Environments for Decision Support

Steven H. Gutfreund

COINS Technical Report 84-10

June 11, 1984

Preparation of this paper was supported in part under a grant given to Professor David D. McDonald, National Science Foundation Grant IST-810-4984.

UNIVERSITY OF MASSACHUSETTS
DEPARTMENT OF COMPUTER AND INFORMATION SCIENCES (COINS)
Graduate Research Center
Amherst, Massachusetts 01003

Environments for Decision Support

Steven H. Gutfreund
June 11, 1984

A paper submitted in partial fulfillment of the
requirements for the degree of
MASTER OF SCIENCE

Abstract

White collar workers have been helped immensely by the introduction of computers and office automation equipment. However, most of the systems that have been introduced only deal with the clerical aspects of office oriented jobs: meeting scheduling, letter writing, and book-keeping. Considerably less work has been aimed at assisting people at doing the central portion of their job: that of decision making. In this paper we look at the challenges that system designers face when trying to build systems that do more than solve computational problems and at how our language constructs and programming tools may need to change to meet these challenges.

1. DSS: A CHALLENGE

Today it is a very popular thing to talk about "profession based workstations". People use this term to highlight the fact that they desire something much more than commercially available word processors. A profession based workstation is a system for higher level staff: executives, administrators, financial analysts, treasurers, plant managers, and chief executive officers.

The first part of the term "professional based workstation" is "professional". The word "professional" is used to highlight the fact that the users are professional people, not secretaries or clerks. Unlike clerical workers, professionals are not "plug-compatible" people, they cannot be acquired and trained in wholesale fashion nor given generic tools.

Professionals have specialized skills and training and thus need specialized tools. Not only do professionals need more customized tools than clerical workers, they need more flexible tools. The jobs of technicians and clerks tend to be fairly well defined and delineated. Not so with professional workers. They are expected to deal with, and decide on, a wide range of problems, many of which may be unplanned. Their tools must be adaptable enough to deal with these unplanned situations.

The second part of the term "profession based workstation" is workstation. The workstation metaphor usually implies a powerful CPU, Bit-map terminal, hard disk, peripherals and local network connection. But more is meant by this term than mere hardware. The workstation metaphor also implies an integrated electronic environment for integrating, processing and analyzing intellectual matter.

For a professional, the workstation is meant to become the central tool in his arsenal. It will be used for collecting new communications, messages, and data, then modeling and analyzing those facts, and finally hypothesizing courses of action. It will be the primary vehicle for issuing the orders to execute actions, and for maintaining supervision of the course of current projects. Most of all, a workstation is an "electronic" integrator, able to centrally store and cross-reference related facts and task related data with the computer's renowned speed and efficiency.

This, then is the *vision* of the profession based system. Its appeal comes not only from the potential profit from selling more expensive and sophisticated machinery, but also from the fact that almost every white collar worker can see something in it that would be a great boon in his job.

The problem with this vision is that it comes from the marketing department. It describes

all the wonderful things that the system is going to do for you, but little of what it contains. The result has been systems which are called "professional workstations" that consist of little more than souped up clerical tools: ledgers, visicalcs, data-base managers, meeting schedulers, calendars, and electronic mail systems.

Where did the vision fail? I would maintain that the fault does not lie with the marketing department. Afterall, they have defined and characterized for us what our goal should be, and given us a useful framework for exploration. Instead, I maintain the responsibility for realizing this vision now rests with engineers and scientists to supply the technological substance to what is now only a dream.

The reason we computer professionals have had difficulty in implementing the professional workstation vision is our limited model of the function of a computer. Traditionally, we tend to view a computer as a finite state automata for slavishly carrying out boring and repetitive tasks. [32] This leads us to focus on tasks which can easily be regularized and formalized. As long as engineers are fascinated with the computational aspects of computers, we will tend to be limited to systems for solving clerical tasks: spelling checkers, desk calculators, and word processors.

The reason for this tunnel vision on the part of computer scientists is easy to discover. Programmers produce code that has to run on finite state automata. Because of the "literalness" with which the computer interprets its instructions, the programmer has learned to be exceedingly formal and precise. We find this predilection to formal and precise problem solving seeping into the designs of user interfaces and tools. Instead of trying to build extensible or adaptive environments for problem solving, computer scientists develop AI formalisms that attempt to structure the users problem solving approach.

Unfortunately, professionals do not work in a problem solving domain that lends itself to easy formalization. Afterall, in any well organized firm, the first step an administrator takes to achieve greater efficiency is to divide labor and responsibilities into small, well-defined, and formalized clerical tasks. These tasks are then relegated to the clerical staff, and the professionals are free to deal with less easily defined problems. Therefore, professionals have little interest in the formalized and structured problems that many computer scientists are tackling - they have already been freed of those responsibilities. It is not very likely that any "professional workstation" composed of even the fanciest clerical tools is going to make a professional want to re-assume responsibility for basically clerical office tasks.

There is a new view of computer that is gaining greater recognition: that of a medium for

modeling and representing the real world. When one approaches the design of computer tools with this model in mind, one does not solely construct trip planners and meeting schedulers. Instead, one looks to build such things as physics microworlds [29], electronic circuit and musical instrument design kits [16], and reactive maps [15].

When one looks to provide support environments for decision makers one does not concentrate on linear programming tools and inventory control programs but on providing:

Military planners: a reactive war simulation and tactics planning environment

Stock brokers: an investment microworld for testing investment tactics

Architects: structure and building design kits

We define a *Decision Support System*[1] as a computer system which is designed to provide an environment for modeling and analyzing the real world. Because of its different focus, it will lead to a different environment and context for problem solving than that arrived at by viewing the computer as an automaton for computational tasks.

The goals of decision support are:

- Assisting decision makers in exploring decision alternatives (what-if? questions).
- Training and simulation
- Helping professionals better understand the real world constraints and context via interactive modeling.
- Giving executives sharper insights into the underlying mechanisms that drive the model (and hopefully also the real world).
- Better analysis of data through instrumentation and monitoring of models that would not be possible in the real world.
- Centralized, integrated storage for better cross-referencing, comprehensive modeling, and strategic analysis.

As can be seen, the overwhelming thrust of Decision Support Systems (DSS) is a system that

[1] Sadly, this term is also becoming a marketing buzzword (it tells what the system does for you but not its technical consistency). However this paper is an attempt to flesh out the term by providing a technical basis for this term.

works symbionically with the professional to make him a more effective and conscientious decision maker. The mechanisms employed by DSS are mostly modeling, simulation and analytic tools. It is in this framework that the marketing department's profession based workstation dream can best be achieved.

Having now sketched out the goals and challenge of the DSS we need to describe what will constitute its substance. Our first step will be to give a more technically rigorous definition of DSS than a mere elaboration of goals.

2. DSS: A DEFINITION

Our definition of DSS is based on a framework developed by Peter Keen [18].

	Management Activity			
Type of Decisional Task	**Operational Control**	**Management Control**	**Strategic Planning**	**Support Needed**
Structured	Inventory Reordering	Linear Programming for Manufacturing	Plant Location	Clerical, EDP or MS Models
Semi-Structured	Bond Trading	Setting Budgets for Advertising	Capital Acquistion Anal.	DSS
Unstructured	Selecting a Cover for TIME magazine	Hiring a Manager	R & D Budgeting	Human Intuition

Figure 1. A Framework for Decision Support Systems

A *structured* task is a task where the algorithm has been formalized to a point where it can be immediately implemented with current EDP and MS tools. An example of this type of task would be a payroll system. The requirements and means for implementating a payroll system are so well understood that one could conceivably write a generator program that would produce one on demand. A good test for whether a task is highly structured would be if one can formalize the requirements and outputs of the program in a denotation formal enough so that one could actually prove that the program meets the requirements.

Unstructured tasks exist at the other end of the spectrum. One could not possibly state

the requirements in a formal enough notation so as to make formal proofs of sufficiency possible. These tasks are usually thought of as being carried out in an individualistic and "artistic" manner.[2] Most unstructured decision making is made by an expert in an area, who by dint of his extensive knowledge in a field, can make decisions on which correct path to take. Keen's examples of designing the cover of *TIME* magazine and deciding which R & D projects to fund are good examples of tasks that are largely unstructured.

There is a definite tendency by some technologists to minimize the extent to which decision tasks are unstructured. Frequently this results in packaged structured systems that purport to be complete decision support systems. When the technologist's system is (justifiably) rejected by the decision makers, he frequently ridicules and rejects the customer's criticisms. Alter presents one technologist's caricature of his client:

> **"I just don't understand that kind of stuff. It simply is not my bag. I'm intuitive. I don't understand equations and that kind of stuff. I just use them when I have to put them in reports. I look at ads and see if they turn me on. I try to get a feel for the market. It's and intuitive thing. In summary I'm an intuitive animal, so don't bother me with facts." [1]**

Keen has done us a great service in describing the true nature of structured/unstructured systems. For, although we technologists understand that great progress comes from formalizing and structuring unstructured tasks, we still need a healthy respect for the complexity and richness of the complete decision and problem analysis process. Understanding the limits of structured tasks and the nature of unstructured tasks is especially important when a decision problem is of a semi-structured nature and a mix of approaches is needed.

Semistructured tasks fall in-between these two extremes. For portions of these tasks there may exist mathematical and formal problem solving techniques. Yet, there is usually some unstructured element of the task that requires expert knowledge on the part of the user. Too much reliance on one's formal tools can lead to disastrous consequences.

To give an example of this we can look at trading in corporate bonds. There exist well known techniques for determining yields on bonds, and linear programming tools are powerful enough to handle the straightforward optimization of yields. Still, one would be foolish to completely rely

[2] There may exist books that present "algorithms" and procedural methods for decision makers to use in solving unstructured problems. However, these algorithms are not formal, involve a fair amount of human intuition, and cannot be implemented with today's software technology.

on these tools, since unstructured aspects of bond investment (devaluations, bankruptcies, and nationalization of corporate assets) could easily wipe out years of investment gains.

Thus, in semistructured tasks, while there may exist many computational decision support tools, the complete analysis of the problem milieu will be dependent on the decision maker's intuitive feel of the problem domain.

Keen has defined Decision Support Systems as systems designed to support problem solving in the semistructured task domain. This definition complements the one given previously. Before we defined DSS in terms of a particular problem solving style. Now we see that there is a specific domain of decision and support problems that requires our particular type of problem solving environment (a environment designed to give one an intuitive feel of the problem milieu through a micro-world simulation). Given our new unified definition of DSS, we need now to take a look at the problem domain.

3. DSS: THE PROBLEM DOMAIN

Decision support systems are supposed to provide an assist to decision makers in making decisions. But what is the nature of these decisions? Some of the questions that one might ask are:

- How do changes in the jet-stream affect California weather?
- Would increases in SO_3 emissions in the Midwest affect acid rain in the Northeast?
- What biological processes are affected by digoxin? THC?
- What advertising media are most effective for specific types of products? age groups?
- If I change the shipment policy how will it affect inventory levels?
- If I change pricing, how does it affect the bottom line?
- What changes can we make to operations to improve cash flow?

