DEVELOPMENT OF AN EXPERT SYSTEM USING OBJECT ORIENTED PROGRAMMING

by

ANDRE PLANTE, 8612297

REPORT SUBMIT TO FULFILL THE REQUIREMENTS OF

COURSE 336 - 490 D PROJECT

DATE

MARCH 20, 1989

PRESENTED IN WINTER SEMESTER 1989

MACDONALD COLLEGE OF MCGILL UNIVERSITY

DEPARTEMENT OF AGRICULTURAL ENGINEERING

TABLE OF CONTENTS

ACKNOWLEDGMENTS

The author wishes to express his sincere thanks and appreciation to Professor S.O. Prasher, the research supervisor, for his unfailing support, encouragement and guidance. His classical yet rigorous attitude has helped the author through the course of this experience.

The author also wishes to express his gratitude and appreciation to "The Whitewater Group" and especially to Zack Urlocker, Manager of Developers Relations for the Actor package, for their interest and suggestions with Actor-related difficulties.

Many thanks are also extended to the following students, B. De Serre, and J. Hebert, for there friendship, help and encouragement which contributed to make this project enjoyable and rewarding. A special acknowledgement to N. Zemanchick for proofreading this report.

Thanks also go to my family and very close friend, Michele, for there constant support and patience, understanding and judicious advices with subjects in relation with this research.

ABSTRACT

Object-oriented programming was used to design an expert system in water management studies. The purpose of the system was to test if subirrigation and controlled irrigation would be desirable for a given farm. Years of personal expertise by the leading intellects in this field was put into gouverning rules. The expert system was developed using "Actor", an object-oriented language. Due to the limited time available, full scale testing was not done but a prototype was built. Therefor, partial results are given in this report. Additionally, the benefits and encapsulating methods of object oriented programming in relation with artificial intelligence in agriculture are fully discussed.

INTRODUCTION

Agriculture is still a way of life for many but it is becoming an increasingly sophiticated business. Computers in agriculture have been around for a long time but they are just starting to take the strain off the owner/operators. One way of relieving this pressure that was recently made available by new technology is expert systems. The need for decision support systems in agriculture is increasing with the impact of decisions on the financial growth of the enterprise. Expert systems can provide help with these decisions by their knowledge base. They also have the power to communicate, to act as technology transfert medium. The use of artificial intelligence as a farming tool is starting to develop in many areas and countries. Some new specialized languages are appearing on the market, and with this has originated a need by certain faculty members to see if their use was justified in agricultural systems. These new languages offer a challenging approach to old dreams. Their logical development is closer to the human thought process, which therefore makes them more adapted to the building of expert systems. Conventional programming followed a mathematical approach to problems. The internal workings of object oriented programming have new revolutionary concepts, such as encapsulation and inheritance. Their relative importance is discussed further. In conventional programming a new project was always started from a blank page. A object-oriented programmers can start were one as stop. Reusability of source codes is an important time saving factor. Also discussed later. Through the creation of an expert system pertaining to subirrigation and controlled irrigation, the appreciation of the object oriented approach will be discussed. The possibilities of object-oriented programming for agricultural uses will also be discussed. One of the new languages offered in this trend will be used to build the prototype of the expert system and a description will be given. The language utilized is commercially available on the market. It is produced by "The Whitewater Group", and sold under the name " Actor ".

REVIEW OF LITERATURE

COMPUTERS IN AGRICULTURE

Although agriculture is still a way of life, it is also an increasingly sophisticated business. Tomorrow's farmer will be a better businessman as well as a better production technician. He/she will benefit from a rapidly growing array of electronic technologies which will provide more information, on a timely basis, with more analytical capabilities. How well farmers manage this information will be an important factor in business success (Agriculture 2000, 1983). As Beasley (1983) puts it " a computer won't make a bad manager good, but it will make a good manager better".

The computer technology as an agriculture tool is not new. Some applications have been around for several years. One of the earliest and still in use applications is the dairy herd improvement (DHI) services which have been widely available since the 1960's (Macdonald College, 1986). The purpose of this organization is to provide dairy farmers with performance results of each cow by analyzing milk samples and monitoring each of the cow's milk production on an individual basis. General management recommendations are then made to the herdsman in order to achieve financial growth.

As Gautier (1987) pointed out, most computer applications in agriculture can be classified into the following categories: 1) financial record keeping and accounting, 2) physical record keeping, 3) planning and decision support, 4) communications, and 5) farm automation. Financial budgeting and accounting packages, in general, attempt to adapt the type of software used by small and medium-sized businesses to the farm enterprises. These imitations range from simple applications written in BASIC (Legasy et al., 1984), with some specific spreadsheet programs, to more dedicated and comprehensive systems.

Since the start of the electronic era, research in agriculture has been relying on computers for laborious and time consuming tasks such as statistical analyses and theoretical simulation of physical processes. This was done by using mathematical models. Researchers are now working on projects which use such models in decision-making applications at the farm level (Kjelgaard and Wu, 1983) through the use of personal computers and computerized data bank systems.

General purpose software for ledger, accounts receivable and payable, can also be used for farm management. Record keeping is basically a mass of organized information concerning different fields of action of the enterprise such as; fields, animals, machines. From these data the farmer can retrieve the desired information at the press of a key. An increase in both the quantity and quality of information accessible to the farmer enables what Peart and Puckett (1982) label a "management by exception" approach where one manager can effectively handle alarge farm by concentrating on the critical or extraordinary cases identified through the record keeping system. This is becoming a standard throughout the industrial world.

Data acquisition is done via automation and is used mainly in automated feedback control devices. I believe that the dairy farm is one of the main concerns of both research and industry in Quebec, probably due to the economic importance of this industry. Computerized record keeping allows rapid and accurate identification of individuals in the herd, this system is usually linked to an on-farm computer (Agriculture 2000, 1983). As Speicher (1981) views it: "The computerized dairy herd management system is one utilizing electronic animal identification, automatic milk weighing and recording and data entry at the parlour, the lot, the barn or the computer itself". Commercial and experimental systems which attempt to achieve this level of functionality already exist (Spahr et al., 1985). Many of the commercial systems make an attempt to integrate many hardware and software functions in one turn key system. Other programs in this category are for machinery selection or replacement, sire selection for herd im-

provement, forecasting production and herd population modelling. These packages are designed to help solve very specific problems. Therefore, they constitute a legitimate decision support systems, as reported by Gautier (1987).

Microprocessors are being incorporated more and more in the design of agricultural machinery. Computer-based systems to monitor machinery operating parameters such as wheel slippage, fuel consumption, seeding or spraying rates, engine load etc. are already commercially available. Manufacturers are incorporating subsystems such as sensors, on board computers and intelligent control programs in their equipment, and this trend will most likely continue. Research has also been carried out on automatic tractor and combine guidance systems (Harries and Ambler, 1981), as well as computer controlled irrigation (McLendon et al., 1983). The above authors have discussed some of the design considerations which must be observed when creating control programs, to automate the operation of irrigation control structures. Kok and Desmarais (1985) have discussed the use of an integrated hierarchical control system for greenhouse management where four levels of control and/or data acquisition (physical, instinctual, Pavlovian and intelligent) are used to copy the human decision process.

The market now carries affordable communication hardware/software which allows computer systems to exchange information. Thus, the farmer is less isolated with his problems and can contact with the outside world for information or advise. This is done through electronic mail, modems or conference services. Most services also support some kind of interactive computing in the form of electronic mailing, continuing education, banking, shopping etc. Electronic devices and computers used in agriculture and more specifically in farm management are attracting the interest and imagination of theagricultural community to higher and higher levels. Many institutions are publishing newsletters or reports on the preliminary concerns with this field. Most agricultural or rural publications regularly include an article on the use, or potential of electronics and computers for farming. The use of computers as ,information management tools, is considered by many

authors to be the key to any further increase in farm productivity (Cardiff, 1985; Riddle, 1985).

EXPERT SYSTEMS

The organisation or classification of knowledge is one of fundamental features of the human thought process (Walton 1988). To create artificial intelligence is to take a computer and have it mimic the human reasoning. The artificial intelligence field is subdivided in many sectors of activity, the most common ones being; vision, hearing and speaking, robotics, learning abilities, and expert systems (Rheault 1988). I will be discussing expert systems in this section.

Not so long ago, computers were mainly used to handle numbers, like calculators. One would put in numbers and would get numbers back. With expert systems we attempt to simulate knowledge and reasoning, and actually get the computer to "think" in one's field of expertise. This is usually done by applying IF-THEN condition-conclusion pairs in order to form a decision tree (Jones, 1985). So the goal is to make the know-how of leading intellectuals accessible to the rest of the population. Expert systems, according to McKinnon and Lemmon (1985), distinguish them selves from conventional programming by three major aspects:

(1) We apply reasoning to knowledge instead of applying mathematical functions to numbers. Judgement as well as reasoning is used to achieve this.

