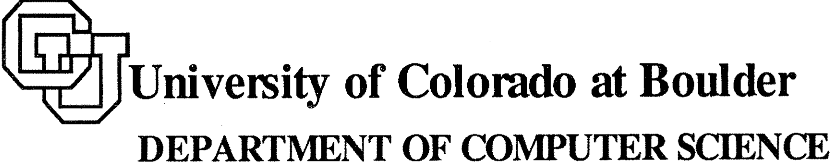
DATASHEETS: DESIGNING AN END-USER PROGRAMMING ENVIRONMENT TO SUPPORT A SPECIFIC DOMAIN

Nicholas P. Wilde

CU-CS-733-94



DATASHEETS: DESIGNING AN END-USER PROGRAMMING ENVIRONMENT TO SUPPORT A SPECIFIC DOMAIN

Nicholas P. Wilde

CU-CS-733-94 1994

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430 USA

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

DATASHEETS:
DESIGNING AN END-USER PROGRAMMING ENVIRONMENT TO SUPPORT A SPECIFIC DOMAIN

by

Nicholas P. Wilde

B.S., Cornell University, 1981
M.S., University of Wisconsin, 1984
B.G.S., University of Nebraska, 1987
M.S., University of Colorado, 1990

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

1994

Wilde, Nicholas P. (Ph.D., Computer Science)

DataSheets: Designing An End-User Programming Environment to Support a Specific Domain

Thesis directed by Professor Clayton Lewis

Abstract

Technologies such as direct-manipulation and graphical user interfaces have allowed many people to do productive work on computers with little or no formal training. Such end-users typically stay within the bounds of functionality provided by one or more applications such as word processors, electronic mail, and the like. Such an application-centered view of computing is naturally limited in what can be accomplished to the functionality provided by the designers of the applications used. There are some domains for which this is clearly quite limiting - the nature of the task(s) involved is such that the functionality required may change from day to day or task to task.

For end-users in such domains, it seems more likely that some combination of application functionality and end-user programming language, designed to support the particular task(s) they may need to accomplish, is the most promising direction to go. In such cases, it is important that the application and language functionality, the overall programming language paradigm, and the interface they are embedded in, are all designed to support the particular domain of interest.

One such domain is exploratory data analysis. I report on using a task-analysis based method to design a environment in which the computational paradigm, the interface, and the language functionality are all designed to support tasks in the given domain. DataSheets is an environment intended to allow such users the power to craft interactive data displays of their own choosing, without learning a conventional programming language. It does this by combining an alternative paradigm for computation, the spreadsheet, with a set of interactive graphical primitives for the display of data sets. The programming aspects of the environment and the graphical aspects are linked in a way designed to allow the user to build interactive data displays, and work with them quickly and easily.

Acknowledgments

To Clayton Lewis – my advisor. To Wayne Citrin, Mike Eisenberg, Mark Gross, and Gerhard Fischer – members of my committee:

To NASA's Center for Space Construction and to US West Corporation, for partial support during the completion of this thesis:

To friends and family – Mom, Dad, my sisters Annelie and Jacqueline:

And most especially to the one person who has made this possible – my wife, Dawn:

A sincere, heartfelt, and enthusiastic THANKS!

Contents

1. Introduction..1

1.1. A scenario..1
1.2. Implications of the scenario for end-user computing........................8
1.3. Limitations of the application-centered view of end-user computing 10
1.4. The need for end-user computing languages 12
1.5. The notion of task-oriented end-user environments........................ 13
1.6. Making the paradigm fit the task(s) 15
1.7. Thesis 16
1.8. Contributions 16

2. Background and related work.. 19

2.1. Other attempts at combining application functionality with programming language concepts. 19
2.2. Psychology of programming and problem solving 19
2.3. Cognitive and Programming Walkthroughs – theory based design of applications and languages. 22
2.4. Visual programming and program visualization 25
2.4.1. The spreadsheet as an end-user programming environment.. 26
2.4.2. Other Spreadsheet-based visual programming environments.. 29
2.4.3. Other visual paradigms.. 30
2.4.3.1 Constraint-based 30
2.4.3.2 Demonstrational.. 32
2.4.3.3 Rule Based.. 33
2.5. Graphical display of information.. 34
2.6. Statistical packages and EDA.. 39

3. Design Methodology 42

3.1. The Forward Walkthrough – Pushing the programming walkthrough.. 42
3.1.1. Starting with a blank slate doesn't work 44
3.1.2. If not a blank slate, then what?........................ 46
3.1.3. Functionality, level, packaging 51

4. Original Design 56

4.1. The Computational View 57
4.1.1. The problem with spreadsheets 57
4.1.2. The cell 60
4.1.3. The cell range 61
4.1.4. Mathematical Operations 63
4.1.5. Addressing operations 65
4.1.6. Input Ranges 66
4.1.7. Abstraction Mechanism 67
4.2. The Graphics View 68
4.2.1. Coordinate System 70
4.2.2. Attributes 71

5. Prototype implementation and initial testing 73

5.1. Implementation details 73
5.1.1. Propagation algorithm 73
5.1.2. Handling equality constraints 75
5.2. Prototype testing 78

6. Redesign 80

6.1. Proposed modifications 80
6.2. Adopted modifications 83

7. Further Testing 86

7.1. Further testing – procedures 86
7.2. Further testing – results 88

8. Discussion 96

9. Conclusions 108

References 111

Appendix A: Testing materials 115

List of Tables

Table 1. Actual vs. predicted data set for a new method of predicting Boulder's daily maximum temperature 1
Table 2. Task suite for the initial design of DataSheets. 48
Table 3. Binary operators available in DataSheets. 64
Table 4. Unary operators available in DataSheets. 65
Table 5. Available addressing operations in the initial design of DataSheets. 66
Table 6. Available cell attributes and the significance of numerical values in the controlling cell. 72
Table 7. Subject completion of tasks during initial prototype testing. 79
Table 8. Modifications proposed as a result of prototype testing. 80
Table 9. Modifications adopted as part of the redesign. 83
Table 10. Tasks used in second round of testing. 86
Table 11. Time to completion for all subjects and all tasks during second round testing (min:secs). 87
Table 12. Interface components that were classified as indicative of computation, drawing, or linking behavior. 91
Table 13. Number of each class of episode per subject and task. 92

List of Figures

Figure 1. A simple scatterplot of actual vs. predicted. 2

Figure 2. Determining the cutoff value for outliers. 4

Figure 3. Testing a hypothesis that outliers are linked to relative humidity (RH) > 30 %. 5

Figure 4. Removing outliers from consideration to concentrate on the rest of the data set. 6

Figure 5. Measuring the distance between two curves. 7

Figure 6. A classical case of graphical misperception. (reproduced from Cleveland [15]). 37

Figure 7. The Box-and-Whiskers task. 49

Figure 8. Proto-solution for subtask of drawing Box-and-Whiskers median line in a spreadsheet-based language. 49

Figure 9. Proto-solution for subtask of drawing Box-and-Whiskers median line in a constraint-based language. 50

Figure 10. Proto-solution for subtask of drawing Box-and-Whiskers median line in a demonstration-based language. 50

Figure 11. Original and revised filter operations. 55

Figure 12. A DataSheets worksheet consists of a computational part (the top pane) and a graphics view (the bottom pane). 56

Figure 13. The basic parts of a cell. 60

Figure 14. Examples of simple vertical and horizontal cell ranges 61

Figure 15. Examples of nested cell ranges. 63

Figure 16. Worksheet "Calling" Calls Worksheet "Ave" three times. 69

Figure 17 Propagation algorithm used in DataSheets. 74

Figure 18. Skeleton code for updating equality constraints. 77

Figure 19. The naming/cell formula convention adopted in redesign. 82

Figure 20. The modal dialog that is part of the redesigned equality constraint definition procedure. 84

Figure 21. The redesigned linkage between computational and graphics view. 85

Figure 22. An example fragment of a journal file of interface actions taken as part of the second round testing procedures. 88

Figure 23. An interactive scatterplot built in the DataSheets environment. 105

Figure 24. An interactive line chart built in the DataSheets environment. 107

1. Introduction

1.1. A scenario

Imagine you are a research meteorologist working on a new method of predicting Boulder, Colorado's maximum daily temperature. You've been working on this project for almost a year now, and you think you've finally got it to a point where you're ready to test your method against some actual observations. Accordingly, for the past 20 days you have been collecting statistics of Boulder's actual daily maximum temperature, and what your method predicted the temperature should have been (Table 1). You want to explore this data set, with an eye towards understanding what connection (if any) the actual temperature has with what your method predicted.

Table 1. Actual vs. predicted data set for a new method of predicting Boulder's daily maximum temperature

Actual	Predicted
10	20
21	13
30	35
41	54
52	57
61	66
70	75
80	89
90	95
100	105
9	14
18	23
32	37
38	49
52	57
63	68
72	63
81	86
91	96
99	104

This is really an exercise in *exploratory data analysis* (EDA), so you turn to the EDA software package on your personal computer. Knowing that an often fruitful place to start when you have a data set of actual vs. predicted is with a simple scatterplot, you create such a display by reading the data in to an area on the screen reserved for calculations, and assigning each pair of numbers (actual and predicted) to the X and Y coordinates, respectively, of a set of marks created in the data display area of the EDA package. The resulting display looks promising – there appears to be a strong correlation between the actual and predicted values, as evidenced by the general 45 degree slope of the cloud of points plotted on the display.

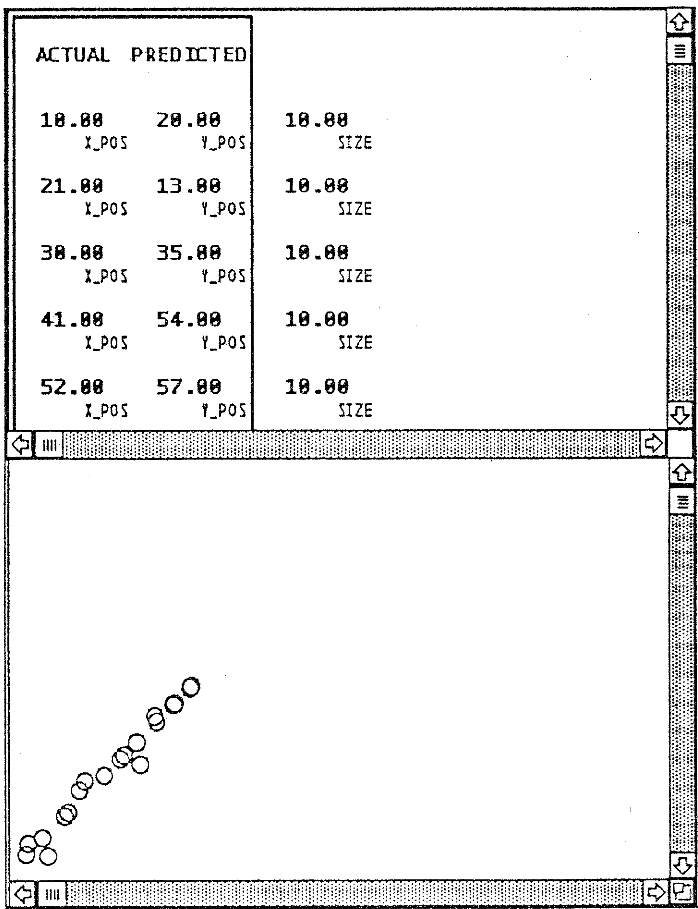


Figure 1. A simple scatterplot of actual vs. predicted.

Not every point appears to be so well behaved, though, as there appears to be some noise, or outliers, in the data set. You wish to investigate the cause of this noise further. The first thing you need to do is isolate those outlying values from the rest of the data set. To do so, you design a virtual device which allows you to interact with the data set and its display in a way that lets you determine an appropriate cutoff value (Figure 2). This virtual device is represented as a vertical bar on the data display, and works as follows: the height of the vertical bar is tied back into the calculation area of the EDA package and represents the cutoff value that determines what is and is not treated as an outlier in the display of the data set. The display is set up so those points that have a difference between actual and predicted greater than the cutoff value are represented as solid circles, while those points for which the difference is less than or equal to that value are represented as hollow circles. The display is updated continuously as you manipulate the vertical bar representing the cutoff value, so the dynamic visual feedback allows you to try various values and determine what an appropriate cutoff value appears to be – about 5 degrees.

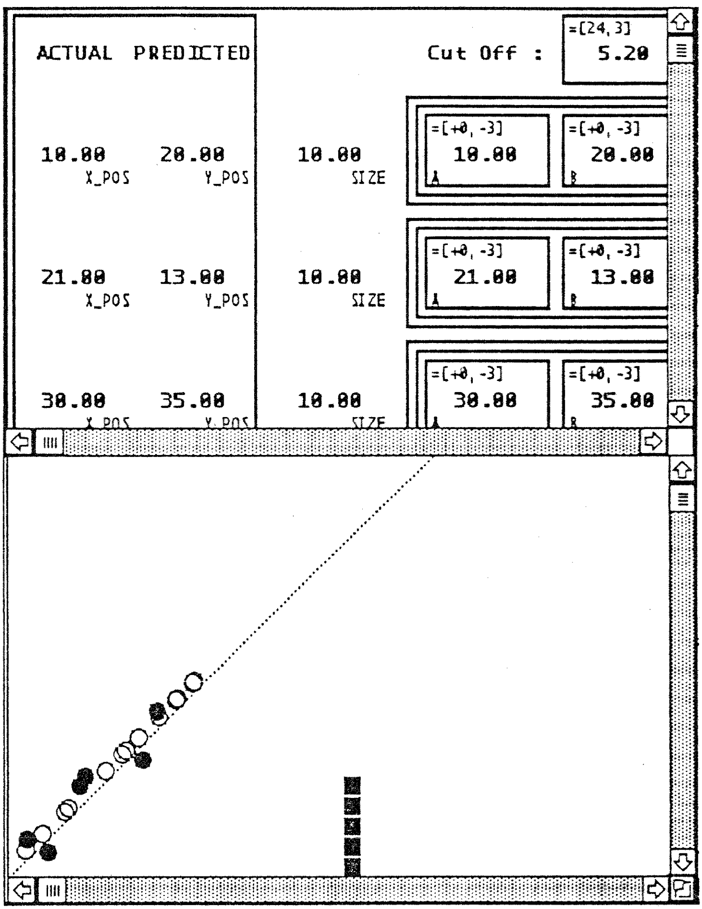


Figure 2. Determining the cutoff value for outliers.

The number 5 strikes a familiar chord with you. You remember doing something differently in your prediction method on certain days, when the relative humidity (RH) was greater than 30%. That would have made a difference of about 5 degrees in the predicted value . Since you have the relative humidity data for the days in your data set available to you, you read those values into the calculation area of your EDA package and choose to display those points representing days with RH greater than 30% differently – in this case by choosing a square as a plotting symbol rather than a circle (Figure 3). When you do so, something immediately jumps out at you – those values you had designated as

outliers are precisely those days for which the relative humidity was greater than 30%.

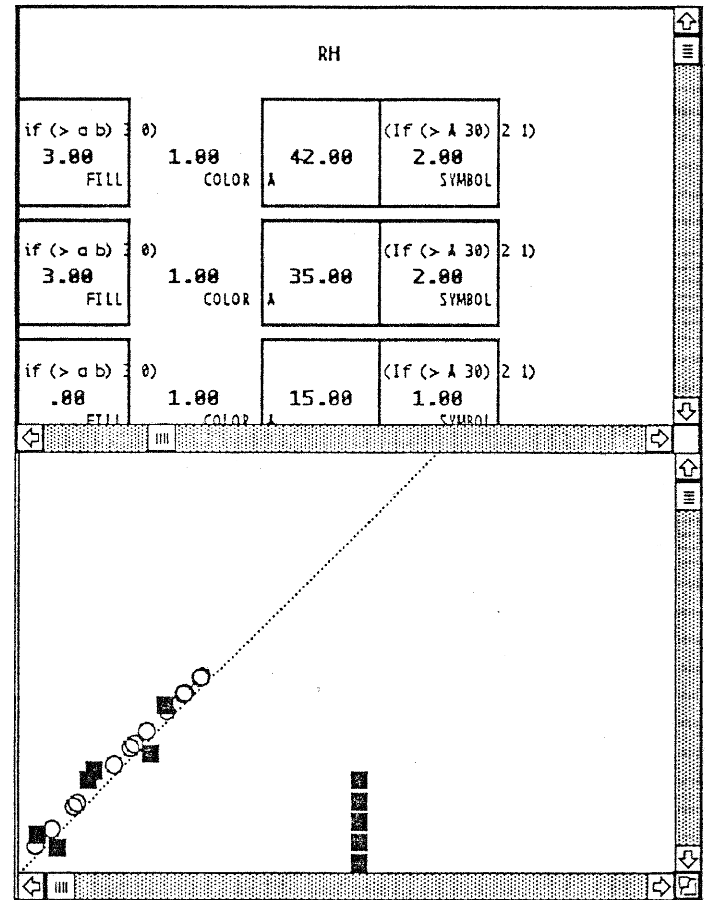


Figure 3. Testing a hypothesis that outliers are linked to relative humidity (RH) > 30 %.

Confident that you have a handle on the cause of this unusual behavior on certain days, you decide to examine the rest of the data set to determine how well your method is working in the more normal case. To do so, you want to get rid of the outlying points and concentrate on the others (Figure 4). The resulting plot looks very promising - there is a strong linear correlation between actual and predicted, although it appears that a constant offset has crept

into your predicted values somehow, as they all appear to lie to one side of the line that represents (actual = predicted). To further investigate this, you use your EDA package to plot both the actual and predicted values as two separate time series on the same plot (Figure 5). When you do so, you note the two time series curves appear to be parallel and there appears to be a constant difference between them.

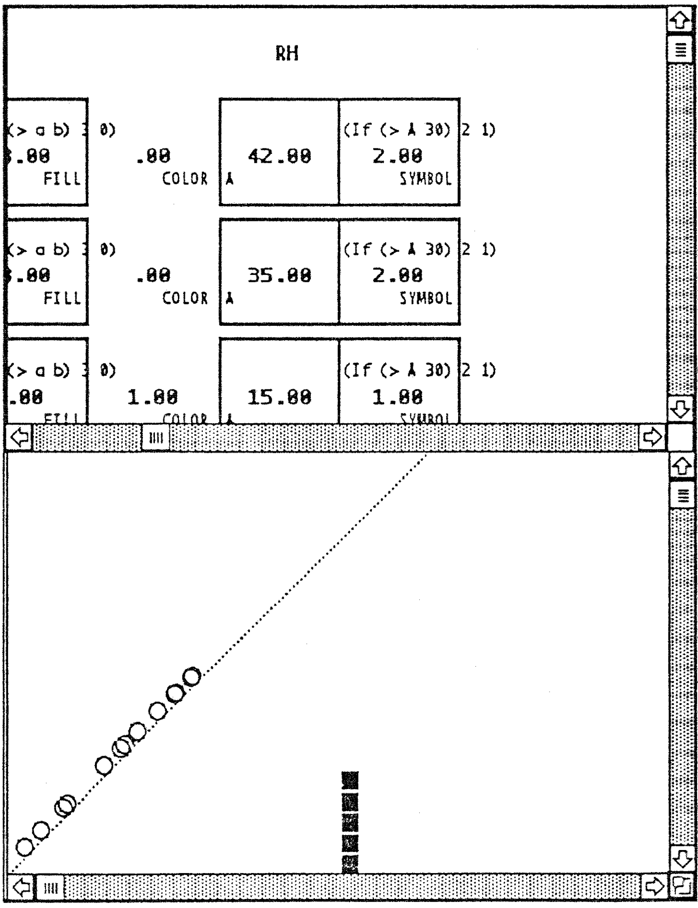


Figure 4. Removing outliers from consideration to concentrate on the rest of the data set.

Knowing that the eye can sometimes be fooled into thinking that curves appear parallel even when they are not, you decide to investigate just a little further. You create another virtual device on your data display which

allows you to interactively measure the vertical distance between the two curves at various points along them and get an instantaneous reading of the difference. It works in the following way: Y coordinates of a vertical line are tied to the Y coordinates of the two curves at the horizontal distance along those curves at which the vertical line is positioned. The line is constrained to be vertical by having the X coordinates of each end of the line constrained to be equal to each other. The Y coordinates of each end of the line are fed back into the calculation area of the EDA package, where the difference between them is computed and the result displayed.

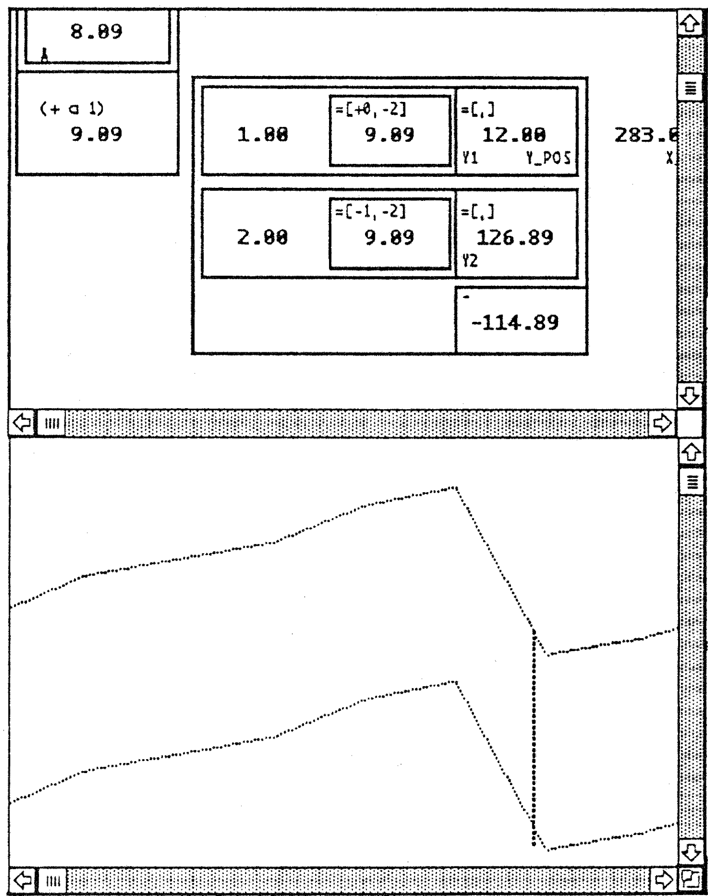


Figure 5. Measuring the distance between two curves.
The numbers are different from the first display to emphasize the distance between the curves.

