techniques that would be of benefit to the architecture proposed in this thesis.

There are areas that do address some of the visualization issues that are problematic in the current Q'Nial environment. Three of them are of specific relevance. These are the areas of Data Structure Visualization, Algorithm Animation and Data Navigation.

While Nial arrays differ from data structures in traditional languages (such as C and Pascal), where the user can define the fields and name the structure, arrays are none the less data structures. They provide structure for encapsulation of data, but provide much more flexibility. The Nial language has only one type, the array, but that one type is capable of representing almost any type of traditional data structure. This
leads to the examination of how other languages and systems attempt to display data structures.

Chang [Chang 1987] provides some classifications for languages that use different degrees of graphical interfaces for user interaction. While Nial is not considered a graphical language, the objects within the language (arrays) have a natural graphic representation, and thus falls into classification of “Languages that support visual interaction”.

It is often the case that using an appropriate visualization for traditional data structures can greatly facilitate the debugging or understanding of the data being displayed or the behaviour of a program. Linked lists and tree data structures are examples where a good visualization can provide a clear understanding of the behaviour and use of such data structures. This area of study is call Data Structure Visualization.

While the underlying implementation of Nial does involve pointers to handle hierarchically nested arrays, these details are entirely hidden from the user. Thus it could be argued that Nial arrays do not require the user to switch between an implementation view and a conceptual view.

Algorithm Animation is a research area that, by its own definition, subsumes the field of Data Structure Visualization. The idea behind Algorithm Animation is to impart to the user/viewer some aspect of the execution of the program through some graphical
representation or representations. These representations are often graphic views of traditional data structures. Depending on the desired property of the executing program to be shown, these views are sometimes heavily abstracted from the underlying data structure.

Algorithm Animation systems incorporate methods to display data structures, or at least abstractions of them. It is this aspect of this research that is of interest.

While the previous two areas may help provide insight on how to improve the display of arrays, we must remember that, for most arrays, Q'Nial is very capable of presenting an understandable view. Where Q'Nial is weak is in the area of Data Navigation.

The area of Data Navigation deals with the problems that arise when the information of interest to the user exceeds the screen capabilities. The basic premise is, through some display technique, to help prevent the user from becoming lost in the data. This is one of the weaknesses of the current Q'Nial display method. Several techniques in the area are examined.

Other research that does not fall into the above categories is also presented. This research may provide some additional insight into the problem.
2.2. Data Structure Visualization and Algorithm Animation

Data Structure Visualization and Algorithm Animation have been combined because they share many of the same concepts. Specifically, Algorithm Animation embodies and extends Data Structure Visualization into a more complete system for visualization of program execution.

2.2.1. Array Diagrams and the Nial approach

Jenkins and Schmidt [Schmidt 1982] describe a method for displaying Nial arrays that is the basis of the display method for arrays in current QNial implementation. Array displays are constructed in a hierarchical manner. At the bottom level are "primitive diagrams". These are methods of drawing the simple arrays in Nial (singles, list, and tables/matrices). Array nesting is dealt with next, followed by higher dimensional arrays. The special cases of empty arrays are discussed, as well as a sketch mode of display which reduces the screen clutter by eliminating the lowest level of container drawing.

The display method discussed is based purely on text based graphics and does not deal with issues involving effective display techniques for arrays with large extents or deep nesting.

2.2.2. An Iconic Description Language

The authors of DISL (Data Structure Iconic Language) [Seminar 1990] developed a system that allows for the definition of how Modula-2 data structures can be displayed. The visual objects that the user can define are referred to as Icons, though in reality
there are more like tables. These tables can be defined to represent the attributes and contents of data structures using a simple Icon description language.

The system deals with traditional data structures with simple base types and the contents of the icons tend to be traditional base types such as integers and reals.

2.2.3. Incense: A system for displaying data structures

The Incense system [Myers 1983] was one of the early systems that truly attempted to provide a general system of data structure display. At the time, few systems were able to create dynamic displays of data structures during program execution. Those that could were usually restricted in some significant manner.

The Incense system provided default formats for the display of the base types in the system language\(^1\). Additionally, Incense was the first language to allow multiple formats for a single data type/structure to be displayed concurrently. Most prior systems usually allowed data structures to be displayed only after program execution was complete.

One of the fundamental design principles of Incense was that the traditional display method for a data structure might not be the most informative, but a system that would allow the user to define analogical pictures for data structure would be significantly more useful. These user defined analogical pictures are simply higher level abstractions of the data structure involved.

\(^1\)The language used was called Mesa, a language similar to Pascal
The authors of Incense discussed the capability of interactive editing, but had not yet implemented it. The system does allow the user to select, move and change formats for currently displayed objects.

Myers deals with the problem of presenting data that is too large to be displayed completely (in full detail). A basic capability to deal with this problem is present in the system. The system allows complex algorithms to be embedded into the display formats. The system user can resize the view window of a data structure at any time, and the view format must be able to redraw the display to maintain the same format.

2.2.4. Conceptual View of Data Structures

Graham's thesis [Graham 1988] builds on work started by Myers in the Incense system and other systems [Brown 1985a] [Brown 1985b]. The central idea in Graham's thesis and the prototype system Weasel is the "Conceptual View". A conceptual view is created using the mapping language designed specifically for the task. The intention is that the Weasel user would never program in the mapping language directly, but would use a higher level graphic interface to define the mapping from data structure to conceptual view.

The mapping language is the critical link between traditional data structures and their conceptual views. The mapping language Graham has developed is suitably powerful and flexible enough to deal with complex graphic representations.
Graham does discuss the details of data structure layout and elision and discusses techniques to deal with the situation where the contents of a view will not fit in the window. These capabilities were not actually implemented in the Weasel system, but the intention is that the system would include a number of predefined layout methods, and predefined elision and navigation strategies that the programmer could draw from.

2.2.5. Automated Drawing of Data Structure Diagrams

Ding and Mateti [Ding 1990] take a unique approach to data structure visualization. They take a very general approach to the problem of drawing data structures diagrams. They don't examine a particular system with particular data structures but instead, study a diverse group of data structure diagrams in order to develop a general methodology.

Their approach deals with all aspects and uses of data structure diagrams (DSDs). Their uses are generally classified into the following broad categories: Debugging, Program Understanding, Program Design, Program Visualization and Visual Programming. In particular they are interested in the following points:

1. What makes the displays look pleasant or unpleasant,
2. How to draw diagrams aesthetically and automatically,
3. How do diagrams affect our productivity.

Some of the factors that affect the aesthetics of a DSD are visual complexity, regularity, symmetry, consistency, modularity, sizes, shapes, separation, and traditional ways of drawing.
The authors discuss each of these factors in detail and develop a set of objectives and rules to guide DSD drawing. For each of the above factors they develop implementable rules that satisfy the general objectives. The key aspect of these rules is that they are implementable, and can be incorporated into a system.

The authors describe a rule system capable of implementing a general system for DSD drawing. In such a system, rules are hierarchically organized data structures\(^2\). Each sub-rule (data structure) is a specialization of the parent rule, defined by some measurable condition. Each node in the hierarchy will have a method for the DSD layout, and the ability to compute the "badness" of a particular layout attempt. Weightings can also be assigned to each node. The weightings and "badness" factors allow the system to resolve conflicting rules with the intended result being the better choice in the conflict.

This approach is significantly different from many other systems. It is common that the aesthetics are hard wired into a diagram method for a particular system, but here, the authors propose a system where the aesthetics of a DSD are a dynamic and flexible part of the drawing system.

\(^2\)Not to be confused with the DSD actually being drawn
2.2.6. The Balsa System

In [Brown 1985a], Marc Brown describe a system that is based around a high level interface that allows a user to interact with dynamically changing graphical representations of his or her programs.

The system has four distinct modes of operations. These modes are:

Algorithm Designer
   In this phase, the designer actually develops the programs of interest and identifies the interesting events and data. Work is also done to design the graphical representation of the data objects of interest.

Animator
   The animator composes "views" from the results of the previous phase.

Scriptwriter
   The scriptwriter combines views, ordering and program execution into a complete presentation.

User
   The intended user of the system (student).

Brown discusses the entire system in significantly more detail in [Brown 1985b].

2.2.7. The Information Visualizer

The Information Visualizer [Card 1991] is a user interface for an information retrieval. The system is based on the concept of 3D Rooms or Information Workspaces. Also discussed are techniques to match the display and animation capabilities of the interface with the perceptual capabilities of the user.
Each of the Rooms in the experimental system they developed presents a different user interface that is suited to the information retrieval task at hand. What is relevant to this thesis are the techniques used in these Rooms.

While all of the user interfaces developed are 3D in nature, the basic principles apply to many visualization problems, including the issues raised in chapter 1.

Of particular interest are the Cone Trees [Robertson 1991]. This interface technique uses 3D Cones to represent hierarchical information such as file system directory structures or organizational charts. The interface include full user interaction capabilities so that information not at the front of the visualization can be easily brought to the front.

Another user interface used in the Information Visualizer is the Perspective Wall [Mackinlay 1991]. This technique is similar to work on Fisheye view by Schaffer [Schaffer 1993] (see section 2.3.2). The technique address the issue of how to provide a navigable presentation of information that would otherwise not fit onto a single screen.

The technique uses a folding metaphor for displaying information. A 2D representation is mapped to 3D panels. In the centre of the screen is a true 2D panel of the area of information in focus. On each of the left and right sides on the centre panel are panels that appear to fold back into the distance. These folded panels provide visual context
for the current area of focus. The detail on the folded panels is almost complete near the fold and becomes indistinguishable as the folded panel fades into the distance.

2.3. Navigation Techniques

This area of research has the most relevance to addressing the weaknesses in the current Q'Nial environment.

2.3.1. Display of large two-dimensional spaces

Beard and Walker [Beard 1990] have attempted to deal specifically with the problem of displaying information that has an existing and usable 2-D representation, but where the volume of information exceeds the capabilities of the display screen.

Some examples of such situations are systems such as Hypertext systems, where the idea is to provide links between related textual information. The information is always considerably more vast that the available screen space. Another example is that of Geographical Information Systems. The detail in which maps are stored necessarily disallows the user from seeing fine detail in the context of the entire map.

These, and other types problems lead to the study of Navigation. The main goals are to provide a interface to the data which reduces the mental overhead needed to handle the following situations:

  * "I want to be at such-and-such a location" ... "get me there fast!"
  * "Where am I?"
The second situation is the problem of “getting lost”, and the first relates to the efficiency of the interface. Both of these points become moot when the entire data space can fit onto the screen.

To deal with the two problems stated above, the authors use a system that includes a map window. The map window will always be visible in addition to the normal data window. The map window contains a graphical picture of the entire data space, the details (graphic details) of which may not be discernible. The normal data window will only display a subset of the data represented in the map window, but in full detail. Additionally, in the map window, there will be a rectangular outline that indicates the area of the data currently in the normal data window.

The second question “Where am I?” will be answered by the location of the rectangle in the map window.

To deal with the first problem of how to change locations in the data efficiently, the authors use a combination of two techniques. The first technique is called *Zoom Navigation* and the second is called *Roam Navigation*.

*Zoom navigation* allows the user to change the location and magnification of the current main display by dragging a new rectangle out in the map window. Using this technique, the user can directly specify the location to be viewed, and by varying the size of the rectangle, they can control the zoom factor or scale used in the main display window.
The roam technique is similar, in that it uses a rectangle in the map window to represent the current view in the main display window. The difference is that the size of the rectangle in the map window is fixed, but the user is able to drag the entire rectangle around the map window. This way, the user can move about the data by simply dragging the rectangle outline in the map window.

These systems use the map window along with the roam technique (the Zoom technique would not be particularly practical for general windowed interfaces) to allow the user to have a virtual screen many times larger than their physical screen.

2.3.2. Fisheye Technique

Schaffer et al. [Schaffer 1993] expand on a navigation technique that takes a different approach that addresses some of the issues raised in chapter 1. Specifically, the authors believe that the number of windows on the screen should be minimized.

Their approach to the problem of “getting lost” is to use what they termed a *Fisheye* technique. The basic idea of this technique is to use a single window to display all of the data, but allow the user to *focus* on a particular area of the data. The area the user has chosen to *focus* on will be disproportionally expanded relative to the rest of the window. This is analogous to using an extreme wide-angle lens on a camera, in that the area in the centre of the lens appears much closer to the viewer than the periphery of the image.
The authors acknowledge that there are a variety of different approaches to the fisheye technique. The three general categories are:

**Distorted Views:**
This method involves altering the size and location of some data objects on the screen in order to accommodate the **focused** objects which will be expanded (in size) to allow their entire detail to be seen.

**Partial Views:**
This method is an extension of the Distorted View, but with the additional feature that objects that become too close to the periphery of the **focused** object(s) may be left out of the diagram all together.

**Alternate Representations of Screen Information:**
This method involves having a variety of different screen representation for the objects being viewed. These different representations would necessarily be of different sizes.

The authors developed a system that used a variable zoom technique, which falls into the category of the Distorted view method. The difference is that in an **unfocused** state, many individual objects would be grouped into encapsulating objects (the grouping reflects the semantics of the data). As the user focuses on an object on the screen, the selected object would expand to reveal its component parts, and the remaining objects, previously displayed on the screen would shrink in size, and possibly move towards the periphery.

The authors tested this method against a full-zoom technique, where, as the user focused on a grouped object, its component parts would be displayed, and the previous objects on the screen would disappear.

Once again, the results demonstrated that providing the user with a context in which to view the data improved the user's ability to successfully navigate through the data.
2.3.3. Treemaps

An approach for dealing with large amounts of hierarchical data called Treemaps has been developed [Johnson 1991] [Turo 1992]. This approach deals with both data structure visualization and navigation at the same time. The technique is only suitable when the fine details of nodes in a data hierarchy are not important, but the overall structure is.

The technique involves abstracting each successive level in a hierarchy into nested rectangles. The orientation of the rectangles at each level of nesting alternates from vertical to horizontal. The top level rectangle might be subdivided horizontally to hold its immediate children. Each of the children will then be subdivided vertically to hold the next level of children. This repeats to the bottom level. Additionally, rectangles may include other visual cues such as colour to aid the viewer.

While the applications of such a method may seem restricted in its uses, it is an interesting technique that utilizes the entire screen space available, and can impart a great deal of hierarchical information. One of the examples for such a technique might be a directory/file hierarchy on a typical computer system. Rectangles are coloured to indicate file type. This might allow a system administrator to locate unexpected buildups of user files, or temporary files somewhere on the system.