(2) This can be a learning tool because it explains the meaning and reasons of each question. This is also referred to as transparency.

(3) The updating of the knowledge bank is easier due to physical separation between the knowledge and the program, and by this practice the application achieves flexibility.

From the preceding discussion, we can see that expert systems are composed of the knowledge base, the inference engine and the user interface. Let's discuss these sections individually.

First the knowledge base or heuristics, these are the private parts of the system and are usually held by individuals. This includes the rules of thumb, judgement, and sometimes experience based guesses that typically characterize the human expert-level decision making. Secondly the inference engine or procedure include the approach to the define problem, it decides which heuristic to apply to the problem, accesses the rules in the knowledge base, executes and determines when acceptable solutions have been found. The inference engine can be in the form of a shell such as "Exsys 3" written in the C language (Exsys, 3 1985) which are available on the market or in the form of a personal program developed using conventional languages. The remaining part permits the bi-directional communication and should use an approach adapted to the aimed users. Self explanation of the conditions or reasoning for an answer is very important for knowledge transfer to the users.

A computer is capable of handling more than payroll, inventory, tax returns, orbits and trajectories, and Database management. But let's not overestimate its capabilities. Tasks such as designing new tools, stock market predictions, and for that matter, every day language, are not in reach. Presently the computers can handle diagnosing and troubleshooting, production scheduling, and equipment layout for example (VanHorne, 1986). There are three important sectors of application for these systems, as reported by Reault (1988):

(1) as a diagnostic tool. This is used mostly in preventative maintenance of machinery, medical diagnostic, and in management of big projects.

(2) to interpret situations, to examine alternatives, to make recommendations accordingly to pre-establish criteria. This is used in the insurance field.

(3) and in intelligent documentation which is used to replace manuals, procedures, data cards. A very good example of this is the MUSE system used by McGill university for its library card services.

There are other systems that are used for selection in complex decisions, process control, or as a support for conception and development. It is the author's opinion that the future and most promising use for agriculture is in decision support. Other fields will also have their share of the stage (Huggins 1986). Here are some of the existing expert systems cited in the literature and reported by Dluschizt and Schmisseur (1988);

NAME	FIELD OF APPLICATION
PLANT/ds	Diagnostic of soybean diseases.
PLANT/cd	Prediction of corn damage from the black cutworm.
COMAX	Integrated crop management in cotton.
POMME	Pest and orchard management of apples.
PLANT/tm	Diagnostic of weeds in turf.
GMA	Determination of grain marketing alternatives.

Why is it so important to store the knowledge of experts on computers? First of all for economic reasons. A good example of this is the expert system called REPFARM which is used for farm machinery selection. Big savings can be made due to the importance of the needed investments in this sector (Kline et al., 1988). But there is more to it, permanent access to knowledge and consistency in judgement. Experts are expensive, rare and they are not always available. This shortage of experts is responsible for holding back the development of science as well as the development of human kind. An increase in productivity of experts can be achieved by freeing them from the routine jobs so they have more time for research. The formation of junior personnel can be executed through this

transfer of technology. As we can see consistency and reliability, not creativity, are paramounts (VanHorne 1986).

OBJECT ORIENTED PROGRAMMING; A DIFFERENT APPROACH

The words Object-Oriented Programming have been gaining a lot of commercial attention from those who must built advanced computer systems. Several claims have been made that computer program writing time could be cut by half, that conventional programming was obsolete, etc... But is this silent revolution part of our future? Here is an analogy given by Cox (1987) that illustrate the situation:

Before the industrial revolution, the firearms industry was hardly an industry at all but a loose coalition of individual craftsmen. Each firearm was crafted by an individual gunsmith who built each part from raw materials. Firearms produced in this way were expensive and each was the distinctive product of a gunsmith's personal inspiration. After a big contract to an individual gunsmith by the government he innovated by dividing the work so that each part was produced by a specialist to meet a specified standard. Each gunsmith focused on a single part, using sophisticated tools to optimize that task. This produced economies of scale that drove down manufacturing costs and permitted easy repairs.

As Cox (1987) points out, the importance of object-oriented programming is comparable to that of the interchangeable part innovation, and for many of the same reasons. Both redefine the unit of modularity so that workers produce subcomponents instead of complete solutions. The subcomponents are controlled by standards and can be interchanged across different products. Programmers no longer build entire programs from raw materials, the bare statements and expressions of a programming language. Instead they produce reusable software components by assembling components of other programmers. These components are called Software-ICs to emphasize their similarity with the integrated silicon chip, a similar in-

novation that has revolutionised the computer hardware industry over the past twenty years or so. This new approach has the following advantages:

1. Systems can be made malleable by retaining some elasticity into run-time. This involves relaxing the usual demand that everything be accomplished at compile-time, also known as linking. Dynamic binding is a move in this direction (Philips, 1986) but in object oriented programming this concept is pushed to a maximum. Once an object receives a message, it takes over the systems until it passes the priority to another object. In contrast, the traditional well structured sub-routines eventually pass the control back to the main program (Verity, 1987).
2. Systems can be made more changeable by making them smaller and lighter in weight, thus transforming system like functionality into program like size. The encapsulation and inheritance techniques, described later, do this by integrating reusability into the mainstream of the software development process.
3. Systems can be more tightly encapsulated as objects that behave as armor-plated black boxes to limit the ripple effect when a penetration of the static defense does happen. A change to one part of the system need not affect the rest of the system, but can be dealt with inside the part directly affected. This is very desirable when dealing with big applications (100 000 lines +) because as the understanding of the entire system goes down, the ability to deal with deficiencies goes down as well. This is best illustrated by comparing it to "Occam's Razor" rule (Actor, 1987).

These primary new features are summarized in two key words, encapsulation and inheritance. These new tools and concepts can help to produce software that are far more tolerant to changes. Encapsulation is the foundation of the whole approach. Its contribution is restricting the effects of change by placing a wall of code around each piece of data. All access to the data is handled by methods (procedures) that are put there to mediate access to the private part (Madsen, 1986). These encapsulated operands are called objects. This encapsulation makes systems more malleable by restricting the amount of damage a change

can cause. This benefit is called encapsulation. It is the first and foremost contribution of the dynamically-bound style of object-oriented programming.

A second benefit, called inheritance, becomes possible once encapsulation is available. Inheritance is a technique for defining new data types by describing how each new type differs from some pre-existing type. The methods that characterize a class are also inherited by the descendants (Smalltalk/V, 1986). Inheritance is the more innovative part of this approach because it is not provided by conventional languages. It is a tool for automatically broadcasting codes to classes developed by different members of a team. Programmers no longer start each module with a blank page, but instead write a single statement that references directly some class that is already in the library. The ancestor class can be shared by an infinite number of descendent class (Agha, 1986). Each subsequent statement describes how the new class differs from the one in the library. Inheritance is a way of defining some useful construct in a central place and then automatically broadcasting that construct to all the places where it could help (Cox, 1987). Encapsulation means that the user no longer applies operators to operands while taking care that the two are type-compatible. Instead the user tells the object what needs to be done, and the object chooses the correct operator from a table of things that the object knows how to do. Actually the user does not need to know how the object works to use it in his own programs (Robison, 1988). The result is that the effect of adding a new kind of object does not spread beyond the place where the new type is defined.

Unlike more traditional programming methods that are based on concepts such as data flow or mathematical logic, object oriented programming is based on directly modelling the application(Thomas, 1989). Mostly all of the languages that are object oriented use an environment which is based on the concept of windows (Petzold, 1988). Therefore creating the window is the starting point of building an application. This window is composed of the objects that support the program (SmalltalkV/286, 1988). The following example is a step by step development of an application, simple but clear as is the path to follow in this type

of programming. I found this example in the SmalltalkV/286 (1988) package tutorial book;

Step 1. Problem statement: a phone book with the list of persons stored in alphabetical order. Selecting a name will retrieve the associated phone number. Names and phone numbers can be added or deleted.

Step 2. Draw the window, concentrate on the appearance and don't think at this stage of the internal workings.

Step 3. In order to build this window, you must identify the classes and select the ancestor classes. In this example we need to create the PhoneBook class whose instances are the phone book window application. We also need the following classes; Menu, ListPane, TextPane, and TopPane. After consideration the Dictionary class is the most appropriate for associating names with phone numbers. Names and phone numbers will be represented by String.