The common aspect present in all of these modeling systems is that they are involved in answering questions concerned with the consequences of indirect actions. In dealing with weather systems, small local storms and disturbances can cascade to create macroscopic effects. With digoxin poisoning, it is not the case that digoxin itself is a carcinogen, but rather that it unbalances the production of certain enzymes which lead to cells being much more susceptible to other

carcinogens. The acid rain problem in the Northeast is not a direct result of sulfur emissions by coal burning generators in the Midwest, but arises through a long sequence of chemical reactions and by-products.

In decision making we value most those individuals who are able to estimate the indirect consequences of their actions. The investment manager who can balance opportunities in the stock market versus the gold market, the real estate market, and the rare book market is more valuable than one who has confined his knowledge to stocks alone. The meteorologist who can balance the meteorological effects of ocean thermals, solar activity, and thermal inversion layers, is more valuable than a meteorologist who has no training in oceanography, astronomy, or environmental science.

If what we value most in decision makers is their ability to discover the indirect consequences of actions and balance heterogeneous systems, we need a DSS system that supports these activities. There are three attributes that I consider to be most important in producing this type of decision support environment: cascadability, extensibility, and extemporaneousness.

Cascadability: It should be possible to construct a model (simulation) out of component modules from many different knowledge domains. In assembling a model one imports components from other models, cascading them together to create an integrated model. Thus to assemble a system for studying inventory controls, one would import the output portions of the factory model, create a warehouse model, and import part of the distribution and sales systems. The designer supplies the common sense reasoning needed to interface these tools on the fly.

Extensibility: A DSS system must lead to designs that can be readily extended as need demands. One starts with simple undifferentiated components that are cascaded together as black boxes to form larger structures. But, as needs demand, the boxes become transparent and one embroiders on their internal workings and external interfaces. For example, a biologist may start with a simple Pitts-McCulloch model of a neuron with which he builds his brain structures. As the model develops, the model of the workings and structure of the neuron can be elaborated and improved, and yet the original macro-structures will still hold together as long as the basic interface assumptions still hold.

Extemporaneousness: A decision support environment must not only support dynamic changes in its operation and inference rules, but must make its model (simulation) visible. One needs to see the production bottlenecks forming, cash flow accumulation, and enzyme production imbalances. The best way to assist the decision maker in discovering unsuspected secondary and tertiary

indirect consequences is to animate the operation of the model.[3]

To see how these features would come together in a single system, let us look at our biologist who is studying digoxin poisoning. First he construct a simple model of a cell and its external interfaces. He then cascades cells together to construct simple organs. After adding metering equipment to the simulation he watches the model and tinkers with it extemporaneously. He then goes back, extends, elaborates, and differentiates the cells, making their internal mechanisms richer, yet preserving the basic interface assumptions on which the organs were built. He then goes back and runs the simulation and repeats this cycle.

We now turn our attention to the technology needed to construct a DSS environment. We will begin by making a quick survey of the entire spectrum of methods proposed for DSS construction.

4. DSS: THE TECHNOLOGIES FOR MODELING

There are many technologies that have been used for DSS tools, our survey will examine several major areas where work has been done: Analysis Information Systems, Operations Research, Artificial Intelligence, and exploratory research.

4.1 Analysis Information Systems

An Analysis Information System (AIS) is a system for data analysis, statistical correlation, program management, and financial tracking.[18] In their books Keen and Alter present many examples of AIS systems that have been effectively put to use in corporations. We will look at two such systems, ISSPA [1,18,19,20] and PMS [1,18], and explore what functionality they presented.

4.1.1 *ISSPA.* ISSPA was a simple data presentation system for administrative planning in educational institutions. It had a small set of verb oriented command such as:

List: show data variables

Rank: rank data on one variable

Plot: draw two dimensional charts

Regres: regression analysis

Histo: plot histogram

ANOVA: analysis of variance

[3] How could one put into formal structures algorithms for looking for unexpected tertiary consequences?

NTILES: break up the data into quartiles or deciles

WTILES: break up data into quartiles or deciles by equity

The last two commands were particularly useful in ISSPA's main roles: that of determining school staffing levels based on demographics, and determining the equity spread among projects.

4.1.2 *PMS.* PMS was a stock analysis system implemented at Great Eastern Bank. Like ISSPA, it was a verb oriented system with commands like: **Plot** (a stock), **Plot** (an industry), **Group** (use one attribute to collect related stocks into a group), **Histogram** (a group), and **Scatter** (plot one attribute against another for all securities). **Scatter** and **Group** were especially useful commands in helping analysts discover unexpected correlations in investment strategies.

4.1.3 *Discussion.* The positive features of such systems can be quickly identified: they are quick to learn (approximately one hour for ISSPA according to Keen [18]), they adapt quickly to new problems (the commands in PMS are almost identical to that of ISSPA), and they sometimes uncover unexpected correlations and trends. Unfortunately, one can also quickly point out the weaknesses of these systems. They are not true modeling/simulation systems, and thus testing a strategy (such as an investment strategy where there will be feedback from the model) is not possible. Also they present the end user with a fixed set of commands, which in the case of ISSPA could only be extended by an APL programmer.

4.2 Operations Research Systems

Operations Research covers a wide range of quantitative methods used in business for decision making.

4.2.1 *Network Models.* Network analysis can be used for discovering production bottlenecks or least cost paths of production. Two common applications of Network Analysis are PERT charting [5,13], and CPM [7]. In PERT charting one draws a network diagram such as shown in figure 2, assigns costs to each path, and solves the network to find the cheapest path.

4.2.2 *Linear Programming.* Linear Programming is a technique for maximizing or minimizing a variable which is subject to a set of constraint equations. For example, a manufacturing executive many have a set of equations describing the cost of production at his various plants, the cost of transporting the goods to market, and the number of units of goods required at each market. He can then write a set of linear equations and use Linear Programming techniques to

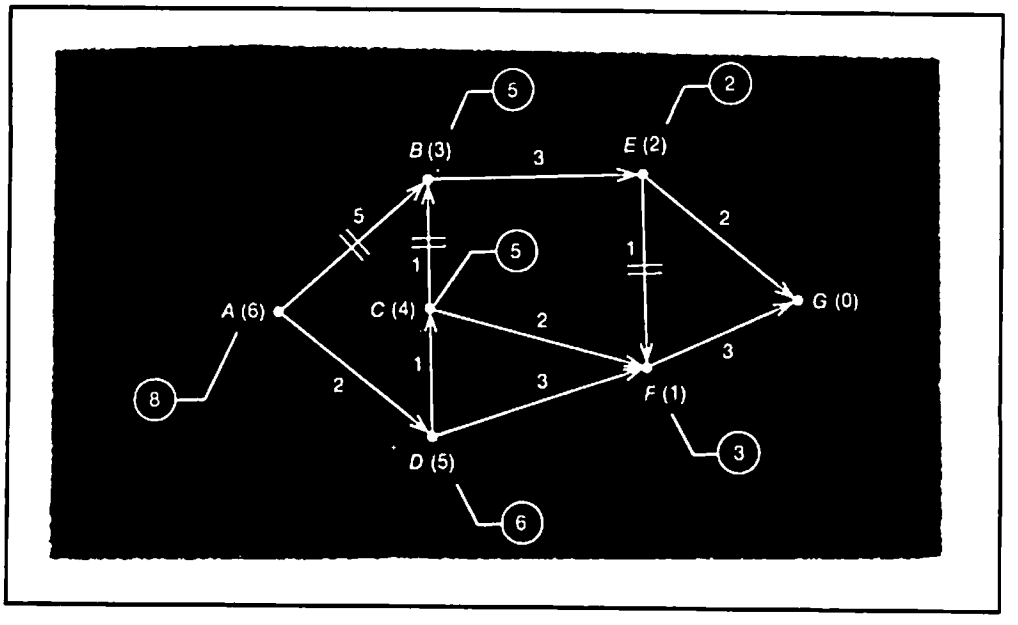


Figure 2. A PERT chart

find the optimal strategy for maximizing his profits. Figure 3 shows a graphical representation of a Linear Programming solution.

There also exist algebraic solution techniques such as Simplex for solving Linear Programming problems that are too difficult to describe pictorially. If one can only accept integer solutions to one's equations, such as when one can only order supplies in units of 50 or 100, then one can use Integer Programming techniques to obtain optimal results. Dynamic Programming can be employed when one has to make a sequence of optimal solutions such as in a multi-year investment plan.

4.2.3 *Statistical Modeling.* Statistical methods use probability distributions, Markov chains and loss matrices to model business activities. Typically these techniques are uses in conjunction with simulation tools to test business strategies. For example, a car rental office may model customer arrivals as a probability distribution, and use Markov chains to describe the probabilities of one, two or three day rentals. With this model one can construct a simulation to test different service policies and determine the resulting profits.

4.2.4 *Discussion.* Operations Research modeling techniques have been widely and successfully applied to many decision making problems. One common reason for their success in their power. Because they are well formalized and mathematically sound, they can give very complete

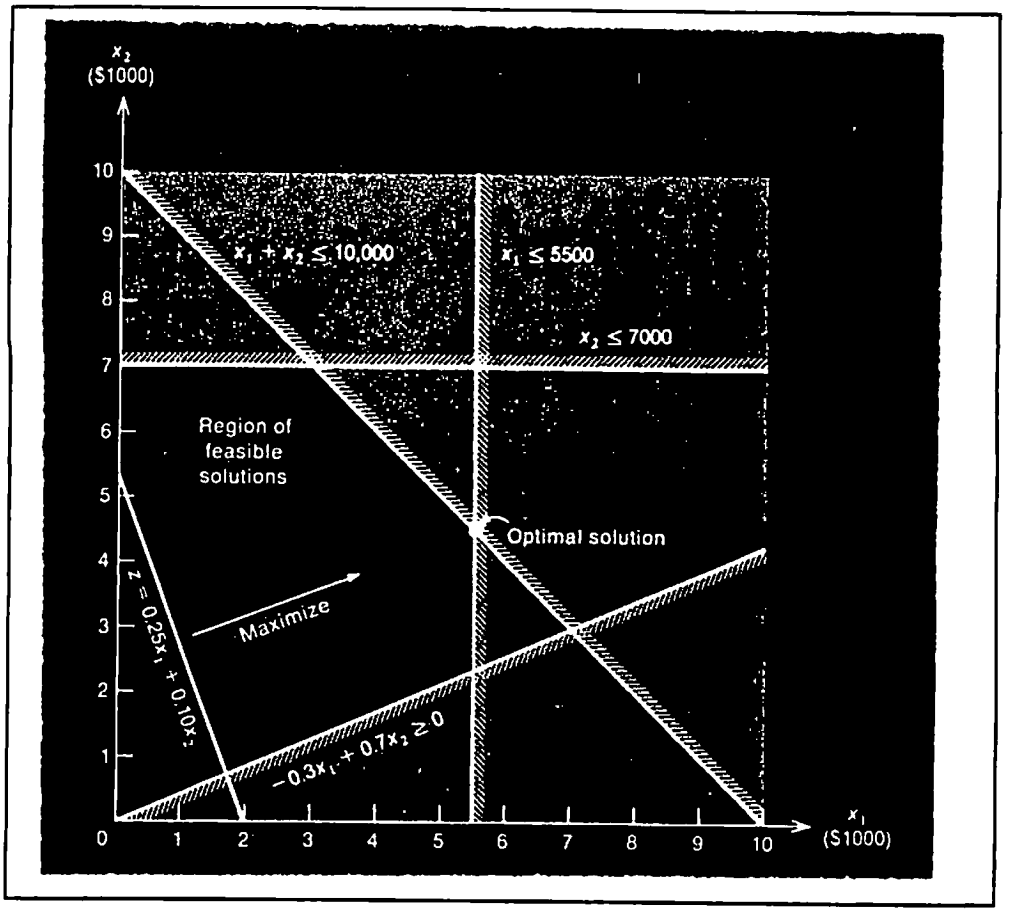


Figure 3. Graphical solution for a financial problem

and sound solutions to problems. Because they provide for good abstraction, they are applicable to a wide variety of decision making domains.