2.4. Other Research

This section presents some additional research that does not fall into the previous categories but may be relevant to the ideas presented in this thesis.
2.4.1. Elision

A system such as TuringTool [Cordy 1990], a system for software management, attempts to deal with the problems of large software systems and the maintenance task. The large size and hierarchical nature of software systems provides a strong analogy to large 2-D data spaces [Beard 1990] [Schaffer 1993](discussed above).

TuringTool has two clear objectives. The first is to aid the programmer/maintainer in moving around the software system (navigation), and the other is to aid the programmer/maintainer in locating areas of interest that relate to some aspect of program context.

The first objective is a navigation problem. The authors specifically attempt to avoid graphical abstractions of software hierarchies that require the user to “switch” between views (from graphical to textual). Instead they use a system called “hierarchical source elision” as the normal navigation technique.

This technique is achieved by expanding and compressing text. The user can examine source code at any level\(^3\). Levels below the current viewing level are not shown (an elision), but represented by ellipsis (...). The user may select and expand any source text that is elided, as well as selecting any currently visible text and compressing it. This has the effect of hiding the level selected and all levels below that.

---

\(^3\) Levels: meaning software levels such as: system, modules, procedures, blocks, statements, etc.
In addition to the hierarchical source elision strategy, that authors also include a powerful non-structural elision system. This is a system called “rule based source elision”. This technique is used to locate areas of interest in a program in a non-structural manner. For example, if the user was interested in seeing all occurrences of a particular variable throughout the program, or the user wanted to see all calls to a particular procedure, they might normally have to use complicated, non-integrated tools such as the Unix command grep to accomplish the task of locating these areas of interest. Using rule based source elision, the task described above is straightforward and tightly integrated into the TuringTool system.

The rule based elision system provides a set of primitive rules to select text by. These rules are based on program content such as assignment, declaration, calls, etc. These primitive rules can be combined with standard set theoretic operations to provide an exceptionally powerful text selection technique.

2.4.2. Automated Presentation Design

Mackinlay provides an early work [Mackinlay 1986] in an effort to automate visualization of relational information. He refers to these visualizations as Presentations.

His Presentations are based around a formal definition of graphical languages. He defines the expressiveness of these languages as a function of their syntax and semantics. He uses a conjectural theory to establish the effectiveness of these languages.
He goes on to explore the concept of composition of these graphical languages. With a set of primitive graphical languages and using a composition algebra, in conjunction with the expressiveness and effectiveness criteria, he is able to produce general automated Presentations that can adapt to the information currently being visualized.

2.5. Application to the Nial language

A general conceptual view for Nial arrays exists [Schmidt 1982] and is well accepted by current users of Nial. For this reason I don't believe that a complete redesign is necessary. An integration of some of the techniques described in the above research could extend the current capabilities of Nial.

The Map window technique [Beard 1990] could be used for displaying Nial arrays. The current display method could be left unchanged, except for the inclusion of a map window. If the array being displayed became too large for the viewing area, a map window might appear and the main display would only show a zoomed portion of the entire array. The user could then zoom and or roam using the map window. Though such a system would be an improvement over the current system, I hesitate at the use of extra windows that could clutter the current interface to Nial.

A modified system that combined both elision [Cordy 1990] and a navigation technique such as the Fisheye technique [Schaffer 1993] could be effective. Small arrays could appear as they currently do, but any array that would exceed the available screen space (while maintaining all detail) could have some of its lower level nesting elided. If the user wanted to see more detail, they could mouse click on the elided sub-array, and it
would expand to a readable size. To accommodate the focused sub-array, the remainder of the array diagram could have the Fisheye technique applied to it.

Though somewhat beyond the scope of the problem, the possibility of having a rule based display strategy could be of some use. In a debugging situation a particular atom value or sub-array might be of specific interest, and one could imagine a rule that allows the user to specify that all atomic values be elided in the array except those which match some pattern. This kind of a technique could be used while debugging to track down erroneous data.

In a related application, any improved system for the display of Nial arrays could be included into a debugging or animation extension to the language (as discussed in section 2.3). This could prove to be a valuable tool for the Nial programmer.

If the system for display of arrays were efficient enough, animation could added to the interpreter in the form of a “data watch”. The user could specify a particular variable to “watch” in a window. Subsequently during further program execution the “watch” window would maintain a current view of the variable. This could be a valuable debugging feature, as well as a powerful tool for experimenting with Nial.
Chapter 3. Proposed Presentation Techniques

3.1. Introduction

The chapter will discuss and analyse, in advance, the Presentations that were developed with the EPS prototype (chapter 7). There are presented early in the thesis to provide a concrete motivation for many of the architecture and design decisions that are discussed in the next two chapters.

The Presentations discussed were chosen to both validate the architecture and design proposed in this thesis, and to address the current visualization issues motivating this thesis (chapter 1).

Section 2 briefly outlines each Presentation and its intended capabilities and goals. A simple output sample is provided for each Presentation. Section 3 provides an analysis of each Presentation, including the output from the EPS prototype demonstrating the capabilities and application of the technique.

3.2. Proposed Presentations

The following sections provide a overview of the basic capabilities and desired functionality of the Presentations used in this thesis.

3.2.1. Traditional Presentation

This Presentation models the current display method used for Q'Nial. Nested arrays are depicted using nested rectangles, lists, or grids. Arrays of simple values (atoms)
are draw without a bounding container, and all other arrays include containers to illustrate the shape of the array.

Figure 3.1: Traditional Presentation

Figure 3.1 shows 4 matrices nested within each other. The deepest matrix (containing the values 1, 2, 3 and 4) does not have bounding grid lines.

3.2.2. Flat Text Presentation

The Flat Text Presentation is modelled after the output of the Nial primitive 'display'. The 'display' primitive produces a simple character string as output. The output is a pure Nial expression, and thus can be reinterpreted to reproduce the original array.

Figure 3.2: Flat Text Presentation
Figure 3.2 illustrates a simple nested array. This technique provides a greater benefit as a representation form that can easily be stored and reinterpreted to reproduce array data than it does as a presentation technique.

3.2.3. Infinitely Diminishing Presentation

This Presentation is an adaptation of the Traditional Presentation. It maintains all of the basic behaviour of the Traditional Presentation with the additional behaviour that font size and cell padding is reduced incrementally at each deeper level of nesting.

Figure 3.3: Infinitely Diminishing Presentation

Figure 3.3 shows the same array data as in figure 3.2. The font size used for the numbers is reduced at deeper levels of nesting. The padding between the nested array grid lines is also reduced. There are lower bounds on how small the font size can be and how small the cell padding amounts can be.

The propose of the presentation technique is to less on screen area to display nested arrays. Structure and content are preserved to the degree allowed by the lower bound on the font size.
3.2.4. Nested Lists Presentation

This Presentation technique does not use nested boxes or grids to depict nesting, but instead uses nested lists. These lists use extra annotation to assist in understanding the presentation.

---

Figure 3.4: Nested Lists Presentation

Figure 3.4 again shows the same data as in figure 3.2 and figure 3.3. The first node of each list is a meta-node used to indicate the shape and type of the array in the list. Subsequent entries in the list represent the array data. Entries whose value is a nested array are denoted by a small square. Entries without the small square are atomic values.

3.2.5. Information Hiding Presentation

This Presentation is again based on the Traditional Presentation. The difference is that, initially, all array data at a nesting depth greater than 2 are hidden. A small grey rectangle is used to indicate hidden array data. The user may then select any element of the display and request that the depth at which hiding occurs be changed. The
depth of hiding can be increased or decreased, and the change is localized to the array selected and its children.

Figure 3.5: Information Hiding Presentation

Figure 3.5 shows a sequence of screen shots showing the process of ‘unhiding’. The array data is the same as in figure 3.1. Notice the localized effect of the unhiding behaviour.

This Presentation technique is intended to allow exploring of large and deeply nested data.

3.2.6. Elided Presentation

The technique used in this Presentation is similar to the Information Hiding Presentation in that some array is not displayed in order to make more efficient use of screen space.
This technique differs in that this elision behaviour is not controlled by the nesting depth of the data, but by the extent of the data. Array data with any extent greater than 5 (for the sample in figure 3.6) is reduced to 5 by replacing elided data by elipses.

![Figure 3.6: Elided Presentation](image)

User interaction is a key aspect to this presentation technique. Arrays can be selected and their elision parameters can be adjusted to show more or less data.

3.2.7. Composite Presentations

A composite Presentation is the use of more than one base Presentation to produce the desired output. Which Presentations are used for each array can be controlled by nesting depth, array type, array shape, content information, or user input.
Figure 3.7 shows a simple composite Presentation where the Nested Lists Presentation is used for the top two levels of nesting, and the Infinitely Diminishing Presentation is used thereafter.

3.3. Presentation Analysis

The previous sections in this chapter outline the basic capabilities of the Presentations developed for this thesis. This section discusses the capabilities and applicability of these Presentations.

All of the Presentations developed include the ability for the user to interact with the on-screen display of the data. User interaction is explored in more detail in Chapters 5 and 7.
3.3.1. Traditional and Flat Text Presentations

These two Presentations were chosen as benchmarks for the EPS prototype system. Because they are based on the same capabilities that are available in Q'Nial, they possess the same limitations outlined in Chapter 1.

They do add significant capabilities, in that font size and typeface can be altered, padding and other layout parameters can easily be adjusted and most importantly that user interaction is allowed.

3.3.2. Infinitely Diminishing Presentation

Figure 3.8 shows a pruned solution tree for the Tic Tac Toe game. The recursive data structure consists of a 2 element array. The first element is the current board layout, and the second element is a list of recursive pairs representing the list of possible “next” moves. The recursion ends when a “won” game occurs.

The Infinitely Diminishing Presentation used in figure 3.8 allows a significant portion of the solution tree to be displayed. If the Traditional Presentation were used, very little of the tree could be displayed.
The main drawback of this presentation technique is that the detail at deep nesting levels becomes very hard to read. The structure and nesting of the output is maintained. This presentation technique might be applicable to situations where it is important to verify the structure of the data being viewed without being overwhelmed by detail.

The technique could be enhanced by adding the ability to alter or override the scaling factor used to adjust the font size and cell padding. The ability would allow the user to focus on the detail that would otherwise be obscured at deep nesting levels. The Hiding Presentation (Section 3.3.4) illustrates how such a parameter override works.
3.3.3. Nested Lists Presentation

The List drawing capability available in the EPS prototype is quite simple but it can be effective. Figure 3.9 shows the display of an annotated Nial parse tree for the simple expression “2 * (3 + 4)”. QNial uses its own nested array data structure to store the internal parse trees used by the evaluator. The tree structure of the parse tree make it well suited to the Nested List Presentation technique. The user's eye is easily able to following the tree structure and reconstruct the meaning of the tree.

\[ \text{Figure 3.9: Nested List Presentation of a Nial Parse Tree} \]

By contrast, the Traditional Presentation technique can become very confusing for larger parse trees. Figure 3.10 show a portion of the same parse tree using the Traditional Presentation technique.
Figure 3.10: Traditional Presentation of a Nial Parse Tree

Much less of the parse tree is visible and for that reason the tree is harder to reconstruct mentally.

The Nested Lists add extra annotation to indicate the original shape of the array data. This is done because the Lists flatten the array into lists regardless of their original shape. This extra annotation can be confusing. It adds extra screen clutter, especially when the array are already lists.

3.3.4. Information Hiding Presentation

Figure 3.11 show a similar Tic Tac toe solution tree to figure 3.8. This tree is pruned and does not contain the solutions down to “won” games. In this figure the state of the display is that of unhiding the solution path of 2,6,5 (0 based origin).
Figure 3.11: Hiding Presentation of a Tic Tac Toe Solution Tree

The hiding technique allows almost complete and readable representation of the nesting structure and shape of the data. At the same time the user can selectively explore the data in full detail by following paths of nesting. The obvious drawback, which is also the benefit, is that not all of the data detail can be visible.

3.3.5. Elided Presentation

The elision technique used in the Elided Presentation shares that same basic characteristics as the Hiding Presentation. The sacrifice is that some of the data is not visible for the benefit of a higher level view of the data.

Figure 3.12 show a data structure that hold sentences as lists of words. Browsing such data structures when developing or debugging text processing software could quickly become overwhelming.
Figure 3.12: Traditional Presentation of a Text Data Structure

Figure 3.13: Elided Presentation of a Text Data Structure

Figure 3.13 show the same data using the Elided Presentation. The elision parameters are such that there is a limit of 5 elements displayed in any one dimension and that the elision location is the second last element of the array.
It is clear that this technique can reduce the amount of screen space needed for a visualization. This technique, as in the technique used in the Hiding Presentation, is limiting unless the user is able to interact with display in order to alter the elision parameters.

This Presentation is not well suited to the display of heterogeneous arrays where the elements may vary widely in shape and type. In such cases, dramatic structure and data may be hidden from view and provide the user with a false sense of regularity in what they can see. Consequently, the applicability of this Presentation technique would appear to be for regular homogeneous data.

3.3.6. Composite Presentations

It is clear from the previous sections that some the presentation techniques shown are well suited to particular types of array data. It is very likely that there is no single presentation technique that is useful for all types and combinations of array data.

The ability to combine Presentations to form composite Presentations is an important factor in the system design presented in this thesis (Section 6.5). Through the combinations of different presentation techniques, it possible to create a Presentation that can effectively display varied and complicated data.

Figure 3.14 shows the same parse tree as shown in figure 3.9, but now using a composite Presentation. The presentation is a combination of the Nested Lists
Presentation and the Traditional Presentation. The presentation uses the following simple rule for determining which presentation to use at each nesting level in the array data: “If the array consists of atomic (non-nested) elements, then use the Traditional Presentation, otherwise use the Nested List Presentation.

![Figure 3.14: Composite Presentation of a Nial Parse Tree](image)

The result is that the output show in figure 3.9 is now significantly more readable. The Nested List Presentation is used to display the tree structure of the parse tree, and the Traditional Presentation technique is used to display the node data.
Chapter 4. Architecture

4.1. Introduction

Chapter 1 presents the current capabilities of the Q'Nial interpreter for displaying nested array data and discusses some of the limitations of its display capability. In summary these limitations are:

1. Limited customization of the display (sketch, decor).
2. Inability to effectively display large arrays (in dimension or extents).
3. Inability to interact with the resulting display (change view properties or edit).
4. Reliance on character based stream output.