Step 4. List the user interfaces, in this case we have; add, list, remove, and text.

Step 5. Implement these methods in the application, which include the starting point for a window; opening it. The dictionary which will contain the information is also opened at this point.

Probably the biggest draw back with this new technique is the adaptation period; inheritance and encapsulation is a really totally different field. One can estimate that a few months working full time will be needed before really knowing what is going on. Large collections of predefined objects and classes can be purchased and put away until needed. They will usually include the corresponding methods and algorithm (Verity, 1987). To use these a good understanding is again required. Object-oriented programming is not so much a coding technique as it is a code packaging technique, a way for code suppliers to encapsulate functionality for future consumers (Cox, 1987). Functionality is no longer developed by coding each line from scratch, but by inheriting some useful class and describing only how the new one differs. The effect is to put reusability squarely in the mainstream of the software development process.

OBJECTIVES OF THE PROJECT

In this dissertation the following objectives and guidelines were used:

1. Explore the possibilities of object oriented programming for agricultural uses;
2. Design an expert system using object oriented programming;
3. Demonstrate that the system can be operational.

SCOPE OF THE PROJECT

To achieve the preceding objectives, commercial hardware and software were used and a prototype was created. The effort focused on the understanding of object oriented programming and expert systems. The expert system was designed to give an appreciation of the conditions under which subirrigation and controlled irrigation could take place in humid regions. Full scale testing was not done but a demonstration program was built. Years of personal expertise by the leading intellects in this field was put into governing rules. The knowledge gained from this experience was assessed and the findings and conclusions are discussed in detail. Also guidelines are given for the development of future systems.

DESCRIPTION OF ACTOR PACKAGE

Actor is an object oriented programming language with incremental compiling. This means that instead of separating the active programming instructions from the passive data, Actor integrates the two in what is called an object. It is said to be a pure language because all the supporting classes are in the form of objects. They are written in Actor, like the application we will be producing. This new language was designed with artificial intelligence as it's primary use.

Actor is a Microsoft windows application. As such it is supported by any computer that will run MS-Windows. To run Actor the following hardware is required: hard disk, 640 K RAM, Graphics display and adapter, mouse or other pointing device. All these peripherals are supported via MS-Window.

Actor is a complete programming environment. It uses all the power of Microsoft windows to help you organize and analyze your work. For the less initiated readers, MS-Window is a multitasking graphical-based object environment. This permits you to see all your work at the same time and trace the influence of one part on another as you make changes. Windows are acted upon as any other object blended in the Actor philosophy.

Among other programming tools supplied with the Actor package, we find the "Browser", which is a highly specialized editor customized to work with Actor source code. The "Inspector", which can be imagined as a window to peak inside Actor objects. The "Debugger", which is used in handling source code errors by interacting with the compiler and giving you the operations that preceded the system halt. And the "File Editor" which its main purpose is to deal with MS-Window option editing.

Actor also offers dynamic linking with other languages such as; C, Pascal, Assembler, or Fortran. The librairies written in the preceding languages can also be accessed by Actor. This package has the ability to pass data in C structure as well. Artificiel intelligence support includes; frames, symbols, dictionaries, lists, symbolic programming, and functional arguments. It also as parsing possibilities and lexical analysis YACC compatibility.

When you purchase Actor, the package can be seen as a collection of objects just waiting to produce descendant classes. The actual possibilities of Actor are as large as your imagination. Actor is produced by "The Whitewater Group", the version I used was purchased in 1986 but more recent versions are available now. These newer versions can support OS/2 system environment.

DEVELOPMENT OF AN EXPERT SYSTEM USING OBJECT ORIENTED PROGRAMMING

The expert system data knowledge was constructed from the work of Moin Darbary and Prasher (1987). This work was composed of some 70 rules reunited in an expert system shell, Exsys 3. From these 70 rules I found that the desired information could be gathered from 12 questions. Here are the question with there possible answer;

Question 1: Is your region a humid region? Possible answers: yes, no.

Question 2: Is there a water supply? Possible answers: yes, no.

Question 3: What type of power supply do you have? Possible answers: diesel, electric, none.

Question 4: What is your type of soil? Possible answers: sandy clay, clay, silt, silty clay, silty clay loam, silt loam, clay loam, sandy clay loam, loamy sand, sandy loam, loam, sand.

Question 5: Is there an impermeable layer? Possible answers: less than 2.5 M, more than 2.5 M, none.

Question 6: Was some backfill used at installation time? Possible answers: yes, no.

Question 7: What is the salt concentration in your irrigation water? Possible answers: less than 1500 ppm, between 1500 and 10000 ppm, more than 10000 ppm.

Question 8: Is your average water table deeper than 1 M during the growing season? Possible answers: yes,no

Question 9: What is the effective rooting depth of the future crop? Possible answers: less than 30 cm, between 30 and 60 cm, more than 60 cm.

Question 10: What is the slope of your field? Possible answers: less than .1%, between .1 and .2%, between .2 and .3 %, between .3 and.4%, between .4 and .5%, more than .5%.

Question 11: What is your drain spacing? Possible answers: less than 15 M, between 15 M and 20 M, between 20 M and 25 M, between 25 M and 30 M, between 30 M and 35 M, more than 35 M.

Question 12: Where will the harvested fruit grow? Possible answers: below ground, above ground.

Note that with each question stated precedingly a "help" choice is given. When selected the user will get the necessary explainations concerning the purpose or importance of the question he is answering. After reading this information the user can return to the question an answer.

All of the proposed answers have attached to them a probability of the event being successful. The event referred to are the questions which dictate the conditions for subirrigation and controlled irrigation. These probability are in fraction form, e.g. 3/10 or 10/10. Two separate numbers are given for the two types of irrigation. Some questions depend on the answers given to previous questions, in this category we have questions; 6, 7, 12 which depend on question 4. We also have questions 10 and 13 which depend on question 9. All the other questions are considered to be independent from one another. Some questions are key questions, by which I mean that under certain conditions subirrigation is not desirable and when these conditions are encountered the system will come to a stop and explain why subirrigation is not desirable. It will then terminate all operations.

In the system developed by Moin Darbary and Prasher (1987) the inference engine was formulated by the shell and could not be modified. Due to the object oriented language possibilities, the inference engine in the proposed application was to be developed from the following statistical rules;

CF(A1,A2)=CF(A1) + CF(A2) *(1-CF(A1))

where, CF stands for certainty factor of independent variables

A1 is the first rule or question

A2 is the second rule or question

The primary task is to calculate the probability of an event according to the answers to the preceding questions. By putting the inference engine in separate objects, future upgrading of the statistical part is possible and easy. In contrast with conventional programming the inference engine will not have the control of the application. The use of a main program and subroutines are a thing of the past. The structure of object oriented programming favours a shift of power to one object at a time as explained in the literature review.

Now that all the system requirements have been stated, I will explain the sub-divisions and creation of the necessary objects. In order to report the actual process that was followed to develop the final logic, I will explain the first system which was developed but later proved to be illogical according to the object oriented philosophy.

First system

In this scenario the objects were divided by rules, therefor the system was composed of 12 objects equivalent to the 12 questions outline earlier, plus the inference engine for a total of 13 objects. A flow chart is given in appendix A. Each object was a small self-contained program that was meant to run one after the other. The execution order was directed by one master class. Each object was design to have it's own screen and facilities, such as pushbuttons, it's own windows, messages, and direct access to the inference engine. They all descended from the Window class. The first class to be written was the Power class. Although the syntax was mastered after a few tries, I could never get the logic of the Power class to run correctly.

Second system

In this revised approach the method explained in the literature review section was followed. The Windows were designed as the starting basis. The supporting objects that were needed were then introduced one by one but in no specific order. The system will still have 12 windows but each will have about 3 to 4 supporting classes. Every possible answer will also be an object with it's own supporting classes. The number of classes needed for this application will be in the order of 75. A flow chart of the entire application is given in appendix B. They will mostly descend from Dialog class, OrderedCollection class, ListBox class, Window class, and Object class. Instead of using pushbutton as in the first system, I have used list boxes.

The time required to write the entire system with all it's supporting classes is out of the scope of this project. In order to check that the present system is functional for agricultural purposes I have made a demo program. The demo program was nested with the rest of the Actor Demos. It was done by adding a statement in the OrderedCollection class that maintains the list of demos in the DemosList class. The same type of windows, as in the Actor Demos, were used for consistency reasons as well as for source code reusability testing. The demo contains 2 questions and each have 2 possible answers. After completing the cycle the control is returned to the Actor Demos. The reader will find a flow chart and a print of each class in apendix C. A print of "Demoslist" was included so you could see the lines that were added to nest the demonstration program in the Actor Demos. Pictures of screens during actual runs were taken an put in appendix D. The reader can therefor visualize the impact of the graphical object interface.