Since Operations Research techniques cover so much ground, and are so widely and effectively used, one must be careful when making general criticisms. Still, one problem that has been observed with these techniques is that they require extensive training in formal analysis so that they can be applied appropriately and their results interpreted correctly.[4]

4.3 AI Modeling Techniques

There have been a wide variety of approaches to AI modeling techniques for decision support.

[4] Alter [1] presents an amusing anecdote about a bank manager who used a linear programming system for depositing overnight balances. One day it indicated that all the money should be invested in English banks, the next day there was a major devaluation of the Pound. The linear programming constraint equation for devaluation was missing, and the result was a major loss. Since DSS problems are inherently semi-structured, application of formal techniques such as linear programming must be done with great care.

A complete survey would require the investigation of everything from Prolog-based inferencing systems [3] to natural language fact retrieval [22]. However, the most flexible and powerful systems seem to be the schema driven production systems, so I will concentrate my investigation on these systems.

IMS [11] is a schema driven system for manufacturing decision making. One describes in short schemas the operation of the machinery in the plant, their resource requirements, and the steps required in the production of goods. One can then query the system to obtain optimal scheduling policies, production bottlenecks, order priorities, resource availability, etc.

POISE [4] is a system for describing bureaucratic (office) procedures. POISE uses schemas to describe preconditions, postconditions, resources, and the logical sequence of operations needed to carry out office tasks. One can query POISE to get information on the status of office tasks, or it can prompt the user about incomplete tasks, or alert the user of quicker methods for carrying out tasks.

Odyssey [10] is a system for handling the billeting (assignment to posts) of military personnel. It contains schemas that describe the education and skills of personal and the personnel needs of the military posts with open billets.

While these systems obviously provide some powerfully attractive functions, there are some noticeable problems with the schema approach.[5] Schemas are usually one-level descriptions techniques, they are not typically composed of layers of increasingly abstract constructs, therefore we lose the important cascadability property that was discussed in the last section.

While there do exists techniques to overcome the cascadability limitation, it is more difficult to overcome the domain dependency of schema driven systems. When a AI modeling system such as IMS is confined to the narrow domain of plant management, the schemas are small and easily extended by a user. However, if we wished to construct an AI modeling system for a meteorologist who studies weather systems and how they affect acid rain, incorporating the meteorological, chemical and biological knowledge into the schemas would make both the schemas and the inference rules unwieldy. Furthermore, since we wish to create extemporaneous programming systems, we need to be able to modify the inference rules of the system itself. Unfortunately, this usually requires a skilled knowledge engineer trained in the intricacies of the

[5] It can be very frustrating to get an AI researcher to admit the limitations in his system. For every limitation there is always so-and-so's system, which supposedly takes care of that problem. Upon close examination it usually turns out that either the system is not even yet in the early exploratory phase, or that the system only addressed "toy" applications and itself has serious limitations.

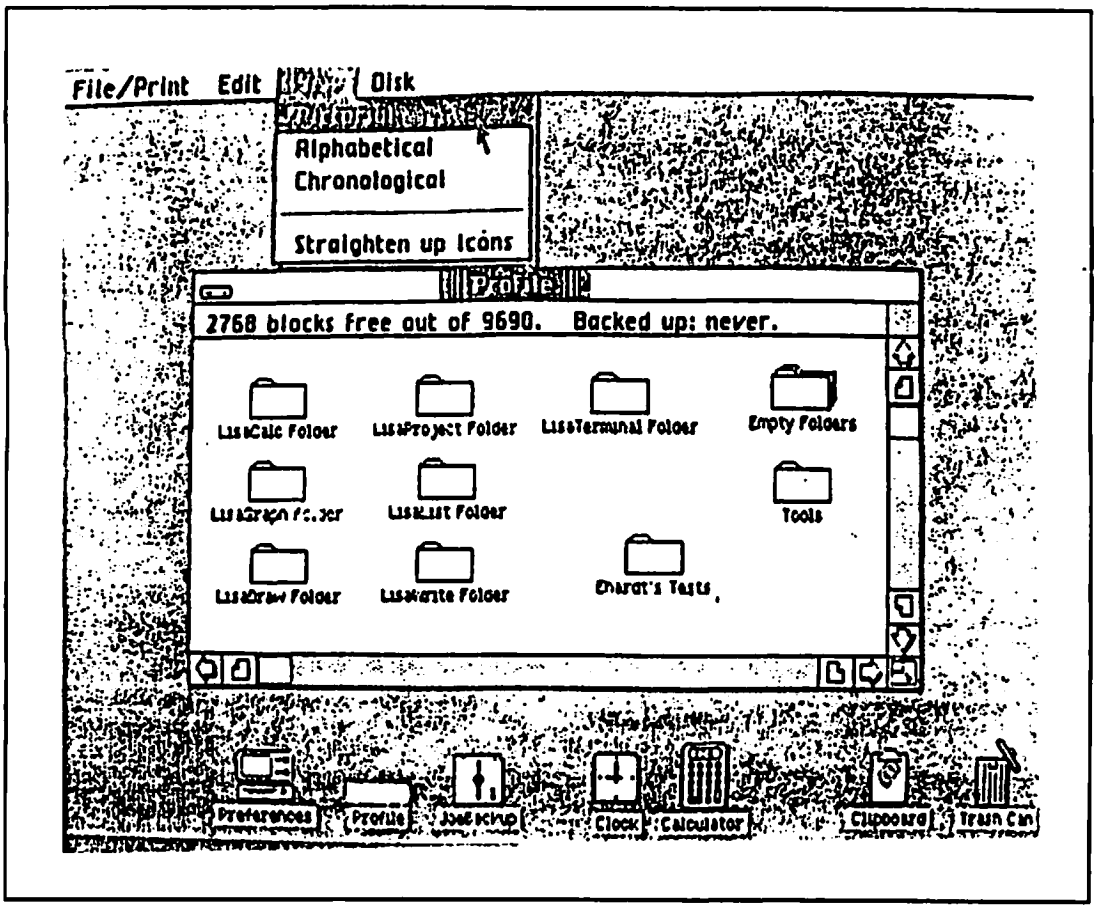


Figure 4. Top Level Lisa Display

production system.

4.4 Exploratory Work

In the next four sections we will examine four systems: Lisa, OBE, PHD, and Smalltalk-80. While these systems may not individually transcend the limitations of the other systems we have examined, they do present interesting constructs which can eventually be incorporated into a true Decision Support System.

5. LISA

Lisa [8] is probably the best example of a class of tools I call "ICON based fixed function tools". This rapidly growing class of systems is best employed in file drawer applications but it has interesting mechanisms for cascading and integrating tools, plus and extremely lucid interface. Lisa has captured a certain kind of nimble interaction style which makes it an appropriate starting point for our study of DSS systems.

5.1 Components

Lisa consists of six major tools: LisaCalc, LisaDraw, LisaGraph, LisaProject, LisaList, and LisaWrite along with interconnection tools for interfacing between these packages. The major tools are named fairly mnemonically: LisaCalc is a Visicalc style spreadsheet calculator, LisaDraw is a picture composition tool, LisaGraph is a pie chart, histogram and graph plotter, LisaProject is a Pert chart and task scheduling tool, LisaList is a list based database tool, and LisaWrite is a word processor. The overall thrust of the major tools is spreadsheet oriented calculation controlled by fixed and pop-up menus of operators.

The role of the interconnection tools is to provide interface between the tools, invocation control, storage management, and most importantly - overall coherence and integration of the environment. Interconnection tools consist of: clipboards for transferring data between the major tools, trash cans for the disposal of old data, folders for the storing of spreadsheets, blank pads for new sources of spreadsheets, and glyphs for invocation control.

5.2 Features

There are three major features of Lisa that are useful for DSS implementers to take note of: the use of structure based tools, a coherent interconnect environment, and graceful context switching.

5.2.1 *Structure based tools.* The six major tools of Lisa are largely structure based. A structure based system utilizes operations and data structures compatible with high level composite structures. Thus LisaWrite operates on pages, paragraphs and figures instead of being line or character oriented as most text editors are. LisaProject operates on task and milestones which are in turn made out of lower level components. Furthermore, Lisa gives these structures dynamically updatable pictorial representations to emphasize the concrete effects of operations. Thus, in LisaWrite if one changes the page format, the page layout will visually change on the screen, if one changes a row in LisaGraph then one sees corresponding changes in the pie or histogram that the row is connected to.

We have previously noted that in DSS environments it is very important to be able to tinker and explore design alternatives. The pictorial structure-based Lisa tools provide a nice environment for this kind of interactive tinkering.