In this chapter, an architecture that supports extensible and varied views of nested array data is developed. Section 2 outlines the objectives the architecture is intended to support. Section 4 describes the division of the architecture into components or layers and describes the core responsibilities of each layer. Section 5 describes each layer in more detail.

4.2. Objectives of the Architecture

The most common process in interactive program development in the Q'Nial environment involves visually examining the results of computations to see if the resulting arrays are correct in shape, structure and content. This process can be complicated by the fact that array results may be wider than the current display screen,
or the dimensionality and nesting may be hard to interpret because of the restrictions of the current display method.

The architecture must be able to address the issues and limitations described in chapter 1, possibly by using adaptations of some of the techniques from chapter 2. The resulting architecture must allow nested array data to be presented in a manner that can preserve certain key characteristics of the resulting information. The architecture should be able to support a number of presentation methods that could be chosen by the system, or by the user. It might be that different presentation techniques are used concurrently in one display (composite presentations [Mackinlay 1986]). Each could be tailored to preserve particular characteristics of the information. Different presentation methods might have the following properties:

- Display shape in favour of content (hiding)
- Restricting the visible nesting in favour of high level shape information (hiding)
- Restrict the extents of long arrays (elision)
- Combinations of the above

The end result would be that the user would be able to evaluate the result more efficiently. Additionally, the presentation of the information would be dynamic in two senses. First, the user could interact with the presentation in order to reveal more or less detail or to change parameters that control the presentation method. This concept is called exploring. Even thought the presentation method may involve hiding some aspect of the results from the user, the user should still be able to request that more detail be displayed to the point of being able to view all information. Secondly, the user
should be able to interact with the presentation in order to edit the actual data values or the shape/structure of the data.

Content editing may be straightforward, but structure/shape editing would allow the user to change shape and dimension using standard techniques like clicking and dragging. The system would dynamically redisplay the entire presentation as necessary as edits are made.

The architecture should provide an environment where the development and design of different visualizations of array data can be made with relative ease. Someone wishing to create or extend a visualization should be able to concentrate their efforts on issues involving the layout and interactions that the visualization requires and not on complicated data structures and management code to allow the visualization to work.

The architecture should allow the visualization designer to rely on built-in support for drawing, layout, editing and interaction. The architecture should support and clarify the understanding of the flow of data and information.

The support for the visualization designer should be as unrestricted as possible. The designer should be able to develop non-traditional visualizations without having to work within the confines of a architecture that is too restrictive.

From a software engineering point of view it is desirable that the architecture provide for the ability to contain all issues of layout, graphics and interaction for a visualization
into a module or component. This component, which this thesis terms a presentation, should be able to conform to an interface specification that would allow different presentations to be used within the architecture transparently (except for the change in the appearance of the visualization).

The next section describes the division of the architecture into 3 distinct layers.

4.3. Layer Design

The Architecture divides the important functionality into three basic layers/modules. The division is based on the flow of information between the graphical user interface and the Q'Nial internal representation. The information flows in two directions (figure 3.1). In one direction the raw nested array data is transmitted to the user interface for the generation of a graphic display of the array. Information flows in the other direction when the user interacts with the user interface displaying the nested array in order to manipulate the raw data (editing).

![Figure 4.1: Basic Data Flow](image)

What is not clear from figure 4.1 is where the responsibility for deciding how to draw a graphical display of the raw data lies. The data flow is also a partial cycle where a
user interface action might modify the data and thus necessitate the regeneration of the display.

To clarify these issues, a presentation layer is introduced to manage the information flow in both directions (figure 4.2). The responsibility for deciding how to draw a graphical display for a nested array lies in the presentation layer. The presentation layer is also aware of when a user interface event requires a modification to the raw array data and also if the display requires updating as a result.

![Figure 4.2: Enhanced Data Flow](image)

The presentation layer acts as a filter for information flowing in both directions. The information flow between the presentation layer and the user interface layer consists of user interface information, and the flow between the presentation layer and the array data layer consists of array data information.

It becomes clear that the presentation layer has the responsibility to decide how to create a visualization of the array data. Is it also takes responsibility for interpreting user interface events and translating them into the appropriate modification requests for the data layer.
The next three sections will provide an overview of the responsibilities of each of the layers.

4.3.1. Array Data Layer

The primary purpose of the array data layer is to provide access to the nested array data. The presentation layer needs to be able to traverse the nested array data and extract the atomic value information in order to create a visualization.

To support editing and manipulation of a visualization, this layer must also provide access to Nial language features that can accomplish the required edits. Access to the full capability of the Nial language provides the presentation designer with unlimited flexibility in how the user can interact with the data, and thus the visualization.

4.3.2. User Interface Layer

The visualization layer provides support for creating graphic visualizations of the nested array data as prescribed by the presentation layer. This layer is responsible for an on screen window where various graphic elements are drawn by request of the presentation layer.

This layer acts much like a traditional drawing program where graphic elements can be created, and subsequently edited. Interface events such as mouse clicks and drags must be attributed to the appropriate graphics element, and subsequently passed to the presentation layer.
A selection of graphics elements is provided by this layer. These elements are the building blocks of different visualizations. In order to minimize the effort needed to implement a presentation, these should include high level graphic elements such as grids that suit the visualization, in addition to the traditional elements such as lines, text, and rectangles.

The graphic elements should be able to manage their own selection, dragging, and other aspects of visual appearance. Some of these capabilities may result in extra graphic annotation on the screen, and should not be the responsibility of the presentation layer.

4.3.3. Presentation Layer

The fundamental division in the architecture is that of the presentation layer. A presentation, from a conceptual point of view, means a method of displaying nested array data. The presentation layer should isolate all capabilities needed to achieve a particular visualization from the other layers in the architecture.

The presentation layer requires support and management for tracking which nested arrays created which lines, rectangles and other graphic elements. The array data layer needs some bookkeeping capabilities to provide this support.

The presentation layer must be able to examine the raw array data and use arbitrary algorithms to traverse the data and create graphic elements to represent the data.
As important as creating particular visualizations of the data, the presentation layer must also be able to activate editing and manipulation capabilities. The details of which user interactions have what specific effect on the data is a matter of presentation design, but this layer must be able enable various interactions for graphic elements and be able to direct them to the correct nested array to be acted on.

4.4. Summary

The next chapter outlines how the architecture presented in this chapter can be implemented using Object Oriented (OO) techniques. Through the use of classes and methods, the information flow between the layers in the architecture can be well defined.

Each layer can be composed of classes with methods for the purpose of communicating within the layer and methods for communicating with other layers. Object Oriented design can be of great benefit in the User Interface layer especially. It is in this layer where there will be many graphical elements that can be modelled very effectively using a well designed OO inheritance hierarchy.
5.1. Introduction

This chapter presents a system design that follows the architecture described in chapter 4. The next section outlines the impetus for using an Object Oriented (OO) Design to support the architecture. Section 2 details the interaction between classes in the design. Sections 3, 4 and 5 describe the Objects/Classes needed for each layer in the architecture and their important methods and interfaces. Section 6 explores some of the important design issue that span layers in the architecture.

5.1.1. Object Oriented Design

The design details in this chapter will be described using object oriented techniques. The encapsulation and reusability aspects of OO Design fit well with the architecture of the system.

Graphic elements in the user interface layer are suited to OO Design. This chapter will introduce the VisualObject class and its role in the design. This class and its sub-classes will rely heavily on OO inheritance in order to provide consistent and predictable behaviour in the user interface elements.

Virtual classes (C++) or Interfaces (Java) can be valuable paradigms for defining the capabilities and interfaces for event handling.
The sample program segments and methods descriptions are presented in a pseudo Java language style.

5.1.2. Classes in the system

This section describes the necessary classes that need to be represented in the system and how they relate and communicate with each other. The following classes will be discussed in detail in this chapter:

NialArray class       Contains the raw array data and provides methods for accessing and manipulating the data.

DisplayArray class    A sub-class of NialArray objects that extends the functionality to include management capabilities where association with a specific nested array is required.

ArrayCanvas class     A class that provides an on screen window for the display of graphics elements that comprise a visualization.

VisualObjects classes A collection of classes that provide the graphic elements used to build visualization. They include lines, text, rectangles and other elements.

Presentation class    A class that contains all of the details of a particular visualization and is responsible for how the user can interact with it.

The table in figure 5.1 shows which layer in the architecture the classes are located in.

Subsequent sections will describe the important functionality for these classes.
5.2. Relationships between Objects

Figure 5.2 shows the class inheritance hierarchy and the class associations for the classes described in this chapter. The classes are arranged horizontally according to the architecture layer in which they belong. The classes shaded in grey are sample instances that are not specifically defined by the architecture or design. The other classes are part of the system design.
The relationships between the Objects in figure 5.2 are an important part of the design. This is especially true of an OO system, where the developer must explicitly define which methods of Objects in the system are visible to other Objects.

The central object in this system is the ArrayCanvas class. While it is a relatively passive object, it is the junction for many of the interactions between other objects in the architecture. The ArrayCanvas class keeps track of a list of VisualObjects (lines, text, etc) that must be displayed on the ArrayCanvas. These VisualObjects are active. This means that once they are displayed, the ArrayCanvas object must subsequently track user input for the VisualObjects.
The ArrayCanvas object does not create or delete VisualObjects. The Presentation Class is responsible for creating the VisualObjects and passing them to the ArrayCanvas object.

5.3. Classes in the Data Layer

The next two sections discuss the NialArray class and the DisplayArray class.

5.3.1. Nial Array Class

This class stores and provides access to the raw Nial array data. This class has no knowledge of visual representations of the array nor does it provide any support for this. This class is purely for access and manipulation of the array data.

It provides methods that allow access to the raw array data for any calling application.

This is a recursive class in order to represent the recursive nature of Nial arrays.

Methods in this class can be divided into three sections.

1. Constructor methods
2. Data Access methods
3. Language Primitive methods
The presentation layer will make extensive use of the data access methods in order to retrieve the structure and atomic information needed to generate a visualization. The data access methods include the following capabilities:

1. Array type requests
2. Array shape requests
3. Nested array retrieval
4. Atomic value retrieval

Use of the language primitive methods will typically be used to support the editing capabilities of the presentation layer. The example in figure 5.4 illustrates how a potential visualization might allow the selection of a column in a simple matrix. The column is then deleted and the resulting array is displayed.

The Nial code to apply such an edit might be as follows:

```nial
deleteCol IS OP data ColIndex {
    transpose mix (((tell (1 pick shape data)) except ColIndex)
               choose (cols data))
}
```
Chapter 5

The equivalent OO language equivalent of the Nial code above might be:

```nial
NialArray
deleteCol(NialArray currentArray, NialArray colIndex)
{
    // determine the number of columns
    NialArray numColumns = currentArray.shape().pick(NialArray(1));
    // create an array of integers to select the columns we will keep
    NialArray remainingColumns = numColumns.tell().except(colIndex);
    // select only those columns and rebuild the matrix
    return(currentArray.cols().choose(remainingColumns).mix().transpose());
}
```

While it is possible for the presentation layer to accomplish these types of edits using the low level data access methods, access to the language primitive methods provides a much richer and more expressive way in which to manipulate the arrays.

### 5.3.2. Display Array Class

![DisplayArray Class](image)

This class is a sub-class of the NialArray Class. It takes advantage of the existing recursive structure of NialArray class in order to provide extra management capabilities for use by the presentation layer.

The extra facilities that this sub-class adds are those where the information being stored or the functionality being added must be associated with a specific NialArray instance. The following represents the information that this sub-class manages:
1. The list of VisualObjects that represent the NialArray
2. The locally overridden Presentation instance that this NialArray uses.

This information is stored at each level in the nested DisplayArray and is specific to the array information at the current level only.

The DisplayArray class holds references to the VisualObjects that were created by the Presentation to represent this NialArray. VisualObjects are objects that represent graphic elements such as lines, text, circles or containers. They are described in more detail later. The methods that support the list of VisualObjects are as follows:

```
addVisualObject(VisualObject vo)
clearVisualObjects()
VisualObject[ ] getVisualObjects()
```

The DisplayArray class provides the following methods for accessing the Presentation instance to use for generating the visualization for the current array and possibly its children.

```
setPresentation(Presentation p)
Presentation getPresentation()
```

These access methods enforce an inherited behaviour of Presentations. The `getPresentation()` method searches up the parent/child hierarchy of the DisplayArray instances to locate the Presentation instance to use. This is the method that the Presentation class uses when deciding which Presentation instance to use to generate the visualization for the next level of nesting of a DisplayArray.
Chapter 5

The `setPresentation()` has a symmetric effect of assigning the specified Presentation instance to the current DisplayArray, and then recursively clearing all of the local Presentations for all the child DisplayArrays.

5.4. Objects in the User Interface Layer

The following sections describe the VisualObject and VisualObjectContainer classes. These are classes that implement the capability to draw graphics elements on the screen. In addition, these classes can be sources of user interface events. The ArrayCanvas class provides the management and control of the VisualObjects and their interface events.

5.4.1. The VisualObject Class

![VisualObject Class](image)

The `VisualObject` class is a virtual or abstract class that provides a framework for implementing sub-classes that constitute the graphic elements used to build a presentation.
The class provides default functionality that all sub-classes require. The functionality includes:

- VisualObject selection grouping
- VisualObject event identification
- Offset and Origin management
- Visibility and Attribute information

Selection Grouping

Figure 5.7 shows a possible presentation of a simple array of integers. The picture is composed of comma separated strings and a enclosing rectangle. It may be the case that the presentation designer would want to select the all the elements (numbers, commas, and the bounding rectangle) if the user clicked the bounding rectangle or any of the commas with the mouse. Alternatively, if any of the numbers are selected then only the specific number is selected.

![Diagram of selection grouping]

This functionality is achieved through the concept of selection peers. Once a instance of a VisualObject has been created, it can be supplied with a list of other VisualObjects
that should be selected if the current object is selected. This functionality is supported through the following methods:

```java
addSelectPeer(VisualObject vo)
setSelectPeers(VisualObject [ ] voList)
VisualObject[ ] getSelectPeers()
```

When VisualObjects are created in the Presentation class, calls to `addSelectPeer()` or `setSelectPeers()` can be made to indicate which other VisualObjects are considered a single object for the purposes of visual depiction of selection.