RESULTS AND DISCUSSION

As computers gain more and more importance in agriculture, learning to use this new type of language will become essential in order to follow the technological trends. During the development of this project it became evident that an adaptation period is necessary. In the first system I developed, I was able to master the syntax after a few rewrites. However, understanding the interaction between the objects you are creating and the rest of the system is the hardest part. The failure of the first system can be attributed to this type of error. This system was essentially conventional programming; it relied on dynamic binding to reunite the classes at run time, therefore no shift of power from an object to another was occurring. The actual size of each object was also found to be inadequate. The objects where too big for practical purposes, signifying that the full power of this new approach was not being utilized.

The second system is a sample of what you can achieve with object oriented programming when you use the right approach. Objects are reusable and should be reused. In order to demonstrate the usefulness of this feature I used the same windows throughout this demonstration program, reusing the object which had the windows for the Actor Demos, found under the name "DemosWindow". The encapsulation was also demonstrated in this application because when I modified the "DemosList" class file the rest of the objects that form the Actor Demos were not affected by the change. By having smaller objects updating is made a lot easier, for example if the list of possible choices for a question have to be changed this can be done by replacing the object which contain the list only. The rest of the system will not be affected. In the demo system I wrote two classes, responsible for storing and handling the appropriate lists of choices or objects, that is "SoilList" and "PowerList". The SoilList class could be given it's full potential easily by replacing it with an updated version which includes

all soil types. To display the needed text in my application I used the "error-Box" function but I would strongly suggest that a special message box be created and used throughout the system for a better understanding on the user's part. This message box should be a descendant of the message window, therefore inheriting the full power of its ancestor.

The purpose of the second system was to prove that a full size system could be built using this approach. In order for the system to be completed the remaining questions should be added to the existing structure, either at the beginning or at the end, depending on the desired results. The inference engine was discussed, but it never reached the final stage of completion. The internal workings would need adjustments but when ready it can be integrated to the system like any other object. When the system is complete the main support window coulod be added to it and the application can be sealed off. The Actor book contains a recipe to seal off applications. It gets rid of all the unwanted objects and classes, and keeps only what is necessary for the support system of the new application. Some classes are essential for developing the application, such as the text editor, but are useless afterwards therefore getting rid of them makes the system smaller and time efficient.

In the agricultural field no two enterprises are the same, therefore this new modularity approach could give birth to a generation of software which will be adapted to the individual needs of each enterprise. With this reusability and sealing off technique it seems that such applications are not far away.

SUMMARY AND CONCLUSIONS

An expert system was designed using a new approach called object-oriented programming. A partial system was built using a commercially available package. A description of this new language, ACTOR, is given. The practical possibilities of this new approach for agriculture were explored.

Object oriented environment makes it essential to start by designing the window and from this window it is then possible to conceptualise the support objects. The difficulty with this new approach is the absence of the main program in order to favour a distribution of the tasks. Each object should serve only one purpose in order to use the full potential of this approach. Due to this subdivision of task, called encapsulation, the debugging process is simplified. Additionally the chain effect of a bug is also stopped by this method. Updating is simplified by this modularity, and objects can be replaced when or if they become obsolete. Reusability is a time saving factor with great potential, it should be used to its maximum. The second system I designed can be developed to its full size by using the modularity and the inheritance mention before. Objects can be added when developed, therefor the expert system can be written by a team of programmers or by succeeding senior year project students. From the demonstration program I made to test the second system, I can conclude that the entire system can be operational. With the available technologies computers are capable of producing more than mathematical function and to use them only at that level is under utilizing them. Expert systems will continue to develop as the demand for artificial intelligence in agriculture grows. Agriculture will benefit from this new approach by gaining a new way of coping with the diversity of the demand due to the variance between enterprises. Object oriented programming will be a tool of great importance for agriculture in years to come.

SUGGESTIONS FOR FURTHER WORK

The type of system described in this dissertation was not specifically designed to support growth. Actually no thought at all was given to it. Object oriented programming by it's nature has the ability to be upgraded intensively. Consequently, the number of objects and their methods can be improved and expanded without limits. The package can therefore be made to fit the need of the potential users.

The following are suggestions for system enhancements or additions which were identified by the author:

- the entire system should be built according to the second system described above;
- the finished system should be tested by users for it's accuracy and transparency;
- a study of inference engines should be conducted in order to design an upgraded version;
- series of objects capable of interrelating between users and rules to permit easy upgrading of rule base should be added;
- the possibility to make the system learn from it's mistake so we can have an intelligent software;
- the possibility of adding a voice digitizer to facilitate access for people with no computing background.

LITERATURE CITED

Actor, 1987. Actor language manuel. The Whitewater Group. IL. 643 pages.

Agha,G. 1987. Actors; a model of concurrent computation in distributed systems. The MIT press, MA. 274 pages.

Agriculture 2000. 1983. Batelle Press. Columbus, OH. 250 pages.

Beasley J.O. 1983. Microcomputers on the farm. Howard W. Sams publishing compagny, IN. 204 pages.

Cardiff, J. 1985. Farming and the computer. A changing horizon publication, Seattle, WA. 226 pages.

Cox, B.J. 1987. Object oriented programming; an evolutionary approach. Addison-Wesley publishing compagny. 274 pages.

Dluschitz R. and W.E. Schmisseur 1988. Expert systems; Application to agriculture and farm managtement. Comput. Electron. Agric., 2: p.173-182.

Exsys 3 1985. Expert system development handbook. publ. by Excess inc. 100 pages.

Gautier, L. 1987. Devellopment and use of a database and program package for farm production management, PH.D. Thesis. Macdonanld College of McGill University. 534 pages.

Harries, G.O. and B. Ambler. 1981. Automated ploughing: a tractor guidance system using opto-electronic remote sensing techniques and a microprocessor based controller. J. of Agricultural Engineering Research. Vol 26(1): 33-53.

Huggins, L.F. and J.R. Barrett and D.D. Jones 1986. Expert systems: Concepts and oppurtunities. Agricultural engineering jan/feb 1986, pages 21-23.

Jones J.W. 1985. Using expert system in agricultural models. Agricultural engineering vol(66) no.7, pages 21-23.

Kjelgaard W.L. and Z. Wu. 1983. Micro-computer program for field machinery management. ASAE paper no. 83-1536. 20 pages.

Kline,D.E. and D.A. Bender, B.A. McCarl, and C.E. VanDonge 1988. Machinery selection using expert systems and linear programming. Computers and Electronics in agriculture, 3, pages 45-61.

Kok, R. and G. Desmarais. 1985. An intergrated hierachical control system for an intelligent greenhouse. ASAE paper no. NAR85-403. 12 pages.

Legacy, J. and T. Stitt and F. Reneau. 1984. Microcomputing in agriculture, Reston Publishing. 254 pages.

Macdonald College. 1986. Dairy herd analysis service. Dairy herd analysis services Report 1980-1985. Macdonald College. St-Anne de Bellevue. The service Report. 78 pages.

Madsen, O.L. 1986. Block structure and object oriented languages. Sigplan not.(USA) vol 21 no. 10 p 133-142.

McKinnon, J.M. and H.E. Lemmon 1985. Expert systems for agriculture. Computers and Electronics in agriculture. vol(1) pages 31- 40.

Mclendon, D. and S.J. Thomson and J.L. Chesness. 1983. Irrigation Scheduling - A valid option with microprocessor-based controls. Agricultural Engineering. Vol 64(9): 12-14.

Miller, D. 1983. Videotex: Science fiction or reality?. Bute magazine. Vol 8(7). pages 42-56.

Moin Darbary M. and S.O. Prasher 1987. Water management systems in humid regions. Macdonald College of McGill University. unpubl. MSc disseration.

Muse 1988. McGill University librairy automation news. vol 4 num 2, 17 pages.

Peart M.R. and H.R. Puckett. 1982. Feed processing and animal freeding: management by exception. Agricultural Engineering. Vol 63(2): pages 18-19.

Petzold C. 1988. Programming Windows. Microsoft press, Wa. 852 pages.

Philips 1986. Disk operating system version 3.10, user reference manual. Philips corp., Montreal, Canada. 356 pages.

Rheault, M. 1988. Les systeme experts. Micro-gazette. October. pages 6-7.

Riddle, W.E. 1985. Agriculture 2000 - A time of technological change. Processing of the Agri-Mation 1 conference. ASAE. pages 21- 26.