5.2.2 *Coherent interconnect environment.* Perhaps a more important feature is the Lisa interconnect environment. The main goal here was to provide a coherent single way of interconnecting

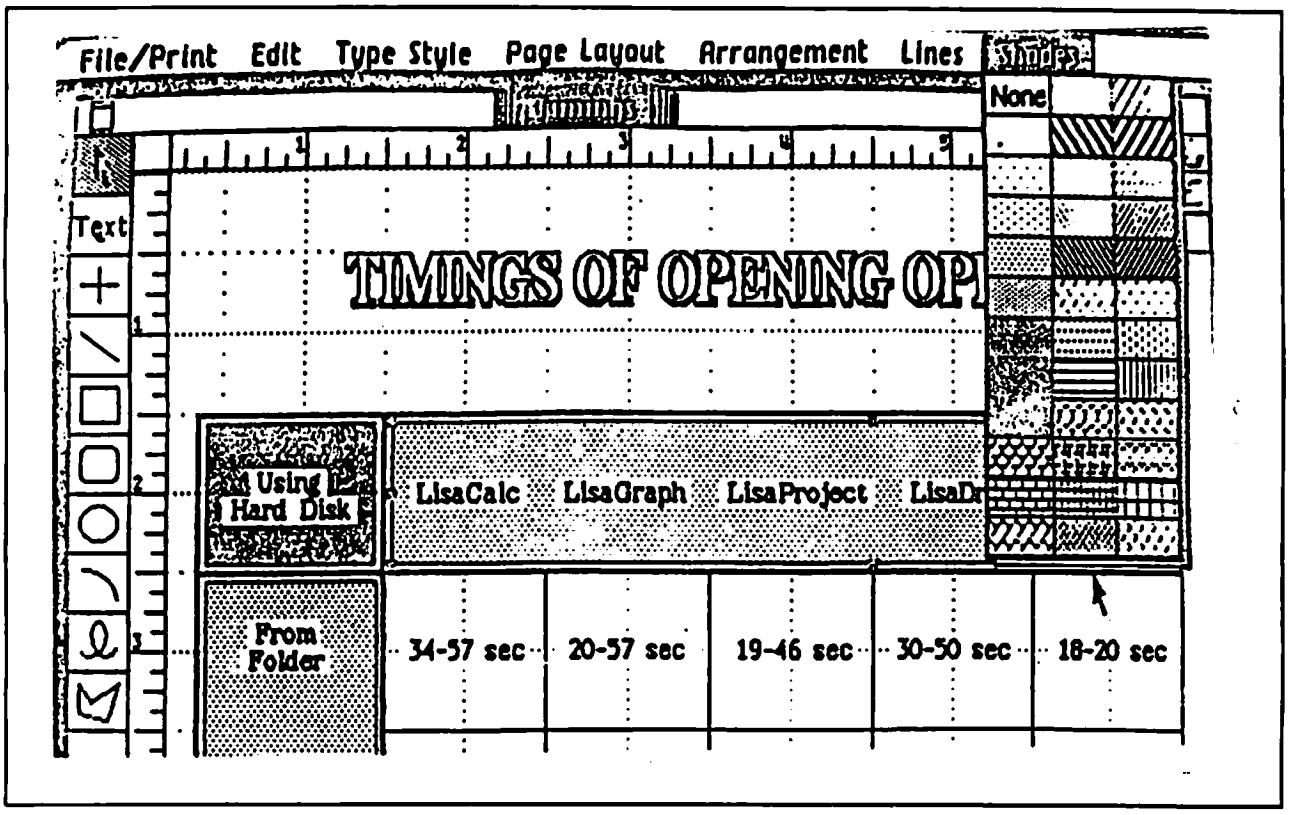


Figure 5. LisaDraw display

the major Lisa tools and controlling invocation and storage management. In conventional systems disjoint utilities (compilers, linkers, backup programs, file manipulation utilities, etc.) are controlled by a command shell. In Lisa there is a more cohesive (and more intuitively simple) tool control environment. Invocation of tools is done by pointing to an Icon of a spreadsheet, storage is done by moving spreadsheets to a folder, deletion is done by moving an object to the trash can icon, copying data is done by cutting out sections from a spreadsheet moving them to a clipboard and then transferring them to another spreadsheet, and backing up is done by moving an object to the floppy disk icon.

Since DSS environments will probably consist of many tools, and since many DSS problems will involve the coordinated working of several tools, powerful interconnection tools are mandatory for DSS environments. Lisa's interconnection tools provide a good starting point for integration and interconnection of tools.

5.2.3 *Context switching.* The ability to shift one's focus of attention from one task to another is very important in DSS environments. Different tools will be providing different perspectives

on the problem. Lisa uses a window mechanism for saving and restoring contexts. Windows provide another way in which Lisa provides concrete physical representations to abstract operations (context save and restore in this case). Lately, attention has focused on windowing systems as the most pleasant way to allow users to gracefully pop and restore contexts.

5.3 Limitations

To the implementers of a DSS environment, understanding the misfeatures and limitations of a system can be as useful as learning what the useful features were. Lisa had three basic limitations: extensibility, interconnectivity, and automatability.

5.3.1 *Extensibility.* The major tools of Lisa are basically canned fixed function tools. This is a definite limitation for many decision support task. Many decision support tasks are either ad-hoc, unpredictable, or content sensitive. The problems with the Lisa tools are that the implementation language is hidden and that operations and data structures are highly specific.

There is no simple way to overcome this limitation. A large part of Lisa's success comes from the fact that its operations and data structures are tightly coupled to the problem domain, and very specific visuals are given to the data structures. For highly structured tasks, such as file drawer systems, word processing and spreadsheet calculating the Lisa approach is fairly satisfactory. Afterall, highly structured tasks by their nature do not require much extensibility. However for DSS tasks that are more unstructured, (representational modeling, etc.) the specificity of the data structures and operators will need to be relaxed to allow more generic (and thus extensible) operations. For example, by fusing the related data types of spreadsheet columns, LisaProject tasks, and character streams into a common ordered data structure class with common operations - one can add accessing methods with fewer worries about specific representation concerns.

5.3.2 *Interconnectivity.* Many tools in Lisa cannot communicate with each other. While data can be moved fairly gracefully from LisaCalc to LisaGraph, no data can be moved from or to LisaList, nor can anything other than non-manipulable bit-map pictures be moved to LisaWrite from the other tools.

This limitation arises from the same source as the extensibility limitation: data structures and operators are highly specific. Easier interconnectivity can only be achieved with adoption of more generic objects, so that specifics dealing with representation and operation can be ignored when dealing across tools. Unfortunately this will only come with the loss of power and intuitive clarity that comes from application specific data structures and operators.

5.3.3 *Automatability.* Currently in Lisa all operations require either direct keyboard or mouse action, there is no facility for automating long sequences of Lisa operations. There are at least two solutions to this problem. One could couple Lisa with a Tinker-like program by example facility, thus allowing both long operation sequences and conditional execution streams to be constructed. Another option would be to develop process and control icons that would specify the data paths and sequence control for the system (the ThinkerToy approach).

6. OBE

QBE [34] presented a major new design strategy for the design of data inquiry and processing systems. Previously data base queries were largely formed as formal mathematical expressions. QBE queries are formed using template filling and pictorial queues. This results in a more humanly tractable and intuitive approach. Yet the QBE methodology has been formally proven to be as powerful as the more obtuse formal query methodology.

Today QBE's successors: OBE [33], SBA [35,6] and Star's record processor [28] have extended the scope of the QBE approach beyond that of simple database queries to that of general office procedures. For the sake of brevity we will concentrate on the analysis of OBE (the most comprehensive and advanced of these systems).

Sales	Salesman	Sales Quota	Sales to Date
TR1(Monthly)	_N	_Q	_S

Conditions
_S > _Q

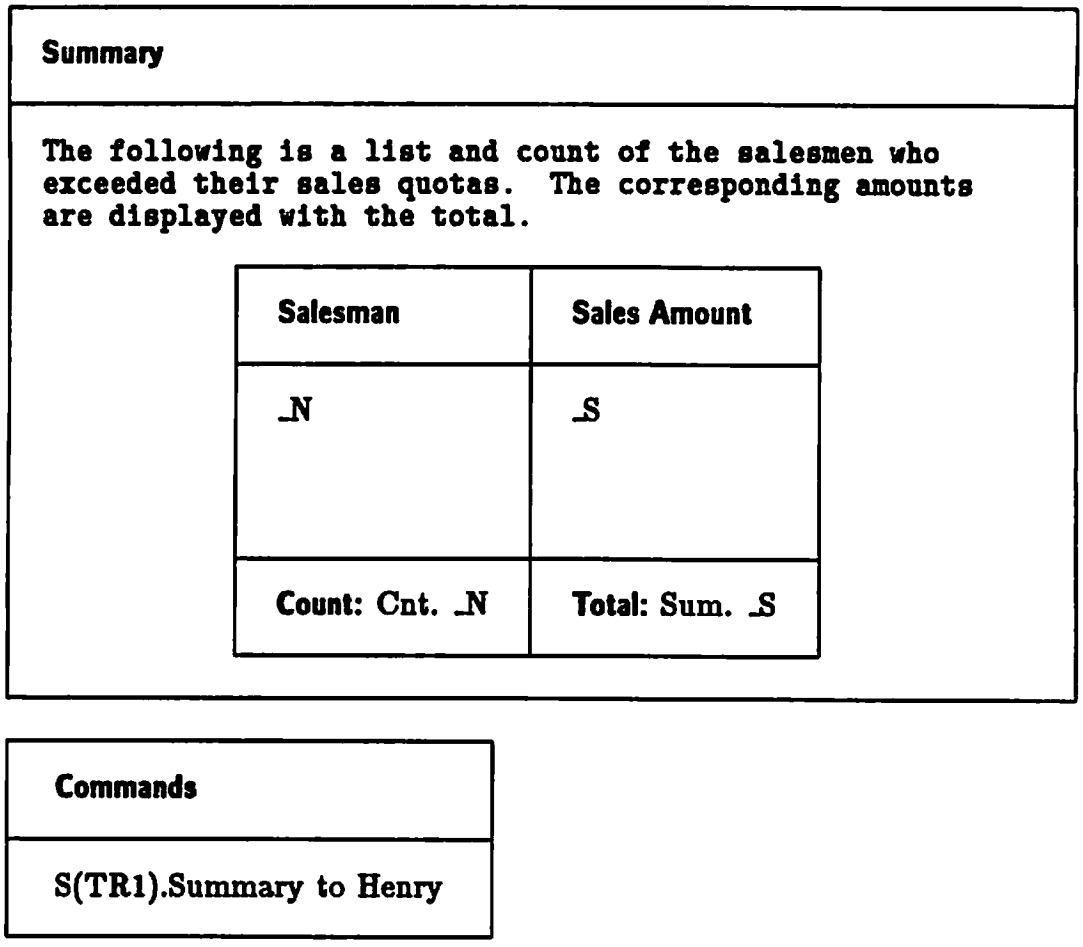


Figure 6. Sales report generator in OBE

6.1 Content

In figure 6 we show an example of how OBE can be used to construct and send a sales report to a recipient (Henry). The command that invokes the send is in the box labeled: **Commands.** The report that is being sent is in the box called: **Summary.** It consists of both text and a table of salesmen's performances. This table is culled from the **Sales** database. The filter that selects which records to pull from the **Sales** database is located in the **Conditions** box (_S > _Q or Sales to Date > Sales Quota). In the **Commands** box we note that the SENDing of the SUMMARY to HENRY is done automatically via a *trigger* mechanism (TR1). This trigger is defined as occurring every month.

OBE can be successfully employed in a variety of applications ranging from the automated generation and distribution of memoranda (the example we just saw) to automated database upkeep. One can construct menu-based command streams or even VISICALC-like spreadsheet programs.

While OBE is not a completely general purpose DSS environment, it does present several components that could be incorporated into a larger DSS context. In the next section we look at selected features that are candidates for porting to a DSS environment.

6.2 Features

The three major features of OBE are its use of examples, the introduction of triggers and filters, and the coherency of the screen manager.