By doing this, you create a device which can be moved back and forth along the two curves and that gives an instantaneous reading of the vertical difference between the two at any point. When you manipulate this device, you are quickly able to determine that these two curves are indeed parallel and that there is a consistent bias in your method of predicting Boulder's daily maximum temperature. Confident that you understand everything your data set is trying to tell you, you turn your attention away from your data set and back to fixing the discovered flaws in your prediction method.

1.2. Implications of the scenario for end-user computing

The above scenario illustrates the type of use I had in mind when building DataSheets, the system discussed in this dissertation, which is expressly designed to support the domain of exploratory data analysis. The displays shown above were built within DataSheets. While perhaps a bit fanciful in its representation of an actual data analysis episode, it represents many of the facets of the domain that were of interest to me in building this system:

The displays and techniques involved included many that are common across episodes and individual users. Such displays as scatterplots and time series charts are classic tools that are part of the data analysts stock in trade. It would seem natural to hope that any package or application built to support data analysis would make such displays easy to build and use.

Details of the displays and techniques involved included some that were highly specific to the task at hand. While scatterplots and time series are part of a

data analysts stock in trade, plotting relative humidity on top of actual vs. predicted temperature is not. Neither is building a dynamic device to determine what values in the data set are outliers.

Dynamic displays and techniques were used effectively. Often, dynamic displays can tell us things static displays can not. Determining the proper cutoff value for outliers, for instance, and determining that two time series really are parallel when they appear to be, are good examples of the types of tasks dynamic displays support especially well.

The process was an iterative, incremental one, in which results from earlier parts of the data analysis session suggested questions and explorations to be performed in later parts of the episode. Data analysis is an experimental, exploratory process, in which the tasks involved might differ from episode to episode, and might even change within a single episode. In the scenario above, the specific task at hand goes from plotting actual vs. predicted, to understanding what an outlier is in the data set, then plotting additional information about the data set, and finally plotting a subset of the data set in a completely different form.

What these factors mean, in terms of building a system to support the domain of EDA, is that no single application with fixed functionality will be able to support the entire process of EDA, because we as designers don't know what functionality will be required each time. While a system might provide tailored support for the common tasks within the domain of EDA, it is too much to ask that the system somehow provide tailored support for the particular displays and techniques a user might want to bring into play during a specific session. For

such highly individual displays, a programmable environment in which the user can develop or adapt their own functionality seems more appropriate.

In the following pages, I will argue against the application-centered view of computing, in which fixed functionality is provided by applications, in favor of task-oriented programming environments, in which tailored support for a particular domain is combined with programming language functionality and extensibility. The main goal and contribution of this thesis will be a process for designing such environments. I will also present the design of prototypical task-oriented programming environment for the domain of EDA – DataSheets, as well as presenting in more detail the two dynamic displays used in the scenario above.

1.3. Limitations of the application-centered view of end-user computing

In the 1950's and 60's computers were seen as specialized machines to be used by those who had formal training. Users either received that training, or found skilled intermediaries (programmers) to deal with the machine in solving problems of their choosing. "Canned" software as we know it today didn't exist. Twenty years later, with the arrival of personal computers, computing power was placed directly in the hands of the user who had a problem to solve – the scientist who needed to analyze data from his or her last experiment, or the business person who needed to understand the financial ramifications of a particular business move. Users were still required to learn much about the computer's ways, however, and to use command languages not far removed from programming language syntax to get any useful work done.

With the introduction of GUIs (graphical user interfaces) and direct manipulation interfaces, the use of computing technology was finally brought to a level accessible to the untrained user. The direct manipulation interface allowed many without specialized training to use computers as productivity tools through the use of applications such as word processors, painting programs, and drawing programs. At the same time, the rapidly decreasing cost of computer hardware and increased power of the available hardware made it possible to put computers on just about every professional's desk. As a result, a new class of user was born – the end-user. Such users typically have little or no training in the use of computers. They often do not program, but rely on the power of commercial applications or applications built by others to provide the functionality they need.

In fact, such users often look upon the computer merely as a device for running canned software. It is not uncommon to walk into a store selling computer equipment and overhear someone asking if such and such a machine can run their favorite word processor. The phenomena of individuals buying a computer to run one or more specific applications is now commonplace. That the particular hardware can later be expanded in its functionality by buying new software packages is in many ways incidental, as long as the machine runs the applications the user is currently interested in. This is the *application-centered view of computing*.

Unfortunately, this application-centered view has some serious limitations. The user is constrained to precisely those kinds of applications that

others provide – usually those applications for which it's commercially viable for someone to do so, or for which someone else has had the need, and the skills, to write a program. This is a problem if the domain is one that interests a small number of people, as it may not be viable for a commercial product to be built to support that domain. If the user wishes to use a computer to solve some problem that lies outside the domains covered by commercial software, they have only a few choices: pay someone to write the program for them, learn a programming language and write it themselves, or hope someone else has had the same problem and has spent the time to implement a solution (and then wishes to share that solution with others).

Even if the domain is supported by commercial software, or by software written by some other user, the functionality then at the user's disposal is precisely that functionality that those other individuals (typically, the application designers and programmers) have decided the user should have, and no more. If the task is a highly individual one there may still be a need to create functionality to support the new user's particular task. The existence of contract programmers, who provide customized software support for an individual customer's needs, is proof that this occurs.

1.4. The need for end-user computing languages

In some domains it is very hard, or even impossible, for programmers to clairvoyantly provide all the functionality a user may ever need, because requirements vary widely with the individual user and the problems he or she is trying to solve. In data analysis, for example, a user may have a highly specialized data set that requires processing in ways unique to that particular

data set, or they may require a display tailored to the particular question they are trying to answer, that is unique to the data set and problem at hand. In decision support, a user may need to simulate a particular process that is unique to his or her particular situation, and that requires functionality no programmer has thought to provide in a ready made simulation package.

Because of the limitations inherent in the application-centered view, end users who go beyond the bounds of standard application domains such as word processing or drawing diagrams eventually find themselves needing to program, or they must remain satisfied with the functionality provided by others for them to use.

1.5. The notion of task-oriented end-user environments

Of course, the idea of non-specialist programming and end-user programming languages is nothing new. The original goal of such languages as FORTRAN, LISP, and BASIC was to provide languages that were somehow easier to write programs in than their predecessors. Yet end-user programming remains an elusive goal. What might be different today, then, that allows us to create languages better suited to learning and use by non-specialists? One possibility is that we can now bring to bear some of the ideas from the *psychology of problem solving* and the *psychology of programming* when designing languages.

Over the last several decades, a principal result from the psychology of problem solving and the psychology of programming literature has been the notion of *plans* – generic "chunks" of code available to seasoned programmers that they can apply in many situations and that produces a given result [45]. A

large part of learning to program is building a repertoire of such plans, and learning when and how to apply each plan to reach the desired goal. Such plan application may involve modifying the plan to fit the specific needs of the situation. Beginners typically have many fewer plans to choose from, and are thus often forced to use less satisfactory plans to produce a given result. Such plans are often couched in the generic concepts of the programming language – arrays and integers in Pascal for instance, or lists and symbols in LISP. A central difficulty in learning to program, then, becomes learning to translate from concepts in terms of the plan (programming concepts) into concepts in terms of the problem (problem domain concepts).

It is thus worthwhile to ask if we can produce an environment/language in which programming plans are closer to concepts found in the problem domain. One way we as designers might provide such a language is to provide functionality tailored to the specific domain the user is working in. Such *task-oriented programming languages* would be targeted to a specific domain of interest, but must allow users to create functionality of their own devising, to fit the situation and problem at hand.

Nardi [40], in her book on end-user programming A Small Matter of Programming, discusses the notion of *task specific* end-user programming languages – i.e. languages that are designed and developed for the purposes of supporting a specific task (or tasks) a user community may have a need for. Although the notion of task specific languages seems overly constrained, the idea of task-oriented languages – that provide functionality designed around a suite of tasks a user community has a need to support, but that are still general enough

to enable the user community itself to generate additional functionality when it is required, seems worthwhile.

1.6. Making the paradigm fit the task(s)

When a professional programmer chooses a language (or languages) to develop an application in, he or she commits to the use of a specific *paradigm* of computing for implementing that particular application. Intuitively, most experienced programmers know this: we rarely choose to implement a rule-based system in FORTRAN, or a weather model in LISP. This is not to say that it absolutely can't be done – after all, at a basic level all languages are equivalent if they contain a few standard structures. It is simply that one language provides functionality in a way more attuned to how we would naturally think about the task at hand – the language paradigm fits the problem domain.

There are many competing languages (and thus basic paradigms) to choose from: procedural, object-oriented, functional, logic-based, etc. Committing to a particular language colors the way in which a programming solution gets expressed – but does it influence the ease with which specific programming problems may be tackled? For non-professional programmers, who may not have as broad a repertoire of programming solutions or methods (plans) available to them this may be critical. They may be unable to surmount the difficulties arising from a bad choice of language/paradigm for their particular problem.

1.7. Thesis

The central proposition put forth by this dissertation is that a viable process for building a task-oriented, end-user programming environment, in which the basic language paradigm, the functionality the language provides, and the environment in which that language is embedded are all designed to support a particular domain, involves task and problem-based design in the form of the paradigm-based, forward walkthrough method.

1.8. Contributions

In this dissertation, I present a case study of designing and implementing an end-user programming environment designed to support a particular domain, that of *exploratory data analysis* (EDA) – the process of exploring and finding relationships and regularities in data sets (as compared to *confirmatory data analysis* – the process of deciding those relationships really do exist). Why pick EDA as a domain to support? The first reason is that it provides a unique meta-domain – a domain of use to scientists and others across many different disciplines. Many different types of users have the need to display, and understand, complex data sets. Yet the requirements for the computational manipulation and the subsequent display of each data set may be unique depending upon the particular user's requirements – what questions they are trying to ask of the data and what they hope to understand by examining it. Exploratory data analysis also provides an ideal domain for an interactive programming environment, as often the solutions to one problem, or the answer to a particular question asked of a data set, bring to mind other problems, other calculations and displays that can be done on the same data set.

Throughout this dissertation, there are three themes that I hope to communicate to the reader:

A vision for end-user computing in the future – This involves going beyond the application-centered view of computing and inventing task-oriented programming environments. Such environments would combine some of the flexibility of more general purpose programming languages with domain-oriented functionality of applications in a meaningful manner.

A process for designing such task-oriented environments – To tailor the paradigm the language presents, and the language functionality itself, I have used a variant of the Programming Walkthrough method of interface/language design [7], that attempts to provide feedback from a walkthrough-like analysis as early in the design process as possible, even before the language framework has been designed. I do this by comparing what solutions to particular tasks *would* look like, if they were programmed in a language of a particular basic type. By comparing such "proto-solutions" in different basic language paradigms, we can decide on a particular language type to best support the given tasks.

The design of the DataSheets system itself – In this instance, a multi-paradigm approach was used, in which a spreadsheet based, computational engine is combined with a constraint based, drawing oriented, data display environment. The resulting environment, called DataSheets, was implemented in Macintosh Common Lisp on a Macintosh Quadra computer.

In the pages that follow, I'll first present background and work related to this dissertation. I'll then discuss in more detail the process used in the design of DataSheets in the chapter titled "Design Methodology", followed by an introduction of to the basic design of the application itself.

Because an integral part of the process of designing usable programming environments involves user testing and iterative design, I'll then go on to discuss testing the prototype implementation and what those tests revealed. User tests of the prototype revealed a number of difficulties in using the environment as originally designed. In the latter chapters I will report on the results of those tests, the subsequent redesign, and further testing of the environment. In the discussion and conclusions sections I reflect back on the process of analysis by which I arrived at the particular design implemented, and how that analysis was and was not supported by the results of the user tests. Finally, I will conclude by presenting two "mini-applications," built in the DataSheets environment, that are representative of the types of things I hoped to make possible by building this system. Both represent dynamic displays for EDA – in the first, a user can manipulate a vertical bar to visually determine the appropriate cut-off value for deciding what is and is not an outlier in a specific data set. In the second, a "virtual ruler" is built that allows a user to measure the distance between two curves (representing plots of data as might be found, for instance, in plotting time series) to determine whether or not the two curves are indeed parallel.

2. Background and related work

2.1. Other attempts at combining application functionality with programming language concepts.

I am not the only one to attempt to build hybrid application and end-user programming systems. In particular, Eisenberg [18; 19] has had considerable success creating programmable applications, in which a programming language interpreter is combined with direct manipulation functionality to provide more than the sum of the parts. Eisenberg's approach differs from the approach used here, though, in that he starts with an existing language (Scheme) and adds functionality on top of that language. While he provides specific "sub languages" consisting of functionality tailored to the domain in question, the user must still contend with the complexities of learning a full blown programming language in addition to the specific sub-language for the domain in question.

2.2. Psychology of programming and problem solving

To design a computer-based environment to support a particular process, we must first understand something about that process. In this case, the overall process I am interested in is that of problem solving or programming in support of a particular domain, EDA. However, the underlying process is still *programming* as a means of specifying a solution or information display. To this end, we must understand a little of the basics of a cognitive model of programming.

Programming is an opportunistic process, which consists of sequence

of steps the programmer must take to arrive at a solution to some problem. Individual steps may be motivated by certain goals the programmer has, or may involve breaking a particular goal into subgoals [3]. Often steps involve selecting and instantiating the proper *plan*, or sequence of actions, to achieve a particular goal. Plans are specific sequences of actions that programmers may have available to them for solving particular problems [45]. They may involve specific actions or programming constructs within the language being used, or may involve breaking a particular goal down into subgoals that are more easily accomplished. Novice programmers may have few plans available to themselves, and thus expend a great deal of time and effort in trying to adapt ill-suited plans to specific goals. Expert programmers have many more plans available to themselves (acquired through a wealth of programming experiences) and are thus more easily able to select a plan that is well suited to a particular goal. In many cases, present day languages make it necessary to *merge* elements of two or more plans together to achieve a coherent program. Expert and intermediate level programmers may adopt a more plan-based or opportunistic strategy in developing code – the section of code that is most promising at the moment (i.e. the most achievable subgoal) becomes the center of attention and what is worked on next [17; 42]. Because experienced programmers jump around in the code, working on the current target of opportunity, programmers must not only be able to represent their ideas in code, but must often reinterpret their own code in light of their ideas [24]. Although the notational aspects of programming languages do not significantly affect the strategies expert programmers use (they still employ a more opportunistic, plan based model than novices), the notation may determine how quickly a novice may be able to take a more opportunistic, plan based approach [17].

In [54] we distinguish several different types of steps, including goal-control steps, coding steps, comprehension steps – both for the notion of comprehending *process* (what happens when this code is executed?) and *purpose*, (what does this code accomplish?), and testing of both process and purpose. We further note that many of these same steps and step-types show up in the various *activities* of programming: creating programs, testing programs, and debugging them.

Novices and experts use the same basic problem solving processes and steps – they just may apply them differently, and in different orders. Learning to program, like learning any complex task, involves mastering the many different types of steps as well as the management of the process as a whole. Not only must the programmer learn the individual techniques available to them to solve problems, they must learn when to apply those techniques, and to what subparts of the problem, in order to be successful.

Programming languages designed to support the entire process of programming should be significantly easier to learn, and easier for non-specialists to learn, than languages that don't, or that only support specific parts of the process. Such languages should not only provide support for the individual steps involved in crafting a problem solution, perhaps by providing the appropriate functionality and control structures to solve specific subparts of given problems, but should provide support for the entire process of programming by making it easier to know when to apply the language functionality and control structures to the appropriate parts of the problem.

These are precisely the questions the paradigm-based and programming walkthroughs, discussed later in this dissertation, are designed to address.

2.3. Cognitive and Programming Walkthroughs – theory based design of applications and languages.

For the user/programmer to take each step in such a process, then, requires *motivation* from some source: whether that motivation comes from the environment, the language, or the task itself. This is precisely what the programming walkthrough is intended to examine: the programming walkthrough is a method for trying to design languages that support at least the higher level process of programming better, by examining the kinds and quantity of knowledge (called *guiding knowledge*) a user would need to apply to solve a particular problem in a given language design [6; 7]. The programming walkthrough grew out of earlier work on the cognitive walkthrough – a method for improving the design of walk-up and use interfaces for such tasks as Automatic Teller Machine (ATM) design, phone-mail system design, etc. [41]. While no one expects programming languages, even those designed for end-users, to be truly walk-up-and-use, we can still use many of the same ideas to design languages to be easier to learn and to express solutions in.

The programming walkthrough is a *task-based* method – it requires a suite of problems for which the language is supposed to be capable of expressing solutions. It is also an *inspection* method – it requires a prototype language to express problem solutions in prior to, or at the same time as, performing the walkthrough. There is no way to derive design guidance for a yet to be designed language using the programming walkthrough. This can be a problem, as it

requires the designer to "pick a language design out of the air" so to speak, prior to applying the walkthrough. This may not seem to be a particularly grave problem for researchers, who often have experiences with many different types of languages and are willing to weigh the various merits and disadvantages of certain language types before designing an initial version. Research is by its very nature, an experimental process – if a researcher has to throw away his or her initial designs, they hope they have learned something in the process, but are not overly concerned by the trial and error nature of the process. This may, however, be more of a problem if we want to incorporate language design at the level of software engineering, as may be necessary if the notion of task-oriented languages were to be included in applications for a specific user group. Software engineering as a profession is supposed to be an engineering process, as opposed to an experimental process – practitioners at least want to maintain the illusion of forward progress through an orderly process, as opposed to relying on trial and error.

Even if one can come up with a prototype language design without too much difficulty, there is still the "your baby is ugly" syndrome – nobody wants to find flaws in their initial efforts and be told they have to redesign large portions of their newly designed language. The result is a perception of wasted effort (which may or may not be true, but is still perceived) on the designer's part which might make them reluctant to change an existing design even if methods such as the programming walkthrough clearly point out problems with the design as it stands. For these reasons, I wanted to bring the programming walkthrough further forward in the design process, and see if one could use the walkthrough, or a variant, to actively derive a language from a set of problems,

rather than just derive criticisms of an existing language.

Nardi [40] distinguishes between 3 levels of end-users – *application end-users,* who only use the products of others with little or no programming/customization; *local developers,* who have an intrinsic interest in computing and bring their expertise in a particular domain to bear in creating customized applications and environments for a particular domain; and *professional support programmers,* who create the higher level widgets and functionality local developers can use in creating domain-specific applications and environments.

As mentioned previously, the view of the application end-user seems too restricted for many domains – it limits the end-user to the scope of the designer's vision and understanding of what the end-user might want to do. Many domains are not amenable to providing all the functionality that may ever be required in advance – as new functionality may be required for the particular task requirements facing the user at some point.

At the same time, not everyone will be lucky enough to have a cadre of professional support programmers able to provide new widgets whenever necessary. Even if a group of users is lucky enough to have professional programmer support, communicating the user's desires to the programmers takes time and effort. And such professional programmers still need to understand how best to provide the functionality required by the users of computing technology. By developing processes and methods by which professional programmers could better design and provide functionality in the

form of end-user computing systems, or task-oriented languages, for others to take advantage of, we hope to make it possible for end-users to employ such task-oriented languages in crafting highly individual solutions to particular problems. This dissertation aims at providing a process by which professional programmers could design such task-oriented languages.

2.4. Visual programming and program visualization

Since the implementation of PICT [23], and even before that with such seminal programs as Sutherland's SketchPad [48] and Borning's ThingLab [10], the field of visual languages, and in particular visual programming, has attracted a great deal of interest.

The hope of such systems has often been that by replacing text with graphics, somehow the end result will be more accessible to non-specialists. Take for instance, Myer's remarks in [39]:

> "Another motivation for using graphics is that it tends to be a higher-level description of the desired actions (often de-emphasizing issues of syntax and providing a higher level of abstraction) and may therefore make the programming task easier..."

This is often done without regard to *why* graphics should be better, and in what specific contexts. Indeed, it is not even clear that graphical or visual programming should be any better from a psychological perspective: Green, et al. [25], note that in some cases textual representations of programs may actually

be easier to understand than graphical representations. Indeed, our own analysis [54] has pointed out that purely graphical programming systems may be harder to use than textual languages, because a large part of programming involves incorporating semantics of the real world problem into the programming solution, via the modification of the program presentation in some way. While this could conceivably be done by incorporating custom icons into the graphical program that represent concepts in the problem domain, or by including digital pictures of actual objects in the problem domain into the program, it seems clear that the primary method for such customization will remain the same as that found in textual languages – the use of well chosen textual names for such things as program procedures or variables.

Mixed graphics-textual languages, where each medium contributes the strengths of its particular presentation to the support of the process of programming as a whole, thus seem to be a promising direction for future programming environments. One such hybrid system already enjoying considerable success is the spreadsheet.

2.4.1. The spreadsheet as an end-user programming environment

The spreadsheet application is often hailed as one of the success stories of end-user computing, as indeed it should be (I use *Spreadsheet Application* here to refer to the overall package a user builds individual *spreadsheets* in – in other words, a household budget might be one spreadsheet developed within a spreadsheet application for a specific computer). One only has to take a look at the great commercial success of spreadsheet application packages to see this. A recent survey of business applications for the Macintosh

personal computer indicated that spreadsheet applications and integrated packages containing spreadsheet applications were 6 of the top 15 sellers [35].

Spreadsheet applications, for the purposes of this dissertation, have two distinguishing characteristics:

- A grid-based matrix of *cells* which can hold data values and/or formulae. Each individual cell can be addressed using a Cartesian coordinate system.

- Formulae, often hidden from casual inspection, which can be placed in individual cells, and can refer to other cells elsewhere on the spreadsheet.