Robinson, P. 1988. CIM's missing link: object oriented databases. Computer graphics world. october 1988. p 53-58.

Smalltalk/V 1986. Programming handbook. Digitalk inc. 514 pages.

Smalltalk/V 286 1988. Programming handbook. Digitalk inc. 561 pages.

Spahr, S.L. and H.B. Puckett and D.E. Dill. 1985. An integrated system for automatic data collection and analysis on the dairy farms. Processing of the Agrimation 1 conference. ASAE. pages 339-345.

Speicher, J.A. 1981. Computerized data acquisition systems for dairy herd management. J. of Animal Science. Vol 53(2): pages 53- 60.

Thomas, D. 1989. What's in an object?. Byte march 1989, pages 231-241.

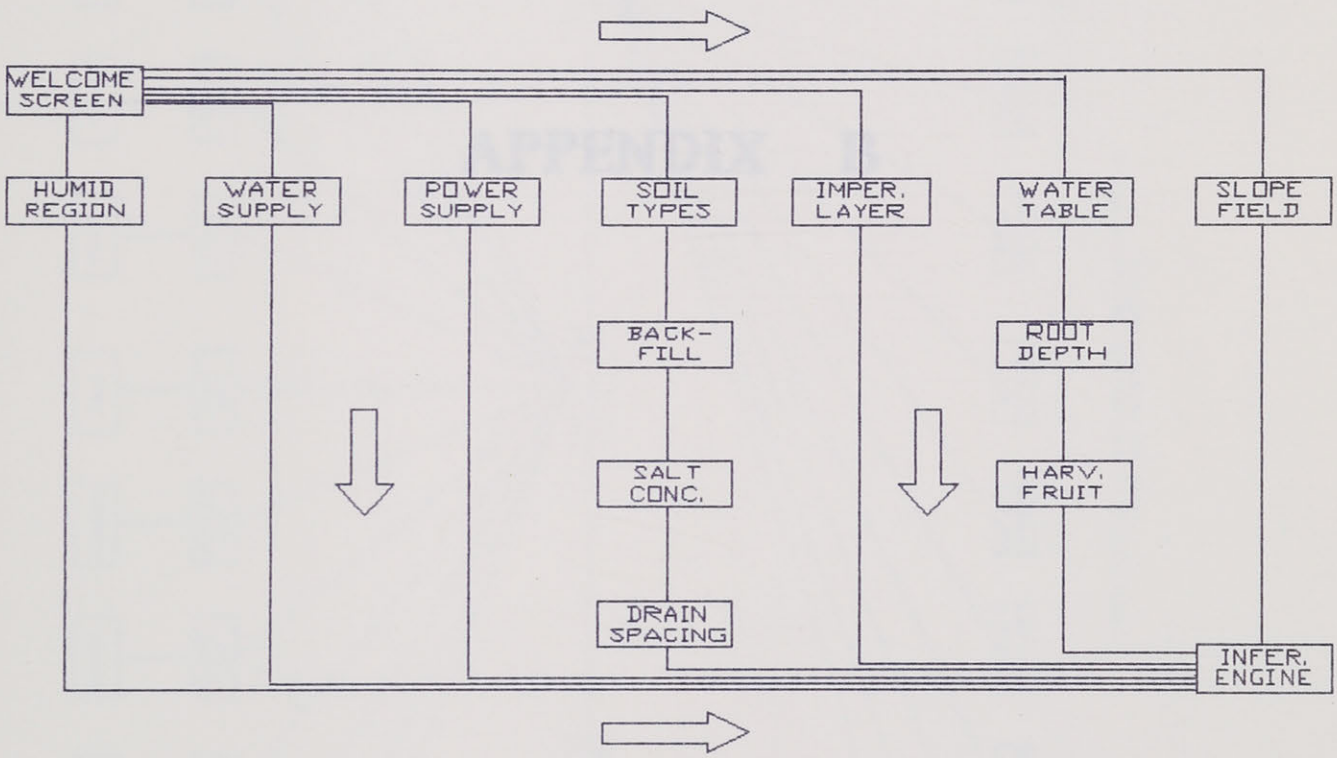
VanHorne, M. 1986. Understanding expert systems. Bantam books, The Waite Group. 233 pages.

Verity, J.W. 1987. The OOPs revolution. Datamation. vol 33 no. 9. p. 73-78.

Walton, P.D. 1988. Principales and practices of plant science. Prentice Hall inc., 438 pages.

APPENDIX A

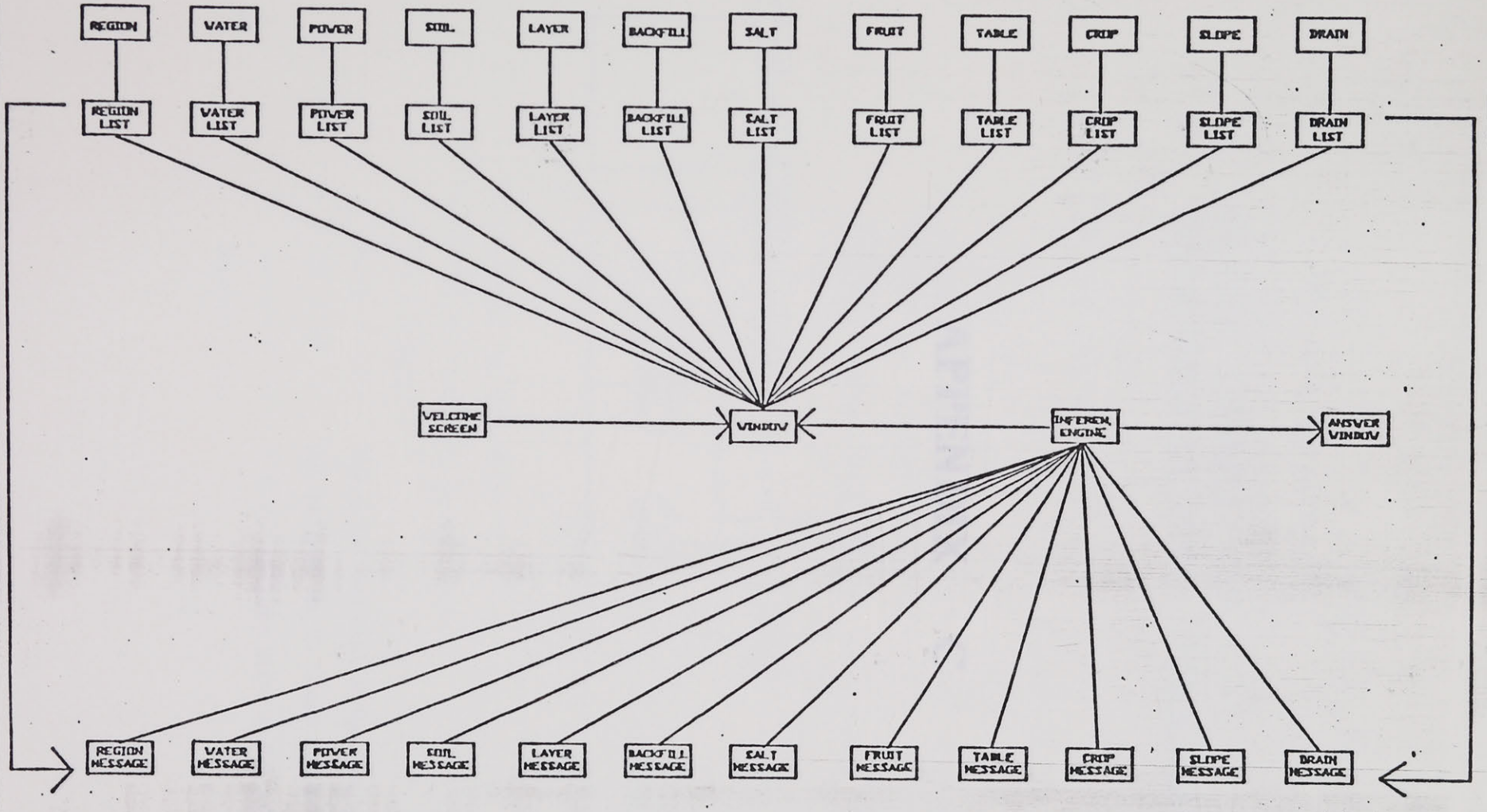
SYSTEM ONE



This is a flow chart of the first system.

Each box corresponce to an object.