**VisualObject Identification**

An important requirement of the design is that the presentation designer be able to use any combination of VisualObjects to create a visualization. This includes the possibility of using more than one instance of a specific type of VisualObject to represent a single NialArray. While the architecture supports this by design, we must provide some extra support to aid in event processing.

![Row and column selection bars](image)

*Figure 5.8: VisualObject Identification*

Figure 5.8 shows a example presentation where there are horizontal and vertical lines drawn outside a grid container that holds the integer matrix data. Suppose the designer has intended that these “extra” annotations allow the selection of columns or rows for subsequent actions (cut, copy, paste, etc). The problem is that once the
VisualObjects (the "extra" annotation lines in this example) have been created the system then needs to be able to uniquely identify which VisualObject created subsequent events.

Each of the line VisualObjects in the figure 5.8 would be identified as being a row or column annotation, and additionally, the lines belong to a specific column or row. To support the identification of VisualObjects for event handling, the following methods allow the presentation designer to assign an arbitrary id to each VisualObject that is created. The presentation designer can then check the id in the event handing code in order to take the appropriate action.

\[ \text{setID(}Object \ s) \]
\[ \text{Object getID()} \]

The \( s \) argument to \text{setID()} is of type Object. In Java all classes are a sub-class of the \text{Object} class, and thus the above methods specification allows the presentation designer to use any class to identify the VisualObject. This class could be a String, Integer, or a complex class that can hold the necessary information.
VisualObjectContainer Sub-Class

The VisualObjectContainer is provided as a template for container VisualObjects. This class is a sub-class of VisualObject and can itself act as a child object, but this class adds the following methods to support the transparent management of DynamicOffsets:

- `add(VisualObject vo)`
- `add(VisualObject vo, int index)`
- `add(VisualObject vo, Object layoutData)`
- `doLayout()`

These methods are used to add child VisualObjects to instances of VisualObjectContainer and then recalculate the layout. Non abstract sub-classes of VisualObjectContainer must implement these methods and should ensure that they call the `setDynamicOffset()` methods of the child VisualObject being added. The `doLayout()` method should recalculate the layout given the current sizes of the child objects.

For the container/child relationship to work properly the VisualObjectContainer specific methods must be implemented such that they manage the DynamicOffsets.
properly. The paint() method, which executes the actual graphics calls, must use the DynamicOffsets properly.

    paint(int xo, int yo)

The following is a sample implementation of the paint() method of a text VisualObject:

```java
class VisualObject {
    public void paint(int xo, int yo) {
        Graphics g = canvas.getGraphics();
        g.drawString(text, xo+x+getDynamicOffset().x, yo+y+getDynamicOffset().y + g.getFontMetrics(font).getAscent());
    }
}
```

Notice how the drawString() call calculates the location to draw the string. The xo and yo parameters are supplied by the ArrayCanvas in case it wants to globally adjust the location of the resulting visualization. The x and y variables are local to the text object and are supplied by the creator of the text object (the presentation designer). The x and y values from the DynamicOffset (usually a reference to the DynamicOffset instance of a parent container object) are added to the location as well.

**Offset and Origin Management**

When constructing diagrams or pictures with tree or nested array visualizations, (or many other types), traditionally the placement of child objects in such visualizations can be a complicated process. It often involves calculating positions and locations where child objects will be drawn.

VisualObjects and VisualObject containers support a nested class called a DynamicOffset. This is a class that can eliminate the need for many complicated layout calculations the presentation designer may otherwise have to code manually.
DynamicOffsets supports a concept that is now common place in user interface libraries (X Widows/Widgets, Java AWT, etc) [OSF 1992][Zukowski 1997]. This is the capability where container objects can manage child objects based on a high level layout strategy.

Use of DynamicOffsets can eliminate the need for most layout calculations only if an appropriate VisualObject is available that can support the desired layout strategy, or if the presentation designer develops a custom layout facility to use them.

The typical use of a DynamicOffset is illustrated in figure 5.10 where the child VisualObject will hold a reference to a DynamicOffset instance managed by the parent container instance. The child VisualObject will use the reference to the DynamicOffset instance in its paint() method to render the child VisualObject in the correct location.

The parent container will make the appropriate adjustments to each DynamicOffset during is layout calculation.
VisualObjects have the following methods available for access to the DynamicOffset instance when this VisualObject is being treated as a child of a container object.

\[
\text{setDynamicOffset(DynamicOffset } do) \\
\text{DynamicOffset getDynamicOffset()}
\]

**Attribute Methods**

VisualObjects also include some basic attribute management. The following methods are examples of some of the attributes that the presentation designer can apply to VisualObjects.

\[
\text{setHidden(Boolean hiddenState)} \\
\text{Boolean getHidden()} \\
\text{setColour(Colour } c) \\
\text{Colour getColour()}
\]

**Abstract or Virtual Methods**

The following methods are abstract methods in the VisualObject class (and as well in the VisualObjectContainer class) and must be implemented in any non abstract instance of a VisualObject or a VisualObjectContainer. The `paint()` method has been described earlier. The `getSize()` method is equally important. This method should be implemented such that it will always return the correct size of the object. VisualObjectContainer instances rely on calls to these methods for each child in order to properly calculate layouts.

\[
\text{paint(int } xo, \text{ int } yo) \\
\text{Size getSize()}
\]
5.4.2. ArrayCanvas Class

```
<table>
<thead>
<tr>
<th>ArrayCanvas</th>
</tr>
</thead>
<tbody>
<tr>
<td>dirty: boolean</td>
</tr>
<tr>
<td>removeVisualObject(VisualObject)</td>
</tr>
<tr>
<td>addVisualObject(VisualObject)</td>
</tr>
<tr>
<td>clearVisualObjects()</td>
</tr>
</tbody>
</table>
```

*Figure 5.11: ArrayCanvas Class*

VisualObjects are always created within two contexts. The first is the DisplayArray that requested the VisualObject be created (section 5.2.2), and second is the ArrayCanvas where the VisualObject will be displayed. The ArrayCanvas class provides the following methods, similar to those in the DisplayArray class, to manage the list of VisualObjects that are currently being displayed.

```java
removeVisualObject(VisualObject vo)
addVisualObject(VisualObject vo)
clearVisualObjects()
```

The presentation developer has little interaction with the ArrayCanvas class other than the above routines. VisualObjects themselves provide the more important interface. The ArrayCanvas class provides the system with an on-screen window area where VisualObjects are actually displayed and it provides important management of the VisualObjects that are displayed there.

The ArrayCanvas is responsible for intercepting user interface events and directing them to the appropriate VisualObject (which will likely pass the event on to the
DisplayArray that created it. It is the VisualObjects that control what events are generated.

How the user is allowed to interact with the objects is based on what interactions have been enabled for each VisualObject. The set of interactions that are available is VisualObject dependent.

The ArrayCanvas class is responsible for updating and refreshing the on screen display area when necessary. VisualObjects may be added, removed or modified and require the ArrayCanvas to update the display appropriately.

5.5. Presentation Layer/Class

```
<table>
<thead>
<tr>
<th>Presentation</th>
<th>(abstract)</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignPresentation(D)</td>
<td></td>
</tr>
<tr>
<td>generatePresentation(D)</td>
<td></td>
</tr>
<tr>
<td>generate(D)</td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 5.12: Presentation Class

The presentation layer has only the Presentation class. It is an abstract class that embodies the specific visualization the presentation designer wishes to accomplish. Different instances of the abstract class can be implemented by the presentation designer and used within the architecture to produce different visualizations for the array data. The techniques for visualization and methods of user interaction are entirely defined within this class.
The Presentation class is an abstract class that specifies three methods that must be implemented in any non-abstract instance of a Presentation class.

\[
\begin{align*}
\text{assignPresentation} & \quad (\text{DisplayArray } da, \text{int } index, \text{int } depth) \\
\text{generatePresentation} & \quad (\text{DisplayArray } da, \text{int } index, \text{int } depth) \\
\text{generate} & \quad (\text{DisplayArray } da, \text{int } index, \text{int } depth)
\end{align*}
\]

These methods are the main entrance points to instances of this class. The \text{assignPresentation}() method is called first by the system. The system expects that the implementation of this method will assign the appropriate presentations to the top level DisplayArray and possibly each nested DisplayArray as well. The method may be a simple as the following:

\[
\begin{align*}
\text{assignPresentation} & \quad (\text{DisplayArray } da, \text{int } index, \text{int } depth) \\
& \quad \{ \\
& \quad \quad \text{da.setPresentation(new TraditionalPresentation());} \\
& \quad \}
\end{align*}
\]

The above implementation will only set the presentation for the top level DisplayArray and thus the nested children will inherit the specified Presentation class. Where the implementation of this method may become more complicated is when the designer wishes to use a composite presentation (see section 6.5).

The \text{generatePresentation}() method is the top level call that the ArrayCanvas class uses to request that the Presentation generate VisualObjects that represent the visualization for the requested Presentation.
The `generatePresentation()` method should control the traversal algorithm for the presentation. Typically this will be depth first as illustrated by the sample `generatePresentation()` implementation below.

```java
generatePresentation(DisplayArray da, int index, int depth)
    if (da.kind() == Array::type)
        for (int i = 0; i < da.tally(); i++) {
            DisplayArray nestedDa = da.fetchasarray(i);
            nestedDa.getPresentation().generatePresentation(nestedDa, i, depth+1);
        }
    generate(da, index, depth);
}
```

The `generate()` method contains code that generates the VisualObjects. In the above code sample, this method is called after the traversal of the children in order to achieve the depth first rendering.

### 5.6. Important Design Issues

This section introduces some important design issues that are more general in nature and are not specific to a particular architecture layer.

#### 5.6.1. Recursive Display Generation

The system is intended to solve display problems of the current Nial diagramming method outlined in chapter 1. The nested array data space for the Nial language is very regular and well defined. While many of the common visualizations of nested data are, as the data is, recursive in nature, the system does not restrict the presentation designer to such visualizations. It does, however, provide extra support for managing recursive visualizations.
5.6.2. Support for Editing and Interaction

A key capability of the system is that should be able to support user interaction. Such interaction comes in two basic forms. First is the ability for the user to interact with parameters of presentation in order to tailor the view to his/her desire. This might involve hiding or showing portions of the display, exploring the display in more detail or eliding more or less information. Second is the ability for the user to interact with the data through the system.

Many of the issues raised in chapter 1 can be solved by using heuristic based display techniques. Parameters to the heuristics would likely be nesting depth, array extents and possibly array type. A presentation designer should endeavour to expose these parameters that control the display to the user of the system through the user interface. By doing this, users will be able to customize a presentation to suit the data they are viewing.

The ability to edit the actual array data given any particular presentation is a very powerful capability. The system architecture provides support for this capability, but providing full editing capabilities to any presentation can be a complicated task. It requires mapping interface capabilities to intuitive transformations on the data. The interactions are typically in the form of selections, drags and resizes. Deciding how to interpret such events when applied to visual components of the display (lines, rectangles, text, etc), requires good interface design skills. Implementing an editing technique within the system may not be as hard as designing it.
The architecture reduces the burden on the presentation designer when dealing with user interface issues by providing a high level view of user interaction events. Every visual object has the capability to generate high level events that the presentation designer can interpret as desired. These events might be:

- Selection
- Double Click
- Right Click
- resize
- move or motion
- drag and drop

Visual Objects are capable of presenting their own selection annotations, drag anchors, and drawing the appropriate graphics when dragged, resized or moved. The presentation designer can enable these events and interpret them according to the desired design.
Chapter 6. Presentation Development

6.1. Introduction

This chapter expands on important issues and concepts involved in the design of instances of the Presentation class. Sections 6.2 to 6.11 discuss some of the basic issues and requirements that assist when designing presentations. Subsequent sections explore the flexibility and power the architecture provides for building complex presentations.

6.2. Basic Presentation Development

Presentation development is not intended to be an end user task. As a result the presentation designer does require a good understanding of all aspects of the architecture and system design.

Foremost is the need for the designer to be able to work with the raw NialArray data. This requires a good understanding of the NialArray class and its access methods. The designer must be able to traverse the NialArray in the desired fashion and extract the necessary data at each point.

Knowledge of the use and functionality of VisualObjects is important. It is necessary to be familiar with the set of available VisualObjects in order to consider how to create a visualization. It may be the case that new VisualObjects are needed to support a particular visualization.
Event handling is critical to creating dynamic presentations that the user can interact with. Including and managing editing and dynamic presentation capabilities requires at least as much effort as creating the visualization.

The following is a list of steps that a typical presentation design might go through:

1. Decide how the visualization will appear.
2. Decide which VisualObjects can be used to represent the components of the visualization.
3. Decide on the traversal technique needs to generate the visualization and implement that in the `generatePresentation()` method of the Presentation class.
4. Implement the `generate()` method so it creates the desired VisualObjects for the supplied DisplayArray. This may require dispatching other local methods for generating the presentation for different types of NialArray data (integer, heterogeneous arrays, etc).
5. Decide how VisualObjects will be grouped for interaction purposes and call the `setSelectPeers()` or `addSelectPeer()` methods for each VisualObject as necessary.
6. Incorporate parameter control in the `generate()` method if this is a feature of the visualization technique. This might include creating different VisualObjects depending on nesting depth, shape, or other parameters of the NialArray.

The above steps will only produce a static picture of the Nial array. To include editing or interaction capabilities, the following must be done as well:

1. Decide how the available events (as determined by the VisualObjects used) are to be interpreted. Some events may adjust visualization parameters (see point 6 above), other events may modify the raw data.
2. If the visualization creates more than one instance of a class of VisualObject for any one DisplayArray, then these must be uniquely identified by calling the `setID()` method of any such VisualObject. This is only necessary if the VisualObjects will be generating events that are to be interpreted and distinguished from each other.
3. Extend the Presentation class though multiple-inheritance (C++) or through Interfaces (Java) such that the class can implement the event handling methods generated by the VisualObjects.
4. Implement the event handling methods such that they accomplish the desired behaviour.

While the description above breaks the process into two phases, it is very important to plan the editing and interaction capabilities of the presentation at the earliest possible stage.

6.3. Presentation Inheritance

The `getPresentation()` method of the DisplayArray class is a dynamic method. It is possible that any nested DisplayArray object may not have an explicit local Presentation instance associated with it. If one is not explicitly associated with the DisplayArray then the nesting hierarchy is used to search up the hierarchy until a Presentation is explicitly set. It is a requirement of the system that all top level DisplayArrays objects have an explicit Presentation set.

This inheritance means that in a typical case, only the top level DisplayArray needs to have a Presentation set. The Presentation would be inherited by all nested DisplayArrays. The flexibility comes when you explicitly set the Presentation of an intermediate DisplayArray. All children of the intermediate DisplayArray will inherit the new Presentation.
6.4. Dynamic Presentation Use

In the context of the inherited behaviour of the `getPresentation()` method, a properly written Presentation class should request the Presentation class for each nested `DisplayArray` object that it will render through the `generatePresentation()` method. While it is not critical that this technique be adhered to, it does allow the presentation to work cooperatively with the system and with other Presentations instance classes.

The following code fragment illustrates a possible technique for implementing the `generatePresentation()` method of a user developed Presentation class:

```java
public void generatePresentation(DisplayArray da, int index, int depth)
{
    if (ds.kind() == ArrayArrayType)
    {
        for (int i = 0; i < da.tally(); i++)
        {
            DisplayArray lda = da.fetchasarray(i);
            lda.getPresentation().generatePresentation(lda, i, depth+1);
        }
        generate(da, index, depth);
    }
}
```

The above routine is a typical depth first recursive rendering routine. The dynamic use of Presentations is archived through the `getPresentation()` method call of the `lda` object and not the current `da` object. This technique assures that if any nested `DisplayArray` does have a different Presentation class installed, then it will be used.

The alternate approach would be to replace the line:

```java
DisplayArray lda = da.fetchasarray(i);
lda.getPresentation().generatePresentation(lda, i, depth+1);
```

with

```java
da.generatePresentation(da.fetchasarray(i), i, depth+1);
```
This would force the system to use a single Presentation class for the whole DisplayArray.

The dynamic approach provides greater flexibility in how Presentations can be used and combined. Section 6.5 discusses in more detail the concept of composite presentations, which take advantage of the dynamic use of presentations.

6.5. Composite Presentations

Dynamic Presentations and Presentation inheritance are available to support the concept of composite presentations. A composite presentation is the use of two or more different presentations to produce a final visualization.

The use of more than one Presentation encourages simpler, more efficient Presentation programming. Different Presentations, designed for achieving different visual effects, can be combined to produce final output that suits the data being displayed.

For example, for the top levels of nesting of a DisplayArray, a Presentation that favours detail instead of shape could be used. At deeper nesting levels, a Presentation that favours shape and compact size instead of detail could be used.

6.6. Presentation Initialization

To implement the concept of composite presentations, it is important to determine how and when to assign different Presentation instances to DisplayArrays at different points in the nested DisplayArray hierarchy.
6.6. Presentation Initialization

To implement the concept of composite presentations, it is important to determine how and when to assign different Presentation instances to DisplayArrays at different points in the nested DisplayArray hierarchy.

It is possible that the Presentation class for the root node of the DisplayArray could preprocess the DisplayArray in advance and explicitly assign different Presentation classes to different DisplayArrays at different nesting levels. Such a technique would allow one Presentation to "seed" the initial Presentation for each nested DisplayArray.

This concept is referred to as Rule Based Presentation assignment.

6.7. Rule Based Presentation Assignment

A simple recursive routine as follows could be used as an initial pass to assign Presentations to a DisplayArray based on the simple conditional statement at the top of the function. This type of routine could be applied to selected sub-trees of the DisplayArray in the case of edits.

```java
int MaxExtent = 10;
int MaxDepth = 4;

public void assignPresentation(DisplayArray da, int index, int depth)
{
    da.setPresentation(new ScaledFontPresentation);
    if (shape > MaxExtent in any dimension)
        da.setPresentation(new ElidedPresentation);
    if (depth > MaxDepth)
        da.setPresentation(new ShapeOnlyPresentation);
    if (da.kind() == Array.atype)
        for (int i = 0; i < da.tally(); i++)
            assignPresentation(da.fetchasarray(i), i, depth+1);
}
The effect of the above method would be that the default Presentation would be the ScaledFontPresentation unless the array had extents (in any dimension) greater than 10 in which case a special ElidedPresentation is used. The ElidedPresentation would limit the amount of data displayed in a particular dimension. Additionally, any array nested at a depth greater than 4 deep would use the ShapeOnlyPresentation that might only display very small versions of the array containers with little or no representation of the atomic data values.

The variables MaxExtent and MaxDepth could be controlled through the user interface to allow the user to show more or less information as desired.

6.8. User Specified Presentations

The user interface for the prototype system allows the selection of arbitrary DisplayArray objects. It is possible that the UI could provide the capability for the user to assign a new Presentation to an arbitrary DisplayArray in an existing Presentation.

This capability would allow the user control of how different data being displayed is rendered. The ability to clear an explicit Presentation for a node (and possible all children) would be valuable too. These two features would allow the user complete control of how data is presented.

6.9. Editing Presentations

With the ability to make arbitrary edits to a display of an array, the system must be able to manage the case where new array data is inserted into an DisplayArray. It is
clear that the Presentation should not "re-seed" the entire DisplayArray. If it did, user
overrides and display customisation would be lost. The system would need to assure
that some expected behaviour would occur.

Atomic value edits would likely require no change in Presentation assignment, but any
change in nesting, shape or structure could possible require a partial "re-seeding" or
re-application of the rules for the affected sub-tree.

In the situation where a copy/paste style edit is being made, the Presentation could
decide that the Presentation of the "copied" data be preserved. Alternately the
appropriate Presentation could be “re-seeded” based on the new location for the data.

An edit where the user interacts with the Nial interpreter to use the Nial language
capabilities to reshape or otherwise manipulate some portion of the data may result is
an edit where new or significantly different array data is introduced. Once again, a
decision needs to be made on what the expected behaviour will be regarding the “re-
seed” of Presentations for the new or changed data.

6.10. Presentation Compatibility

To support the design feature that Presentations can be used together, the system
must define the minimum requirements that will allow Presentations to interact with
each other without conflict.
The base Presentation interface provides the template from which Presentations must be derived. The Presentation interface in the prototype system is extremely sparse. The main requirement of a Presentation class derived from the base Presentation interface is that the entry point methods assignPresentation(), generatePresentation() and generate() are implemented. These methods are expected to be able to initialize the Presentation assignment and generate the Visual Objects required to display a nested array and all of its children. Since they may be called by other Presentation classes in a composite environment, they must support these entry point methods⁴.

6.11. Presentation Specific State Information

Presentation classes will likely need to store state information in order to represent parameter overrides or other customization the user may have requested. Some information may be of a global nature and thus can be stored as member variables of the Presentation class being developed. Other information may be overrides that are specific to a particular nested array of the array currently being displayed and thus are local to a specific DisplayArray instance. The Presentation class is a non-hierarchical class (unlike the Array or DisplayArray classes) and cannot conveniently store such local information.

To support the ability to maintain local state information for each nested array being displayed, the DisplayArray class provides a place to store local state information. With this facility it is not necessary to duplicate the hierarchical structure of the

⁴ This is enforced by the Java language.
DisplayArray in order to store local information. In fact, the exact purpose of the DisplayArray sub-class of the Array class is to provide this type of extra support for the system.

Each DisplayArray contains a “repository” object. This object has two main methods:

```java
void put(Object key, Object value)
Object get(Object key)
void clear()
```

The value that is stored in the repository is of class Object (at the top of the inheritance hierarchy in Java), and thus can be any class in the Language.

Whenever a presentation class is requested to handle an edit event or to render a DisplayArray, it can simply request a value from the repository of the DisplayArray of interest. For example, if a presentation used array nesting depth to control how much detail to display at lower levels of nesting, then the following code example might represent what would happen when the user requested to “see” more depth:

```java
void HandlerFunction(DisplayArray da)
{
    Integer depthOverride = (Integer) da.store.get("DEPTH_OVERRIDE");
da.store.put("DEPTH_OVERRIDE",new Integer(depthOverride + 1));
da.getCanvas().reRender();
da.getCanvas().reDraw();
    //...
    return;
}
```

In fact the above code should recurse up the repositories of all parent nested arrays of the current object in order to accumulate the “DEPTH_OVERRIDE”s. It would be logical that an override at any nesting level would apply to all nested arrays below the
selected one. The following example would then be the function used by the presentation algorithm to calculate the additional depth value to add to the actual depth:

```java
int GetLogicalDepthOverride(DisplayArray da)
{
    int accumulatedOverride = 0;
    DisplayArray currentDA = da;
    while (currentDA != null) {
        accumulatedOverride += (Integer) da.store.get("DEPTH_OVERRIDE");
        currentDA = currentDA.getParent();
    }
    return(accumulatedOverride);
}
```

It is important to consider the issue of when the repository should be cleared of data. For example, if the presentation algorithm or the user changes the presentation being used then it is certainly likely that the data in the repository either has no relevance to the new presentation or will contain inappropriate data.

Use of the repository is unrestricted and can be used by the presentation designer as necessary. The prototype system does provide some rudimentary support to allow repositories to be cleared (recursively) or to clear selected values from the repository.
Chapter 7. Implementation of the EPS Prototype

7.1. Introduction

The prototype system is written in Java [Gosling 1996]. Java was chosen because it provides the necessary Object Oriented capabilities and it is an efficient language to develop prototypes in quickly. For a system tightly integrated with the Nial interpreter, C++ would be the implementation language of choice, but the language similarities between Java and C++ [Gosling 1996][Stroustrup 1991] and the speed at which a Java prototype can be developed made Java a reasonable choice.

The EPS prototype was developed using the Borland JBuilder 1.01 IDE. The classes developed for the prototype system follow the specifications in the design chapter very closely. A few extra classes were needed to build an “application” layer around the architecture. The EPS prototype is a standalone system.

7.2. The ArrayCanvas Class

The EPS system is a vehicle to explore the architecture and design described in this thesis. To reduce the effort involved in producing the prototype the ArrayCanvas class does not support direct user interaction with the on screen picture. This simplification eliminated the effort involved in detecting user input events such as mouse clicks and mouse drags, and attributing these events to the proper VisualObject. While this type of work is not hard it is time consuming.
The ArrayCanvas does provide an equivalent technique for producing the input events for the VisualObjects. Figure 7.1 shows a screen shot of the EPS program. The lower half of the window contains a scrolling list of rows. Each row contains buttons and other interface elements that produce the enabled events for each VisualObject. Using this simplified interface technique, the system is able to simulate the needed user input events to demonstrate the capabilities of the prototype.

![Figure 7.1: Interface Event Controls](image)

**7.3. Interaction with the Q'Nial Interpreter**

Current implementations of Q'Nial do not include an object oriented internal representation for the array data as is used by the architecture in this thesis. The prototype system implements a partial NialArray class. This class includes the core data access functionality and some limited language primitive functionality. To explore the capabilities of the prototype system it was important to have access to complex and realistic Nial array data. The functionality of the prototype NialArray class is not rich enough to easily create interesting array data for test purposes.
To provide access to real Nial array data, the NialArray prototype class implemented a file based interface to the real Q'Nial interpreter. While this file based method of data exchange is not particularly efficient, it is a quick solution for a prototype system.

The implementation involved porting from C to Java the code in the Q'Nial interpreter that is responsible for reading Nial array data from a file. This ported code was incorporated into a NialArray constructor. The constructor takes the name of the file as an argument and creates a recursive NialArray data structure.

The Q'Nial for Windows environment was used to generate all of the sample data used throughout this thesis. The EPS prototype includes a simple “File Open” dialog box that allows selection of these files for display.

7.4. Sample Presentations

This section details some of the presentations that were built with the prototype EPS program. The first two presentations simulate visual styles already available in the Q'Nial interpreter. The Traditional Presentation simulates the basic array diagrams that Q'Nial uses as the standard display technique. These are produced by the Nial operation \textit{picture}. In Q'Nial, these diagrams are created with ASCII characters. The Flat Text Presentation models the output of the Q'Nial primitive \textit{display}. The \textit{display} primitive produces an executable ASCII string representation of the array. The Flat Text Presentation method uses only text VisualObjects to generate the presentation.
Subsequent presentation techniques illustrate the capabilities of the prototype to produce a variety of presentation techniques with little effort. Segments of the Java code for each presentation are included. Appendix A includes the complete code for the Traditional Presentation (discussed next).

7.4.1. Traditional Presentation

This Presentation class implements the same display technique as the current output from Q’Nial [Schmidt 1982] (figure 7.2). The output uses graphic elements instead of ASCII based output. The intent of this sample Presentation is to demonstrate that the prototype is capable of reproducing the current output capabilities of the Q’Nial interpreter.

Due to the fact that the output is graphically generated, it is possible to change the font face, point size, the padding amount the grids uses in each cell, the line thickness of the grids and the colour of any of the graphics elements. None of these capabilities are available in the Q’Nial environment.

Figure 7.2: Emulation of Current Q’Nial Capabilities

The simplified Java implementation for this presentation is shown below. The code in the routines that create VisualObjects has been omitted for simplification. What
remains is the top level structure that illustrates the basic architecture for creating a Presentation.

```java
public class TraditionalPresentation implements Presentation {

    /**
     * This method is required by the Presentation interface.
     * This method is called to generate the VisualObjects
     * that will represent this DisplayArray and all nested
     * DisplayArrays.
     *
     * This is a simple bottom up recursive algorithm.
     */
    public void generatePresentation(DisplayArray da, int index, int depth) {
        //-- If this is a hetero array then recurse through it
        if (da.kind() == Array.ATYPE)
            for (int i = 0; i < da.tally(); i++) {
                DisplayArray da2 = da.fetchasarray(i);
                //-- This is the recursive call
                da2.getPresentation().generatePresentation(da2, i, depth + 1);
            }
        //-- Generating the visuals after the recursion (bottom up)
        da.getPresentation().generate(da, index, depth);
    }

    /**
     * This method is required by the Presentation interface.
     * This method is called first to "seed" the nested DisplayArray
     * with the desired Presentations.
     *
     * This method simply sets the Presentation for the top level
     * DisplayArray. The nested DisplayArrays will inherit this
     * Presentation.
     */
    public void assignPresentation(DisplayArray da, int index, int depth) {
        da.setLocalPresentation(this);
    }
}
```
/**
 * This method is required by the Presentation interface.
 * This method actually generates the VisualObjects.
 * This method assumes that the VisualObject for child nested arrays
 * have already been generated. The bottom up algorithm makes this
 * method very simple.
 */

public void generate(DisplayArray da, int index, int depth)
{
    //-- Check the possibilities and call the appropriate method
    if (da.IsEmpty())
        generateNull(da, index, depth);
    else if (da.IsAtype())
        generateHeteroArray(da, index, depth+1);
    else
        generateHomoArray(da, index, depth+1);
}

/*
 * These routines will generate homogeneous and heterogeneous arrays.
 */

public void generateHomoArray(DisplayArray da, int index, int depth)
{
}

public void generateHeteroArray(DisplayArray da, int index, int depth)
{
}

/*
 * Dispatch routine for each atomic type of array
 */

public VisualObject generateAtomic(DisplayArray da, int index, int depth)
{
    switch(da.kind()) {  
        case DisplayArray.inttype:
            return generateInt(da, index, depth);
        case DisplayArray.realtype:
            return generateReal(da, index, depth);
        case DisplayArray.chartype:
            return generateChar(da, index, depth);
        case DisplayArray.phrasetype:
            return generatePhrase(da, index, depth);
        case DisplayArray.booltype:
            return generateBoolean(da, index, depth);
        default:
            return (null);
    }
}

/*
 * These methods will generate pictures for atomic array data elements
 */

public VisualObject generateInt(DisplayArray da, int index, int depth)
{
}

public VisualObject generateReal(DisplayArray da, int index, int depth)
{
}

public VisualObject generateChar(DisplayArray da, int index, int depth)
{
}

public VisualObject generateNull(DisplayArray da, int index, int depth)
{
}

public VisualObject generatePhrase(DisplayArray da, int index, int depth)
{
}

public VisualObject generateFault(DisplayArray da, int index, int depth)
{
}

public VisualObject generateBoolean(DisplayArray da, int index, int depth)
{
}
The `generatePresentation()`, `assignPresentation()` and `generate()` methods are required by the Presentation interface. These methods must be implemented in order for the rest of the system to be able to interact with the Presentation\(^5\).