Such formulae typically are not visible upon casual inspection of the individual spreadsheet, instead, the <u>result</u> calculated by a formula within a cell is displayed within that cell. There are usually mechanisms for displaying all formulae at once, but typically this is not the normal mode of working with a spreadsheet application. Instead, a user normally accesses one cell at a time and works with the formula within that particular cell.

Unfortunately, spreadsheet applications, while successful as an end-user programming environment, tend to be an error prone programming medium. Several studies have pointed out commonplace errors in the spreadsheets of even experienced users [11; 43]. I shall argue later in this dissertation that a large part of the problem with conventional spreadsheets has

to do with the hidden formula method of specification, and that one way of making a spreadsheet application that is less error prone is to eliminate the need for hidden formulae.

2.4.2. Other Spreadsheet-based visual programming environments

A number of efforts in the past have attempted to extend the spreadsheet model of computation in one way or another. However, most of these have retained the traditional "hidden formula" approach.

NoPumpG and NoPumpG II [31; 55] were efforts to extend the spreadsheet model of computation to interactive graphics: while they retained the traditional formula based approach, they did away with the grid based indexing system of traditional spreadsheets, relying instead upon a system of naming cells to be used in formulae. Since named cells are no longer referred to by their location, they can be "liberated" from the traditional X-Y grid and allowed to move (and be moved) around the spreadsheet as necessary. This proved to have several ramifications:

- The use of mnemonic cell names for data items often resulted in formulae that were much easier to read, in terms of problem-domain concepts, than similar formulae with indices in them.
- Allowing cells to be free-floating created difficulties with accessing a particular value, as the cell could be anywhere on (or, for that matter, off) the screen.

- Allowing users to place cells on the spreadsheet wherever they wanted to turned out to be a double-edged sword: It was useful to organize cells according to function and their place in a given calculation, but often difficult to do so. At times, it was useful to arrange cells close to the particular graphical item they were associated with. But because this was a system for interactive graphics, such items could move or be moved, leaving their associated cells behind.
- It did away with one of the major advantages of traditional spreadsheets – the ability to replicate computations easily by cutting and pasting, or filling rows and columns, combined with relative cell references. I have elected to retain the traditional grid in DataSheets, but have combined it with a slightly different approach to relative cell addressing, which still allows a user to cut and paste position independent chunks of computation.

Action Graphics [28] is a spreadsheet-like visual programming environment aimed at interactive simulations. Like the NoPump family of systems, Action Graphics does away with the traditional grid-based approach to computation, in favor of a named-cell based approach. Action graphics also uses formulae (arbitrary lisp expressions) to specify computations. The Forms family of languages [1; 2; 12] retains the traditional hidden formula method of specification, as well as the traditional grid structure.

2.4.3. Other visual paradigms

2.4.3.1 Constraint-based

One language paradigm that is not the exclusive domain of graphical

programming, but is often associated with graphical presentations of languages, is constraint-based programming. In constraint-based programming, the user specifies the primary quantities of interest in the problem solution and any constraints there may be upon the values they can take, and the system figures out the calculations required to ensure those constraints are satisfied. Sutherland's SketchPad [48], the archetypal interactive graphics program, was a constraint based system in that the drawings created had constraints that specified the relationships between parts of the drawing. Borning's ThingLab [10] was an early constraint-based language that used a visual syntax to specify the constraints. Since constraint systems do away with the need for an explicit ordering of statements on the programmers part, systems based on constraints are often proposed as candidates for end-user programming languages (see, for instance [20]).

Spreadsheets can be viewed as related to fully-constraint based systems. A spreadsheet system of inter-defined quantities is really a forward-only constraint system, in that values are defined to be dependent in specific ways on the values of other values, but the constraint is only in one direction. Fully constraint based systems have bi-directional constraints, in that defining A to be dependent upon B and C has the effect of also defining B and C to be dependent upon A. Unfortunately, this has a significant effect on the complexity of defining constraint-based solutions. It is not always possible to fully derive the backwards relation given the forward relation. If, for instance, A is defined to be the sum of B + C, and A is changed, the question becomes: which value is changed in response B, or C or both? Spreadsheets avoid this problem rather simplistically by declaring that once A is defined as depending upon B and C,

you can't change A at all, just the values it depends upon. This may seem a Draconian solution, but the astonishing variety of things done in modern spreadsheets is eloquent testimony to the power of the forward-only constraint mechanism.

2.4.3.2 Demonstrational

Another paradigm of computing that often gets mixed up with the notion of graphical programming is programming by demonstration (PBD) or programming by example (PBE). According to Myers [39], in programming by demonstration (which he refers to as Programming With Example) a user demonstrates the actual sequence of actions involved in the program solution, while in the PBE a user simply presents input/output pairs to the system and allows the system to fully infer the resulting program.

In the vein of PBD systems, such systems as Maulsby's Metamouse [36], Cypher's Eager [16], Lieberman's Mondrian [33], and Modugno's Pursuit [38] are all examples of the PBD paradigm. Several of these systems have been put forth as a means for end-users to automate routine tasks – in other words, the author's have advanced them as candidates for end-user programming languages. A main difficulty I see with demonstration as a means of programming, especially for end-user's, is that of *sequencing*: to properly demonstrate the actions a system must go through to solve a particular problem, the user has to be able to perform all the required actions, *in the proper order,* for the system to learn them. This seems to fly in the face of many of the results of research in the psychology of programming, that view programming as an opportunistic process in which parts of the program solution may be developed

in many different orders having nothing to do with the actual sequence that must be performed for the final result. One solution is obviously to design the solution off-line prior to demonstrating the completely designed solution to a PBD system. However, this seems unnecessarily cumbersome and effortful, and would appear to obviate one of the purported advantages of PBD.

Programming by Example (PBE) doesn't have the same drawback as PBD, in that the sequence of actions that the user goes through isn't remembered or used by the system in inferring the final program. Instead, what is important is the input/output pair. The system notes the transformation needed to get from initial input to final product, and infers a program to execute that transformation. Obviously, most programming problems are vastly under-specified by one input-output pair, as there may be many ways of making that particular transformation which would result in vastly different outcomes given some other initial input. It then falls to the user to specify a "covering" set of input/output pairs to fully specify the programming problem to the system (or to at least specify it to a point where all possible solutions are equally "good" as far as the user is concerned). This is obviously difficult, equivalent to designing a covering set of test cases for program testing. It's unclear whether end-user, non-specialist programmers will ever be able to consistently derive such a covering set of cases, given the trouble even professional programmers have for anything but the most trivial of programs.

2.4.3.3 Rule Based

Rule-based programming has a long and successful history. Recently, a number of efforts have been made to take the ideas behind rule based, or re-

write systems, and apply them in graphical programming systems as a means of creating end-user programming environments. The results, in such systems as Bell's ChemTrains [6], Furnas' BitPict [22], and Macintyre's Vampire [37], are systems for creating "behaving pictures." Typically, a user specifies a number of re-write rules by giving "before" and "after" pictures. When given an initial input picture, the system then employs a pattern matcher to match parts of the initial picture with the "before" part of the given re-write rules, and applies the appropriate graphical transformation. Such systems have been applied to a number of problems ranging from simulating Turing Machines to ecological simulations and others. Typically re-write systems handle qualitative simulations well, but don't work well for quantitative computations such as are involved in EDA tasks.

2.5. Graphical display of information

Another important thread of related work concerns recent and continuing work in developing a science of information representation. There are two main directions such work has proceeded in: data-oriented display methods and task-oriented display methods.

Along the lines of data-oriented display, several major works have appeared on the proper design of data graphics. Tufte's books [50; 51] are perhaps among the better known works of these. Prior to Tufte, Bertin's seminal works [8; 9] were among the first to put the design and semantics of the graphical display of data on a more rigorous footing. Much of the characterization of basic information types, and the appropriate graphical methods for displaying them, in Mackinlay's [34] and other later work can be found to have its roots in these

two books.

Mackinlay used Bertin's analysis of information types to describe the *expressiveness* and *effectiveness* of a given graphical language to display a particular information-relation in question. Mackinlay broke graphical languages down into a basis set of primitive languages, such as horizontal and vertical position and retinal languages, and a set of composition operators, such as single and double axis composition. From these he was able to demonstrate how many of the standard varieties of data graphics can be built up, and to analyze these displays in terms of their ability to communicate all the information in a given relation (the language's expressiveness) and their ability to communicate the information in a manner readily perceived by the human perceptual system (the effectiveness of the given language).

Larkin and Simon [29] proposed that the graphical display of information is useful for two reasons:

- Graphical displays reduce *search* for information, by arranging information in a way that either directly reflects the structure of the information itself, or that groups related pieces of information together.

- Graphical displays allow the substitution of visual (spatially oriented) operations for comparatively effortful mental (arithmetically oriented) operations.

Building on Larkin and Simon's analysis, Casner [14] used a task analytic approach to design graphical displays of information for a particular purpose. He did this by attempting to design graphics that allowed the substitution of particular visual operations for more effortful logical ones. He was able to show that for a given task, designing a graphic in such a way allowed users to perform the task more easily and more accurately than if it hadn't been designed in this manner.

This works well when a graphic is being designed to support a specific task (in Casner's dissertation, for example, the graphics were displays of information for an airline reservation system). Unfortunately, for the domain of EDA, the task is often much less clearly defined. The actual task may change from data set to data set, from session to session, and even within a session as new hypotheses about the data are formed and tested. There may be several pieces to the task that may suggest different (and perhaps contradictory) designs.

In addition, these visual operators can be inaccurate, or misleading. A famous case of this is the Playfair curve of England's balance of trade with the East Indies during the 18th century [15]. In comparing the two curves, it is natural to assume the visual operation used is scanning vertically from one curve to the other in several places along the curves and judging the distances to be less than, the same, or greater than. Unfortunately, the visual operation the eye uses is somewhat different – instead of using the vertical distance from curve to curve the eye is misled into using the shortest (i.e. normal to the curve) distance from curve to curve. What the eye perceives as two nearly parallel curves are, in fact, quite far from parallel (Figure 6).

For tasks where a specific visual operation may in fact be misleading, it may be better to use interactive display rather than a static one. When examining a table or a graphical display of information, readers will often use a piece of paper as a straight edge, or two fingers to measure and then compare the distances between two sets of points. Casner [13] coined the term "finger operators" for such physical operations that many of us use in daily life. In effect,

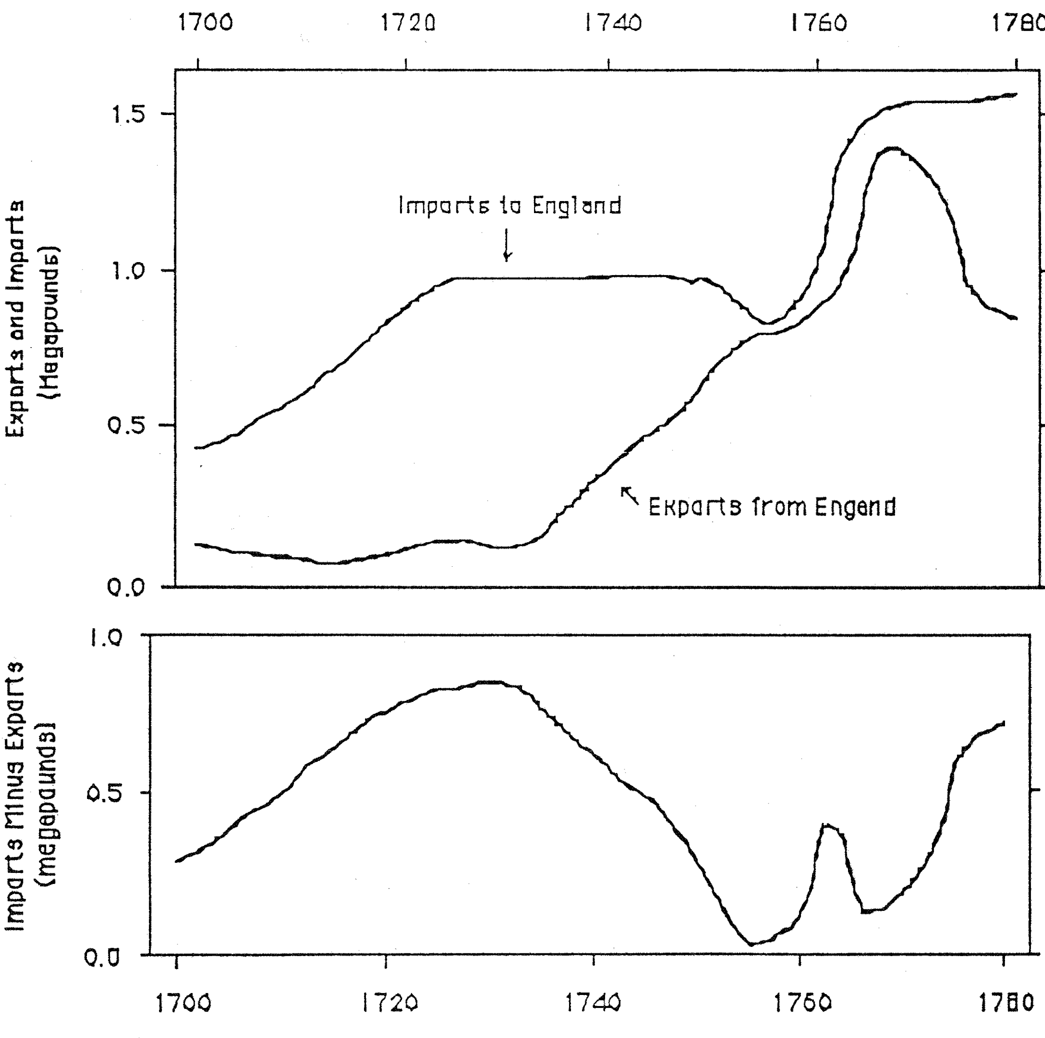


Figure 6. A classical case of graphical misperception. From about 1755 to 1765, the eye perceives the distance between the two curves to be moderate and only slightly varying, when in fact the difference between the two curves varies quite markedly (reproduced from Cleveland [15]).

in such cases we are substituting physical operations for visual ones. Physical operations are less error prone because they are less subject to the vagaries of the human perceptual system. If we use a piece of paper to measure the vertical distance between the two lines at some point in Figure 6 above, for instance, then compare that distance to the distance between those two lines at some other point, we can pay attention to whether or not our measuring device is truly measuring what we are interested in (the vertical distance, not the normal distance), and we can directly compare the distances at two different places on the line segments.

With the proper computation and display environment, we could do the same thing virtually by creating a "measuring tool" – a line that slides horizontally along the space between the two lines, with one endpoint fixed on the top line and the other fixed on the bottom line. This tool could be configured to automatically compute and read out the current vertical distance between the two lines. We could then use the tool to move interactively along the curves, measuring the distance between them and noting whether it increases, decreases, or remains the same. Such a tool is but one example of the interactive techniques the proper environment for EDA might make possible.

For the reasons stated above, (task requirements can give rise to contradictory display requirements, task requirements can change, and that for some tasks the best display is not static but dynamic), it seems that to properly support the domain of EDA the best display environment will be mutable – an environment that allows the user to create a display appropriate to the task or subtask directly at hand, that allows the user to then change that display as the

requirements of the task change, and that allows the user to incorporate interactive techniques or dynamics where appropriate. This is what DataSheets, the system presented here, aims to provide.

2.6. Statistical packages and EDA

Over the years there have been a number of attempts, some quite successful both from a research and commercial standpoint, to support the related domain of confirmatory statistics with computer systems. Exploratory data analysis deals with the finding relationships within a data set, while confirmatory statistics deals with confirming those relationships that have been found actually exist and are not just a product of coincidence. Many of the statistics systems are command-line oriented, and require a great deal of training for the user. Applications such as S++ [4] and Minitab [44], while more directly aimed at confirmatory statistics, provide many functions and displays that can be used for exploratory data analysis. Many of these packages are command-line oriented, though, and while they may provide the ability to compose already supplied functions in different ways, they provide little in the way of mechanisms for designing basic functions and displays themselves. In many ways, such environments represent the application-centered view of computing applied to the domain of statistics.

LISP-Stat [49] is an interesting example of an environment, aimed at confirmatory statistics that allows end-user extension of the environment's functionality. LISP-Stat is an object-oriented toolkit of the kinds of objects useful for the domain of statistics, which a user interacts with and manipulates through the LISP system listener (a command window which serves as the primary

means of interacting with the LISP system itself). Since the whole environment consists of nothing but LISP objects and the LISP environment itself, as well as the source code for those objects already defined, it is by definition extensible. Of course, to be proficient with it the user first has to master LISP and the notions of object oriented programming, as well as to understand the capabilities and functionality provided by the supplied toolkit.

Tukey [52], who defined the field of EDA, designed several environments that were more directly aimed at exploratory data analysis. A primary lesson from such environments as Prim-9 [21] and others was that interactivity and dynamic displays could be quite useful in EDA – a classic dynamic display technique developed by Tukey is that of *spinning*. In spinning displays of data sets with 3 or more dimensions are rotated about one or more axes in order to allow the user to develop a more comprehensive picture of the data set.

Another interactive technique that has proven useful in EDA is that of *brushing* or *painting* in which related sets of data are set up on complementary displays (such displays could show, for instance the various projections of a 3 or 4 dimensional data set onto 2 dimensions) [5; 46]. Using a cursor to select a subset of data points in one of the displays then causes the same subset to be highlighted in the other display – in this manner a user can again explore a more global picture of the data set than might be available solely by looking at 2-D projections.

While such techniques as spinning and brushing represent useful

techniques in EDA in their own right, I believe there are a great deal many other ways in which the dynamic, interactive nature of the computer as a display mechanism can be put to use in EDA. A secondary aim of this dissertation, then will be to develop an environment in which dynamic displays of data could be used to better advantage, and in which interactive techniques for a specific display could be programmed by the user themselves.

3. Design Methodology

When designing a language to support a specific domain, one needs to first understand the requirements of that domain – what problem solving methods and techniques are unique to that domain, and how to properly support the domain and its methods. We need, therefore, a method for analyzing the domain and understanding the specific requirements it imposes on an end-user programming language. The method I have used here is *task-based*, or *problem-based*, design, following work done at the University of Colorado, Boulder on the programming walkthrough method [6; 7].

3.1. The Forward Walkthrough – Pushing the programming walkthrough

The programming walkthrough is a method for assessing a given programming language's *writability* – how easy or difficult it should be for typical users to solve target problems in the language. It is an inspection method, meaning that a language designer must first design the language, then apply the programming walkthrough retroactively in order to discover (and hopefully fix) problems.

Bell [1992] has shown that it is possible to use the programming walkthrough iteratively, designing parts of the target language, then using the method to critique those parts and perhaps to decide between competing alternatives. Even here, though, walkthroughs are used strictly reactively – the design must be generated prior to any walkthrough being performed.

This has several disadvantages. In the first place, most designers don't

really want to iterate a design – they would rather "do it right the first time." Design has a strong emotional and egotistical component to it, as no one wants to have the flaws of their handiwork pointed out to them. But this is precisely what inspection methods such as the programming walkthrough do. They expose all the problems that say in effect "your baby is ugly." Second, while the programming walkthrough can help choose among competing designs, it remains mute about how to produce design alternatives. Design ideas are left completely up to the designer's imagination and intuition, which has already been proven to be flawed (or the programming walkthrough would have nothing useful to say at all).

Thus, it seemed a logical next step to attempt to use walkthroughs more actively in the design process, to help generate a language design rather than to merely critique it. The programming walkthrough focuses on the *process* of designing a solution to a programming problem in the given language. It postulates this process to be a series of *steps*, and attempts to provide justification that an average user could use to arrive at each step given a definition of the language and some *guiding knowledge* about how to apply it.

When the language hasn't been invented yet, it seems hardly fair to ask how a user would specify a problem solution in a nonexistent language. Instead, we must focus on the process of problem solving, without embedding the problem solution in a particular programming language or paradigm. Such *forward walkthroughs* could be carried out by examining hypothetical solutions to problems the proposed language is intended to handle, without embedding them in any particular programming language.

3.1.1. Starting with a blank slate doesn't work

I wanted to push the concept of a forward walkthrough as far as possible. I therefore attempted to start the analysis and design with as little as possible fixed, to see in what direction thinking about the problems in my test suite would lead me. I intended to a paradigm-free forward walkthrough, in which I would sketch out generic solutions to all of the target suite problems, then embed them somehow in the design of the new language.

The programming walkthrough method requires a language designer or critic to place himself or herself in the mental shoes of the user for a while, to examine the process of designing a solution in a given language. The person doing the walkthrough must step back from what he or she knows about the language and problem domain and ask what the user would know, how the user would apply their knowledge, and most critically, where the user could go for guidance as to how to solve the problem. This involves separating the process of doing the walkthrough from the process of solving the problem, especially if the person doing the walkthrough is unsure of the problem solution.

In the programming walkthrough, this can be most easily accomplished by retrospection: the analyst works through the problem, or a portion of the problem, then "steps back" from the problem solving episode to reflect on what knowledge was brought to play in the problem solving process. The concept of guiding knowledge becomes critical. Whatever knowledge is not contained within the language specification or the problem statement becomes part of the guiding knowledge – what the user needs to know in order to solve

the given problem in the given language.

Without a language specification to fall back on, the concept of guiding knowledge becomes much more nebulous. In some sense anything that is not contained in the problem statement is now guiding knowledge. The processes of solving the target problem and designing the language (also a problem solving process) become muddled. Am I solving the given target problem (i.e. the problem the designer wants the language to be applied to), or solving the design problem (i.e. designing the language itself)? Would this step be much easier if a given language feature already existed, or would it be just as difficult as it was prior to inventing the feature?

My experience has been that a truly paradigm-free walkthrough is at the least difficult, if not impossible. In my opinion, in order to solve any non-trivial programming problem, we need to embed the solution in some generic programming paradigm, be it procedural, functional, constraint-based, or something else. This is true for at least two fundamental reasons:

- Memory requirements, and the need for a notational system
- Sequencing requirements – what comes first depends upon the paradigm used.