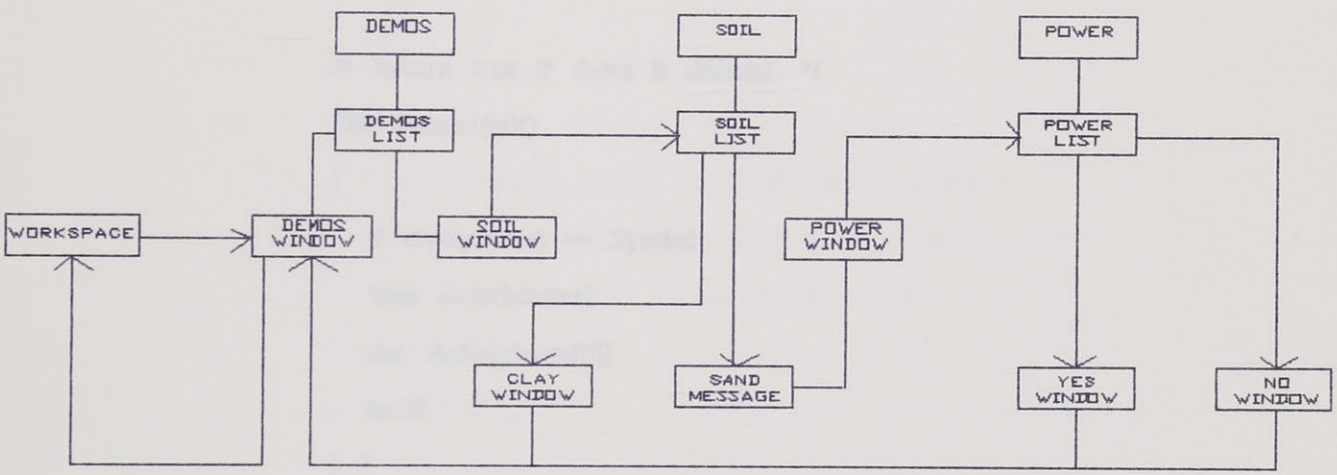
APPENDIX B



NOTE: THERE IS AS MANY MESSAGES FOR ONE OBJECT AS THERE ARE POSSIBLE CHOICES.

APPENDIX C

FLOW CHART OF DEMONSTRATION PROGRAM



This is the flow chart of the demo program.
All the code for the above classes
is in the following pages in this appendix.

```
/* Each Demo object contains information on an Actor
   demonstration program. */ !!

inherit(Object, #Demo, #(desc /* Description for listbox */
classes /* Classes used for demo */
files /* Files used for demo */
memReq /* Memory req'd in bytes */
runBlock /* Code to run the demo program */), 2, nil) !!

now(DemoClass) !!

now(Demo) !!

/* Return true if demo is defined. */
Def defined(self)
{ ^
   if class(classes) == Symbol
   then Actor[classes]
   else Actor[classes[0]]
   endif
} !!

/* Initialize the demo object. */
Def init(self, descrip, cls, fl, mem, blk)
{ desc := descrip;
   classes := cls;
   files := fl;
   memReq := mem;
   runBlock := blk;
} !!
```

```
/* A window class for the Actor demonstration programs. */!!

inherit(TurtleArea, #DemosWindow, #(dlg), 2, nil) !!

now(DemosWindowClass) !!

now(DemosWindow) !!

/* Recreate method for demos window. */
Def   recreate(self)
{ create(self, parent, caption, locRect, WS_POPUPWIND);
  init(self);
  ^nil
} !!

/* This method is needed to transfer eol messages
   sent to self to the objects in OutPorts.  This
   happens during some Actor operations.  */
Def eol(self)
{ printLine("");
} !!

/* Initialize the DemosWindow. */
Def init(self | cStr)
{ init(self:TurtleArea);
  createMenu(self);
  cStr := "Clear!";
  changeMenu(self, 0, lP(cStr), CLEAR, MF_APPEND);
  freeHandle(cStr);
  if Actor[#Turtle]
  then cStr := "Fractals!";
     changeMenu(self,  0, lP(cStr), FRACTAL, MF_APPEND);
```

```
        freeHandle(cStr);
        addTurtle(self, init(new(Turtle), self));
    endif;
    drawMenu(self);
    show(self, 1);
    reSize(self, 0, 0);
    dlg := new(DemosList);
    runModeless(dlg, DEMOSLIST, self);
} !!

/* Respond to command messages. */
Def command(self, wP, lP | aDlg)
{
    select
        case wP == CLEAR repaint(self);
        endCase
        case wP == FRACTAL aDlg := new(FractalDialog);
            runModal(aDlg, FRACTAL, self);
        endCase
    endSelect
} !!
```

```
/* Creates a modeless dialog with a list of demos. */
        !!

inherit(Dialog, #DemosList, #(demos,dlg1 /* An ordered collection
   of strings */), 2, nil) !!

now(DemosListClass) !!

now(DemosList) !!

/* Adds a demo with the specified attributes to the
   demos collection. */
Def addDemo(self, desc, classes, files, mem, blk | aDemo)
{ aDemo := new(Demo);
   init(aDemo, desc, classes, files, mem, blk);
   add(demos, aDemo);
   sendDlgItemMessage(self, DEMOS_LB,  LB_INSERTSTRING, -1, lP(desc));
   freeHandle(desc);
} !!

/* Initialize the Demos list box with the appropriate choices. */
Def initDialog(self, wP, lP)
{ demos := new(OrderedCollection, 10);
   addDemo(self, "soil", #SoilWindow, "classes\soillist.cls", 1000,
   {using(| temp)  temp := new(SoilWindow, ThePort, nil,"What is your soil
type?",nil);
      show(temp,1);
      add(OpenWindows, temp);
   });
   addDemo(self, "Turtle graphics", #Turtle, tuple("classes\turtle.cls",
   "act\turtfrac.act", "act\stars.act"), 1000,
   {
```

```
    if size(parent.turtles) == 0
    then addTurtle(parent, init(new(Turtle), parent))
    endif;
});
addDemo(self, "Track", #(Ellipse, ShapesWindow, TrackWindow),
tuple("classes\ellipse.cls", "classes\shapeswin.cls",
"classes\trackwin.cls"), 1000,
{using(| temp) temp := new(TrackWindow, ThePort, "Track",
    "Actor Track Application", nil);
    show(temp, 1);
    add(OpenWindows, temp);
});
addDemo(self, "HyperCube", #CubeWindow, "classes\cubewind.cls", 1000,
{using(| temp) temp := new(CubeWindow, ThePort, nil,
    "Actor HyperCube", nil);
    show(temp, 1);
    add(OpenWindows, temp);
    run(temp);
});
addDemo(self, "GraphWindow", #GraphDemo, tuple("act\graphcon.act",
"classes\graphdem.cls") , 1000,
{using(| temp) temp := new(GraphDemo, ThePort, nil, "A Graph Window",

    nil);
    show(temp, 1);
    add(OpenWindows, temp);
});
addDemo(self, "Mandelbrot 1", #PlayMandelbrot, "classes\playmand.cls",
1000,
{draw(init(new(PlayMandelbrot), "m1.plt") , parent);
});
addDemo(self, "Mandelbrot 2", #PlayMandelbrot, "classes\playmand.cls",
```

```
        1000,
        {draw(init(new(PlayMandelbrot), "m2.plt") , parent);
        });
        addDemo(self, "Mandelbrot 3", #PlayMandelbrot, "classes\playmand.cls",
        1000,
        {draw(init(new(PlayMandelbrot), "m3.plt") , parent);
        });
        addDemo(self, "Fractal", #Fractal, "act\frac.act", 1000,
        {example(Frac, 5);
        });
        addDemo(self, "N-Queens", #(Queen, QueensBoard, DisplayBoard),
        "act\queen.act", 1000,
        {solve(new(QueensBoard), 8);
        });
        addDemo(self, "Class tree", #TreeNode, "act\clastree.act", 1000,
        {tree(Object);
        });
        addDemo(self, "Actor logo", #ActorLogo, "classes\actorlog.cls", 1000,
        {draw(new(ActorLogo))
        });
        sendDlgItemMessage(self, DEMOS_LB,  LB_SETCURSEL, 0, 0);
    } !!

    /* Responds to command messages. */
    Def command(self, wP, lP | temp str)
    {
        select
            case (wP == DEMOS_LB) and (high(lP) = 2)
                temp := true;
            endCase
            case (wP == RUN_DEMO) temp := true;
            endCase
```

```
        case wP == QUIT_DEMO destroy(parent)
        endCase
    endSelect;
    if temp
    then temp := sendDlgItemMessage(self, DEMOS_LB,
        LB_GETCURSEL, 0, 0);
        ThePort := parent;
        temp := demos[temp];
        if not(defined(temp))
        then load(temp.files);
            if temp.desc = "Turtle graphics"
            then str := "Fractals!";
                changeMenu(parent, 0,lP(str), FRACTAL,  MF_APPEND);
                freeHandle(str);
                drawMenu(parent);
            endif;
        endif;
        eval(temp.runBlock);
    endif;
} !!
```

```
/* Each power object contains information on a soil
    type. */ !!

inherit(Object, #Soil, #(desc /* Description for listbox */
classes /* support Classes  */
files /* support Files */
memReq /* Memory req'd in bytes */
runBlock /* Code to run the necessary programs */), 2, nil) !!

now(SoilClass) !!

now(Soil) !!

/* Return true if soil is defined. */
Def defined(self)
{ ^
   if class(classes) == Symbol
   then Actor[classes]
   else Actor[classes[0]]
   endif
} !!

/* Initialize the soil object. */
Def init(self, descrip, cls, fl, mem, blk)
{ desc := descrip;
   classes := cls;
   files := fl;
   memReq := mem;
   runBlock := blk;
} !!
```

```
/* A window class for the SOIL demonstration programs. */!!

inherit(TurtleArea, #SoilWindow, #(dlg1), 2, nil) !!

now(SoilWindowClass) !!

now(SoilWindow) !!

/* Recreate method for soil window. */
Def   recreate(self)
{ create(self, parent, "soil list", locRect, WS_POPUPWIND);
   init(self);
   ^nil
} !!

/* Initialize the soilWindow. */
Def init(self | cStr)
{ init(self:TurtleArea);
   createMenu(self);
   cStr := "Clear!";
   changeMenu(self, 0, lP(cStr), CLEAR, MF_APPEND);
   freeHandle(cStr);
   drawMenu(self);
   show(self, 1);
   reSize(self, 0, 0);
   dlg1:= new(SoilList);
   runModeless(dlg1, DEMOSLIST,self);
}    !!

/* Respond to command messages. */
Def command(self, wP, lP | aDlg)
{
```

```
    select
        case wP == CLEAR repaint(self);
    endCase
    endSelect
} !!
```

```
/* Creates a modeless dialog with a list of soil types. */

/* march 07, 1989     Andre Plante   */
        !!

inherit(Dialog, #SoilList, #(soils /* An ordered collection
   of soil types */), 2, nil) !!

now(SoilListClass) !!

now(SoilList) !!

/* Adds a soil type with the specified attributes to the
   soils collection. */
Def addSoil(self, desc, classes, files, mem, blk | aSoil)
{ aSoil := new(Soil);
   init(aSoil, desc, classes, files, mem, blk);
   add(soils, aSoil);
   sendDlgItemMessage(self, DEMOS_LB,  LB_INSERTSTRING, -1, lP(desc));
   freeHandle(desc);
}  !!

/* Initialize the Soils list box with the appropriate choices. */
Def initDialog(self, wP, lP)
{ soils:= new(OrderedCollection, 10);
   addSoil(self, "Clay", #Window, "classes\clay.cls", 1000,
   {using(| temp) temp:=new(Window,ThePort,nil,"Clay",nil);
   show(temp, 1);
   add(OpenWindows, temp);
   });
   addSoil(self, "sand", #SandWindow, "classes\powerlist.cls", 1000,
```

```
    {using(| temp) temp:=new(SandWindow,ThePort, nil, "do you have power?",nil);

     show(temp,1);
     add(OpenWindows, temp);
     });

     sendDlgItemMessage(self, DEMOS_LB,  LB_SETCURSEL, 0, 0);
} !!

/* Responds to command messages. */
Def command(self, wP, lP | temp str)
{
   select
      case (wP == DEMOS_LB) and (high(lP) = 2)
         temp := true;
      endCase
      case (wP == RUN_DEMO) temp := true;
      endCase
      case wP == QUIT_DEMO destroy(parent)
      endCase
   endSelect;
   if temp
   then temp := sendDlgItemMessage(self, DEMOS_LB,
      LB_GETCURSEL, 0, 0);
      ThePort := parent;
      temp := soils[temp];
      if not(defined(temp))
      then load(temp.files);
      endif;
      destroy(parent);
      eval(temp.runBlock);
   endif; } !!
```