The remaining methods in the `TraditionalPresentation` class (`generateHomoArray()`, `generateHeteroArray()`, `generateAtomic()`, `generateInt()`, etc) are used to create the `VisualObjects` for the different types of Nial arrays. Below are the implementations of the `generateHomoArray()` and the `generateHeteroArray()`, and `generateInt()` methods. The implementation of `generateInt()` is representative of the other `generate*()` routines.

```java
public void generateHomoArray(DisplayArray da, int index, int depth) {
    //-- This is the main container (0,0 args indicate default origin)
    VOGrid gl = new VOGrid(da.getShape(), 0, 0);

    //-- Use the Grid container to layout the array, but we don't want to see it
    gl.setVisible(false);

    //-- Indicate the originator of this VO
    gl.setDisplayArray(da);

    //-- Make sure we tell the DisplayArray about the object we have created (and the ArrayCanvas too)
    da.getCanvas().addVisualObject(gl);
    da.addDesignatedVisualObject(gl);

    //-- Gather and store the child info
    for (int i = 0; i < da.tally(); i++) {
        //-- Generate the visual object for each atomic value
        VisualObject newvo = generateAtomic(da, i, depth);

        //-- Add this object to our grid container
        gl.add(newvo, i);
    }

    //-- All child objects have been added, and their sizes should be correct now, so layout the grid.
    gl.reCalc();
}
```

\(^5\) In fact Java requires these methods to be implemented.
This routine is creates a hidden VOGrid object for the container. Using a hidden VOGrid allows the use of its layout capabilities without seeing the grid. This models the Nial sketch mode of output where homogeneous arrays do not have their container displayed.

The `generateHomoArray()` method then specifies which DisplayArray this VisualObject belongs to with the `setDisplayArray()` method call, and then announces itself to the ArrayCanvas and stores a reference in the DisplayArray with the call to the `addVisualObject()` method calls. The method then iterates through each atomic element of the array and calls the `generateAtomic()` method. The `generateAtomic()` method (shown earlier) dispatches a specific `generate*()` method depending on the type of the atomic value.

Once all elements have been added to the VOGrid object, the layout is calculated with the `reCalc()` call.
Next is the implementation of the `generateHeteroArray()` method. This method generates displays for arrays that hold mixed types of nested arrays.

```java
public void generateHeteroArray(DisplayArray da, int index, int depth) {
    //-- This is the main container (0,0 args indicate default origin)
    VOGrid gl = new VOGrid(da.getShape(), 0, 0);

    //-- Indicate the originator of this VO
    gl.setDisplayArray(da);

    //-- Make sure we tell the DisplayArray about the
    //-- object we have created (and the ArrayCanvas too)
    da.getCanvas().addVisualObject(gl);
    da.addDesignatedVisualObject(gl);

    //-- Gather and store the child info
    for (int i = 0; i < da.tally(); i++)
        gl.add(da.fetchasarray(i).getDesignatedVisualObject(i));

    //-- All child object have been added, and their sizes should be
    //-- correct now, so layout the grid.
    gl.reCalc();
}
```

The `generateHeteroArray()` is very similar to the `generateHomoArray()`. This method does not hide the VOGrid container object. This method does not need to generate VisualObjects for the nested arrays that will be stored in the VOGrid because then have been created already through the bottom-up recursive `generatePresentation()` call. This method simply calls the `getDesignatedVisualObject()` to extract the VisualObject of the nested array to stored in the VOGrid object.
Below is the implementation of the `generateInt()` method.

```java
public VisualObject generateInt(DisplayArray da, int index, int depth)
{
    //-- Create a text object to represent the integer
    VOText vot = new VOText(String.valueOf(da.fetch_int(index)));

    //-- Indicate the originator of this VO
    vot.setDisplayArray(da);

    //-- Make sure we tell the DisplayArray about the
    //-- object we have created (and the ArrayCanvas too)
    da.getCanvas().addVisualObject(vot);
    da.addVisualObject(vot);

    //-- return the designated VisualObject
    return (vot);
}
```

This method does not require any layout capabilities, so it simply creates a VOText object to represent the integer and does the appropriate bookkeeping and returns. The other `generate*()` methods (real, boolean, char, etc) are almost identical.

This sample presentation provides some examples of editing capabilities. On screen arrays can be selected, and can subsequently be copied, pasted, or deleted. The following figure illustrates a simple copy and paste edit.

---

![Figure 7.3: A Copy/Paste Edit](image-url)
The Java code presented above does not include the activation of the editing capabilities demonstrated in figure 7.3. To enable the Copy/Paste edit capability shown above, two basic additions are required in the TraditionalPresentation class.

First, the VisualObjects that are created must be told that they can produce edit events (COPY and PASTE) on behalf of the DisplayArray they represent, and second, a handler routine must be added to the TraditionalPresentation class. The handler method will make the appropriate changes to the DisplayArray instance when COPY and PASTE events are generated.

To enable the events, the following code is added to the generateHomoArray() and the generateHeteroArray() methods.

```java
//-- Install some simple commands
gl.addCommandListener("CUT", da, index);
gl.addCommandListener("COPY", da, index);
gl.addCommandListener("PASTE", da, index);
gl.addCommandListener("DELETE", da, index);
```

Additionally, in the generateHomoArray() method, the atomic VisualObjects created with the generateAtomic() call have their event capabilities mapped to the parent VOGrid object through the following call.

```java
//-- Redirect events to the parent grid
newvo.setEventPeer(gl);
```

The setEventPeer() call has the effect of indicating that the newvo VisualObject (created by the generateAtomic() call) will have all of the event generating capabilities of the gl object (a VOGrid).
Once the command events have been enabled, a handler routine must be added to the Presentation class to handle the CUT, COPY, PASTE and DELETE events. For a presentation class to handle events its class definition must be changed so that it implements a Java Interface [Gosling 1996] that indicates that this class can handle edit events.

The new class definition is as follows:

```java
public class TraditionalPresentation implements Presentation, VisualObjecteditable {
    ...
}
```

The `VisualObjecteditable` interface requires that this class implement a method that has the following prototype:

```java
public void processCommand(DisplayArray da, int index, String command)
```

This is the method that will be called by the ArrayCanvas object to handle the CUT, COPY, PASTE and DELETE events we have requested.

The implementation in the `TraditionalPresentation` class is as follows:

```java
//-- Global cut buffer for supporting edit "CUT" operations
DisplayArray cutBuffer = new DisplayArray();

public void processCommand(DisplayArray da, int index, String command) {
    if (command == "CUT") {
        DisplayArray p = da.getParent();
        //-- store this DisplayArray in the CUT buffer
        cutBuffer = da;
        //-- the default constructor for DisplayArray creates a null array
        p.store(new DisplayArray(), index);
```
This handler routine simply executes the appropriate code for each type of edit command that has been enabled for this Presentation.

7.4.2. Flat Text Presentation

Q'Nial has a primitive called "display" that returns a flat string representation for any array argument. The resulting string can be re-interpreted to reproduce the original array: ie. The Nial expression "execute display A = A" is true where A can be any array.

While this type of visualization may not be well suited to viewing most array data, it does provide another reference point for comparing capabilities with the EPS prototype.

Figure 7.4 shows the output from the EPS prototype that reproduces the output of the "display" primitive.
The resulting output has the same editing capabilities as the Presentation in section 7.4.1, and can be enhanced with colour and font characteristics as well.

The implementation for Flat Text Presentation is similar to the Traditional Presentation in many ways. The difference is that the Flat Text Presentation uses a simple horizontal layout object (VOContainer) that is by default not visible.

Figure 7.4 shows the same array as shown in Figure 7.4 but with the grid lines of the VOContainer visible and extra cell padding for Flat Text Presentation.

Figure 7.5 shows how a simple text output is created so that the internal layout maintains the nested nature of the data instead of a sequential list of characters.
Some of the cells of the VOContainer are used to hold annotations such as “[“, “]” and “,”. These VisualObjects act as part of the VOContainer through the use of selection peers and thus can be used to select a complete nested array. Extra cell padding has been added to illustrate the nesting of the containers. Without the extra padding the output uses the natural kerning or spacing of the font being used.

Figure 7.6: Annotations for Higher Dimension Arrays

Figure 7.6 shows another Flat Text Presentation diagram (again with grid lines visible to illustrate the nesting) beside the Traditional Presentation output for the same array. The array data consists of a small matrix with one of the values nested more deeply. This example shows how the addition of the reshape annotation is used to represent arrays of higher dimension for the Flat Text Presentation. Reshape is a Nial primitive that can be used to define or redefine the dimension and extents of an array.

7.4.3. Infinitely Diminishing Presentation

This Presentation (figure 7.7) is derived from the Traditional Presentation in section 7.4.1, but has been enhanced to include font scaling and cell padding adjustment. As the nesting becomes deeper, a smaller font is used and the cell padding is decreased.
This is an example of a presentation that directly addresses some of the problems with the current Q'Nial output capabilities. This presentation can display much more information that the Q'Nial output for a give screen area. Structure is preserved and content is preserved as well (as long as the lower bound for font size remains readable).

The implementation for this Presentation extends the Traditional Presentation by adjusting the font size for the VOText objects and the cell padding for the VOGrid container object.

The following method is used within the Presentation to calculate the scaling factor used to adjust the padding amount based on nesting depth.

```java
private double calcPadScale(int depth) {
    return((6-(depth>5?5:depth))/(double)6); 
}
```
This method will produce a scaling factor from 1.0 to 0.2, with all nesting depths deeper than 5 using the 0.2 scaling factor. This method is then used to adjust the cell padding for each cell in each VOGrid object with the following calls in the `generateHomoArray()` and `generateHeteroArray()` methods:

```java
gl.getPad(i).setScale(calcPadScale(depth));
```

When creating the VOText objects for atomic values the `adjustFontSize()` method is used. The following code is from the `generateInt()` method. The point size for the font is reduced from the default by the nesting depth. The `adjustFontSize()` method places a lower bound of the smallest available font size the size that can be set using this method.

```java
VOText vot = new VOText(String.valueOf(da.fetch_int(index)));
vot.adjustFontSize(-depth);
```

Other that the additions of the above code, the implementation for the Diminishing Presentation is the same as the Traditional Presentation.

### 7.4.4. Nested Lists Presentation

The technique used in this Presentation (figure 7.8) is modelled after the List Control that is common to many user interfaces. These List Controls are often used to navigate file systems and are well suited to representing recursive information.
The small square indicator displayed on some nodes indicates that the node itself is another nested array. Often these indicators are used to be able to hide or expand the information at that node.

While this type of visualization does lose some structure information (multi-dimensional arrays are flattened into lists), it does provide a clear depiction of the nesting depth and it provides a convenient manner in which to explore or navigate nested arrays.

The structure information is represented by the first element in each list. This element is a list "title" and is not array data. This title is used to display a text representation.
of the type and shape of the array. The title for a 2x2 heterogeneous matrix would be "(2x2) of mixed".

The implementation for this Presentation extends the Traditional Presentation by using the VOList object instead of the VOGrid object. A textual type and shape string is created and supplied to the VOList object for the title cell. The following code illustrates the changes:

```java
String shapeStr = new String();
shapeStr = "" + shape[0];
for (int i = 1; i < valence; i++)
    shapeStr = shapeStr + "x" + shape[i];

//-- This is the main container
VOList 11 = new VOList(da.tally(),"("+shapeStr+") of "+da.KindStr(),0,0);
```

This additional code is used in both the `generateHeteroArray()` and `generateHomoArray()` methods.

Other than the above extension, the code is identical to the Traditional Presentation.

7.4.5. Information Hiding Presentation

This Presentation is an adaptation of the Traditional Presentation. Arrays at a nesting depth 2 (or more) are hidden. They are represented by a small grey rectangle. Double clicking on one of the grey rectangles adjusts the hiding depth by 1, thereby exposing a further level of array nesting. Any visible array can be selected and be requested to be hidden. Adjustments of the nesting level at which arrays are hidden are local to the selected nested array and its children.
Figure 7.9: Independent un-hiding of a Hiding Presentation

Figure 7.9 illustrates the ability to un-hide arrays independently. The three screen shots in the figure show the progressive un-hiding of a nested array until a leaf or bottom level array is reached. If a hidden array (denoted by a small grey rectangle) is un-hidden, then only that selected array is un-hidden and other arrays at the same nesting level remain hidden.

The effect of applying the “unhide” command to a specific array (or hidden marker) is to increment the depth at which arrays are hidden by 1 for all child arrays.

This technique defaults to providing a limited view of the array data, but allows easy exploring. Information at very deep level of nesting can be explored without producing an information overload because of the ability to explore down a single nesting path without affecting adjacent nested arrays (they will stay hidden).