The need for a notational system arises because solutions to any non-trivial programming task will have several pieces, and will likely be large enough that it is difficult to remember all elements of the problem and nascent

solution without at least writing notes. Once you begin committing parts of the problem to paper, you begin committing to a particular paradigm, because whatever notation system you use will depend upon some paradigm for thinking about the problem. Use an English-language-like pseudo-code, and you're thrust into thinking about the problem in a procedural manner. Use formulae to specify relationships between different pieces of the solution without specifying ordering, and you begin working in a constraint-based or spreadsheet paradigm. Try to write out the actions required to solve the problem, and a demonstration-based paradigm starts to emerge. My experience in designing DataSheets has been that you can't get around it – there is no such thing as a truly generic, paradigm-free notational system.

Sequencing requirements are a problem because thinking about any complex problem requires thinking about one part or another first. Different paradigms require thinking about different parts first. Thus, depending upon the particular entry into a problem, the designer ends up drawn into a particular paradigm not by any intrinsic quality of the problem, but simply by the order in which they chose to attack the pieces.

3.1.2. If not a blank slate, then what?

If one can't create a truly paradigm-free, start-from-nothing forward walkthrough and discover the best possible programming milieu for a given set of problems, what can a language designer do? Instead of starting from a completely blank slate, a designer could attempt to imagine problem solutions in more than one particular programming style, and then compare those solutions. The idea here is, instead of trying to decide what a canonical solution would look like to a particular problem, look at what the solution might look like if we

imagine solving the problem in each of several different popular programming paradigms.

Even within a particular paradigm, it becomes difficult to keep the process of programming a solution in mind when dealing with the question of what the programming language should look like – in essence, the search for functionality drowns out much of the concern for writability. In my own experience, I ended up designing solutions for the given problem suite in 3 different paradigms:

Spreadsheet-based	A forward-only constraint system specified on a 2-dimensional grid that holds data values and formulae.
Constraint-based	A user specifies constraints on the various pieces of the graphical display based upon the data values, and the system figures out the details of the display based on those constraints.
Demonstration-based	A user demonstrates a concrete sequence of actions to be taken to solve the given problem – Since much of EDA involves repeated steps over a set of data points, the system could be built to infer what to do for the entire set based on demonstrated actions on one or two data points.

The *paradigm-based, forward walkthrough*, then, consisted of designing proto-solutions to problems in the domain of interest in the chosen language

paradigms. Such proto-solutions are at a level equivalent to pseudo-code for procedural languages – they tell what needs to be done, but not necessarily what specific language features will be invoked to get the work done. The task suite used to design DataSheets consisted of 4 tasks, shown in Table 2. The task suite was chosen to be representative of the variety of tasks a user might be faced with in the domain of EDA.

Table 2. Task suite for the initial design of DataSheets.

Task	Description	Where from	Rationale
Budget	Given a set of names and salaries, and a percentage range for each, calculate: total of salaries, new salaries, total of new salaries, and percent increase of total.	CACM [43]	Typical task in end-user programming and spreadsheet processing.
Scatterplot	Given a set of paired data values, actual vs. predicted, plot a standard scatterplot of the data.	Standard	Typical task in EDA, involves a lot of repeated actions over a set of data points.
Stem and Leaf Plot	Given a set of data values, plot a Stem and Leaf plot of the data.	Tukey [52]	Typical task in EDA, also involves a lot of repeated actions over a set of data points.
Box-and-Whiskers Plot	Given a set of data values representing a sample from some population, plot a Box-and-Whiskers plot of the sample	Tukey [52]	Typical task in EDA, involves non-repeated actions, drawing plots based on values derived from but not directly out of the data set.

Proto-solutions were designed for each task in each of the three target paradigms: spreadsheet-based, constraint-based, and demonstration-based. For example the Box-and-Whiskers plot task involves plotting several horizontal lines that represent calculated values from a specific sample of some population

(see Figure 7). One subtask of that entire task involves plotting the line representing the median value of the data. Example proto-solutions for this particular subtask are shown in Figures 8 – 10.

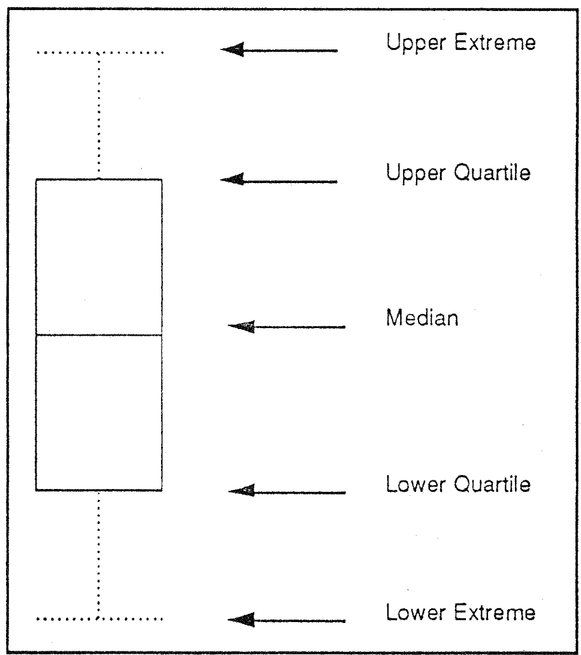


Figure 7. The Box-and-Whiskers task involves plotting several different horizontal lines representing various values calculated from a sample of some population.

Spreadsheet-based

Specify input cell range
Specify sort of input range
Specify count function
Specify median depth = count / 2
Specify median value = value[median depth]
Draw horizontal line
Specify Y position cell as median value

Figure 8. Proto-solution for subtask of drawing Box-and-Whiskers median line in a spreadsheet-based language.

```
Constraint-based
Draw horizontal line for median value
specify vertical value as value where
# of data points > vertical value == # of data points < vertical value
specify X = # of data points > vertical value
specify Y = # of data points < vertical value
specify X ==Y
```

Figure 9. Proto-solution for subtask of drawing Box-and-Whiskers median line in a constraint-based language.

```
Demonstration-based
Draw horizontal line
Read in data value
Place vertically in position
Adjust horizontal line to median of data values so far read:
# of data points below line == # of data points above line
```

Figure 10. Proto-solution for subtask of drawing Box-and-Whiskers median line in a demonstration-based language.

Forward walkthroughs are in useful in identifying the paradigm that best fits a certain class of problems. If the designer of a language has trouble imagining solutions in a particular language style, it is likely the intended users will also. Often certain paradigms seem more "natural" for certain classes of problems.

For the DataSheets language, working through problems revealed that EDA tasks often consist of specifying two different types of things: the computational manipulations designed to put the data in the proper form to be displayed (or to bring out the proper quality of the data to display) and the graphical manipulations designed to constrain a display to show the correct information about the data. In examining both the demonstrational and constraint-based approaches, it became clear that they were more suited to the graphical part of the tasks, while the spreadsheet-based ideas seemed more natural for the computational part. It also seemed somewhat difficult to specify

the display part abstractly – the mechanisms most useful for specifying the computational part of the solution seem least useful for specifying the graphical/display part.

To put it another way, certain parts of the problems lent themselves to a computational approach – calculating the median of a set of values, for instance. Other parts of the same problems lent themselves better to a more direct manipulation approach – drawing a horizontal line that is then assigned a (median) value as its vertical position is more natural than specifying four coordinates to form a line, where two of those coordinates are set equal to each other (the Y coordinates) and the other two coordinates are arbitrary (the X coordinates).

Consequently, the design of the DataSheets language evolved into an attempt to combine two equally powerful paradigms: the spreadsheet for computation, combined with a more direct-manipulation, constraint-based drawing package for image manipulation.

3.1.3. Functionality, level, packaging

The choice of one or the other overall paradigm colors the search for a solution path in the nascent language. Within a given paradigm, examining possible solution paths for target problems helps to define the proper functionality for the language elements – i.e. what the building blocks are upon which other solutions to different problems can (hopefully) be constructed.

This isn't all bad. The right functionality is a primary component of

usability for any system. This is especially true for higher-level or task-oriented programming systems in which much of the gain in programmer productivity is to be obtained through having the right high-level language constructs to match the tasks. Writing an expert system in a spreadsheet, for example, is somewhat ludicrous because the spreadsheet doesn't provide the appropriate functionality for that task – building a year-end ledger in OPS 5 is equally inappropriate.

Not only must the right functionality be incorporated into a programming language, that functionality must be at the right level for the class of users and problems it is intended to deal with. Although it's possible to write any kind of program in assembly language, no one would argue that it is the most productive use of a user's time – the language is simply too low level for many tasks. At the same time, a language that is at too high a level may artificially constrain the use of that language. Forward walkthroughs can help determine the right level of functionality for a particular language. In sketching out solutions to several different target problems, it soon becomes apparent that there are commonly repeated steps, or groups of steps. The right level is where commonly repeated elements are specified by one or a few steps – if you find you're repeating a large number of steps from solution to solution, the language is at too low a level. If you're not repeating any at all the language may be at too high a level – look for ways of factoring common steps out of some of the higher level constructs you've come up with.

A third area where forward walkthroughs can help is identifying when the proposed language facilities seem especially awkward, requiring what seems to be counter productive steps to arrive at the solution. In this case, new

functionality might be called for, or already provided functionality might be better packaged at a higher or lower level to fulfill the same need. This case comes closest to the kind of design criticism that the ordinary programming walkthrough provides – we could say that if a designer is tempted to write specific guiding knowledge aimed at motivating one step, and only that step, then perhaps they should look for a less cumbersome way of doing the same thing.

The right functionality, the right level of functionality, and the right packaging of functionality at the proper level to make solving the target problems easier – these form a continuum of concerns that the designer faces as language design progresses. They also form a continuum of concerns that walkthroughs can address. As the language design within a particular paradigm progresses the designer moves from a forward walkthrough, worrying about the steps a particular problem requires to solve it and the functionality needed to provide those steps, to the programming walkthrough, where it is more useful to worry about packaging the functionality appropriately.

An example will make this concrete. At one point in the process, DataSheets had been designed to work on "columns" or "rows" of data, either applying the same operation to each data item separately (i.e. take the log of all values in the column), or applying one operation to the group as a whole (i.e. find the maximum value in this column). The need for isolating and applying operations to some subset of the data, based either on the data values themselves or on a related column of data, quickly became evident as problems were worked through. This was a significant amount of necessary functionality that wasn't

initially conceived of.

As this need was brought out by several problems, not just one, it was clear that the right level is one that makes this type of operation easy to specify – i.e. in just a few steps. A first cut at providing this was a simple predicate operator, that applied a predicate and built up a series of 1s and **`NoValues`** in a new column depending upon the values in the data column. Once a column of truth values is formed, a user can copy the original column of data over and apply a multiplication operator between the two columns to separate out the original data values that passed the test in the first case (Figure 11).

Although this provides the necessary functionality, it involves several operations that are more an artifact of the programming environment itself, and less directly relevant to the programming solution – the act of copying the original data over and then multiplying by the truth values. It was also noted that these three operations were applied in exactly the same fashion in several different solutions – apply a predicate, copy the original data over, then multiply. To raise the level of the language, and do away with the non-intuitive steps of copying and multiplying, it was decided that these steps could be done by the predicate operation implicitly – the result of applying a predicate operation is no longer 1 or **`NoValue`**, but the original data value back if the predicate passes. This combines the three operations into one, with very little loss of generality (Figure 11).

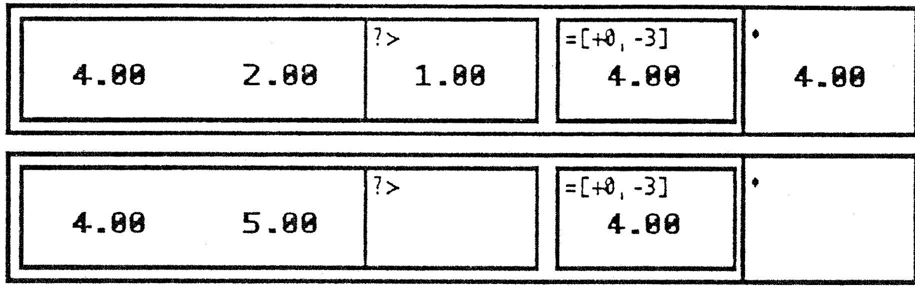


Figure 11a. Original conception for filter operations. The ?> operation results in a 1 if the leftmost of the two cells on the left is greater than the other cell on the left.

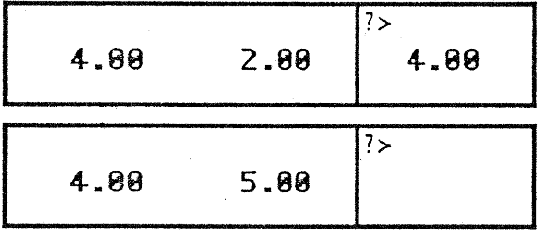


Figure 11b. Revised conception for filter operations. The first value is passed through if all values meet the specified test.

Figure 11. Original conception for filter operations involved passing a 1 through as a truth value and a blank (NoValue) as false. Since this often involved multiplying by the original value to get it back it was decided to pass that value through as a truth value and NoValue for false.

4. Original Design

A program in DataSheets consists of one of more worksheets, each consisting of a computational view and a graphics view (Figure 12). The set of worksheets that comprise a single program is called a workspace, and can be managed (i.e. saved, opened, etc.) as a single entity. The computational view of each worksheet is similar to a standard, two-dimensional spreadsheet in both appearance and method of computation.

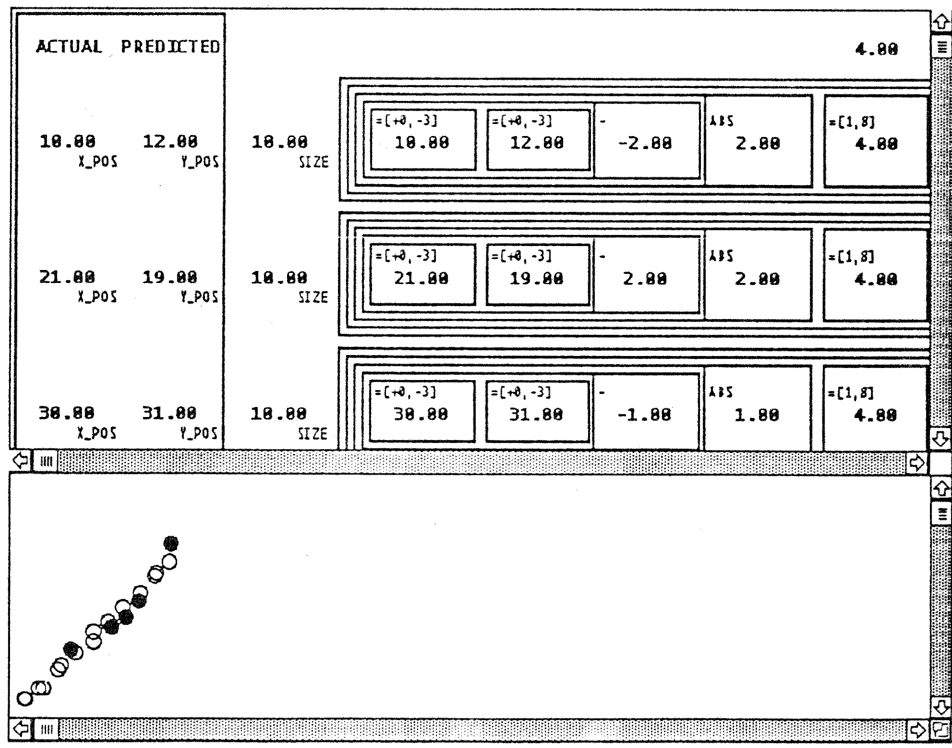


Figure 12. A DataSheets worksheet consists of a computational part (the top pane) and a graphics view (the bottom pane). Although the graphics and data views of a single worksheet are constrained to the same physical window (to emphasize that they are merely different views of the same information), each is independently resizeable as panes within the window.

In addition to the computational view of the worksheet, there is an additional graphics view that serves as the main visual representation of the data values. *Marks* on the graphics view can be associated with a particular row of values in the computational view – each cell in that row may have a cell *type* which corresponds to a particular graphical *attribute* of the mark's visual representation. The linkage between graphical attribute and cell value is two-way – if the cell has no computation constraining its value, its value is free to change to reflect changes in the display (for instance, if a user drags a mark around with the mouse).

4.1. The Computational View

4.1.1. The problem with spreadsheets

According to Lewis and Olson [32], some of the advantages of spreadsheets include:

- a familiar, concrete, visible representation
- suppressing the inner world (no *pumping* of data required from the programs internals to the screen or output device)
- automatic consistency maintenance
- absence of a control model
- low viscosity (difficulty of changing program without affecting other parts of the program)
- availability of meaningful aggregate operations (sum, avg., etc.)

- immediate feedback

As mentioned earlier, one of the primary disadvantages of spreadsheets is that they are an error prone programming medium. In one study by Brown and Gould, 41% of the spreadsheets created by experienced users of spreadsheets contained significant errors [11]. Of the nine participants involved in that study, all made at least one significant error in spreadsheet creation during the course of the study. Yet, it was reported the participants felt a "high degree of confidence" in the correctness of their solutions. Of the 17 different reported errors, 11 were errors in formula specification.

Despite their problems, spreadsheets remain one of the most popular applications on personal computers to date. And although a small industry providing pre-built spreadsheet applications has developed, the majority of users buy spreadsheet programs without such support from professional programmers. One can only assume that such users are building their own applications, and that they believe the results they are obtaining are indeed correct. Even users with programming experience aren't immune; in the Brown and Gould study, eight out of nine participants reported having "some" or "a lot" of programming experience.

Why the large gap between the perception of correctness in spreadsheets, and the actuality? In previous work, we asked users to perform some simple tasks involving querying and understanding a spreadsheet-based model of a simple household budget, and a graph-based model of the same material [53]. When using the spreadsheet-based model, few, if any of the users

paid much attention to the formulae that actually specified how the model worked. Instead, they relied on the overall formatting of the spreadsheet (row and column layout, italicized and underlined labels, etc.) to form hypotheses as to the underlying computational structure of the spreadsheet, and then tested those hypotheses by entering numbers into the spreadsheet and observing changes.

The fundamental problem, then, is that of the *visual presentation of the spreadsheet leading the user astray* – the computational structure (the formulae), and the visible structure (the row and column layout, and formatting) are completely independent of each other, and in fact, can even contradict each other. That the visible structure of the spreadsheet and the computational structure can be at odds is not only possible, it is likely – the two must be specified independently by the spreadsheet builder, and if care is not taken, changes to one are often not reflected in the other. Accordingly, in DataSheets I attempt to address this problem by doing away with the traditional "hidden formula" approach to spreadsheet specification – instead, computations are built up by specifying that one or another *range* of cells, by virtue of its placement, computes a specific value.

The computational view of the individual worksheets in DataSheets looks much like an ordinary spreadsheet – it is a two-dimensional array of cells, into which values may be entered and other values calculated. In a standard spreadsheet it is usual to address columns by the letters A through ZZ, and the rows by whole numbers, starting at 1. In DataSheets, numeric values are used for both, making each cell addressable in a manner similar to an array reference (i.e.

A[1,2] addresses the value in the row 1, column 2 of worksheet A). In addition, the indices in a worksheet address are not constrained to be integers – if a fraction is used, a linear interpolation of neighboring cells is used to provide the appropriate value (i.e. the address A[1,2.5] names the value that is the linear interpolation between the value at A[1,2] and A[1,3]). If one of the neighboring cells has value `NoValue` (i.e. is blank) or `Error` then the value of the address being interpolated is also `NoValue` or `Error`.

4.1.2. The cell

The cell is the basic data unit within DataSheets. Each cell consists of three parts: The *value*, *operation*, and *type* fields (Figure 13).

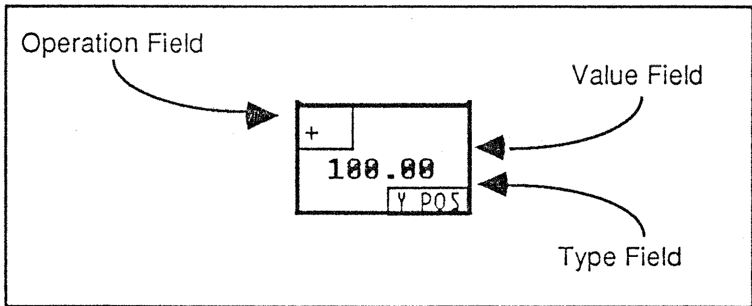


Figure 13. The basic parts of a cell.

The value is the data value this cell holds at the moment – if the operation part of the cell is blank, this field is editable or can take on new values when the corresponding parts of the graphic display are manipulated. Values can be either numeric or textual data. The cell operation tells what operation has given this value as a result – i.e. if the cell operation has a "+" in it, this value is the result of adding other cells values within the same cell range together. The

cell type, reflects the cell's ties to the graphics view, and names the attribute of some mark in the graphics view that this value corresponds to.

4.1.3. The cell range

The cell range is the basic unit of computation in DataSheets. A cell range consists of a horizontally or vertically contiguous set of cells, which comprise a computation. The last (bottom-most in a vertical cell range, rightmost in a horizontal cell range) cell (called the result cell) in a defined range always has its operation field filled in. The specified operation is applied to the values of all the cells in the range save the last one, and the resultant value is the value of the result cell. The central idea is that a sum should reflect the user's intuitions of what a sum should look like – a column of numbers with the total at the bottom or the right hand side. Note that when a cell gets its value specified as the result of computation, its value can no longer be edited. Examples of simple cell ranges are shown in Figure 14.