6.2.1 *Examples.* OBE employs a type of example-based programming.[6] Instead of the usual methodology of programming by specifying abstract operations on abstract data types, OBE employs concrete data representations and mechanistic operations. Thus to select data records, one inserts fixed values into the fields of the records where one wants fixed values and conditionals into those fields where one is selecting a range of values. The results of selection are available immediately, giving instant feedback for debugging. This is a very different situation from that of conventional programming where specification of the algorithm and running the program are separate and distinct steps.

6.2.2 *Triggers and Filters.* Triggers allow OBE users to automate operations once they become mundane and fixed. Triggers can be one-shot or occur at repeated intervals (monthly, daily, or hourly). While triggers allow one more automatability than available in Lisa, we will see later it is a very limited form of automatability.

Filters are a fairly concrete and intuitive way of specifying database operations. All six of the relational algebra operations are possible via filters: selection, projection, join, intersection, union, and difference. A simple facility like this, with so much general purpose power epitomizes the type of tool we want to include in our DSS environment.

6.2.3 *Screen Management.* Screen management in OBE has been carefully tuned for conceptual coherency at all levels. For example, when one points to a window's vertical edge and invokes the EXPAND operation, the window expands vertically. When one points to a horizontal edge of a window, EXPAND expands horizontally. Inside a table or report, EXPAND will add either rows or columns to the table or report. Inside a field EXPAND adds space to the field, and in text it adds space before the cursor. The other operations of ERASE, MOVE, SCROLL, LOCATE and ZOOM are similarly leveraged for hierachical power.

[6] Though not as powerful or computationally complete as Tinker or PHD.

6.3 Limitations

OBE achieves its power and generality at the expense of several important limitations. Some are design limitations which arise from the attempt to make the tool and its problem domain more compact and coherent. Others involve only implementation difficulties that can be overcome by some re-thinking of the problem.

6.3.1 *Variable Names.* As seen in our example of memo generation, OBE has a serious problem with variable names. The ludicrous use of one letter variable names in OBE is symptomatic of a deeper problem: there is no provision for variable name scoping. How should name space boundaries be determined? How does one name higher level abstract entities like conditions, triggers, and command streams? These issues must be addressed before OBE progresses beyond "toy" applications.

6.3.2 *Explicit iteration.* There is no explicit iteration construct in OBE. Therefore operations involving double-dimensioned data is either difficult or impossible. One impossible operation is cross-referencing, since one cannot explicitly iterate through the columns of a table and process them against a one dimensional vector.

6.3.3 *Conditionals.* There are no true conditionals in OBE. Filters cannot themselves cause different execution streams to be invoked and triggers are only actuated by gross events and have little control over data streams.

6.3.4 *Applicability.* OBE, STAR and SBA are all extremely limited in the scope of their applicability. OBE operates only on forms, reports and tables. General purpose modeling environments contain more and richer structures than office forms and need a more generally applicable tool set.

6.3.5 *Extensibility.* OBE is limited to five operations: display, insert, update, delete and send a data object. General purpose programming is not possible. Thus we cannot do general purpose DSS simulation and modeling.

6.3.6 *Inexplicit Sequence.* By design the sequence of data selection and filtering is not specified in OBE. This was done deliberately so that the system could automatically make performance improvements by optimizing query evaluation. Unfortunately, this means that the user is never quite sure when collections are available, partial updates to the database complete, or any other timing information that is required for complex database operations.

In the next section we will examine Tinker and PHD to see how control and sequence can be made more explicit within the OBE metaphor.

7. TINKER AND PHD

In the previous sections, the systems we have dealt with were special purpose tools. They could deal with programming only within limited domains, and provided limited means for extending an algorithm. Both Tinker and PHD are general purpose programming environments, powerful enough to carry out any of the DSS activities we examined in earlier sections.

7.1 Philosophy

When Henry Lieberman designed Tinker, he took a look at programming and decided that it was not primarily a process of formally describing algorithms (as some textbooks might describe it), but one of teaching a machine how to handle situations. Since the best pedagogy in teaching is by example, this is the method employed by Tinker.

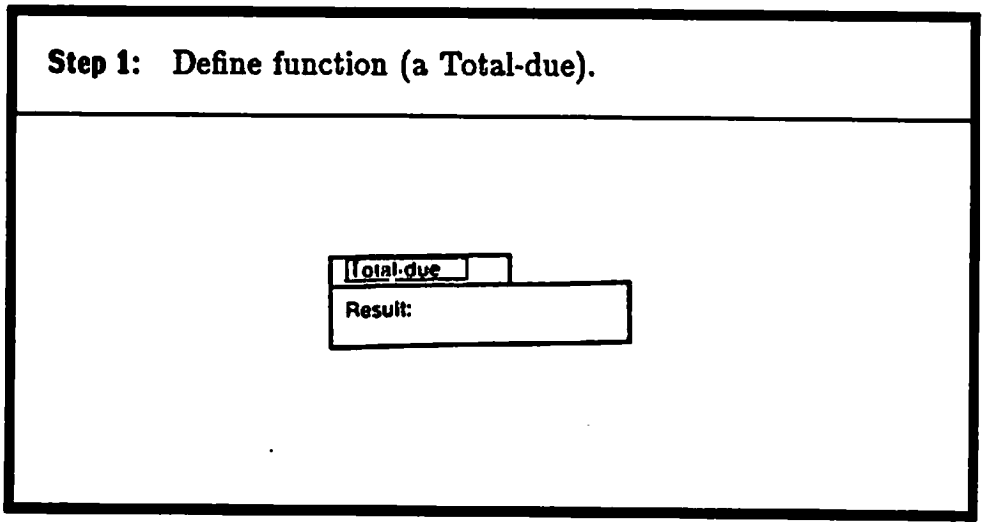
For example: to construct an $\alpha-\beta$ tree search function, one takes a specific tree and manually performs the $\alpha-\beta$ search for a specific node. General purpose code is produced in this manner by parameterizing out the specific example that the user worked on, and applying the resulting function recursively on the tree. Since not all nodes are solved in an identical manner – when the user does supply of a node solved in a different manner, the system prompts the user for a distinguishing conditional to tell apart the two cases. In this way conditionals are entered into the function, which will eventually result in a function which is generally applicable and capable of searching the entire tree properly.

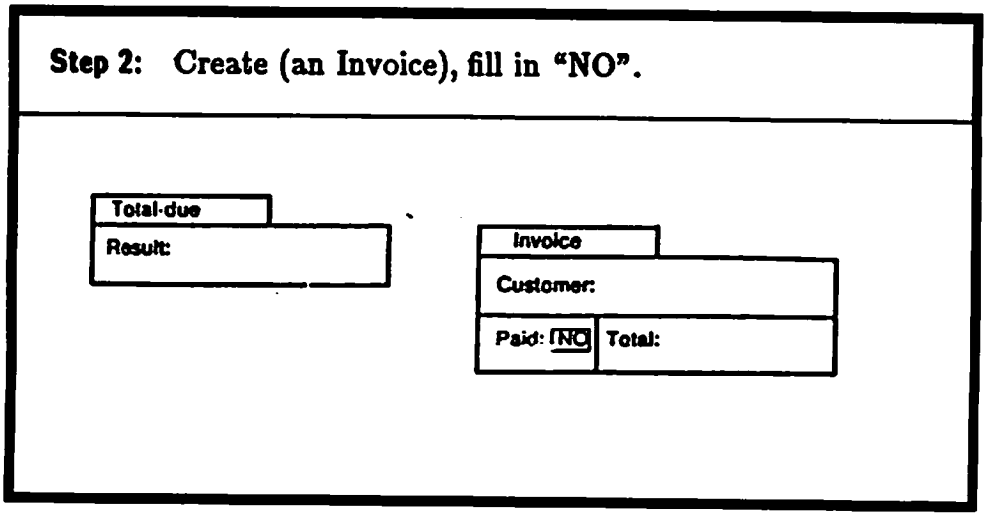
Tinker's generality comes from the different way it uses examples than that of OBE. The examples that the user supplies in OBE can only come from the existing database, while in Tinker they are constructed by the user. Also, in OBE, examples are only used for selection, while in Tinker they specify more general purpose functions such as conditionals and recursion. The result is a general purpose programming environment that has been used for constructing user interfaces, symbolic differentiation, and games.

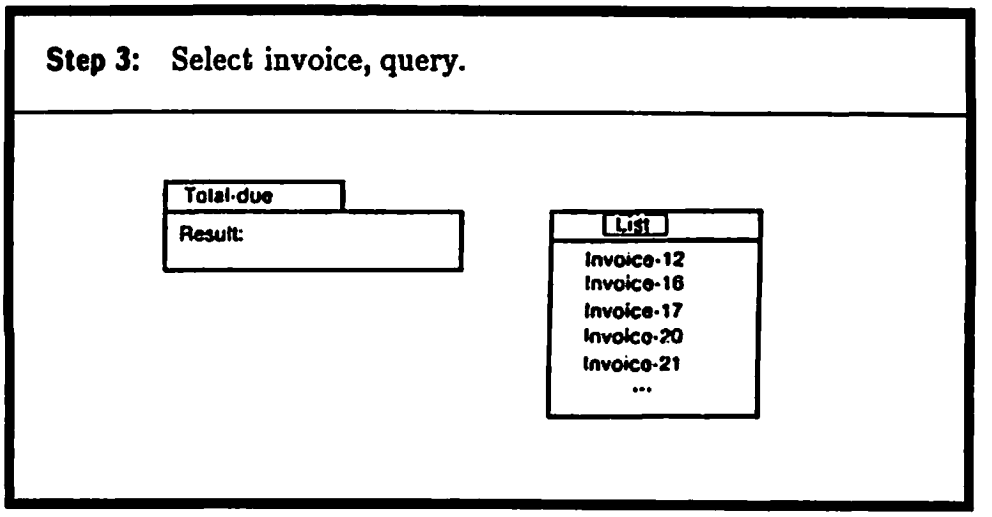
Tinker spawned numerous progeny research projects of which Programming by Homorphisms and Descriptions (PHD) [2] is the most interesting. The authors of PHD realized that one of the potential pitfalls of the Tinker approach is its reliance on the machine to generalize code from complex examples. Afterall, how is a machine to distinguish the relevant from the irrelevant in a particular example. To avoid this pitfall the authors of PHD raised the level of conversation

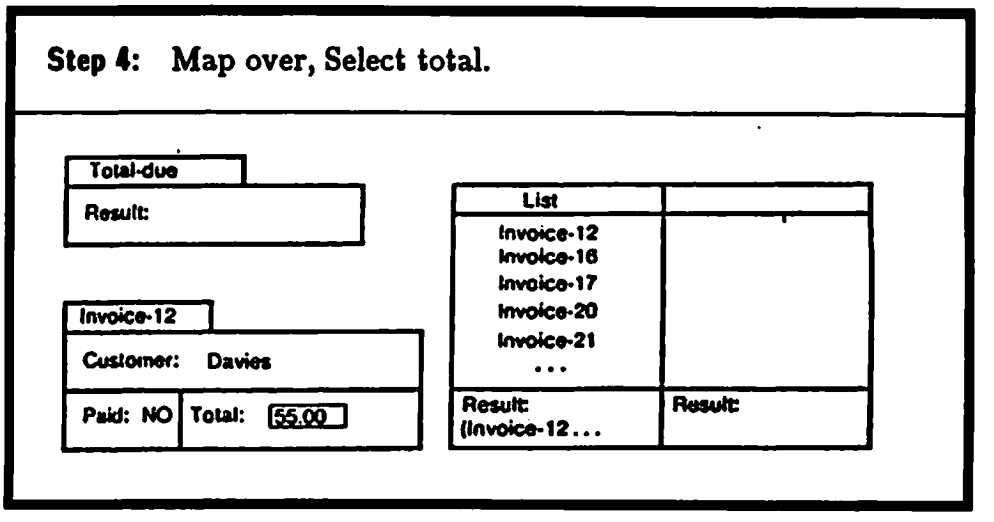
between the user and the system from specific objects to operations involving items defined on a higher semantic level.