The implementation for this presentation involves modifying the `generatePresentation()` method that recurse down the nested arrays. It is modified
to limit recursion by taking into account the default depth limit and local overrides of the default that the user applies through interaction.

```java
public void generatePresentation(DisplayArray da, int index, int depth)
{
    if (da.kind() == Array.type)
        for (int i = 0; i < da.tally(); i++) {
            DisplayArray lda = da.fetchasarray(i);
            //-- object recurse down if we are above the default hiding depth
            if (depth+1+getAdjustments(lda) < hideDepth)
                lda.getPresentation().generatePresentation(lda,i,depth+1);
            else
                generateHidden(lda,i,depth+1);
            //-- Once again limit the depth to which we recurse
            if (depth+getAdjustments(da) < hideDepth)
                generate(da,index,depth);
            else
                generateHidden(da,index,depth);
    }
}
```

The `getAdjustments()` method (called above to establish the local depth overrides) is implemented as follows.

```java
protected int getAdjustments(DisplayArray da)
{
    //-- Check to see if this array has a depth override
    Object tmp = da.store.get("HIDING_ADJUST");
    int adj = 0;

    //-- convert it to an Integer
    if (tmp != null)
        adj = ((Integer) tmp).intValue();

    //-- If we have a parent array then recurse up through the parent arrays and accumulate the HIDING_ADJUST value
    DisplayArray p = da.getParent();
    if (p != null)
        return(adj + getAdjustments(p));
    else;
        return(adj);
}
```
This method uses the DisplayArray repository to store local depth hiding overrides (section 6.11). This method will walk up the DisplayArray nesting hierarchy until the top DisplayArray is reached (has a null `getParent()` result). It will accumulate the "HIDING_ADJUST" values stored in the repository at each DisplayArray.

The following is a new method used to create the small grey rectangle used to denote an array that is hidden. This small grey rectangle has the "UNHIDE" command added.

This command allows the user to adjust the hiding depth.

```java
public VisualObject generateHidden(DisplayArray da, int index, int depth)
    throws ArrayException {
    VORectangle vor = new VORectangle(5, 5);
    vor.setColor(Color.gray);
    vor.addEventListener("UNHIDE", da, index);
    da.getCanvas().addVisualObject(vor);
    vor.setDisplayArray(da);
    da.addDesignatedVisualObject(vor);
    return (vor);
}
```

The following code is added to the `processCommand()` event handling method in order to adjust the local overrides when the "UNHIDE" command is applied to the VORectangles created in the `generateHidden()` method.

```java
if (command == "UNHIDE") {
    Integer tmp = (Integer) da.store.get("HIDING_ADJUST");
    Integer newval = new Integer(-1);
    if (tmp != null)
        newval = new Integer(tmp.intValue() - 1);
    da.store.put("HIDING_ADJUST", newval);
    da.getCanvas().reRender();
    da.getCanvas().reDraw();
} else if (command == "CUT") {

```
This code extracts any current override stored in the repository and adjusts the value such that one additional level of nesting will be displayed at or below the current DisplayArray.

7.4.6. Elided Presentation

This Presentation is modelled after the Infinitely Diminishing Presentation (section 7.4.3). In addition to the basic capabilities of the of the Infinitely Diminishing Presentation, this Presentation adds the ability to Elide information.

This elision capability is accomplished through an Elision class. This class provides the ability to specify when and how much information is to be elided. The Elision class provides methods that have the same names as methods in the DisplayArray class. The difference is that the versions of these methods in the Elision class translate shape, structure and extent information of the real DisplayArray data to the Elided form (and vise versa).

These translation methods in the Elision class are:

```java
int[] shape()
int tally()
int elidedIndexToRealIndex(int index)
boolean isElided(int index)
```

By using these methods of the Elision class as a “filter” for the real array data, an elided form of the array can be easily displayed.
The Elision class allows the specification of the maximum size to display for any extent of an array. The location where array information is to be elided can also be specified.

The following figure (7.10) shows the result of the Elided Presentation with 5 elements as the maximum extent to be displayed on any dimension. The default location to hide the information is at the second last cell in each extent.

The ".." indicates that information has been elided. While this sample does not support the capability, it would be straightforward to complete this Presentation by allowing varying elision characteristics at different depths. The ".." indicator could also act as an interface element, such that interaction with it could alter the elision characteristics (increase or decrease the maximum extent).
The following excerpts from the ElidedPresentation class illustrate the use of the Elision class.

```java
//-- Create a global Elision object for the Presentation and initialize;
//-- the maximum elision extent to 5 and the elision location to be
//-- the second last position in the extent
Elision e = new Elision(5,1);

public void generatePresentation(DisplayArray da, int index, int depth)
{
    if (da.kind() == Array.atype)
        for (int i = 0; i < e.tally(da); i++) {
            //-- Use the Elision class to get the real index
            int realIndex = e.indexToRealIndex(da,i);
            DisplayArray lda = da.fetchasarray(realIndex);
            if (!e.isElided(da,i))
                lda.getPresentation().generatePresentation(lda,realIndex,depth+1);
            else
                generateElided(lda,realIndex,depth+1);
        }
    generate(da,index,depth);
}
```

An Elision object is created for use within the Presentation. The `generatePresentation()` uses the Elision object (Elision method calls are in bold) to map the size and extent characteristics of the real array data.

The following code segment is used in the `generateHeteroArray()` and `generateHomoArray()`. It show the use of the Elision class to map the size and extent characteristics of the real array data.

```
VOGrid gl = new VOGrid(e.shape(da),0,0);

//-- Gather and store the child info
for (int i = 0; i < e.tally(da); i++) {
    int realIndex = e.indexToRealIndex(da,i);
    gl.add(da.fetchasarray(realIndex).getDesignatedVisualObject(),i);
}
```
The next method is used to create the ".." text that represents the elided array data.

```java
public VisualObject generateElided(DisplayArray da, int index, int depth)
{
    VOText vot = new VOText("..");
    da.getCanvas().addVisualObject(vot);
    vot.setDisplayArray(da);
    da.addDesignatedVisualObject(vot);
    return(vot);
}
```

7.5. Composite Presentations

Section 5 in chapter 6 discusses the design support for composite presentations. In this section two simple examples of composite presentations are discussed. The samples make use of the base Presentation classes discussed above. These composite Presentations do not provide any capability to create VisualObjects. Instead they rely entirely on other non-composite presentations presented earlier.

Both sample presentations follow the basic technique of having an initial Presentation to be used and changing to a different Presentation for arrays at a depth greater that 2. The two sample presentations differ only in which two presentations they use.

Figure 7.11 shows a presentation where the initial Presentation is the Flat Text Presentation and at nesting depths greater than 2 the Infinitely Diminishing Presentation is used.
Chapter 7

Figure 7.11: Composite Presentation (sample 1)

Figure 7.12 shows a presentation where the initial Presentation is the List Presentation and the at nesting depths greater that 2 the Infinitely Diminishing Presentation is used.

Figure 7.12: Composite Presentation (sample 2)
The following code is for the Presentation class that generates the output in figure 7.11. This Presentation relies on the visual capabilities of the Flat Text Presentation (figure 7.4) and the Infinitely Diminishing Presentation (figure 7.7) and has no capability for generating VisualObjects. In fact the generate() method throws an exception if it is called.

```java
public class CompositePresentation implements Presentation {

    FlatTextPresentation ftp = new FlatTextPresentation();
    DiminishingPresentation dp = new DiminishingPresentation();

    public void generatePresentation(DisplayArray da, int index, int depth) {
        if (da.kind() == Array.atype)
            for (int i = 0; i < da.tally(); i++) {
                DisplayArray lda = da.fetchasarray(i);
                lda.getPresentation().generatePresentation(lda, i, depth+1);
            }
    }

    public void assignPresentation(DisplayArray da, int index, int depth) {
        if (da.kind() == Array.atype)
            for (int i = 0; i < da.tally(); i++) {
                DisplayArray lda = da.fetchasarray(i);
                assignPresentation(lda, i, depth+1);
            }
    }

    private void setRuleBasedPresentation(DisplayArray da, int depth) {
        if (depth <= 1)
            da.setLocalPresentation(ftp);
        else
            da.setLocalPresentation(dp);
    }
}
```

The setRuleBasedPresentation() method is used by the assignPresentation() method to initialize each nested DisplayArray to use one of the two base Presentations.
(Flat Text or Infinitely Diminishing) depending of the nesting depth. The `setRuleBasedPresentation()` method is extremely simple. There is no restriction on what criterion could be used in determining which Presentation to assign to a DisplayArray.

After the `assignPresentation()` method is called to initialize the Presentation the `generatePresentation()` method is then called by the EPS prototype. This method, in fact, is barely used. The following line from that method:

```java
lda.getPresentation().generatePresentation(lda,i,depth+1);
```

will immediately pass the recursive control to one of the core Presentation classes because of the `getPresentation()` call. Because all of the Presentations developed in the EPS prototype following the same `generatePresentation()` method algorithm, they will always pass control to the appropriate Presentation class.

The next code segment is a simplified implementation for the Composite Presentation shown in figure 7.12. Code has been omitted where it is identical to the previous Presentation. This Presentation uses the List Presentation instead of the Flat Text Presentation.
public class CompositePresentation implements Presentation {

    ListPresentation lp = new ListPresentation();
    DiminishingPresentation dp = new DiminishingPresentation();

    public void generatePresentation(DisplayArray da, int index, int depth) {
        // code omitted (same as previous Presentation)
    }

    public void assignPresentation(DisplayArray da, int index, int depth) {
        // code omitted (same as previous Presentation)
    }

    public void generate(DisplayArray da, int index, int depth) {
        // code omitted (same as previous Presentation)
    }

    private void setRuleBasedPresentation(DisplayArray da, int depth) {
        if (depth <= 1) {
            da.setLocalPresentation(lp);
        } else {
            da.setLocalPresentation(dp);
        }
    }

    These composite Presentations show that the system design can easily support the use
    of different presentation styles.
Chapter 8. Analysis and Conclusions

8.1. Introduction

This chapter provides a summary of the architecture and system design presented in this thesis and how it relates to the thesis objectives. The prototype system is evaluated. Open questions and areas of future work are then discussed.

8.2. Analysis

This thesis has attempted to address limitations in the array display capabilities of the current Q'Nial implementation. These limitations, discussed in chapter 1, include the non-interactive nature of array diagrams and the inability to effectively display arrays with long extents, deep nesting, or high dimensionality.

The architecture implemented in the prototype EPS system demonstrated the following basic concepts:

1. An intuitive framework for designing and testing presentations.
2. Simplified technique for the creation of graphical visualizations.
3. An Architecture that provides a clear and understandable division of responsibilities.
4. The use of a Component / Container model of visual elements to eliminate layout calculations.
5. A simple and powerful event model for implementing user interactions.
8.3. The Prototype System

This section presents some of the issues resulting from the EPS prototype implementation.

8.3.1. Application of the Architecture and Design

The object oriented design [Booch1994][Fowler 1998] specified in chapter 5 provided a fairly specific definition of the Java classes and their interfaces. The resulting prototype follows the design very closely and provides all of the functionality specified.

The VisualObject and VisualObjectContainer classes and their instances provide the building blocks for the Presentation designer. The interaction between VisualObjectContainers and other VisualObjects is critical to usability of the system. VisualObjects must cooperate with each other and provide consistent and predictable behaviour.

The EPS prototype implemented event handling capabilities specified in the design, but was limited to simple interactions with the visual displays. Great benefit might be gained through more complex interactions. The basic interaction design was validated, but more complex interaction would likely introduce complications. A specification for generalized interaction capabilities for VisualObjects would be a benefit in developing enhanced user interaction capabilities.
8.3.2. Extending to 3-D

The EPS prototype system is not able to display 3-D visualizations of arrays. This is not a limitation of the architecture or the design presented in this thesis, but rather of the capabilities of the VisualObjects that were created for the prototype and the ArrayCanvas.

To extend the prototype to be able to create 3-D visualizations, a set of 3-D VisualObjectContainers and VisualObjects are required. Presentation development would be virtually identical to the current 2-D prototype.

Enabling user interface interaction with 3-D visual objects requires more complex issues. One would expect to be able to rotate a 3-D visual object in order to view the information from different orientations. This requires additional user interface capability from the ArrayCanvas class. The ArrayCanvas would require some additional capabilities in order to allow the selection and interaction with 3-D visual objects and containers.

All of these issues should be isolated from the presentation designer. 3-D issues would be almost entirely confined to the user interface layer of the architecture.

8.3.3. Composite Presentations

The EPS prototype experiments with a few composite presentations. It appears that this is a very powerful capability of the architecture and design. In fact the composite
presentations presented in chapter 7 involve no rendering code at all. These composite presentations simply make use of existing non-composite presentations.

The rules the sample composite presentations use to assign or seed the initial presentations are very simple. If there were a set of non-composite base presentations, very complex composite presentations could be created by creating appropriate seeding rules.

The sample composite presentations use a simple depth rule to trigger a change in presentation style. A more elaborate composite presentation could example the type, shape, extents and nesting depth to determine the appropriate base presentation to use. Elision characteristics could be applied dynamically to suite the particular array being seeded.

To enable the use of base presentations to build composite presentations it becomes necessary that base presentations are "compatible". Their component/container behaviour must be compatible in order for arbitrary base presentations to "contain" other base presentations.

They may also need to maintain consistent editing behaviour. It would be confusing to the end user to have dramatically different methods of interacting with different parts of a composite presentations.
8.3.4. Elision

Section 2.4.1 in chapter 2 [Cordy 1990] discusses the use of Elision for viewing software source code. The Elision Presentation in chapter 7 demonstrates how effective an Elision strategy is in presenting array information that has large extents.

The sample presentation implements elision through an array data transformation. An Elision class was created that maps the real shape, extent and size information of the array to the elided shape extent and size that is to be displayed. This class simply provides replacement operations for those already existing in the Array Class. A presentation simply uses the replacement access routines and the resulting mapped array will be elided according to the parameterized elision characteristics.

A presentation can use a global Elision class such that all arrays being displayed by the presentation will have the same elision characteristics (as the sample presentation does).

For greater flexibility, a presentation could create an Elision instance for each nested array and store it in the DisplayArray repository for each nested array. The characteristics of each Elision instance can be controlled by initialization, rule based characteristic assignment, or through user interaction. Such an implementation would provide great flexibility and control over how the arrays are displayed.