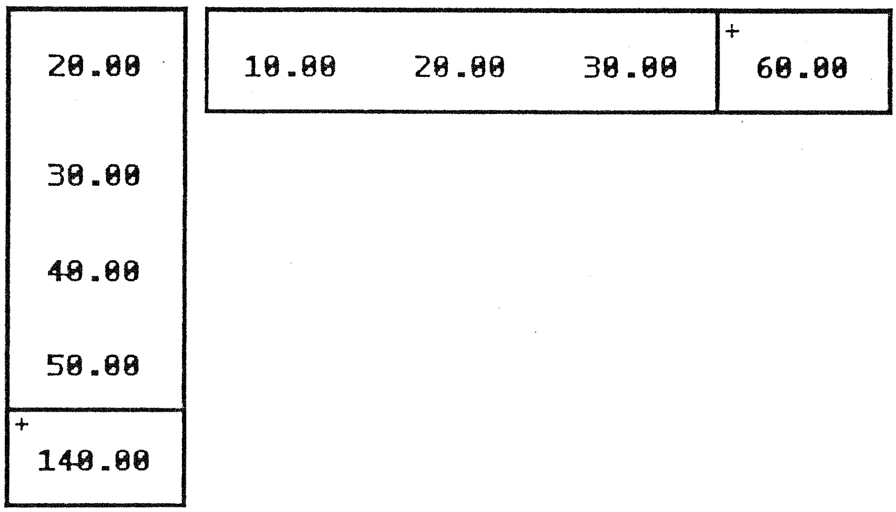


Figure 14. Examples of simple vertical and horizontal cell ranges

The process of specifying a computation is fairly simple – the user selects either a horizontal or vertical range of cells (by dragging with a mouse across the cell range), then specifies the computation they are to take part in, selecting from a palette of available functions. The operation field of the last cell gets filled in automatically, the range gets highlighted with a heavy black border to indicate what is and what isn't part of the computation, and the appropriate value gets computed in the last cell. The part of the range that comprises the input to the function (the argument subrange) is separated from the result cell by another heavy black border to distinguish it visually.

Ranges can be nested, in which case the value of the inner range is the value of the last cell in the range – this is the only value that takes part in the computation specified by the outer range. Ranges must be completely nested if at all – i.e. ranges can't overlap such that only part of one is contained in the other (Figure 15).

DataSheets supports the standard mathematical operations, as well as a set of logical operations designed to allow filtering subsets of data, addressing operations, and operations with side effects designed to allow the introduction of new data from a file. All operations work on numeric data only – textual data within the argument subrange is ignored and the cell containing the text treated as if it had a value of `NoValue`.

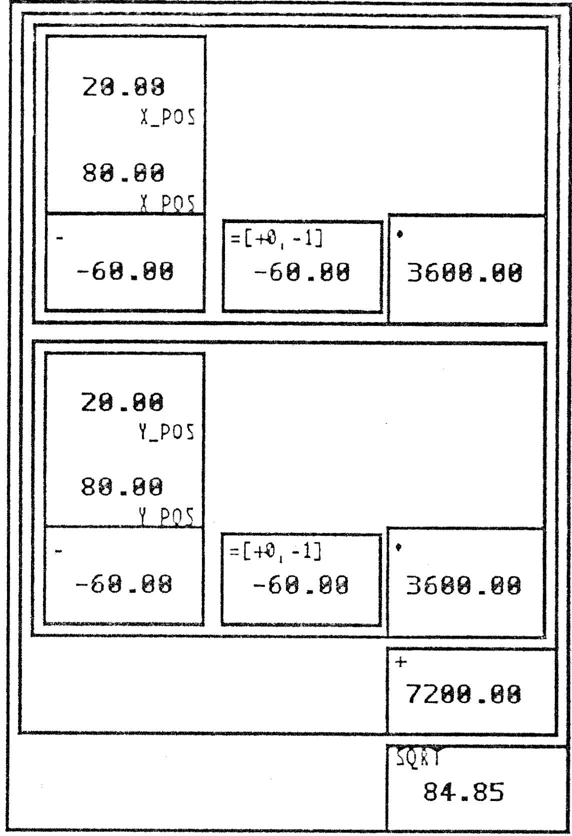


Figure 15. Examples of nested cell ranges.

4.1.4. Mathematical Operations

For binary operations, if the number of cells in the argument subrange is greater than 2, the result is built up by first applying the operation to the first pair of arguments, then to the intermediate value resulting and the next argument, etc. The order of evaluation is from left to right, then top to bottom. For unary operations, if the argument subrange contains more than one cell, the value of the result cell is simply the value of applying the appropriate operation to the first non-blank cell in the range.

In all cases, if an operation results in an error condition (for instance, divide by 0 or square root of a negative number), the result will contain a value of **Error**. Any operation with **Error** as the value of any of its arguments has **Error** as the value of its result. Tables 3 and 4 list the available binary and unary operators, respectively.

Table 3. Binary operators available in DataSheets

+	adds the values of all cells or ranges in the argument subrange and returns the value in the result cell.
-	subtracts the values of all the cells or ranges in the argument subrange and returns the value in the result cell.
*	multiplies the values of all the cells or ranges in the argument subranges and returns the value in the result cell.
/	divides the values of all the cells or ranges in the argument subranges and returns the value in the result cell.
div	integer division – divides the values of all the cells or ranges in the argument subrange, taking only the integer part of each division, and returns the value in the result cell.
count	returns the number of cells in the argument subrange in the result cell.
max	returns the maximum value of the cells in the argument subrange.
min	returns the minimum value of the cells in the argument subrange.

Table 4. Unary operators available in DataSheets

log	returns the common logarithm (base 10) of a cell
√	returns the square root of a cell
sin	returns the sine of a cell (as radians)
cos	returns the cosine of a cell (as radians)
tan	returns the tangent of a cell (as radians)
arcsin	returns (as radians) the arc-sine of a cell
arccos	returns (as radians) the arc-cosine of a cell
arctan	returns (as radians) the arc-tangent of a cell
floor	returns the floor of a cell
roof	returns the ceiling (roof is shorter) of a cell
abs	returns the absolute value of a cell

4.1.5. Addressing operations

A set of addressing operators, or *equality constraints*, allow values to be copied from other parts of the worksheet. This is necessary in order for a value to be used in more than one particular computation. The addressing operations all have the same basic form, but take from zero to three arguments in their argument subrange, allowing either or both of the indices in an address to be specified dynamically. The indices in a cell address need not be integers – if they contain a fractional value, the appropriate interpolation is done between cells to determine a value. Indices may be unsigned, in which case they refer to an absolute position on the worksheet, or signed, in which case they refer to a position relative to the result cells position on the worksheet. As with other operations, if a particular operation is used with the wrong number of arguments its result is `Error`. Each addressing operation takes an optional worksheet name as part of the reference – thus references from worksheet to worksheet are possible – if the worksheet name is omitted, the reference is assumed to be to the local worksheet. Table 5 lists the available addressing operations.

Table 5. Available addressing operations in the initial design of DataSheets. (foo) is an optional worksheet name – if supplied, the reference is to that cell on worksheet "foo."

=(foo)[a,b]	Specifies that the value from the cell in row a, column b is to be placed in the result. This is a fixed reference to another part of the worksheet, in that the indices are specified and can not change dynamically.
=(foo)[,b]	Takes one argument, and specifies the value from the cell in the row specified by the value of the one cell in the argument subrange and column b is to be placed in the result – a dynamically varying row reference, specified by the one argument in the subrange.
=(foo)[a,]	Takes one argument, and specifies the value from the cell in row a, and the column specified by the value of the one cell in the argument subrange is to be placed in the result – a dynamically varying column reference, specified by the one argument in the subrange.
=(foo)[]	(equivalently, =[,]) Takes two arguments, and is a fully dynamic reference – the value of the first argument cell is used as the row number, and the value of the second argument cell is used as the column number

4.1.6. Input Ranges

Within each worksheet one range can be designated as an **input range,** where values that can be read in from a data file are to be placed. Values are read into the input range by selecting a file to open from the "Input DataFile..." menu item in the file menu of the DataSheets application (The format of the data files is tab, space, or comma delimited within a row, and carriage return delimited from row to row). The input range can be multi-column and multi-row in extent, and data is read in to the input range filling cells from left to right, top to bottom. Input ranges compute one value in their result cell – the number of data items read in.

4.1.7. Abstraction Mechanism

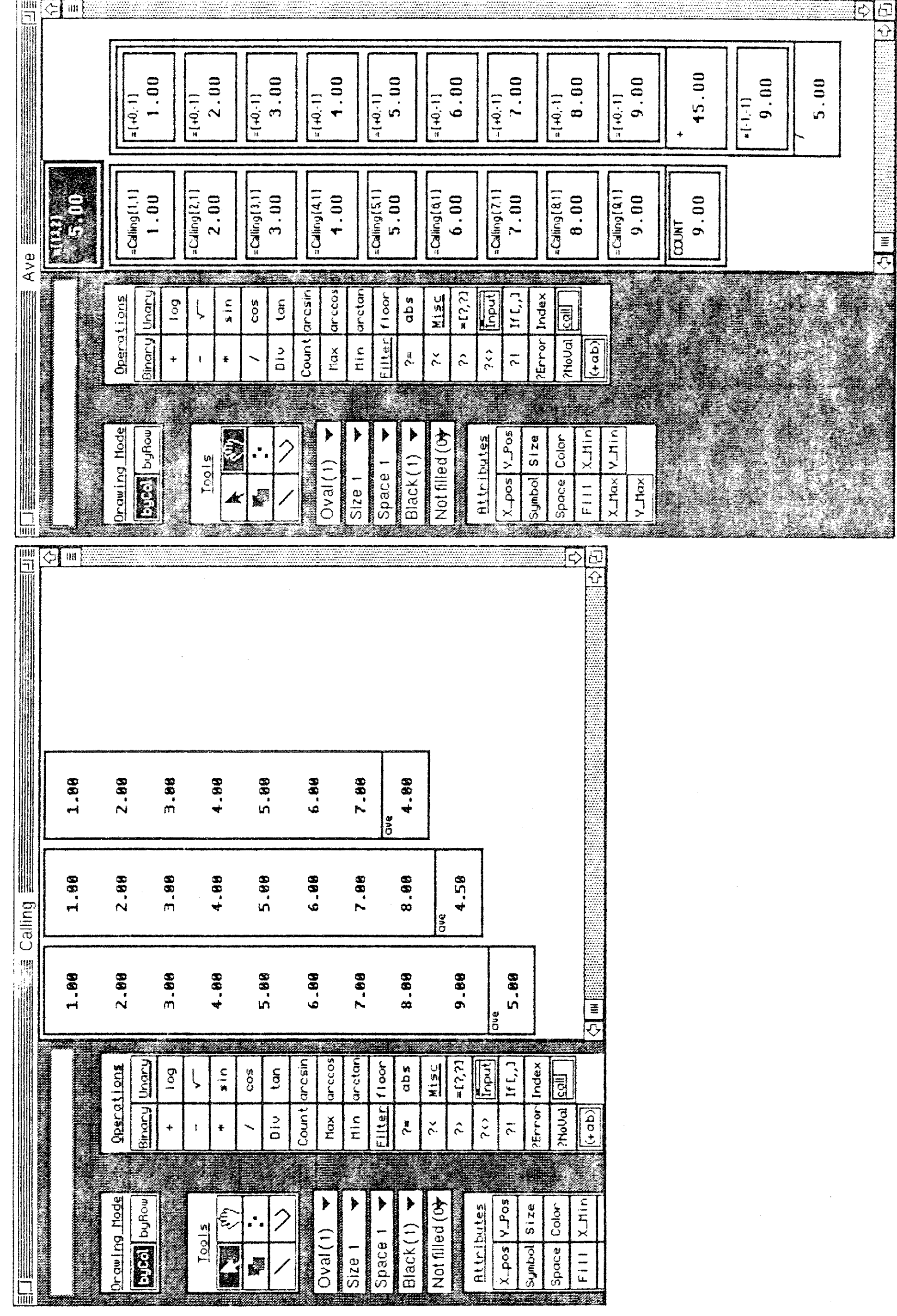
DataSheets has an abstraction mechanism to facilitate structured problem solving and re-use of solution steps. The abstraction mechanism uses the addressing facilities described above, so as to be readily comprehensible to the user. The abstraction mechanism works as follows: anytime a worksheet is created it is given a name (either by the system as a default or by the user if they wish a more semantically meaningful name). This name then becomes another operator that may be used in other worksheets. When a user calls one worksheet from another (i.e. includes the name of the first worksheet in the operation field of the result cell of a range in the calling worksheet), the following occurs: Addresses pointing to each of the cells in the calling worksheet's argument subrange are placed in the corresponding cells, starting in row 2, column 1 of the called worksheet. This comprises the input to the called worksheet. The cell in position [1,1] of the called worksheet is used as the result to pass back to the calling worksheet. Thus, the user can arrange to have a computed result copied into cell [1,1] of the called worksheet, and have that value show back up in the calling worksheet (see Figure 16).

One of the primary benefits of abstraction mechanisms is being able to call the same routine from a number of places in the calling routine – i.e. providing code reuse at the routine level. The mechanism described above really works as described when the called routine is called from one place only. To use the same mechanism for more than one call to that same routine, we have to reset the equality constraints in the called routine each time any of the values in the argument subrange of the calling routine change. As a result, only the constraints

and results from the last call to the called routine are typically displayed in the called worksheet. In practice, this turns out to be a useful state of affairs. It turns the abstraction mechanism into a truly visual mechanism – whenever a value in the calling routine is updated, the user can follow the subsequent updating of the constraints in the called routine and the value that is computed and returned to the calling routine. Which particular call is displayed can always be selected by the user simply by updating one of the argument values in the calling range.

4.2. The Graphics View

The graphics view of each worksheet consists of a set of marks, each of which may be associated with a row (or column) of cells in the computational view. Each cell in that row of the computational view may have an *attribute* associated with it, via its type field – if it does, the cell then both controls and reports the value of that attribute of the mark in the graphics view. If the cell's operation field is filled in, the attribute is fixed and cannot be varied by any other means than the computation of values. If, on the other hand, the cell's type field is filled in but not its operation field, the attribute is not fixed and can be changed by direct manipulation of the graphics view itself – the cell value merely reports the value of that attribute of the mark.


Calling
Ave

Marks can be created in one of two ways: by specifying attributes in a particular row of the computational view of a worksheet, a mark will be created to match those attributes (with default values for attributes not specified). Alternatively, a palette of drawing tools allows the creation of marks through direct manipulation in a draw-program-like manner. Some marks may be associated with a row which does not have cells naming all the attributes of that mark. Attributes of existing marks can be changed without there being a cell controlling that attribute – selecting the mark and then the proper attribute has the affect of changing that attribute to the selected value.

The choice between having rows or columns associated with individual marks is left to the user, although by default columns are associated with marks. The linkage between a column or a row and a particular mark is managed implicitly by the system – if a mark is created by drawing, it is automatically assigned to the next free (unassigned) column or row. Similarly, if attributes (cell types) are assigned to cells in a row or column of the computational view that has no mark currently assigned to it, a mark is created and assigned to that row or column.

4.2.1. Coordinate System

The worksheet graphics view uses standard Cartesian coordinates, with 0,0 in the lower left hand corner of the worksheet, X and Y positive to the right and top, respectively. The range of X and Y coordinates can be determined by 4 cells which are given the special types: X_Min, Y_Min, X_Max, Y_Max. Each worksheet can have one and only one of the each of these special types of cells – if the user subsequently attempts to define more than one of any of these four, a

dialog gives them the option of A) defining the new cell of this type and replacing the old one, or B) canceling the operation and leaving the old cell in place.

4.2.2. Attributes

Any mark can have its attributes controlled by cells in the computational view. The available attributes and numerical values that correspond to particular attributes are shown in Table 6. Each mark may have any number of cells controlling its X and Y position attributes. If a mark has more than 1 of either X or Y cells, it becomes a polyline of symbols the size and shape of which are specified by its other attributes. In this way, lines, and polylines can be specified by one row or column of the computational view. The pairing of X and Y positions to form coordinate pairs is done from left to right, with the first available X coordinate paired with the first available Y coordinate, etc. If there are more X position cells than Y the last X value is paired with the remaining Y values – similarly if there are more Y position cells than X.

Table 6. Available cell attributes and the significance of numerical values in the controlling cell.

Shape	1 = circle 2 = rectangle 3 = rounded rectangle
Fill	O = mark is solid (filled) 1 = mark is hollow (not filled)
Color	0 = white 1 = black 2 = pink 3 = red 4 = orange 5 = yellow 6 = green
Size	the value of the size attribute represents, approximately, the length of the side of a square covering the mark, in the coordinate system of the current worksheet.
Spacing	the value of the spacing attribute represents, approximately, the distance between symbols making up the current mark, in the coordinate system of the current worksheet.
X_Pos	the X position of the current mark (or one of its vertices) in the coordinate system of the current worksheet.
Y_Pos	the Y position of the current mark (or one of its vertices) in the coordinate system of the current worksheet.

5. Prototype implementation and initial testing

5.1. Implementation details

Once the initial language was designed, a prototype version was implemented using Common Lisp on Macintosh Quadra hardware. There are a few details of the implementation worth noting:

5.1.1. Propagation algorithm

The main computational mechanism in any spreadsheet consists of a forward-only constraint system. Typically, when a spreadsheet user enters a number into a cell or edits an existing value, a network of dependencies is kept in the background, and this network is checked for any cells whose formulae may depend upon the particular cell updated. If any are found, their values are updated, cells which depend upon those cells are updated, and so forth, with the changes propagated breadth-first until the network settles out or a cycle is detected.

In DataSheets, because there are no formulae, the propagation algorithm is simpler. Because there are equality constraints, some of which can be dynamically computed, the propagation algorithm is more complicated.

When a user enters or edits a value in a DataSheets cell, that cell's *containing range* , the range that contains the cell in question, may have the value of its result cell affected. The containing range of the cell's containing range may also be affected, and so forth, up to the level at which there are no more

containing ranges. Leaving aside the question of equality constraints for the moment, these are the only cells that can be affected by a change in a particular cell's value. Since overlapping ranges are not allowed, a cell can have one and only one containing range – that range can have one and only one containing range, etc. The propagation algorithm, then becomes straight forward – simply continue up the chain until all containing-ranges have been updated (see Figure 17).

```
(defun calculate-range
 (ws range &optional (start-depth nil) &key (update-shapes
t))

  (let*
      ((result-cell (get-cell ws (bottomright range) nil))
       (op (op result-cell)))
      (set-cell-value result-cell
        (if (cell-contains-formula-op result-cell)
          (calculate-formula-op ws range)
          (case op
         ..  list of operations and function calls deleted..
            (otherwise
              (calculate-call ws range start-depth))))
       :update-shapes update-shapes)))

(defun set-cell-value
    (cell value &key (update-shapes t) (calc-ranges t))

  (unless (eql (value cell) value)
    (setf (value cell) value)
    (when update-shapes
     ...code to redraw shapes in graphics view deleted ... )
    (when calc-ranges
      (let ((containing-range
            (if (and (op cell) (containing-range cell))
               (containing-range (containing-range cell))
               (containing-range cell))))
        (when containing-range
           (calculate-range
             (view-worksheet cell) containing-range))
        (update-address-cells
           (view-worksheet cell)(index cell)(index cell)
nil))))
         (refresh-cell-range
         (view-worksheet cell) (index cell) (index cell)))
```

Figure 17 Propagation algorithm used in DataSheets.

5.1.2. Handling equality constraints

Of course, things are seldom this simple. Equality constraints add complexity to the propagation algorithm, because they allow a mechanism whereby updating a particular cell's value can affect any other cell in the worksheet (in fact, any other cell in any worksheet in DataSheets, since we allow constraints from worksheet to worksheet). Not only can one cell affect any other, the cells it might affect are constantly subject to change, due to fact that equality constraints might have argument cells that dynamically select which row and column they are copying from.

There are two times during which an equality constraint may cause one cell to become dependent upon another:

- At specification time, for fully specified absolute or relative addresses to a fixed location.
- At run time, when a cell that is an argument cell of a range specifying an equality constraint gets updated.

Consequently, a table of possible equality constraints is kept by the DataSheets system, indexed by the address (including the worksheet) the constraint is dependent upon. At any time when a cell's value is updated, the set of current constraints is checked to find any constraints that may depend upon this cell. If any are found, that constraint is checked to make sure it is still current (if, for instance, it is a dynamic constraint the values of its arguments might have changed and the constraint may no longer be current). If it is current, the value of the cell at the receiving end of the constraint is updated, and the propagation algorithm is run again from that point. If the constraint is not currently valid, its

entry is removed from the table. The table of constraints is added to, both at specification time (when a fully specified constraint is selected), and at run time (when a cell that is an argument to an equality constraint range is updated). The work of checking for currently valid constraints, updating cell values, and deleting no longer valid constraints is done by the call to `update-address-cells` from within the propagation algorithm. Figure 18 is a code skeleton of the update address routine.

```
(defun update-address-cells (ws i j depth)

;; given a i and j, find any dependent equality
constraints

      (dolist
         (cell
          (gethash (list ws (make-index i j))
            *constraints*))

          ;; given an equality constraint,
          ;; check that it is still valid

          (multiple-value-bind
            (remote-ws remote-index)
            (calculate-remote-address cell))

          ;; remote-index and remote-address
          ;; is the current cell this constraint
          ;; points to

          (let*
             ((other-cell
               (get-cell ws (make-index i j) nil)))

             ;; if still current, update values,
             ;; otherwise, delete
             ;; constraint from table

             (if
               (and (equal remote-ws ws)
                 (equal remote-index (make-index i j)))
               (calculate-range
                  (view-worksheet cell)
                  (containing-range cell) depth)
               (clear-equality-constraint
                 cell ws (make-index i j)))))))))
```

Figure 18. Skeleton code for updating equality constraints. This is not the actual code, for various reasons the actual code is slightly more complex.

5.2. Prototype testing

Once the prototype was running, an informal round of testing followed with three subjects doing four tasks each (see Table 7). Subjects were first given a brief written overview of the DataSheets environment, then asked to "think aloud" [30] as they proceeded to solve the tasks assigned.

Although all subjects did not complete all tasks (mainly due to the availability of subjects' time), several difficulties with the prototype as designed were uncovered:

1) The mechanism for defining equality constraints between individual cells in the underlying spreadsheet was difficult to use. Users found it hard to key in the addresses of the cells manually, and difficult to scan for row/column addresses over large portions of the screen.

2) The use of the same Cartesian grid for specifying both graphical attributes, and for specifying computations, overly constrained the layout of programming solutions and made laying out both a program solution and an image design unnecessarily difficult.