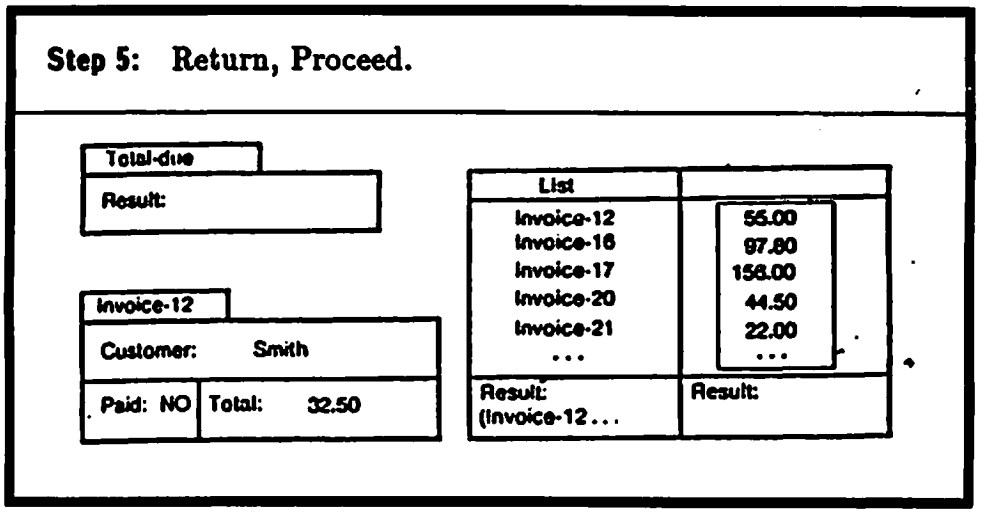
Homorphisms are concrete objects that the user manipulates in the PHD system to provide examples. However, instead of being discrete examples as in Tinker, Homorphisms in PHD serve as placeholders whose semantic characteristics are described by their current role, attributes, properties and relations to other objects.






Step 3: Select invoice, query.
Total-due
Result:
List
Invoice-12
Invoice-16
Invoice-17
Invoice-20
Invoice-21
...


Step 4: Map over, Select total.
Total-due
Result:
Invoice-12
Customer: Davies
Paid: NO
Total: 55.00
List
Invoice-12
Invoice-16
Invoice-17
Invoice-20
Invoice-21
...
Result:
(Invoice-12...
Result:


Step 5: Return, Proceed.
Total-due
Result:
Invoice-12
Customer: Smith
Paid: NO
Total: 32.50
List
Invoice-12
Invoice-16
Invoice-17
Invoice-20
Invoice-21
...
55.00
97.80
156.00
44.50
22.00
...
Result:
(Invoice-12...
Result:

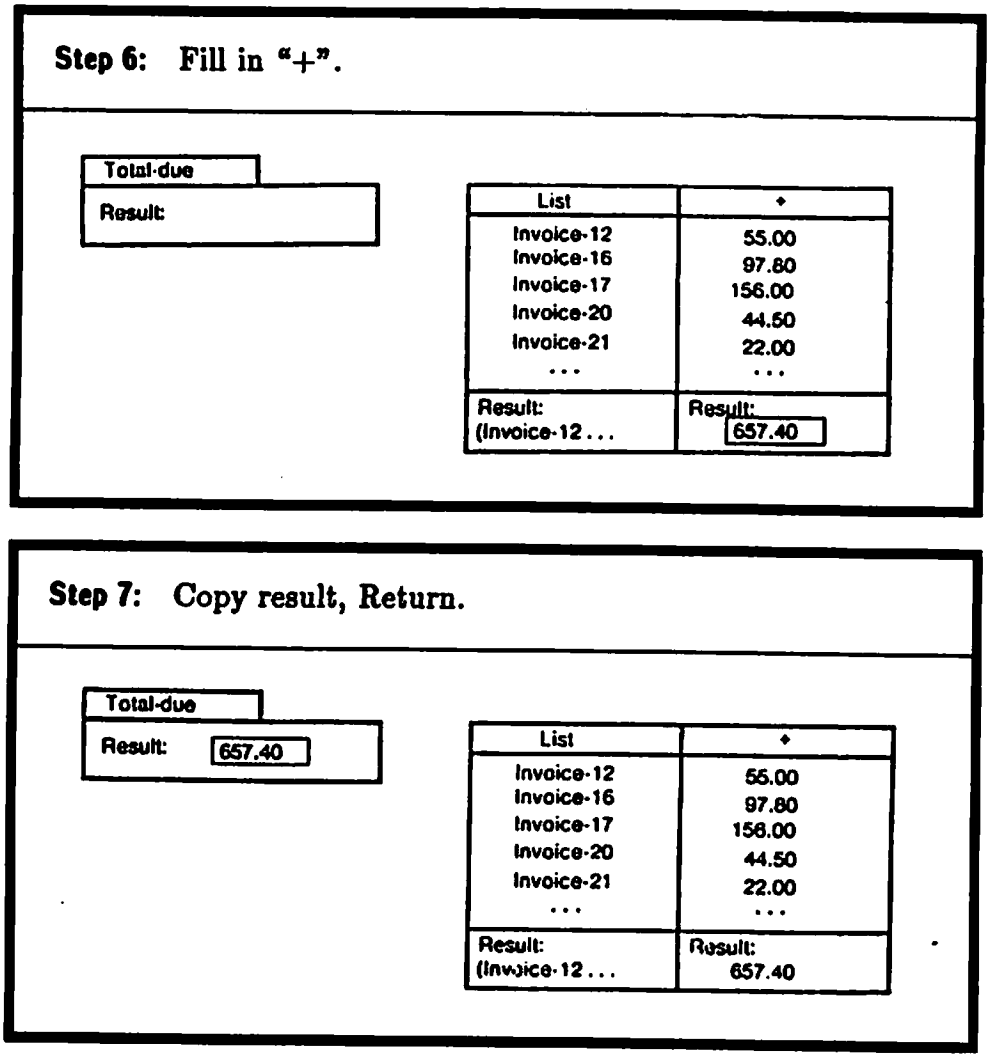


Figure 7. A PHD Programming Example

7.2 Contents

The example in figure 7 shows how to define a function to sum up the total of a collection of unpaid invoices. In **step 1** the user sets up a template for defining the function he is creating: Total-due. To solve our problem, we first need to collect all the unpaid invoices. This is done in **step 2** by creating an example (homorphism) of an unpaid invoice. One then invokes the QUERY operator on the database supplying an invoice homorphism as the example of what we are looking for. A list is returned in **step 3** of all the invoices that have gone unpaid. **Step 4** introduces the most important part of PHD. An operation such as QUERY can be mapped via iteration over an entire set of data in addition to the manner we used it in **step 3**. In **step 4** we select one of the invoices (Invoice-12) and query it for its total, indicating that its value should be returned. We then use this example query as a mapping operator to iteratively apply this query to the entire

list, which is returned as a list of unpaid amounts in **step 5.**[7] In **step 6** the user applies the "+" operator over the totals list to get the sum of the total due. Finally in **step 7** the user completes the definition of Total-due by specifying that the returned value should be the result of **step 6.** Once this is done, Total-due has been completely specified and is stored in the PHD system for use in in other functions and applications.

7.3 Features

7.3.1 *Concrete Programming.* Lieberman points out that one of the factors that makes programming difficult is our inability to visualize the operation of procedures. To illustrate this he draws on an analogy from chess. Below in figure 8 are shown two ways to represent a chess game:

When the chess game is presented in concrete form as a chess board, understanding the state of the game is straightforward. When the chess game is presented in chess notation, it becomes so difficult to follow the play of the game that one might as well be wearing a blindfold. To Lieberman conventional programming is like playing chess with a blindfold, it is very difficult to visualize the state of the program. Therefore PHD and Tinker use concrete objects and mechanistic operations to give to programming the same "concreteness" that one has when playing chess with chess pieces.

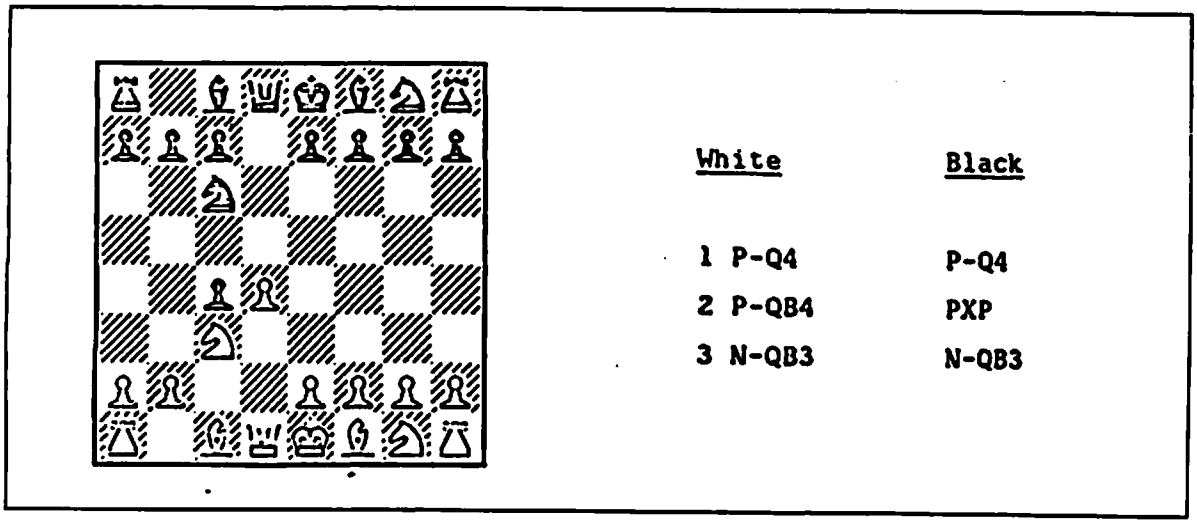


Figure 8. Two representations of a chess game

[7] In PHD not only defined functions such as QUERY can be MAPed, but also user defined functions. Thus PHD could achieve recursion by mapping Total-due over the list, or implement conditional execution by mapping a condition over a list.