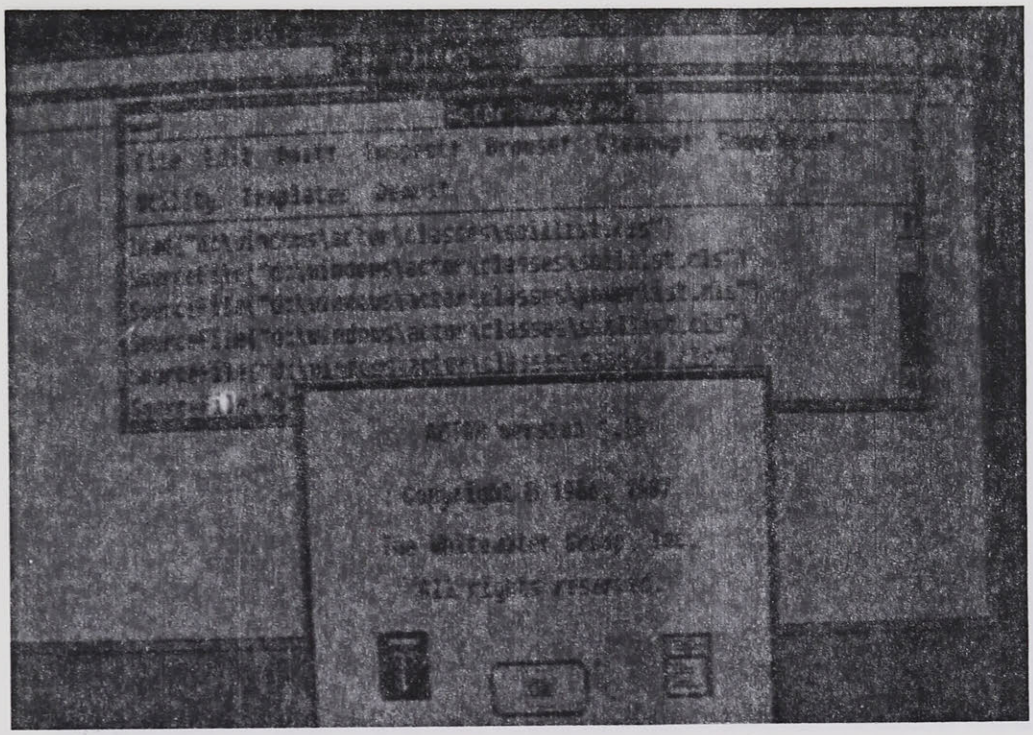
```
/* A window class for the SOIL demonstration programs. */!!

inherit(TurtleArea, #SandWindow, #(dlg2), 2, nil) !!

now(SandWindowClass) !!

now(SandWindow) !!

/* Recreate method for soil window. */
Def   recreate(self)
{ create(self, parent, caption, locRect, WS_POPUPWIND);
   init(self);
   ^nil
} !!

/* Initialize the sandWindow. */
Def init(self | cStr)
{ init(self:TurtleArea);
   createMenu(self);
   cStr := "Clear!";
   changeMenu(self, 0, lP(cStr), CLEAR, MF_APPEND);
   freeHandle(cStr);
   drawMenu(self);
   show(self, 1);
   reSize(self, 0, 0);
   errorBox("SAND", "sand is a good soil for subirrigation");
   dlg2:= new(PowerList);
   runModeless(dlg2, DEMOSLIST,self);
}    !!

/* Respond to command messages. */
Def command(self, wP, lP | aDlg)
```