3) The result of problem 2 above was heavy reliance on the constraint mechanism for copying cell values from one place to another within the spreadsheet. Since constraints were specified by cell address only, a great deal of semantic information as to what a computed value had meant in the underlying problem domain was lost, making partial solutions difficult to parse and thus to complete.

4) Links between graphical items and computations were implicitly assigned and managed by the system, a connection that proved too subtle for many users.

Table 7. Subject completion of tasks during initial prototype testing.

Task	Description	Subject 1	Subject 2	Subject 3
Budget	See Table 2	Yes	Yes	Yes
Scatterplot	See Table 2	Yes	Yes	Yes
Modified Scatterplot	Modify the scatterplot in previous task to display all outliers differently	Yes		Yes
Box-and-Whiskers plot	See Table 2	Yes	Yes	

6. Redesign

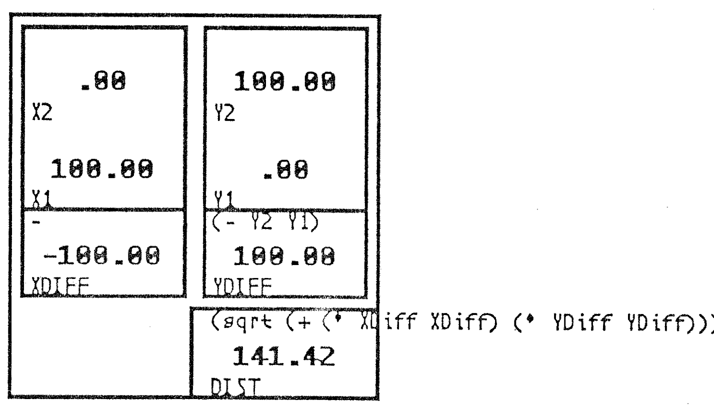
6.1. Proposed modifications

For each difficulty discovered during prototype testing, solutions were proposed, and the resulting design tested again by comparing paper-and-pencil solutions. Modifications that were proposed for each problem are listed in Table 8.

Table 8. Modifications proposed as a result of prototype testing.

Problem / Modification
Equality constraints too hard to specify:
• Equality constraint definition by point and click.
Overly constrained layout:
• Allow overlapping ranges, either just for computation or for both computation and cell-mark correspondence.
• Allow cell-mark correspondence by range or cell grouping, vs. by row/column.
• Allow explicit overriding of automatic cell to mark linkages when needed.
• Allow more complex formulae, instead of just binary operations in cell ranges.
Loss of problem semantics
• Allow named cells / cell ranges
Cell to Mark linkage not apparent
• Create a more visible indication of cell to mark correspondence.
• Allow explicit overriding of automatic cell to mark linkages when needed.

Comparing paper and pencil solutions using each of these proposed modifications revealed some interactions between them, and in fact suggested a method of combining possible solutions for problems #2 and #3: A naming mechanism could be used to implement complex formulae. When a user enters a formula as an operation for a cell range, any non-operation symbols in the formula are assumed to be names of cells in the given worksheet. If no such cells exist, cells within the argument subrange that have values are automatically given names corresponding to the arguments of the formula. In order to allow cut and paste of cell ranges containing formula (a technique which proved quite useful to users in the informal testing) names are defined to be local to the range they are found in, except the name of the result cell, which is also used as the name of the range as a whole. Local names can explicitly be referred to outside of their defining range by qualifying the cell name with the name of the defining range (Figure 19).



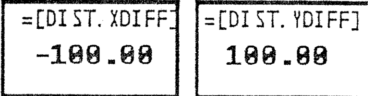


Figure 19. The naming/cell formula convention adopted in redesign. A name field was added in the lower left hand corner of each cell. Specifying a formula as the operation in a cell range results in names being assigned to argument cells. Names are local to a range unless specifically qualified by the name of the result cell of the containing range.

6.2. Adopted modifications

The set of modifications adopted are listed in Table 9.

Table 9. Modifications adopted as part of the redesign.

Modification
Equality constraints hard to specify:
• Equality constraint definition by point and click.
Overly constrained layout:
• Allow more complex formulae, instead of just binary operations in cell ranges.
Loss of problem semantics
• Allow named cells / cell ranges
Cell/Mark linkage not apparent
• Create a more visible indication of cell/mark correspondence.
• Allow explicit overriding of automatic linkages when needed.

Allowing equality constraint definition using point-and-click was accomplished via a modal dialog that occurs after a user selects an equality constraint as the operation to apply to a range – the cursor changes to indicate that a constraint definition is in progress, the next click by the user determines the constraint's arguments, and a dialog box allows user confirmation or changing of the selected arguments (see Figure 20).

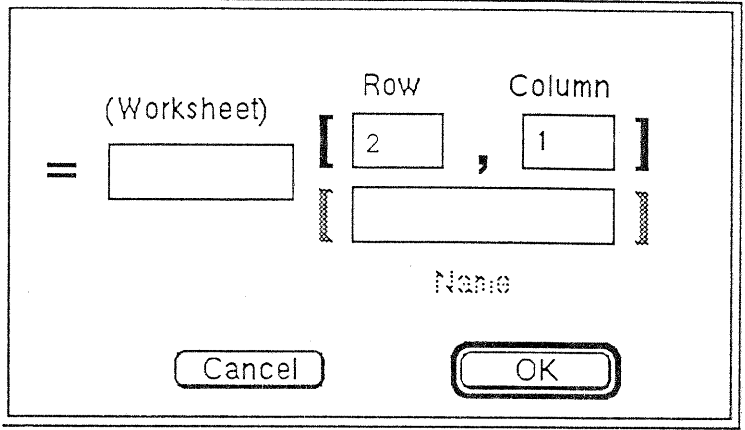


Figure 20. The modal dialog that is part of the redesigned equality constraint definition procedure. When a user chooses the equality constraint operation, a unique cursor prompts them to specify by clicking the arguments for the operation. The above dialog gives the user a chance to override or change the choices thus made.

Allowing complex formulae and named cells was implemented using the combined naming/formula mechanism described above. A more visible indication of cell/mark correspondence was created using a series of icons that appear at the beginning of each row or column in the computational view of the worksheet. The icon is a miniature version of the graphics view of that worksheet with only one mark in it – the mark associated with this particular row or column. Since the icon's view is updated to reflect dragging and manipulation of the corresponding mark in the graphics view in real time, the link between computational and graphics views becomes immediate and obvious (see Figure 21). The icons also allow the user to explicitly override the automatic correspondences set up between marks and row/columns of the computation view, by clicking on an icon representing a particular mark and dragging it to a new row or column.

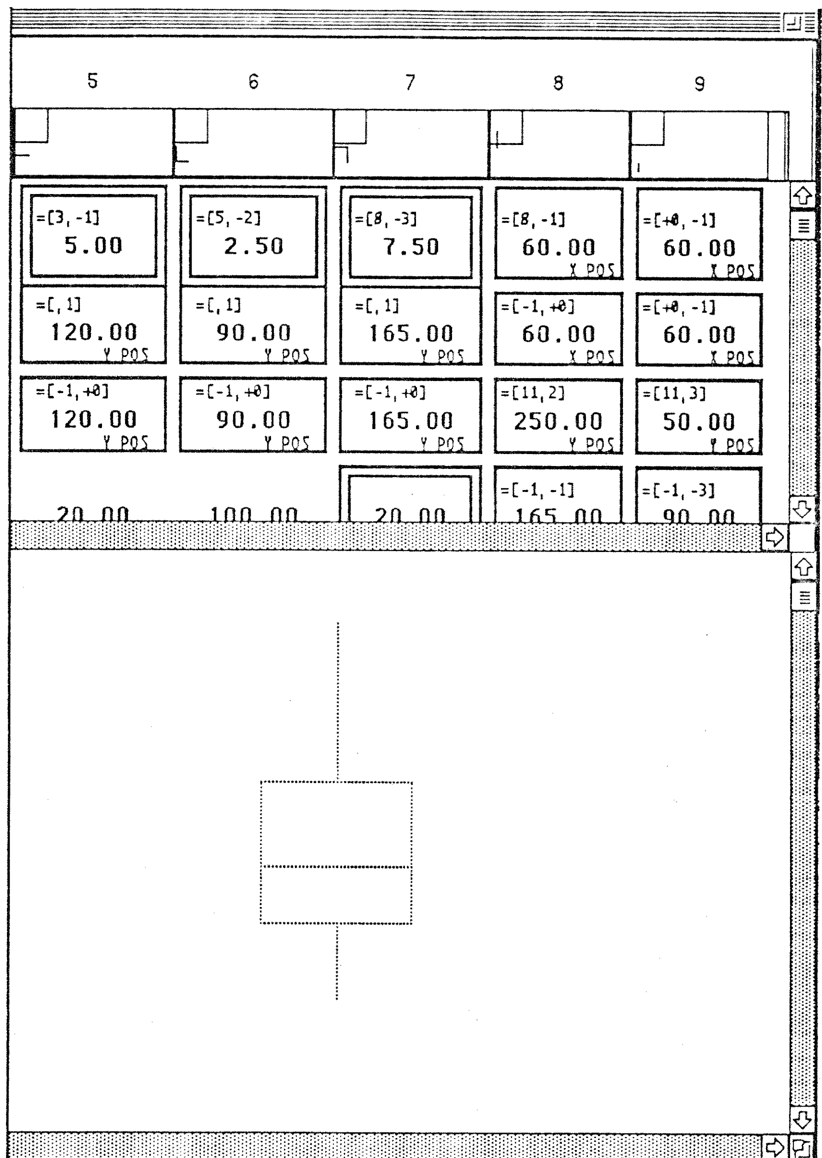


Figure 21. The redesigned linkage between computational and graphics view. The icons at top between the column numbers and the spreadsheet cells each represent one of the marks on the graphics view. Dragging or changing the appearance of the corresponding mark on the graphics view causes a corresponding change in the appearance of the icon.

7. Further Testing

7.1. Further testing – procedures

A second round of prototype testing was done with six users each doing three tasks using the revised prototype. The tasks used in this round of testing are listed in Table 10.

Table 10. Tasks used in second round of testing.

Scatterplot	given a set of 10 pairs of data, actual vs. observed, plot a scatterplot of the data with observed along the Y axis and actual along the X.
Modified Scatterplot	modify the display from task 1 to plot outliers (values where abs (actual - observed) is greater than some fixed value) are plotted in a different color.
Box-and-Whiskers plot	given a sample data set of 10 data points, build a Box-and-Whiskers plot of the sample.

One user had difficulty completing the tasks, due more to difficulties with the Macintosh interface conventions, than with difficulties in understanding the programming and specification of the task solution required. Table 11 lists time to completion for all subjects and tasks. Due to difficulties with testing procedures and bugs discovered in the program during the first testing session, the time for Subject 1, Task 3 is unusually long and is not reflected in the summary statistics.

To help with subjects' time management and to allow subjects to pick up much of the language details in a relatively short period of time, I first conducted a short introduction to the language and environment. This consisted of the subjects reading a page or two of documentation on particular parts of the

environment, followed by a demonstration of those features. In all, three separate episodes of reading about language features, followed by short demonstrations, were used to introduce the major features of the language. Total time for language introduction was about half an hour. During the actual trial, the experimenter was available to answer specific questions about interface capabilities or language features, rather than having users search documentation for the answers (see Appendix A for testing materials). Think-aloud protocols were taken, and the screen videotaped during each session. In addition, the environment was instrumented to write a journal file for each session, consisting of time-stamps and user interface actions (see Figure 22).

Table 11. Time to completion for all subjects and all tasks during second round testing (min:secs). *Time for Subject #1, Task 3 exceptionally long due to program/testing errors and is not included in summary statistics.

	Task 1	Task 2	Task 3
Subject 1	19:00	12:19	56:57*
Subject 2	8:10	11:39	31:00
Subject 3	14:07	15:01	36:00
Subject 4	11:31	11:33	28:52
Subject 5	7:28	15:19	33:28
Subject 6	7:48	13:33	36:00
Mean	11:21	13:14	33:04
Stand Dev.	4:33	1:39	3:08

```
Window Opened at 10:24:36
HAND selected at 8
+ pushed at 20
edit cell range VALUE 22 1,1 x 3,1 at 26
LINE selected at 36
DOTS selected at 43
chose menu symbol-size 10 at 44
ARROW selected at 50
edit cell range ATTR X_POS 1,1 x 1,1 at 63
X_POS pushed at 64
edit cell range ATTR Y_POS 2,1 x 2,1 at 66
Y_POS pushed at 66
edit cell range ATTR X_POS 3,1 x 3,1 at 81
LINE selected at 128
BYCOL selected at 135
edit cell range ATTR X_POS 1,4 x 2,4 at 141
X_POS pushed at 142
edit cell range ATTR Y_POS 3,4 x 4,4 at 146
Y_POS pushed at 146
edit cell range VALUE 10 1,4 x 1,4 at 150
ARROW selected at 163
edit cell range ATTR X_POS 5,4 x 5,4 at 175
X_POS pushed at 175
edit cell range ATTR Y_POS 6,4 x 6,4 at 177
Y_POS pushed at 177
edit cell range VALUE 300 5,4 x 5,4 at 181
edit cell range VALUE 200 6,4 x 6,4 at 183
edit cell range OP FORMULA 1,4 x 2,5 at
539
FORMULA pushed at 563
edit cell range OP FORMULA 3,4 x 4,5 at
638
FORMULA pushed at 645
```

Figure 22. An example fragment of a journal file of interface actions taken as part of the second round testing procedures.

7.2. Further testing – results

Results of testing the revised prototype were positive. Each modification contributed to making the revised prototype a good deal easier for users to pick up and use than the original prototype.

Some of the redefined features proved more useful to users than

others, with the point-and-click specification of constraints being by far the most widely used new feature of the system. This is not surprising – given the reliance of users on the constraint mechanism during the prototype testing, and the fact that users are required to click somewhere during the task of defining an equality constraint. However, watching individuals point to and specify equality constraints in the point-and-click manner versus the fill-in-a-form technique used previously made it obvious that users preferred the new method. The new technique did create confusion in one area – when specifying a dynamic constraint, in which one or more of the row and column numbers are computed and specified via arguments to the equality constraint range, users are still required to click somewhere and specify a row and column as in a fixed reference, which must then be edited out of the resulting dialog box and left blank. This caused some consternation – users understand "clicking over here when I want that cell to have the same value as this one," but when arguments were to be computed from other cells, specifying those same arguments via point-and-click made little sense. In retrospect, an interaction technique set up to require a click from the user when and only when there are less than two argument cells to the range might make more sense.

Naming of cells was used by many subjects in the place of textual comments – names with semantic meaning were assigned to various parts of the worksheet. Surprisingly, those names were almost never actually referred to in formulae. Perhaps because of the ease of the point-and-click method of constraint specification, users instead relied on that mechanism (most often with absolute addresses, sometimes with relative addresses) for copying data values from place to place. Think-aloud data indicate that cell names were important to users in

remembering where to copy values *from*, however.

Explicit management of cell and graphics linkages was not used by as many subjects, although those who did use it did so extensively. There is also evidence that the visual indications, provided as part of the explicit management scheme, helped even those who didn't explicitly make use of it to understand the linkages between graphics view and computational views.

Subjects were split on the usage of complex formulae and implicit naming of argument cells. Some subjects made extensive use of this facility, others none at all. Those who did use this feature did so with little difficulty.

As mentioned earlier, one subject had difficulty, not with language conventions and solving the given task, but with the interface conventions used on the Macintosh computer that were different from those of his home system. It has been suggested that one way to insure you are testing the language, and not the interface itself, is to have the tester actually manipulate the environment, with the subject dictating to the tester the language elements he or she wants to use [47]. In this manner issues of language design can be separated from interface design, which can be tested separately or after the language design issues have been cleaned up.

My original conception, born of the forward walkthrough process, was to try to support two problem solving paradigms as well as possible – working forwards, from computation to image, and working backwards, first defining the image then the constraints imposed on that image by the data. To

this end, the design of DataSheets combines an extended spreadsheet mechanism with an object-based drawing tools one might find in a drawing package.

To determine whether or not users did indeed make use of the multi-paradigm nature of DataSheets, the journal files were examined for each session. As the journal files reflect a user's actions at various times within each session, it is possible to characterize usage episodes based on what interface component the user was accessing. In this way, episodes within each session were characterized as to whether or not the user was exhibiting *computational, drawing*, or *linking* behavior at that time. The actions that were classified as indicative of each of these types of behavior are identified in Table 12.

Table 12. Interface components that were classified as indicative of computation, drawing, or linking behavior.

Computation:	Editing Cell Values Cell Operation Palette Buttons (+, -, *, etc.)
Linking:	Cell Attribute Palette Buttons (X_Pos, Y_Pos, Size, Shape, etc.) Drawing ByRow Palette Button Drawing ByCol Palette Button
Drawing:	Line/Dots/PolyLine Tools Symbol Size Menu Symbol Shape Menu Symbol Spacing Menu Symbol Color Menu Symbol Fill Menu
Neutral:	Arrow/Hand Tools Window open/resize Cut/Copy/Paste Menu Items Font/Visibility Menu Items

Table 13 shows the numbers of each class of episodes for each of the three categories. Journal data was lost for Subject 1, Task 2 due to a program error. The most interesting thing to note is that, while the numbers show a good deal of computing and linking behavior on the part of all users on all tasks, most users showed comparatively little interest in "drawing" parts of the solution as the original analysis suggested.

Table 13. Number of each class of episode per subject and task. Numbers for each subject-task pair are (computation/linking/drawing) behavior, in that order.

	Task 1	Task 2	Task 3
Subject 1	3/4/3	na	11/11/0
Subject 2	/3/1	1/1/1	12/11/1
Subject 3	2/2/4	2/2/0	6/6/1
Subject 4	3/3/1	2/3/0	9/8/2
Subject 5	2/2/1	2/2/1	11/10/1
Subject 6	3/4/0	3/1/2	8/6/2

Why is this so? In retrospect, the tasks may have been too constrained, and the graphics environment may not have provided exactly the right support to make drawing parts of a solution workable. The tasks were constrained in that even the Box-and-Whiskers task (one of the original tasks used in the forward walkthrough process) has every significant endpoint of each mark somehow controlled by a value. For most users, it was simply easier to first compute that value, then ask for the necessary mark to be created, than to work in the other direction. In fact, many of the marks in the Box-and-Whiskers case have values that are in common with other marks on the same display – since the value has already been created in association with that other mark it is a relatively simple

exercise to copy the value to a different row or column and ask that a mark be displayed. However, the original analysis was in part based on these tasks – that analysis suggested drawing at least parts of the solution to be a viable strategy.

Gross [26; 27] has noted that computers don't provide the right support for sketching – often important in design processes where the designer deliberately wants to express ambiguity or lack of commitment to parts of the evolving design by purposely leaving things vague. With conventional computer-based drawing environments the tendency is to want to produce precise, "clean" drawings because the nature of the computer as a drawing medium is that every measurement has a definite, exact value whether its meaningful or not. In retrospect, a similar phenomenon may have been occurring here. Even though, in the Box-and-Whiskers task, for instance, the X extent of each horizontal line was not intended to mean anything, the tendency was to want a) the horizontal extent of all three horizontal lines to be the same, and b) for them to have "reasonable, round" values. Thus, it made more sense for users to specify an endpoint value as a nice round number (100, say), than to draw a line and let the endpoint come out to "whatever" – especially since there was little support for drawing precisely without specifying endpoint values. Easier "cloning" (i.e. by cut and paste) of marks in which unspecified attributes are kept constant, dragging of marks with constant X and Y positions, and a "grid snap" drawing mode might have made drawing some solution parts a more likely behavior.

If it was merely a case of not supplying the right type of drawing support, however, I would have expected to see more <u>attempts</u> at using the

direct manipulation functionality than was evident. The fact is that it often simply didn't occur to users to attempt to use the direct manipulation functionality to draw parts of the solution.

In addition, the tasks themselves were inadvertently presented in a more procedural style – a presentation style that may have directed users' solutions in ways not anticipated. The task for the Box-and-Whiskers problem, for instance, was presented as text which explained the relevant relations between graphic items in a sequential format – perhaps presenting the task in a less sequential manner (i.e. as a labeled diagram of the relevant pieces) might have made a difference.

Another possibility is that my test subjects were simply too sophisticated, and too used to solving problems computationally – most of my test subjects were graduate students or faculty members from computer science or related departments (you take what you can get). While it is true that it would have been preferable to test DataSheets using more computationally naive users, I don't believe this was a real problem here. The subjects, while fairly sophisticated computer users, weren't familiar with the DataSheets environment itself, so in a sense there were "DataSheets" naive. They were not overly familiar with the tasks presented, so it wasn't a question of having a solution to each of the tasks internalized. Furthermore, since these subjects were used to dealing with computers and strange computational environments, they should have been more willing, not less, to experiment slightly and try more unorthodox ways of doing things, such as using direct manipulation functionality in the middle of programming a solution to some problem given them.

A final possibility is that "paradigm switching" is simply difficult. A paradigm represents a mental scaffolding for solving a problem – a way of organizing one's thoughts and deciding what to do next. Switching from one organizing framework to a different one within the context of solving a specific problem may be too much to expect of users, especially those that are struggling to learn a new programming environment and solve some problem at the same time. The unfortunate reality of conducting user testing on programming environments is that you have a finite amount of time to introduce the language and interface and get a subject to perform some task – perhaps if subjects were given the environment to learn and use for some period ahead of time, and then asked to perform some task, they might be more aware of all the available functionality and be better able to put it to use.

8. Discussion

The central proposition put forth by this dissertation is that a viable process for building a task-oriented, end-user programming environment, in which the basic language paradigm, the functionality the language provides, and the environment in which that language is embedded are all designed to support a particular domain, involves task- and problem-based design in the form of the paradigm-based, forward walkthrough method.