7.3.2 *Extemporaneous Programming.* In the section on the DSS problem space we emphasized the importance of extemporaneous programming in DSS environments. Tinker and PHD are the first systems that really lend themselves to this sort of programming. Instead of the usual design methodology of:

1. Design algorithm
2. Code algorithm
3. Remove bugs
4. Execute and draw tentative conclusions
5. Iterate till satisfied

Tinker supports a one-step integrated process. In Tinker, as the name implies, one tinkers with ideas, draws partial conclusions, and constructs test models simultaneously. The Tinker/PHD approach allows one to explore semi-structured problems in an extemporaneous way by freeing one from the rigor required in producing conventional production systems.

7.3.3 *Revisionist Programming.* No system proceeds from conception to working production model without revision. Many times the conception is incomplete and experimentation must be done. The extemporaneous nature of program development in Tinker makes revisions much cheaper than the elaborate development/revision cycle involved in conventional programming.

7.4 Limitations

There are three basic limitations to the Tinker/PHD approach: Lack of permanent documentation, need for careful timing in the presentation of examples, and extensibility limitations arising from large flat name spaces.

7.4.1 *Documentation.* Tinker/PHD lacks a means of generating documentation to describe the created procedures. If one wishes to edit a program at a later date, one only has an unordered set of examples that were generated in a stream-of-consciousness session. Deciphering ones thoughts after a lapse of weeks or months can be very difficult. To be fair, Tinker does generate Lisp code for the function. Nevertheless, it would be as reasonable to expect a user to be able to decipher this machine-generated code as it would be to expect an assembly language programmer to understand a disassembly of his machine code. This limitation seems to be an inherent (and potentially serious) drawback to the type of programming by examples that is in

Tinker/PHD.

7.4.2 *Example Presentation.* The Tinker/PHD program by example methodology sometimes requires intimate knowledge of the algorithm the machine is using to generate code. It quickly became clear that the presentation of examples for the generation of the $\alpha - \beta$ tree search program was not done in a straightforward manner. The programmer knows that if he just started presenting all the different cases at the top level, a jumble of procedures would develop. Instead, he carefully timed the introduction of examples, when to invoke recursion, and when to introduce new functions, so that reasonable code was produced.

That Tinker had to resort to this careful meta-programming methodology is not an altogether unexpected occurrence. Afterall, pedagogues in our schools do not toss out hundreds of special case examples and expect the student to make sense of it. Instead examples, homework, and concepts are carefully chosen to produce in the student's mind the formal model that the teacher already knows. Nevertheless, the need for intimate knowledge of the computational algorithm argues against the extemporaneousness of the Tinker/PHD approach.

7.4.3 *Extensibility.* In creating Tinker/PHD algorithms, one must create ones own examples (as opposed to OBE/QBE where the examples can be fetched from the database). Unfortunately even our best teachers have trouble producing good examples and frequently "cop-out" and resort to didactic methods instead of example-based teaching. The same problem exists in Tinker/PHD where the user may be taxed to his limits to generate cogent examples.

Another problem in the Tinker/PHD approach is that it may not scale upwards well for large complex problems. When the problem is small, the user of Tinker/PHD can use it in a stream-of-consciousness mode to whip out a solution. But for larger problems, where multiple sessions may be needed to produce the algorithm, keeping track of all the examples may become very difficult.

What is needed is a system that can provide more structure to the "code" that is produced. Therefore we move our investigation of DSS tools to Smalltalk-80 which does provide a more structured environment for extemporaneous programming.

8. SMALLTALK-80

The last system we will examine is Smalltalk-80. The earliest versions of Smalltalk were intended to act as concrete simulation environments for children. After five generations of Smalltalk languages, it has now become a large system capable of not only modeling and simulation, but also

of the actual construction of disk file systems, compilers, text editors, simulation tools, painting kits, electronic circuit design tools, and other applications. [12,16,17]

8.1 Contents

One of the most important features of Smalltalk-80 is the blurring of the distinction between the operating system, the tools and the programming language. When one extends the Smalltalk environment by adding tools one is also is adding constructs to the language. Similarly when one is adding new abstract types to the language, one is creating new tools in the environment.

To give a concrete example, in Smalltalk one would not create five different tools for keyed access to information: help files, object libraries, file directories, spelling dictionaries and ISAM files. Instead, one creates a **Dictionary** which serves both as an abstract data type in the language and as a tool for these "five different OS functions". The blurring of distinction between OS and language lends itself to highly leveraged systems. For now we can cross-reference and correlate information that previously would have been isolated in five separate tools each with their own representations and interface utilities.

A second key design principle of Smalltalk is the use of communicating objects as a single uniform metaphor around which the system is structured. In Smalltalk everything is an object: Integers, Reals, Arrays, Bags, Dictionaries, Simulation Objects, Debuggers, and Browsers. One performs operations by sending a message to an object to carry out one of its methods. For example, a Bag will carry out the following methods:

size -	Return the count of the items in the bag.
add: anObject -	Add the object the to the bag.
occurancesOf: anObject -	Return the number of times the object occurs in the bag.
remove: oldObject -	Remove the object from the bag.
collect: aBlock -	Peform aBlock on each item in the bag and collect the results into a bag.
do: aBlock -	For each item in the bag, apply the code in aBlock.
select: aBlock -	Perform aBlock on all elements in the bag, collecting the results into a bag for those items that return true.
includes: anObject -	Test if the bag has this object.

The advantage to this scheme is that one programs by manipulating homorphisms. One is operating on objects according to their high level semantic attributes, not low-level data representations as in traditional programming languages. This approach is similar to PHD which also uses homorphisms to "raise the level of conversation" from low level structure details to higher level conceptual properties and semantics.

8.2 Features

Smalltalk is a large system. For a full description of it we refer the reader to the Goldberg and Robson book, Smalltalk-80: The Language and its Implementation [12]. The key features in Smalltalk-80 for a DSS implementer are its coherence, polymorphism, and the amount of leverage it provides.

8.2.1 *Coherence.* In OBE and PHD the basic structures of the system were built around records and forms. The designers of OBE and PHD were afraid to move away from these structures because their operators were tuned to operate specifically on forms and records. Designing a complete system to operate on all sorts of abstract data types (charts, streams, figures, etc.) was a risky business. There was good cohesion between the OBE and PHD operations and their data structures, but not generally applicability to all types of objects.

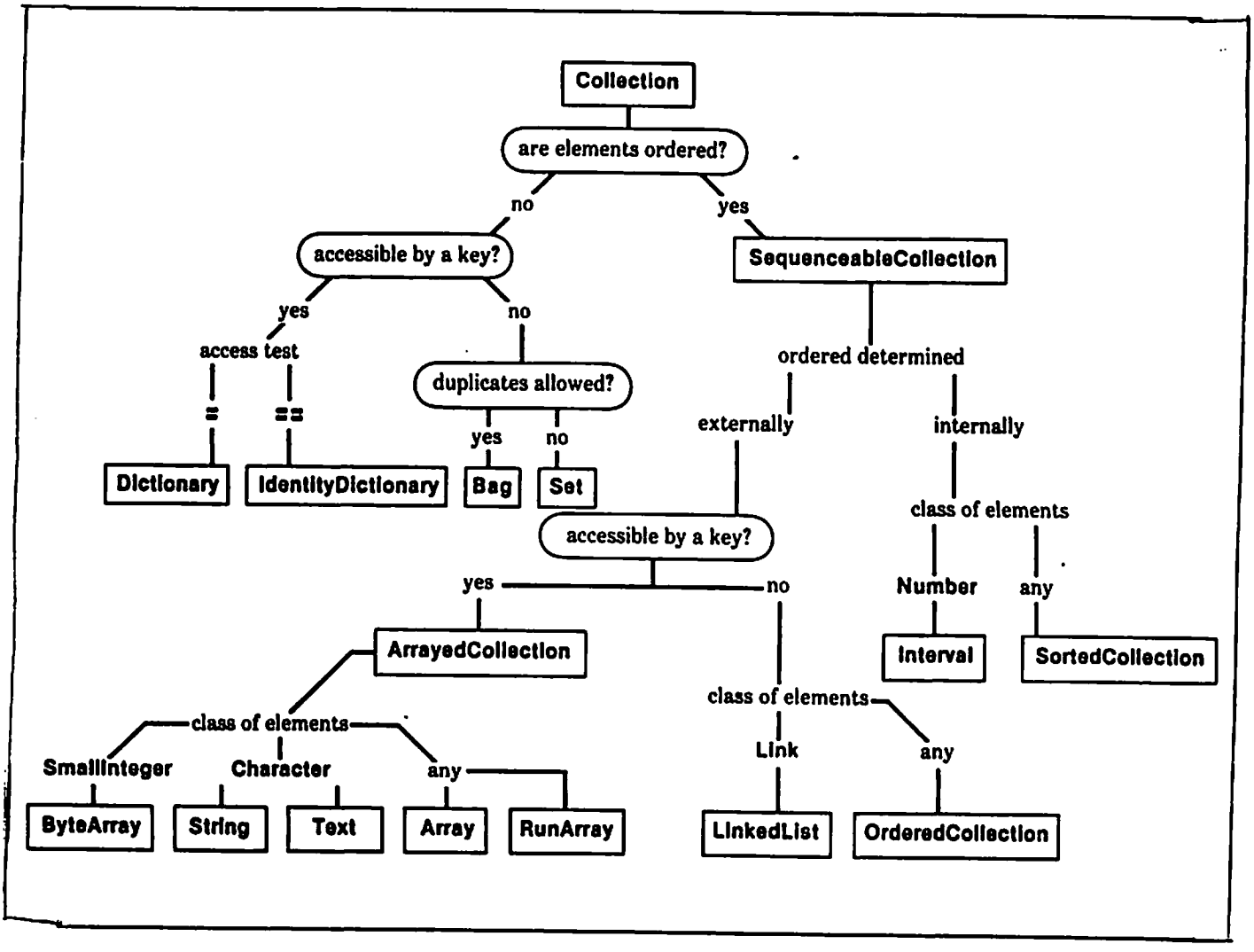


Figure 9. Smalltalk Collection Classes

Smalltalk is able to maintain the tight coherence between operators and objects that PHD and OBE have, while incorporating a much richer and diverse set of objects. Smalltalk is able to do this by using *Class Hierarchies.* In figure 9 we see the **Collection** Class hierarchy.

Operations common to all Collection Classes: add, remove, size, includes - is provided by **Class Collection.** As we move down the tree more specific operations are added by Classes for itself and its progeny. Thus, **ArrayedCollection** adds methods for adding at specific keyed locations (**add: anObject at: key**), for getting the first item (**first**), and for obtaining the index of an item (**indexOf: anElement**). There are similar customizations provided by the other Classes in the **Collection** Class hierarchy, as well as the hierarchies for **Numbers, DisplayObjects, Streams, FileObjects, etc..**

The Smalltalk Class hierarchy can serve as a useful organizing framework for the DSS designer seeking to deliver the same high leverage operations as PHD and OBE and yet be able to deal with a richer data domain.