The Elision technique used in chapter 7 could be applied to each of the other sample presentations in chapter 7. The basic layout and appearance of each of the sample
presentations would not change, but each would gain the additional benefits that data elision provides.

Elision was kept entirely within the presentation class in the prototype system. The Elision class provides no graphical capabilities and is purely a data transformation/mapping capability. Elision was not incorporated into the data layer of the architecture even though that may seem to be the appropriate location for such a transformation. Placing the elision capability in the data layer would immediately restrict the presentation designer to the supplied elision model and limit the ultimate flexibility of the design.

8.3.5. On Screen User Interactions

The prototype system does not allow direct interaction with the on-screen graphic elements. Interaction is performed through traditional user interface elements (button, etc) that provide the equivalent events to what a completed system would allow through direct interaction.

To extend the prototype system to support the desired direct interaction, the relationship between the ArrayCanvas and VisualObjects would need to be carefully defined and designed.

The ArrayCanvas would receive low level mouse events such as clicks, double-clicks and drags. The ArrayCanvas would require a standard method of determining which VisualObject is to receive these events. Events such as mouse drags are complicated
by the fact that a drag may begin in or over one VisualObject, but may end anywhere else, possibly outside the application window.

The architecture relies on having high-level VisualObjects in order to simplify the creation of presentations. Indirectly this requires that the presentation designer be able to create new VisualObjects to suite the desired visualization.

A well defined and designed interface between VisualObject and the ArrayCanvas objects should allow for all of the features mentioned without complication.

8.4. Future Work

The ultimate goal of the architecture proposed in this thesis would be to include it in a production Q’Nial interpreter environment. Q’Nial for Window [Jenkins 1993] provides an implementation environment where a graphical array view could be implemented. The current Q’Nial for Windows implementation still relies on a character based interface, but the Q’Nial application has full access to the graphical capabilities of the Win32 API [Microsoft 1992] to implement the architecture.

Current versions of the Q’Nial interpreter do not yet have an object oriented internal data structure. To overcome this, the design would have to implement an array class much as the EPS prototype has. The complicating issues in the EPS prototype of exchanging of array data with the Q’Nial interpreter through files would not be needed in an integrated version. The integrated array class would have constructors that
would be able to create nested instances from the current internal array data representation.

The remaining classes discussed in chapter 5 (System Design) could be ported from the prototype and enhanced and completed to provide the full functionality as specified in the architecture and system design.

Once implemented, the architecture could be used in two manners. The basic display capabilities that the architecture provides could be used to entirely replace the ASCII character based array output currently used in the interactive environment. Secondly, an explicit edit mode could be added where the user requests to edit an array variable. Once modified, the user could commit the modified value back to the workspace.
References

[Beard 1990]
David V. Beard and John Q. Walker, "Navigational techniques to improve the display of large two-dimensional spaces", Behaviour and Information Technology 9,6 (November-December 1990).

[Booch 1994]

[Brown 1985a]

[Brown 1985b]

[Card 1991]

[Chang 1997]

[Cordy 1990]

[Ding 1990]

[Fowler 1998]

[Graham 1988]
References

[Gosling 1996]

[Jenkins 1986]

[Jenkins 1993]

[Johnson 1991]

[Mackinlay 1986]

[Mackinlay 1991]

[Microsoft 1992]
"Microsoft Win 32 Application Programming Interface : The Programmer's Reference" (Programmer's Reference Library), Microsoft Corporation, 1992

[More 1973]

[More 1976]

[More 1979]

[Myers 1983]
[OSF 1992]

[Robertson 1991]

[Schaffer 1993]

[Schmidt 1982]

[Seminar 1990]

[Stroustrup 1991]

[Turo 1992]

[Xiao 1992]

[Zukowski 1997]
Appendix A Example Presentation

The following is a complete implementation of the TraditionalPresentation Class discussed in chapter 7, section 7.4.1.

```
//Title: Presentation Class for TraditionalPresentation
//Copyright: Copyright (c) 1998
//Author: Chris Walmsley
//Description: This non-abstract implementation of the Presentation Class
//
//*-- All EPS files are part of this package
package thesis;

public class TraditionalPresentation implements Presentation, VisualObjecteditable {

    //-- Global cut buffer for supporting edit "CUT" operations. The default
DisplayArray
    //-- constructor creates a null Nial array.
    DisplayArray cutBuffer = new DisplayArray();

    /**
     * This method is required by the Presentation interface.
     * This method is called to generate the VisualObjects
     * that will represent this DisplayArray and all nested DisplayArrays.
     *
     * This is a simple bottom up recursive algorithm.
     */
    public void generatePresentation(DisplayArray da, int index, int depth) throws ArrayException {
        //-- If this is a hetero array then recurse through it
        if (da.kind() == Array.atype)
            for (int i = 0; i < da.tally(); i++) {
                DisplayArray lda = da.fetchasarray(i);
                //-- This is the recursive call
                lda.getPresentation().generatePresentation(lda, i, depth + 1);
            }
        //-- Generating the VisualObjects after the recursion (bottom up)
        da.getPresentation().generate(da, index, depth);
    }

    /**
     * This method is required by the Presentation interface.
     * This method is called first to "seed" the nested DisplayArray
     * with the desired Presentations. This is where composite
     * presentations can be generated
     *
     * This method simply sets the Presentation for the top level
     * DisplayArray. The nested DisplayArrays will inherit this
     * Presentation.
     */
    public void assignPresentation(DisplayArray da, int index, int depth) {
        da.setLocalPresentation(this);
    }

```
/**
 * This method is required by the Presentation interface.
 * This method actually generates the VisualObjects.
 * This method assumes that the VisualObject for child nested arrays
 * have already been generated. The bottom up algorithm make this
 * method very simple.
 * This method dispatches more specific routines to do the real work.
 */
public void generate(DisplayArray da, int index, int depth)
  throws ArrayException{
    //-- Check the possibilities and call the appropriate method
    if (!da.isEmpty())
      generateNull(da, index, depth);
    else if (da.isAtype())
      generateHeteroArray(da, index, depth+1);
    else
      generateHomoArray(da, index, depth+1);
  }

/**
 * This routine will generate homogeneous arrays.
 */
public void generateHomoArray(DisplayArray da, int index, int depth)
  throws ArrayException{
    //-- Track the list of objects that should behave as
    //-- a single object with this variable.
    Vector localSelectPeers = new Vector(20);
    //-- This is the main VisualObjectContainer container
    VOGrid gl = new VOGrid(da.shpptr, 0, 0);
    //-- Add our self to the select peer list and then
    //-- pass the list reference to the grid object
    localSelectPeers.addElement(gl);
    gl.setSelectPeers(localSelectPeers);
    //-- Give the grid a name (to assist identification for editing)
    gl.setID("CONTAINER");
    //-- Use the Grid container to layout the
    //-- array, but we don't want to see it
    gl.setVisible(false);
    //-- Indicate the originator of this VO
    gl.setDisplayArray(da);
    //-- Store depth information in the repository
    da.store.put("DEPTH", new Integer(depth));
    //-- Install some simple edit commands
    gl.addCommandListener("CUT", da, index);
    gl.addCommandListener("COPY", da, index);
    gl.addCommandListener("PASTE", da, index);
    gl.addCommandListener("DELETE", da, index);
Make sure we tell the DisplayArray about the object we have created (and the ArrayCanvas too)
da.getCanvas().addVisualObject(gl);
da.addDesignatedVisualObject(gl);

Gather and store the child info
for (int i = 0; i < da.tally(); i++) {
    //-- Generate the visual object for each atomic value
    VisualObject newvo = generateAtomic(da,i,depth);

    //-- Because the grid object is not visible, we can reduce some of the padding around the cells.
    if (leftEdge(da,i))
        gl.setPads(i,0,2,0,0);
    else if (rightEdge(da,i))
        gl.setPads(i,2,0,0,0);
    else
        gl.setPads(i,0,2,0,0);

    //-- Add this object to our grid container
    gl.add(newvo,i);

    //-- It shares the same list of select peers as the grid
    localSelectPeers.addElement(newvo);
    newvo.setSelectPeers(localSelectPeers);

    //-- Redirect events to the parent grid
    newvo.setEventPeer(gl);
}
//-- All child objects have been added, and their sizes should be correct now, so layout the grid.
//gl.reCalc();

/**
 * This method is used to generate the VisualObjects for Atomic values
 * It simply dispatches the appropriate method for the specific atom type.
 */
public VisualObject generateAtomic(DisplayArray da, int index, int depth) throws ArrayException {
    switch(da.kind()) {
    case DisplayArray.inttype:
        return generateInt(da,index,depth);
    case DisplayArray.realtype:
        return generateReal(da,index,depth);
    case DisplayArray.chartype:
        return generateChar(da,index,depth);
    case DisplayArray.phrasetype:
        return generateInt(da,index,depth);
    case DisplayArray.faulttype:
        return generateInt(da,index,depth);
    case DisplayArray.booltype:
        return generateBoolean(da,index,depth);
    }
    return(null);
}
/**
 * This method returns true of the specified index is at the left edge of this grid.
 */
private boolean leftEdge(DisplayArray da, int index) {
    int[] address = e.indexToAddress(e.shape(da), index);
    if (address.length == 0)
        return (true);
    if (address[0] == 0)
        return (true);
    else
        return (false);
}

/**
 * This method returns true of the specified index is at the right edge of this grid.
 */
private boolean rightEdge(DisplayArray da, int index) {
    int[] shape = e.shape(da);
    int[] address = e.indexToAddress(shape, index);
    if (address.length == 0)
        return (true);
    if (address[0] == (shape[0]-1))
        return (true);
    else
        return (false);
}

/**
 * This method generates heterogeneous arrays. It assumes that child/nested DisplayArrays have already had their VisualObject generated.
 */
public void generateHeteroArray(DisplayArray da, int index, int depth) throws ArrayException {
    //-- This is the main container
    VOGrid gl = new VOGrid(da.shapeptr(), 0, 0);

    //-- Indicate the originator of this VO
    gl.setDisplayArray(da);

    //-- Give the grid a name (to assist identification for editing)
    gl.setID("CONTAINER");

    //-- Store depth information in the repository
    da.store.put("DEPTH", new Integer(depth));

    //-- Install some simple commands
    gl.addCommandListener("CUT", da, index);
    gl.addCommandListener("CCOPY", da, index);
    gl.addCommandListener("PASTE", da, index);
    gl.addCommandListener("DELETE", da, index);

    //-- Make sure we tell the DisplayArray about the object we have created (and the ArrayCanvas too)
    da.getCanvas().addVisualObject(gl);
    da.addDesignatedVisualObject(gl);
Gather and store the child info
for (int i = 0; i < da.tally(); i++)
    gl.add(da.fetchasarray(i).getDesignatedVisualObject(), i);

All child objects have been added, and their sizes should be
correct now, so layout the grid.
gl.reCalc();
}

/**
 * This method generates an atomic integer picture using a simple
 * Text VisualObject.
 */
public VisualObject generateInt(DisplayArray da, int index, int depth)
    throws ArrayException {
    VOText vot = new VOText(String.valueOf(da.fetch_int(index)));
    vot.setID("DATA_TEXT");
    da.getCanvas().addVisualObject(vot);
    vot.setDisplayArray(da);
    da.store.put("DEPTH", new Integer(depth));
    da.addVisualObject(vot);
    return (vot);
}

/**
 * This method generates an atomic real picture using a simple
 * Text VisualObject.
 */
public VisualObject generateReal(DisplayArray da, int index, int depth)
    throws ArrayException {
    VOText vot = new VOText(String.valueOf(da.fetch_real(index)));
    da.getCanvas().addVisualObject(vot);
    vot.setDisplayArray(da);
    vot.setID("DATA_TEXT");
    da.store.put("DEPTH", new Integer(depth));
    da.addVisualObject(vot);
    return (vot);
}

/**
 * This method generates an atomic character picture using a simple
 * Text VisualObject.
 */
public VisualObject generateChar(DisplayArray da, int index, int depth)
    throws ArrayException {
    VOText vot = new VOText(String.valueOf(da.fetch_char(index)));
    da.getCanvas().addVisualObject(vot);
    vot.setDisplayArray(da);
    vot.setID("DATA_TEXT");
    da.store.put("DEPTH", new Integer(depth));
    da.addVisualObject(vot);
    return (vot);
/**
 * This method generates an atomic picture for a null array using a simple
 * Text VisualObject.
 */

public VisualObject generateNull(DisplayArray da, int index, int depth)
    throws ArrayException {
    VOText vot = new VOText("null");
    vot.setDisplayArray(da);
    vot.setID("DATA_TEXT");
    da.store.put("DEPTH", new Integer(depth));
    //-- Install some simple commands
    vot.addCommandListener("CUT", da, index);
    vot.addCommandListener("COPY", da, index);
    vot.addCommandListener("PASTE", da, index);
    vot.addCommandListener("DELETE", da, index);

    da.getCanvas().addVisualObject(vot);
    da.addVisualObject(vot);
    return (vot);
}

/**
 * This is the edit handler that is required by the fact that this class
 * implements the VisualObjectEditable interface. This method simply
 * examines the edit command supplied as an argument and performs the
 * appropriate action.
 */
public void processCommand(DisplayArray da, int index, String command) {
    try {
        if (command == "CUT") {
            System.out.println("cut..." + index);
            DisplayArray p = da.getParent();
            da.clearVisualObjects();

            //-- store this DisplayArray in the buffer
            cutBuffer = da;
            p.store(new DisplayArray(), index);
            da.getCanvas().reRender();
            da.getCanvas().updateLabels();
            da.getCanvas().reDraw();
        } else if (command == "PASTE") {
            System.out.println("paste...");
            DisplayArray p = da.getParent();
            da.clearVisualObjects();
            p.store(DisplayArray.myClone(cutBuffer), index);
            //p.store(cutBuffer, index);
            da.getCanvas().reRender();
            da.getCanvas().updateLabels();
            da.getCanvas().reDraw();
        } else if (command == "COPY") {
            //-- here we need to clone a DisplayArray
            cutBuffer = DisplayArray.myClone(da);
            System.out.println("copy...");
        } else if (command == "DELETE") {
            System.out.println("delete..."+index);
            DisplayArray p = da.getParent();
            da.clearVisualObjects();
            p.store(new DisplayArray(), index);
            da.getCanvas().reRender();
            da.getCanvas().updateLabels();
            da.getCanvas().reDraw();
        }
    }
}
Appendix A

} catch (Exception e) {
    e.printStackTrace();
}