What of the paradigm-based, forward walkthrough as a method for designing programming environments? Use of the paradigm-based forward walkthrough method had several salutary effects on the design of the DataSheets system:

It turned out to be a useful exercise in setting the overall tone of the environment that resulted. While I admit to having some bias towards spreadsheet-based environments in the first place, the forward walkthrough process confirmed my original suspicions that an extended spreadsheet made sense for EDA. The forward walkthrough did point out, however, that many of the tasks within the domain might be better supported by a more direct-manipulation approach – something I had not anticipated prior to the walkthrough process. And, while during testing users tended not to access the direct-manipulation functionality, that direct-manipulation component of the environment may be more useful in the long term than the usability testing results actually indicated.

The walkthrough process pointed to functionality that had not been thought of in the original design, in some places making problem solutions within the overall language design much simpler. The filter operator functionality discussed earlier is a case in point – although some operators similar to filters were originally planned to be part of the language, the actual form of the operator was discovered through the use of the walkthrough. The idea of fully dynamic equality constraints, where the arguments to the constraint are calculated at run time, is another specific piece of functionality that arose from the walkthrough process. Originally, the equality constraint mechanism allowed a user to specify one cell to copy from – i.e. the constraint was fixed. During the walkthrough process, it was noted that several problems resulted in a need to access values based on their order within the overall set of values, a process that was facilitated by providing fully dynamic constraints that could access a specific value based on the total number of data items read in.

While the forward walkthrough process was helpful, it was not completely successful – the walkthrough failed to show that the overly constrained layout in the problem solution would be as big a problem as it was. In retrospect, the walkthrough process failed to adequately capture the dynamics of putting together a solution. The process allowed me, as the designer, the luxury of revisiting a proposed solution over the course of many days – an opportunity an actual user does not have. And while the process is designed to account for this (by asking the walkthrough analyst to "step outside" their own capabilities and examine the process of arriving at a solution from the point of view of the user), it misses much of the dynamics of a truly opportunistic, trial-and-error, backwards and forwards, problem solving session. For instance, what

may be of paramount importance to the idea of usable non-specialist programming solution is the ability to back up and repair errors in solutions when a wrong step is taken. The walkthrough process captures none of this but instead asks what a user would have to know in order to do everything correctly. In fact, users are always going to make mistakes – making errors is a part of the learning process. The walkthrough process really needs to take this into account, and ask, given that a user makes a mistake here, what knowledge will be required for successful repair of the problem solution?

The original walkthroughs suggested that certain parts of the tasks were better supported by one programming paradigm and that other parts were better supported by other, competing paradigms. I do not believe this was in error, although as already discussed users made less use of the direct-manipulation, drawing functionality than I had anticipated. The tasks themselves may not have been well chosen to reflect that true nature of the EDA process – the fact that they were chosen to begin with, rather than arising as a natural part of exploring the data and its relationships, may have constrained the solutions that users were apt to provide. The way they were presented, which in retrospect was too procedural in nature (rather than declarative) may have influenced the methods users took to solve them. An incomplete set of drawing/direct manipulation interactions (most noticeably, lacking adequate graphical cut and paste facilities) may have made using the direct manipulation facilities harder than it should have been. Finally, it is possible that the subject's themselves were too sophisticated in their use of computational environments for this to be a fair test.

In retrospect, the walkthrough process did not go far enough in suggesting the best way to combine these different paradigms of computing. Rather than look at two paradigms as competing methods of solving the same problem, and then trying to combine them to provide the best of both worlds, a better approach might be to search for ways of embedding parts of one paradigm within the overall framework of the other. The walkthrough process as defined suggested the combination of two paradigms – it was mute on how best to combine those two paradigms. The method chosen to do this (through essentially two separate modules with linkages between them) may not have provided enough support for users to feel comfortable switching from one method of working to another in mid-task.

A solution might be to continue the walkthrough process from the point reached, by selecting a "dominant paradigm" (in this case, the spreadsheet) and using walkthrough-like analyses to understand where support from other paradigms might fit into that overall framework. It would be critical in such an enterprise to pay particular attention to the user's presumed mindset at the time a particular step in a solution is reached. The walkthrough process, as it stands, tends to regard steps in isolation and ask what the necessary guiding knowledge is for the user to accomplish this step. A better question to ask might be "what is the necessary guiding knowledge, given what the user has accomplished so far and what they have learned from that experience?"

It has been suggested that the walkthrough process is inherently a conservative process, in that there is no defined path for innovation to easily enter into the process and thus make its way into the defined language. I don't

believe this is true, for the following reasons:

The process of studying competing language paradigms from a task centered perspective may suggest ways of combining ideas from one paradigm into others in novel ways. The filter operation is really the result of taking an idea (passing meaningful data back as truth values) from one paradigm (the pseudo-functional language of LISP), and applying to another (the spreadsheet paradigm).

Searching for ways to combine paradigms may produce innovative features. In DataSheets, the result of the walkthrough process was a decision to support two different paradigms within the same environment. While the process did not go far enough to explore the most appropriate way of combining these two methods of solving problems, one of the unique features of the environment as it stands is the linkage between direct manipulation functionality and the spreadsheet functionality. Continuing the walkthrough process may suggest other novel ways of combining features from different language paradigms.

Finding ways of supporting the problem space directly within a paradigm may produce innovative features. In DataSheets, equality constraints can address a linear interpolation of cells by specifying a non-integer number as the address to copy from. This is an innovative feature for a spreadsheet-based language that came directly out of the supporting the problem space.

If there is one maxim that I have learned during the design, implementation, and testing of DataSheets, and other systems prior to it, it is that users will always use systems in new and surprising ways not envisioned by the

designer. This is not a bad thing. The test of a process, such as the paradigm-based forward walkthrough, is not so much whether it predicts every nuance of a user's behavior correctly, but in whether that process 1) makes a difference in the end result of the artifact being designed and 2) results in a better artifact than what might have been produced had the process not been followed. DataSheets is an interesting, innovative system with features in it that are the direct result of the paradigm-based walkthrough process. User testing suggested that features introduced as a result of the walkthrough process resulted in a more useful and usable system.

Within the purview of problem-based design for application environments, there are basically two methods of attack: one is to use the task or problem suite to distill the appropriate functionality for the environment, then provide that functionality at a high enough level that the user never has to do more than click a button or pull down a menu to accomplish his or her task(s). The other is to use the task or problem suite to distill the appropriate functionality for the environment, then provide support and functionality at a low enough level that the user can accomplish their own tasks by combining the lower level functionality built into the environment in the appropriate manner.

The former is just the application-centered view in disguise. We are then taking it upon ourselves as the designers of application environments to provide <u>all</u> the functionality a user might ever need – a design task that is not only daunting but which experience has proven to be almost impossible to do. Most commercial applications start out providing a certain core functionality, then expand to provide more and different functions as the user community

demands more and more that the original designers never anticipated. There's even a name for such a phenomena in popular computing literature – "creeping feature-itis."

The second method really doesn't preclude the first, because once the correct lower level functionality, i.e. the right building blocks, are discovered, designers can always provide pre-built assemblages of such components that provide those more commonly used higher level functions within the domain. In other words, for EDA, we don't have to provide a Box-and-Whiskers button – but we can provide a pre-built Box-and-Whiskers plot once the right building blocks are distilled. The abstraction mechanism built in to the environment makes this, in principle, a reasonable thing to do. One weakness of the current setup is that there is no way to link the graphics view of one worksheet to the computational view of another. It would be useful, for instance, to be able to define a Box-and-Whiskers worksheet, which when called from a previous worksheet, not only computes the required values and displays the appropriate plot, it does so in the graphics view of the calling worksheet. At present this is not possible.

The hope is that discovering the right building blocks will allow users to build things above and beyond the demands of the original tasks. And there is some evidence that this is the case with environments like DataSheets: I close with the two interactive displays introduced in the very beginning of this dissertation that were not part of the task suite used for design: an interactive scatterplot (Figure 23) and an interactive line chart (Figure 24).

The interactive scatterplot in Figure 23 allows a user to dynamically

choose a cutoff value for determining what points in a data sample are considered outliers, and should thus be treated differently than the rest of the sample. A simple scatterplot of a set of data is built up, by reading in pairs of observed and predicted values from a data file. The data are plotted as a scatterplot by assigning the observed values as the X coordinates of some set of marks, and the predicted values as the Y coordinates of the same set of marks. The size and symbol-shape, and color of each mark are set arbitrarily to some constant value.

This defines a simple, standard scatterplot of the data. To turn it into a interactive plot, the value of Abs(observed-predicted) is computed for each pair of values. If that value proves larger than some arbitrary cutoff value (tested using a filter operation on the worksheet), the fill attribute of the resulting mark is set to 1 (indicating filled), if not, the fill attribute is set to 0 (indicating unfilled). If all pairs of data are tested in the same way, we end up with a simple scatterplot wherein data pairs with a difference greater than some value are displayed differently on the graph. If we make sure the number tested against is the same for each data pair, by, for instance, dedicating a cell on the worksheet to hold this value and then copying it into the computation for each data pair by an equality constraint, we can easily change that cutoff value and note the resulting changes in the display.

To make the display even more dynamic, we can create a new line on the graphics view, constrain it to be vertical by constraining both X coordinates of the endpoints to be equal, and make the cell containing the cutoff value above equal to the difference between the two Y coordinates of the endpoint of this line.

Now, by simply moving one end of the vertical line relative to the other, the cutoff value is changed, computations are updated indicating what is and isn't an outlier given the new cutoff value, and the display is updated, in real time as the user manipulates the vertical line.

It is interesting that the development of this "mini-application" proceeded directly along the lines described in the paragraphs above. In other words, I did not set out to build an "interactive scatterplot" – I set out to build a scatterplot. As a result of that exercise, I saw that it was possible to compute the differences between data pairs and use that to display outliers differently. I then noticed I could make all the computations depend upon a specific cutoff value. I finally noted that I could easily make that cutoff value depend upon some graphical mark I manipulated on the display itself, creating the final application.

This process of building some data display, modifying it in response to a question that arises (what is an appropriate cutoff value?), then further modifying the display when the process of answering that question evolves into a useful technique for data analysis in general, is precisely the scenario I would envision for users of a task-oriented language – the process of developing programs and solving problems becomes one and the same.

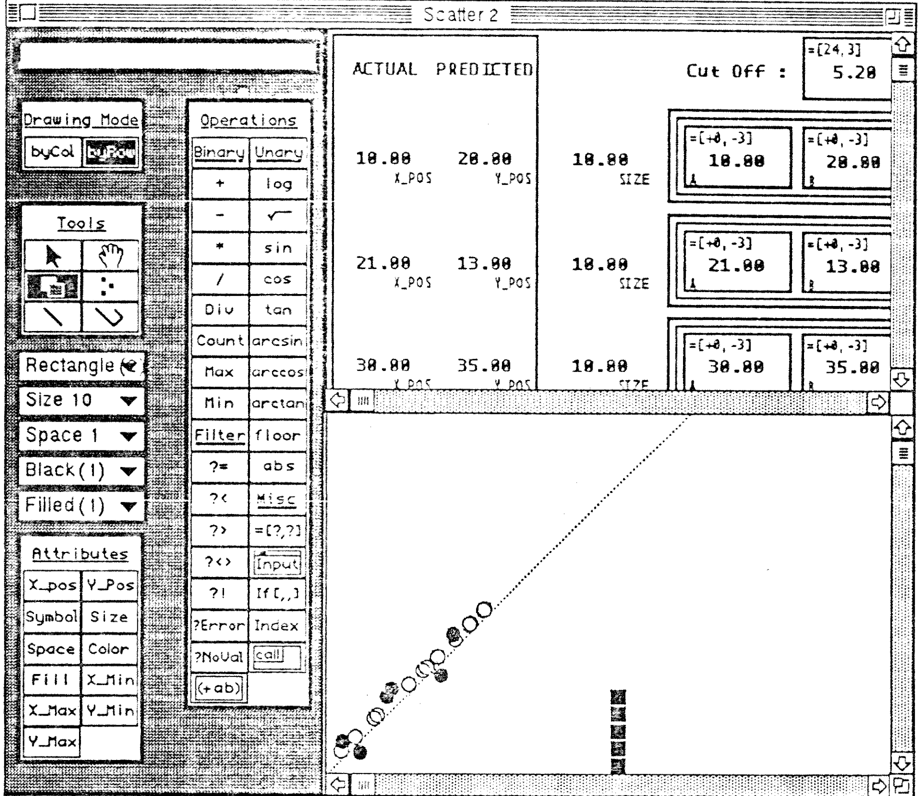


Figure 23. An interactive scatterplot built in the DataSheets environment.

The interactive line chart in Figure 24 represents a different development scenario. In this case, the display and interaction were developed in direct response to a problem noted prior to developing the whole DataSheets system – the problem of graphical misperception when presented with two closely following curves described earlier (see Figure 5).

In this case, two "line charts" are plotted on the same display – a series of data points are read in from a file, and each data point is assigned as the Y coordinate of some mark. An starting X coordinate is arbitrarily assigned, and each succeeding X and Y coordinate for the same mark pair one value from the data file with the previous X coordinate incremented by a constant amount. Two line

charts are easily built up in this manner.

The original conception was to simply build a "virtual ruler" – a mark on the display that is constrained to be vertical (to insure the Y coordinate difference is actually what is computed), and that computes the difference in Y coordinates between its two endpoints. It was envisioned that a user would simply manipulate this virtual ruler by physically moving each endpoint to lie on the corresponding plotted line, then read the difference out as it is computed on the computational view. When this was completed, however, it was noted that the Y coordinates of each end of the line could actually be set automatically – using the X coordinate of the virtual ruler, it is possible to index in to the row of data values representing the Y coordinates of each line plot using a dynamically specified equality constraint, then using the value thus found to set the Y coordinates of the virtual ruler itself. In this manner we can build up a tool that follows both lines of the two data plots, directly computes the difference between the two, and displays it on the computational view, again in real time as the user manipulates the tool.

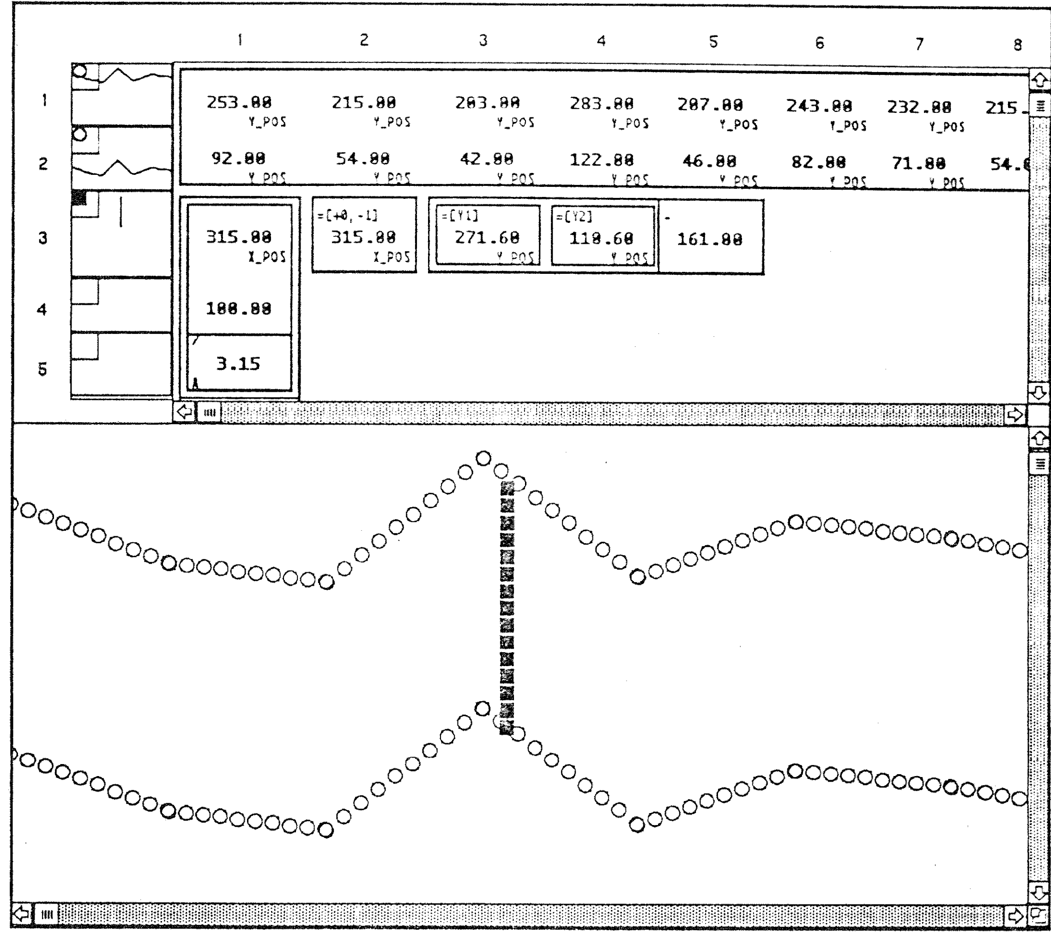


Figure 24. An interactive line chart built in the DataSheets environment.

9. Conclusions

In conclusion, there are three themes that I hope I have communicated to the reader of this dissertation: a vision for what end-user computing might look like in the future, a process for trying to attain that vision, and a particular system, DataSheets, which embodies that vision and was built using the process described.

The vision is of *task-oriented programming environments* – environments that combine some of the domain-specific functionality inherent in applications with the flexibility of some programming medium, hopefully aimed at end-users rather than at professional support personnel. Designing and implementing DataSheets has convinced me that this vision is fruitful one – that combining application functionality with programming language concepts (in this case, spreadsheet concepts) can produce an environment in which artifacts can be produced that are beyond the scope of applications alone, without requiring the user to learn a general purpose programming language. The interactive scatterplot and line chart described here are two examples of artifacts that are simple to produce in environments like DataSheets that would be difficult to produce using a general purpose language without any special support built in, and that would be beyond the scope of simple applications. I hope to continue research involved in combining application functionality and programming language concepts in a manner that provides more flexible and powerful environments for end-users, without getting them lost in the morass of learning FORTRAN or LISP.

The central thesis of this dissertation involves a process for designing such task-oriented environments. This process involves the use of *task-* or *problem-based design*, based on the use of paradigm-based and programming walkthroughs, followed by prototype implementation, user testing, and subsequent iterative design and modification of the prototype. This process worked here. The early use of paradigm-based walkthroughs allowed me to investigate the question of the overall tone for the environment – what variety of programming support an environment for EDA might provide. Continued use of paradigm-based and programming walkthroughs were instrumental in discovering the appropriate functionality to include in such an environment, the appropriate level to provide that functionality at, and in some cases what exact form (i.e. in the case of filter operations) that functionality should take. There were some problems with the prototype that the original analysis did not discover – mainly having to do with the overall solution being too constrained in its use of an X-Y grid for two purposes, and with the lack of reasonable support for error correction in the environment as it stands. Furthermore, the environment did not seem to encourage the kind of opportunistic, iterative problem solving in which solutions are partially computed and partially drawn by the user that I had originally anticipated in my analysis. In retrospect, the design process was stopped too early - another round of walkthrough analysis focusing on the integration of the two paradigms of computing supported may have been instructive. Prototype testing, however, and modification of the original design provided an opportunity to at least partially alleviate the difficulties that arose.

Finally, there is the DataSheets environment itself. I make no pretense of believing that this is the "best" possible environment for EDA, or even a truly

usable one as it now stands. At the moment, the speed of the environment (written on Common Lisp on a Macintosh) and bugs inherent in any prototype preclude its use for really "serious" problems. The environment as it stands, however, validates some of the original design decisions in terms of combining a spreadsheet and direct-manipulation functionality in an environment for EDA work. It points out the validity of the task-oriented environment as a vision of end-user computing in the future. The extended spreadsheet ideas, incorporated into this design, proved to be a reasonable method of building a spreadsheet that does away with one of the primary causes of errors in conventional spreadsheets. The graphical view capabilities of the environment, combined with the computational engine of the extended spreadsheet, provide an environment in which interactive "mini-applications" of the type demonstrated here can easily be built.

16. A. Cypher, *Eager: Programming Repetitive Tasks by Example*, in Proceedings CHI '91, Human Factors in Computing Systems, Addison-Wesley, New Orleans, LA, 1991.

17. S.P. Davies, *The Role of Notation and Knowledge Representation in the Determination of Programming Strategy: A Framework for Integrating Models of Programming Behavior*, Cognitive Science, Vol. 15, 1991, pp. 547-572.

18. M. Eisenberg, *Programmable Applications: Interpreter Meets Interface*, MIT Artificial Intelligence Laboratory Technical Report 1325, 1991.

19. M. Eisenberg and G. Fischer, *Programmable Design Environments: Integrating End-User Programming with Domain-Oriented Assistance*, in Proceedings of CHI '94 Conference on Human Factors in Computing Systems, Addison-Wesley, Boston, MA, 1994.

20. G. Fischer and C. Rathke, *Knowledge-Based Spreadsheets*, in Proceedings of AAAI-88 7th National Conference on Artificial Intelligence, St. Paul, MI, 1988.

21. M.A. Fisherkeller, J.H. Friedman and J.W. Tukey, *PRIM-9: An Interactive Multidimensional Data Display and Analysis System*, in W.S. Cleveland (ed.), The Collected Works of John W. Tukey, Graphics: 1965-1985, Wadsworth and Brooks/Cole Advanced Books and Software., Pacific Grove, CA, 1988.

22. G. Furnas, *New Graphical Reasoning Models for Understanding Graphical Interfaces*. in Proceedings CHI '91 Conference on Human Factors in Computing Systems, New Orleans, LA, Addison-Wesley, 1991.

23. E.P. Glinert and S.L. Tanimoto, *Pict: An Interactive Graphical Programming Environment*, IEEE Computer, Vol. 17, No. 11, 1984, pp. 7-25.

24. T.R.G. Green, R.K.E. Bellamy and J.M. Parker, *Parsing and Gnisrap: A Model of Device Use*, in G.M. Olson, S. Sheppard and E. Soloway (eds.), Empirical Studies of Programmers: Second Workshop, Ablex Publishing, Norwood, NJ, 1987, pp. 132-146.

25. T.R.G. Green and M. Petre, *When Visual Programs are Harder to Read than Textual Programs*, in Proceedings of ECCE-6 (Sixth European Conference on Cognitive Ergonomics), 1992.