8.2.2 *Polymorphism.* One of the more useful features of Smalltalk is its usage of the object metaphor. Alan Kay has compared objects to biological cells. Like one uses cells to construct creatures, one uses objects to construct Smalltalk programs.

When the workings of an organ are described in terms of cells, one is not concerned about the internal representation and construction details of the cells. The basic life processes of the cell are not important in the function of an organ. What is important is the interface the cell presents to the outside world.

In Smalltalk one constructs larger programs out of the abstract interfaces that an object presents to the outside world. The housekeeping and "life processes" that the object carries out internally, as well as its data representation are irrelevant. As with PHD one is encouraged to create and describe objects in terms of higher level semantic properties. The polymorphism in the system arises from the ability to freely interchange objects. When one is only concerned about the semantic interface a construct presents, one is free to embroider on internal structure and operation of the objects without fear of unexpected consequences.

The ramification of polymorphism for DSS designers is that a user can design a simulation environment with certain internal construction rules, and then later go back and embroider on the design without worry about mundane operational details. For example, one could design a stick figure robot factory, and later go back and fill in the details of the individual assembly processes that one only sketched in before.

8.2.3 *Leverage.* How does one extend ones tool environment and yet still maintain coherent operation with pre-defined tools? In Smalltalk one creates a ***subClass*** of an existing Class. The subClass will inherit the protocols, properties, and data structures of its ***superClass.*** One can then add new methods and data structures knowing that one is only embroidering new richness into the object, and that the new subClass will still maintain all the semantic properties of its superClass. For example, say one is unsatisfied with the **Dictionaries** Class, and would like to create a subClass **Thesaurus** that would maintain the properties of a **Dictionary,** but would also provide extended operations for exploring semantic relations between entries. One could then create **Thesaurus** as a subClass of **Dictionary** and be confident that all existing code for maintaining, operating, and

making use of **Dictionaries** would also work for **Thesauruses**. Only users wishing to make use of the added properties of **Thesauruses** would be aware that they were there.

8.3 Limitations

8.3.1 *Size.* The major limitation in Smalltalk comes from its size. While originally third graders could quickly learn it, Smalltalk-80 is now a very large system with 77 basic Classes. When one adds to this all the protocols (methods) for these Classes, one realizes that Smalltalk is a very large system that is not as easily learned and explored by the novice as it once was.

The solution to this difficulty is not simple. It requires that the DSS implementer go back and repeat the design process that led to Smalltalk in the first place. One must search for coherence in objects and operations germane to the DSS domain and implement only those entities that give the greater total leverage to the system.

8.3.2 *Clumsiness.* Another problem with Smalltalk is the way it handles statements such as conditionals and simple arithmetic expressions. Because Smalltalk makes completely uniform use of the object metaphor, both conditions and arithmetic operations are expressed as message sending to objects. A conditional evaluation is done by sending a block of code to a Boolean, and arithmetic operations are expressed as messages to numbers.

To some extent the user can adapt to this unexpected and non-intuitive manner of expression. Still, it is incumbent of DSS implementers to strive to make use of the long-held intuitive biases that users have, to cut down on the learning cost of a new system. Above all else the DSS implementer is a pragmaticist, not a doctrinairian.

9. CONCLUSIONS

In conclusion, we must re-emphasize the importance of a DSS environment and the need of technologies that would foster such an environment. Inasmuch as the requisite technologies are not yet developed, the approach toward implementing the creation of a suitable DSS milieu calls for innovative action.

ThinkerToy [14], the author's proposed PhD thesis, addresses itself to this objective. The suggested research program provides for pictorial simulation using the object metaphor to generate a rich and general purpose programming environment. While the metaphors are largely drawn from the Smalltalk heritage, there is a deliberate effort to present concrete visualizations that are at least as strong as in the OBE system.

The focus of ThinkerToy is on simulation and modeling. The intended goal is to quickly produce animated simulations of manufacturing plants, industrial processes, computer architecture, and biological systems, all within quality standards that compare favorably with animated explanations in science-type films. The simulation and modeling should provide a rich milieu for tinkering and exploration of a knowledge domain. In accomplishing this goal, the author hopes to open an avenue into the area of exploratory programming which heretofore presented problems that have not found successful solutions.

10. ACKNOWLEDGEMENTS

I wish to thank Professors David McDonald and Bruce Croft (the First and Second Readers of my Master's Project) for their help with this paper. Without their numerous suggestions and patient editing this report would have been much the poorer.

REFERENCES

1. Steven Alter, "Decision Support Systems: Current Practice and Continuing Challenges", Addison-Wesley, Reading, Massachusetts, 1980.

2. Guiseppe Attardi and Maria Simi, *Extending the Power of Programming by Examples*, ACM SIGOA Conference on Office Information Systems. Philadelphia, Pennsylvania (1983), 67 – 78.

3. Robert H. Bonczek, Clyde W. Holsapple, and Andrew B. Whinston, *Specification of Modeling Knowledge in Decision Support Systems*, "Processes and Tools for Decision Support", North-Holland Publishing Company, Amsterdam, Netherlands (1983), 65 – 78.

4. W. Bruce Croft and Lawrence S. Lefkowitz, *POISE: An Intelligent Assistant For Professional Based Systems*, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, COINS Technical Report **No. TR-82-19** (1982).

5. K. Roscoe Davis and Patrick G. McKeown, "Quantitative Models for Management", Kent Publishing Company, Boston, Massachusetts, 1981.

6. S. Peter DeJong and Roy J. Byrd, *Intelligent Form Creation in the System for Business Automation (SBA)*, IBM T. J. Watson Research Center, Yorktown Heights, New York., IBM Research Report **No. RC 8579** (10/24/80).

7. Roger D. Eck, "An Introduction to Quantitative Methods for Business Applications", Wadsworth Publishing Company, Belmont, California, 1979.

8. Joseph L. Erhardt, *APPLE's LISA: A Personal Office System*, The Seybold Report on Office Systems ISSN: 0160-9572. **Vol. 6, No. 2** (January 24, 1983), 1 – 26.

9. Craig Field and Nichlos Negroponte, *Using New Clues to Find Data*, Proceedings of the Third International Conference on Very Large Data Bases, Tokyo (1977).

10. Richard E. Fikes, *Odyssey: A Knowledge-Based Assistant*, Artificial Intelligence **Vol. 16** (1981), 331 – 361, North Holland.

11. Mark S. Fox, *The Intelligent Mangement System: An Overview*, "Processes and Tools for Decision Support", North-Holland Publishing Company, Amsterdam, Netherlands (1983), 105 – 130.

12. Adele Goldberg and David Robson, "Smalltalk-80: The Language and its Implementation", Addison Wesley, Reading, Massachusetts, 1983.

13. John H. Grant and William R. King, "The Logic of Strategic Planning", Little, Brown and Company, Boston, Massachusetts, 1982.

14. Steven H. Gutfreund, *ThinkerToy: An Environment for Decision Support*, PhD. Thesis, University of Massachusetts (1984) (to appear).

15. Kristina Spargren Hooper, *Identification of Mirror Images of Real-World Scenes*, PhD. Thesis, Department of Experimental Psychology, University of California at San Diego (1973).

16. Alan C. Kay, *Microelectronics and the Personal Computer*, Scientific American **Vol. 237, No. 3** (September 1977), 230 – 244.

17. Alan C. Kay, *Personal Dynamic Media*, IEEE Computer (March 1977), 31 – 41.

18. Peter G. W. Keen and Michael S. Scott Morton, "Decision Support Systems: An Organisational Perspective", Addison-Wesley, Reading Massachusetts, 1978.

19. Peter G. W. Keen, *Decision Support Systems: A Research Perspective*, Center for Information Systems Research, Sloan School of Business, Massachusetts Institute of Technology, Cambridge, Massachusetts, CISR Technical Report No. 54 and Sloan W.P. No. 1117-80 (March 1980).

20. Peter G. W. Keen and Thomas J. Gambino, *Building a Decision Support System: The Mythical Man-Month Revisited*, Center for Information Systems Research, Sloan School of Business, Massachusetts Institute of Technology, Cambridge, Massachusetts., CISR Technical Report No. 57 and Sloan W.P. No. 1132-80 (May 1980).

21. Peter G. W. Keen, *Decision Support Systems and Managerial Productivity Analysis*, Center for Information Systems Research, Sloan School of Business, Massachusetts Institute of Technology, Cambridge, Massachusetts., CISR Technical Report No. 60 and Sloan W.P. No. 1156-80 (October 1980).

22. Janet L. Kolodner, *Indexing and Retrival Strategies for Natural Language Fact Retrieval*, ACM Transaction on Database Systems **Vol. 8 No. 3** (September 1983), 434 – 464.

23. Henry Lieberman and Carl Hewitt, *A Session with TINKER: Interleaving Program Testing with Program Design*, Artificial Intelligence Laboratory, Massachusetts Institute of Technol-

ogy, Cambridge, Massachusetts., MIT AI Memo **No. 577** (September 1980).

24. Henry Lieberman, *Seeing What Your Programs Are Doing*, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts., MIT AI Memo **No. 656** (February 1982).

25. Henry Lieberman, *Constructing Graphical User Interfaces by Example*, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts., MIT AI Memo.

26. James L. McKenney, "Simulation Gaming for Management Development", Division of Research, Graduate School of Business Administration, Harvard University, Boston Massachusetts, 1967.

27. Donald R. Moscato, "Building Financial Decision-Making Models", American Management Association (AMACOM), New York, New York, 1980.

28. Robert Purvey, Jerry Farrell, and Paul Klose, *The Design of the Star's Records Processing: Data Processing for the Non-Computer Professional*, ACM Transaction on Office Information Systems **Vol. 1 No. 1** (Jan 1983), 3 – 24.

29. Seymour Papert, "Mindstorms: Children, Computers and Powerful Ideas", Basic Books, New York, New York, 1980.

30. Patrick Rivett, "Model Building for Decision Analysis", John Wiley and Sons, New York, New York, 1980.

31. W. Schild, L. R. Powers, and M. Karnaugh, *Picture World: A Concept for Future Office Systems*, IBM T. J. Watson Research Center, Yorktown Heights, New York., IBM Research Report **No. RC 8384** (7/30/80).

32. Herbert A. Simon, "The New Science of Management Decision", 3rd edition, Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

33. M. M. Zloof, *Office-by-example: A business language that unites data and word processing and electronic mail*, IBM T. J. Watson Research Center, Yorktown Heights, New York., IBM Systems Journal **Vol. 21, No. 3** (1982).

34. M. M. Zloof, *Query-by-Example*, AFIPS Conference Proceedings, National Computer Conference **44** (1975), 431 – 438.

35. M. M. Zloof and S. P. deJong, *The System for Business Automation (SBA)*, Communications of the ACM **Vol. 20, No. 6** (June 1977), 385 – 396.