26. M.D. Gross, *The Fat Pencil, the Cocktail Napkin, and the Slide Library*, Submitted to Acadia, 1994.

27. M.D. Gross, *Stretch-A-Sketch: a dynamic diagrammer*, Submitted to IEEE Symposium on Visual Languages, 1994.

28. C.E. Hughes and J.M. Moshell, *Action Graphics: A Spreadsheet-based language for Animated Simulation*, in T. Ichikawa, E. Jungert and R.R. Korfhage (eds.), Visual Languages and Applications, 1990.

29. J.H. Larkin and H.A. Simon, *Why a Diagram is (Sometimes) Worth Ten Thousand Words*, Cognitive Science, Vol. 11, No. 1987, pp. 65-99.

References

1. A.L. Ambler, *Generalizing the Sheet Language Paradigm*, in T. Ichikawa, E. Jungert and R.R. Korfhage (eds.), Visual Languages and Applications, Plenum Publishing Corporation, 1990.

2. A.L. Ambler, *Visual Forms of Iteration that Preserve Single Assignment*, Journal of Visual Languages and Computing, Vol. 1, 1990, pp. 159-181.

3. J.R. Anderson, R.G. Farrell and R. Sauers, *Learning to Program in LISP*, Cognitive Science, Vol. 8, No. 2, 1984, pp. 87-129.

4. R.A. Becker and J.M. Chambers, *S: An Interactive Environment for Data Analysis and Graphics*, Wadsworth Advanced Books and Software, Belmont, CA, 1984.

5. R.A. Becker and W.S. Cleveland, *Brushing Scatterplots*, in W.S. Cleveland and M.E. McGill (eds.), Dynamic Graphics for Statistics, Wadsworth, Inc., 1988.

6. B. Bell, *Using Programming Walkthroughs to Design a Visual Language*, PhD Dissertation , University of Colorado, Boulder, CO 80309, 1992.

7. B. Bell, W. Citrin, C. Lewis, J. Rieman, R. Weaver, N. Wilde and B. Zorn, *Using the Programming Walkthrough to Aid in Programming Language Design*, Journal of Software Practice and Experience, Vol. 24, No. 1, 1994, pp. 1-25.

8. J. Bertin, *Graphics and Graphic Information Processing*, Walter de Gruyter, Berlin, New York, 1981.

9. J. Bertin, *The Semiology of Graphics*, University of Wisconsin Press, Madison, WI, 1983.

10. A.H. Borning, *ThingLab -- A Constraint-Oriented Simulation Laboratory*, Xerox Palo Alto Research Center Technical Report SSL-79-3, July, 1979.

11. P.S. Brown and J.D. Gould, *An Experimental Study of People Creating Spreadsheets*, ACM Transactions on Office Information Systems, Vol. 5, No. 3, 1987, pp. 258-272.

12. M.M. Burnett and A.L. Ambler, *A Declarative Approach to Event-Handling in Visual Programming Languages*, in Proceedings of the 1992 IEEE Symposium on Visual Languages, IEEE Computer Society Press, 1992.

13. S.M. Casner, *Personal Communication*, 1990.

14. S.M. Casner, *A Task-Analytic Approach to the Automated Design of Graphic Presentations*, ACM Transactions on Graphics, Vol. 10, No. 2, 1990.

15. W.S. Cleveland, *The Elements of Graphing Data*, Wadsworth Advanced Books and Software, Monterey, California, 1985.

30. C.H. Lewis, *Using the 'Thinking-Aloud' Method in Cognitive Interface Design*, Technical Report RC 9265, 1982.

31. C.H. Lewis, *Creating Interactive Graphics with Spreadsheet Machinery*, in E.P. Glinert (ed.), Visual Programming Environments, IEEE Computer Society Press, 1990.

32. C.H. Lewis and G.M. Olson, *Can Principles of Cognition Lower the Barriers to Programming?*, in in G.M. Olson, S. Sheppard and E. Soloway (eds.), Empirical Studies of Programmers: Second Workshop, Ablex Publishing, Norwood, NJ, 1987, pp. 248-263.

33. H. Lieberman, *Dominoes and Storyboards: Beyond ``Icons on Strings''*, in Proceedings of the 1992 IEEE Symposium on Visual Languages, IEEE Computer Society Press, 1992.

34. J.D. Mackinlay, *Automatic Design of Graphical Presentations*, PhD Dissertation, Stanford University, 1986.

35. MacWorld, *MacWorld Magazine Best Seller List for December, 1992*, MacWorld, April 1993.

36. D.L. Maulsby, *Metamouse: End-User Programming by Demonstration*, in (eds.), Proceedings CHI '90, Human Factors in Computing Systems, Addison-Wesley, 1991.

37. D.W. McIntyre and E.P. Glinert, *Visual Tools for Generating Iconic Programming Environments*, in Proceedings of the 1992 IEEE Symposium on Visual Languages, IEEE Computer Society Press, 1992.

38. F. Modugno and B.A. Myers, *Pursuit: Visual Programming in a Visual Domain*, Carnegie Mellon University Technical Report CMU-CS-94-109, January, 1994 1994.

39. B.A. Myers, *Taxonomies of Visual Programming and Program Visualization*, Journal of Visual Languages and Computing, Vol. 1, No. 1990, pp. 97-123.

40. B.A. Nardi, *A Small Matter of Programming*, The MIT Press, 1993.

41. P.G. Polson, C. Lewis, J. Rieman and C. Wharton, *Cognitive Walkthroughs: A Method for Theory-Based Evaluation of User Interfaces*, International Journal of Man-Machine Studies, Vol. 36, No. 5, 1992, pp. 741-773.

42. R.S. Rist, *Plans in Programming: Definition, Demonstration, and Development*, in E. Soloway and S. Iyengar (eds.), Empirical Studies of Programmers, Ablex Publishing, 1986, pp. 28-47.

43. B. Ronen, M.A. Paller and J.H.C. Lucas, *Spreadsheet Analysis and Design*, Communications of the ACM, Vol. 32, No. 1, Jan 1989, pp. 84-93.

44. R.L. Schaefer and R.B. Anderson, *Student Edition of MINITAB,* Addison-Wesley Publishing Company, Inc, Reading, MA, 1989.

45. E. Soloway, *Learning to Program = Learning to Construct Mechanisms and Explanations,* Communications of the ACM, Vol. 29, No. 9, 1986, pp. 850-858.

46. W. Stuetzle, *Plot Windows,* Journal of the American Statistical Association, Vol. 82, No. 398, 1987,

47. T. Sumner, *Personal Communication,* 1994.

48. I. Sutherland, *Sketchpad, a Man-Machine Graphical Communication System,* Proceedings of the AFIPS Joint Computer Conference, 1963, pp. 329-346.

49. L. Tierney, *LISP-Stat,* John Wiley & Sons, New York, NY, 1990.

50. E.R. Tufte, *The Visual Display of Quantitative Information,* Graphics Press, Cheshire, CT, 1983.

51. E.R. Tufte, *Envisioning Information,* Graphics Press, Cheshire, CT, 1990.

52. J.W. Tukey, *Exploratory Data Analysis,* Addison-Wesley, Reading, Mass., 1977.

53. N.P. Wilde and C.H. Lewis, *Inventing new information representations through task analysis,* University of Colorado Computer Science Technical Report CU-CS-549-91, 1991.

54. N.P. Wilde and C.H. Lewis, *Supporting the Process of Programming in Visual Programming Languages,* Submitted to IEEE Computer Special Issue on Visual Languages, 1994.

55. N.P. Wilde and C.H. Lewis, *Spreadsheet-based interactive graphics: From prototype to tool,* in J.C. Chew and J. Whiteside (eds.), Proceedings of CHI '90 Human Factors in Computing Systems, Addison-Wesley, Seattle, Washington, 1990.

Appendix A: Testing materials

Things you should know about Datasheets: The overall environment

• In Datasheets, you compute in documents called Worksheets. You can call up a blank worksheet using the "New Worksheet" command under the "File" menu.

• Each worksheet has two main panels - the computation-view area, and the graphic-view area. When you open a new worksheet, the window is split roughly into two equal panels with the top half being devoted to computation and the bottom half to graphics. You can specify the split between computation/graphics by using the hand tool off the tools palette to grab the bottom of the computation panel and drag it up/down.

• As you might have guessed from the two panels there are really two ways of working with datasheets - one is to input data and specify computations on that data within the computational view. The operations palette and the edit box (upper left hand corner of the worksheet), both located in the gray area to the left of the main panels, are used for this purpose. More on this later.

• The other way you can work with datasheets is to build displays by drawing them in the graphics area - the tools palette on the left hand side has a selection tool, resize tool, straight-line tool, symbol tool, and a poly-line tool available in it. You can use these tools to draw various marks in the graphics-view area. The pop-up menus below the tools palette control various aspects of the marks you create, such as color, symbol-shape (a line consists of many individual symbols drawn end-to-end), symbol-size, etc.

• Each mark in the graphics view is also associated with a row, or column of data items in the computational view. The correspondence is created automatically when you create a mark, or when you tell datasheets that a particular cell in a row/column controls a particular attribute of its mark. You can tell datasheets which attribute a given cell controls by selecting a cell, then selecting the appropriate attribute of the attributes palette in the lower, left hand corner of the worksheet.

•The linkage between graphics and computation is bidirectional. That means that if you change the data in the computation view in some way the graphics-view will be updated to match. If you change the graphics view the data view will also be updated to match, via the linkages you have specified between cells and mark-attributes.

[Stop and ask for a Demo]

Things you should know about Datasheets: The computational environment

• The computational view in Datasheets is a grid of cells. Cells hold data and take part in computations. You can enter data into a cell or cells by selecting them with the mouse, and typing a value (the value will be echoed in the edit box in the upper left hand corner of the worksheet), then typing return or tab.

• Cells, and the data they contain, take part in computations by virtue of being contained in a cell range. Cell ranges are groups of contiguous cells that are associated with some operation. The bottom, right most cell in the range (the result cell) contains the result of applying the operation to the rest of the cells in the range
(the argument subrange)

• Cell Ranges can be nested, in which case only the result of the inner computation is used in the computation defined by the outer cell range.

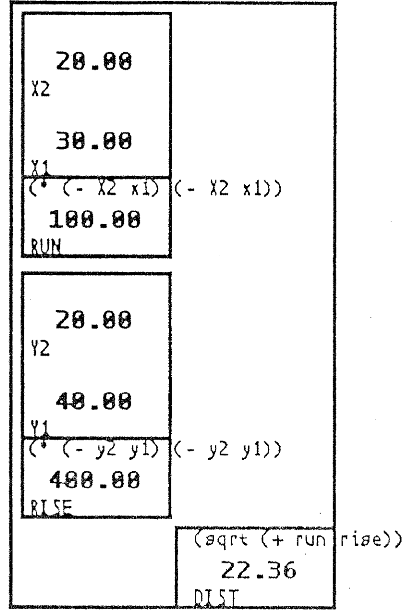
• Cells and Cell Ranges can be named, by selecting the Name Cell menu choice from the Special menu. When selected, the Name Cell menu option brings up a dialog box asking you to specify a name for the selected cell/range. Names for cells are only valid within the range they are contained in - except the name of the
result cell of that range, which is also used as the name of the range as a whole. This policy can be overridden by fully specifying the name of cell from anywhere on the worksheet - i.e. range2.range1.name1 is the cell called name1 that exists within range1, that exists within range2. (See the figure on the next page)

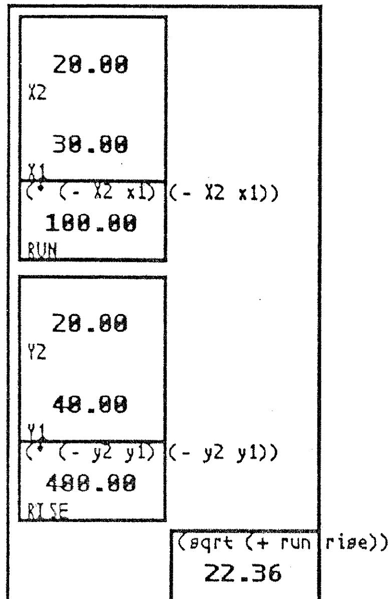
• On each worksheet, on the left hand side, there are a variety of palettes. Ignore all but the operations palette for now. This is how you associate an operation with a range of cells. Select the range of cells using the mouse cursor, then click on the appropriate operation in the palette. The range will be defined by a heavy black border surrounding it, with the result cell identified by having the operation displayed its upper, left hand corner.

• There are several different kinds of operations available to you:

- **Binary** operations aren't really binary, but take as many arguments as you give them, and build up a result by applying the operation to all the argument cells from left to right, top to bottom. Most of them should have self-explanatory names, except perhaps:

Div - performs integer division on all values in its argument subrange.





=[DIST.RISE]	=[DIST.RUN]	/
400.00	100.00	4.00

- **Filter** operations allow tests for certain values. They work by passing the first argument in their subrange through as the result if the test is met by all pairs of cells in the argument subrange, working left to right, top to bottom. If the test is not met filter operations have no value (i.e. a blank cell) in their result cell.

- **Unary** operations only work on one cell at a time. You can supply as many as you like in the argument subrange, but only the first non-blank value will have the operation applied to it and the result passed on as the result of the entire range.

- **Misc** operations: include a grab-bag of operations that don't fit into any other categories:

=[] - allows copying of values from one cell to another. The value of the cell addressed by the =[] operation is copied into the result cell of the subrange. Address operators take 0, 1, or 2 arguments, depending upon what parts of the address are supplied when the operation is specified.

There are 3 types of values you can supply to an address operation:

Fixed address are unsigned, and address the absolute row or column number of their value.
Relative addresses are signed, and address the row or column number relative to the place they are used.
Named addresses simply use the given name of a cell to specify where to copy from.

When you select =[] operation from the palette, the cursor changes to indicate an address is needed. Clicking on a cell specifies that address as the one supplied, and a dialog box comes up allowing you to confirm/change this selection as necessary. The form of the address in the dialog box depends upon the following:

Clicking while holding no keys down specifies an absolute address.
Clicking while holding the option key down specifies a relative address.
Clicking while holding the command key down specifies a named address

Input: Each worksheet can have one input range defined. Data can be read into an input range from a external file by using the "Input DataFile" command on the "File" menu. Data can be read in sorted order, or unsorted order.

If: Usually used in conjunction with one of the filter operations

above. Takes 3 arguments. If the value of the first argument is non-blank, the value of the 2nd argument is returned, otherwise,
the value of the 3rd argument is returned.

Index: Ignore this for now.

Call: Ignore this for now.

- More complex formulae can be specified using an implicit naming scheme. To do this, select a range of cells and click on the item labeled "(+ a b)" on the operations palette. A dialog box will come up. Enter the formula you would like computed, in postfix notation, enclosed in parentheses. The environment will recognize unique symbols within the formula, and assign argument cells the appropriate names to allow the formula to be computed.

- If you have a series of repeated computations that have to be applied to different sets of data you may find it conveniant to build up the series of computations in one column or row, then duplicate it in other areas of the worksheet. This can be done by using the standard Macintosh Cut, Copy , and Paste operations. These work pretty much the way standard cut, copy and paste operations work in other Macintosh applications, with one caveat: you only cut or copy what is visible on the screen when the operation is invoked. You can control what is visible on the screen through the use of the visibilities dialog available through the visibilities command of the format menu in Datasheets. This means, for instance, you can copy a series of data values without copying the computations that produced them, by making cells ranges invisible and leaving cell values visible. The reverse is also possible - copying cell ranges without the values in them.

[Stop and ask for a Demo]

Things you should know about Datasheets: The graphical environment

• Graphics-views are a way of creating pictures of the data you're working with.

• The graphics-view area is in the bottom right hand corner of new worksheets. It can be resized and moved using the hand tool on the tools palette at the left hand of each worksheet.

• Pictures in Datasheets consist of graphical marks, each of which has one or more
attributes. The attributes of each mark that you can control include the shape of the individual symbols making up that mark, the size of those symbols, the spacing, their color, whether or not each element is filled, and the x and y coordinates of the mark.

• There are two ways of creating marks: You can draw them, much as you would in
a MacDraw-like enronment, using the tools available on the tools palette. There are 6 tools available,

- the arrow tool selects marks already created

- the resize tool allows resizing of marks already created.

- the line tool allows creating of lines.

- the hand tool allows resizing of the split between graphics and computation views

- the dots tool allows creating of single symbols, as opposed to lines made up of multiple symbols

- the polyline tool allows the creating of multiple-segment lines.

• When drawing marks in this manner, you can control the attributes of the marks made via the pop-up menus located immediately below the tools palette.

• Each mark is also associated with a row, or column of underlying spreadsheet. The correspondence is created automatically when you create a mark, or when you tell datasheets that a particular cell in a row/column controls a particular attribute of its mark. You can tell which row or column is associated with which mark by the set of
"icons" that mimic the behavior of one mark in the graphics view, associated with each of the rows/columns of the spreadsheet. Selecting one of these icons and dragging from row to row or column to column changes the correspondence of the given mark to the specified row/column.

Things you should know about Datasheets: The graphical environment (cont)

You can tell datasheets which attribute a given cell controls by selecting a cell, then selecting the appropriate attribute of the attributes palette in the lower, left hand corner of the worksheet.

- You can select whether rows or columns of the spreadsheet are associated with marks via the Drawing Mode palette in the upper left hand corner of the worksheet.

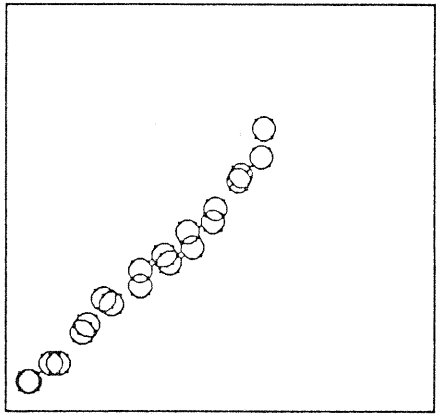
- If no mark has been created for a particular row/column, and you assign attributes to the cells in the column, a mark will be created for you. This is the second way you can create marks on the graphics-view area of a worksheet.

- You can tell which mark is associated with which area of the spreadsheet by selecting. If you select a particular mark using the arrow tool, the first cell of the corresponding row or column of the spreadsheet will be highlighted. The reverse is also true: if you select a particular cell, the corresponding mark on the graphics-view area will be highlighted.

[Stop and ask for a Demo]

Task 1: Defining a simple scatterplot.

Imagine you're a research meteorologist, working on a new method of predicting Boulder maximum daily temperature. You have stored in a datafile the actual and predicted temperatures for 20 days in Boulder. You'd like to get a feeling for how well your new method is working. Use Datasheets to display a simple scatterplot of the data, with actual temperature plotted on the x-axis vs. predicted temperature plotted on the y axis. Your finished plot should look something like this:



A simple scatterplot

A few things to remember: you can examine the datafile by choosing the Open Datafile command from the File menu within DataSheets. You can define an input data range on your Datasheets worksheet using the input operation on the operations palette. Once an input area has been defined, you read data into that area using the Input Datafile command from the File menu.

Task 2: Modifying the scatterplot.

After looking at your meteorological data, you decide you are really only interested in points where the absolute value of the difference between the actual and the forecast temperature is greater than some value (say, 4 degrees). Modify your scatterplot so the points that represent a difference of greater than 4 degrees are plotted in a different color (say, red).

Task 3: Defining a Box-and-Whiskers plot

A colleague asks for you help in examining some wind observations taken from downtown Boulder. The data consists of 10 values, each of which represents the maximum wind speed recorded on one day from downtown.

Your first task is to get a "feel" for the data as a whole. You realize that one of the best ways of visually classifying a sample population such as this is to draw a "Box-and-Whiskers" plot of the sample data. A Box-and-Whiskers plot has the following characteristic (see illustration on next page):

• A horizontal line is drawn at a height whichrepresents the median value of the sample. The median is defined as the value where exactly half the samples values are below it, and half are above it.

• Two more horizontal line are drawn at heights representing the upper and lower quartiles of the sample. The upper quartile represents the value where exactly 1/4 of the samples values are greater than it, and 3/4 are lesser than it. For the lower quartile, the numbers are reversed: 3/4 of the values are greater than the lower quartile, and 1/4 are lesser than it.

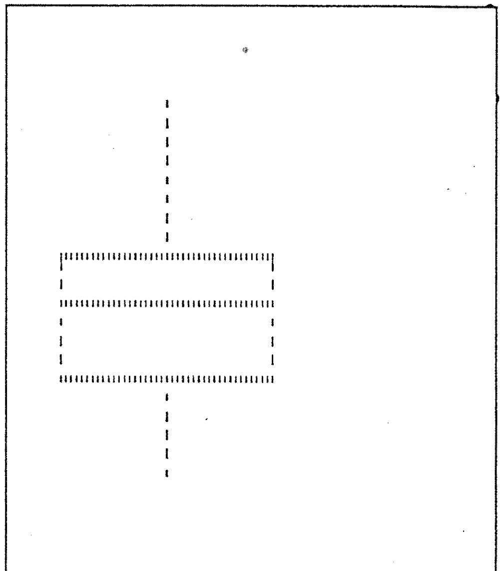
• Vertical lines are drawn from each end of the lower quartile line to the same respective end of the upper quartile line, forming the "box".

• Vertical lines are drawn from the midpoint of the upper and lower quartile lines to
a height which represents the maximum and minimum values found in the sample, respectively, forming the "whiskers".

(continued on next page)

Task 3: Defining a Box-and-Whiskers plot (continued)

You decide to define a Box-and-Whiskers plot using your colleagues data. Your finished plot should look something like this:



A few things to remember:

- You can define an input range, then read the data into the input range in sorted order using the "Input Datafile, sorted" command from the file menu.

- Addressing operations work not only on fixed cell values but on fraction values as well. Thus, if you have data in rows 2 and 3 of a particular column, and you ask for the value in row 2.5 of that column, the values from columns 2 and 3 will be interpolated to synthesize a row 2.5 value for you.