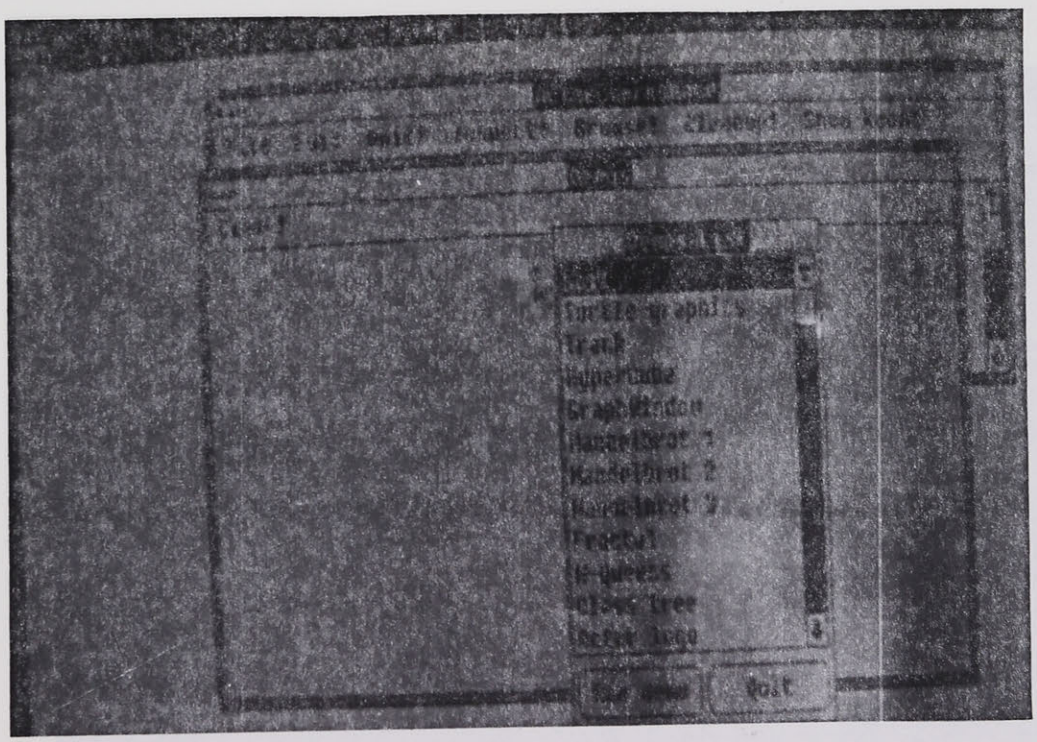
```
{
   select
      case wP == CLEAR repaint(self);
      endCase
   endSelect
} !!
```

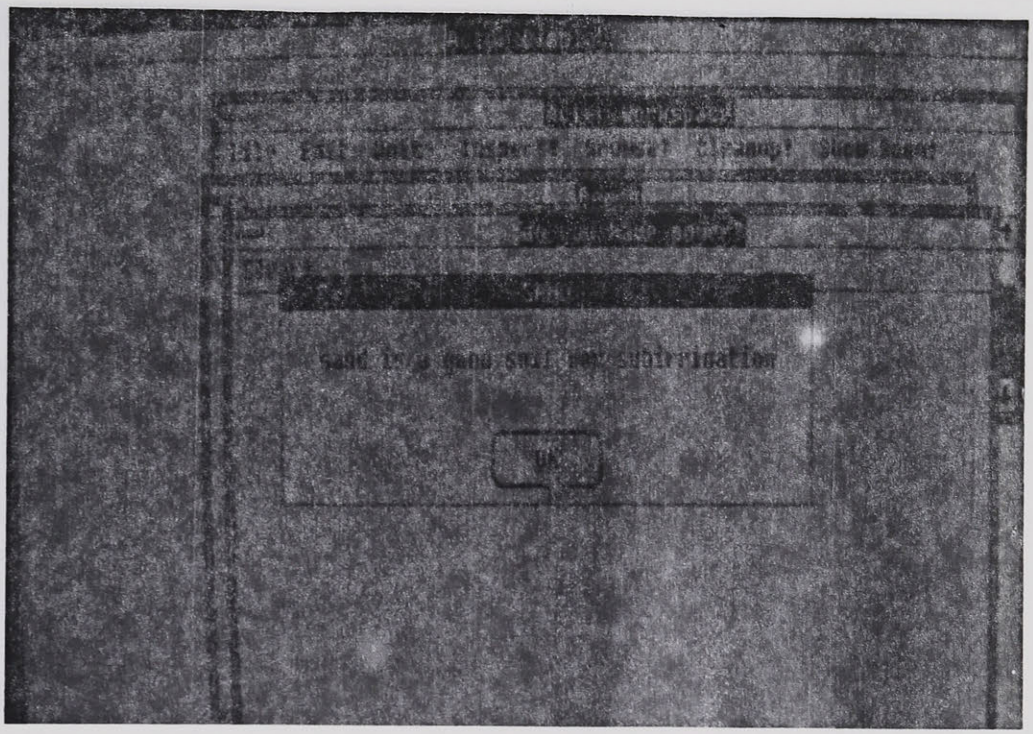
```
/* Creates a modeless dialog with a list of power types. */

/* march 07, 1989     Andre Plante   */

                !!

inherit(Dialog, #PowerList, #(powers /* An ordered collection
   of power types */), 2, nil) !!

now(PowerListClass) !!

now(PowerList) !!

/* Adds a power type with the specified attributes to the
   power collection. */
Def addPower(self, desc, classes, files, mem, blk | aPower1)
{ aPower := new(Power);
   init(aPower, desc, classes, files, mem, blk);
   add(powers, aPower);
   sendDlgItemMessage(self, DEMOS_LB,   LB_INSERTSTRING, -1, lP(desc));
   freeHandle(desc);
}  !!

/* Initialize the power list box with the appropriate choices. */
Def initDialog(self, wP, lP)
{ powers:= new(OrderedCollection, 10);
   addPower(self, "yes", #Window, "classes\clay.cls", 1000,
   {using(| temp) temp:=new(Window,ThePort,nil,"yes",nil);
   show(temp, 1);
   add(OpenWindows, temp);
   });
   addPower(self, "no", #Window, "classes\powerlist.cls", 1000,
   {using(| temp) temp:=new(Window,ThePort, nil, "no",nil);
```

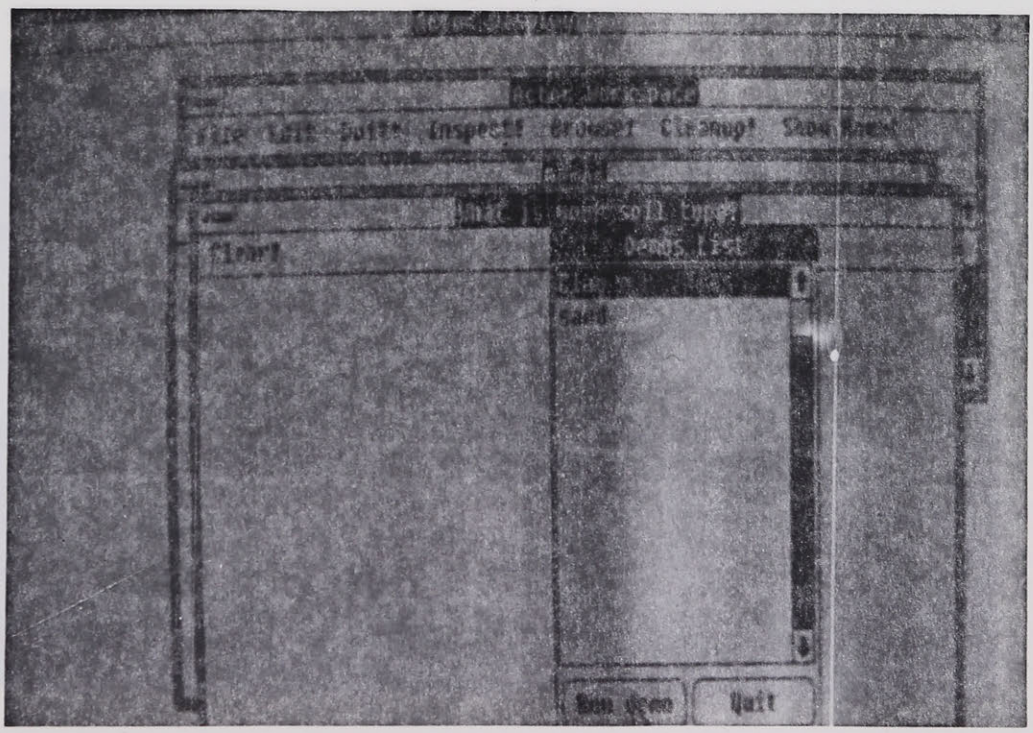
```
    show(temp,1);
    add(OpenWindows, temp);
    });

    sendDlgItemMessage(self, DEMOS_LB,  LB_SETCURSEL, 0, 0);
} !!

/* Responds to command messages. */
Def command(self, wP, lP | temp str)
{
    select
        case (wP == DEMOS_LB) and (high(lP) = 2)
            temp := true;
        endCase
        case (wP == RUN_DEMO) temp := true;
        endCase
        case wP == QUIT_DEMO destroy(parent)
        endCase
    endSelect;
    if temp
    then temp := sendDlgItemMessage(self, DEMOS_LB,
        LB_GETCURSEL, 0, 0);
        ThePort := parent;
        temp := powers[temp];
        if not(defined(temp))
        then load(temp.files);
        endif;
        destroy(parent);
        eval(temp.runBlock);
    endif;
}  !!
```

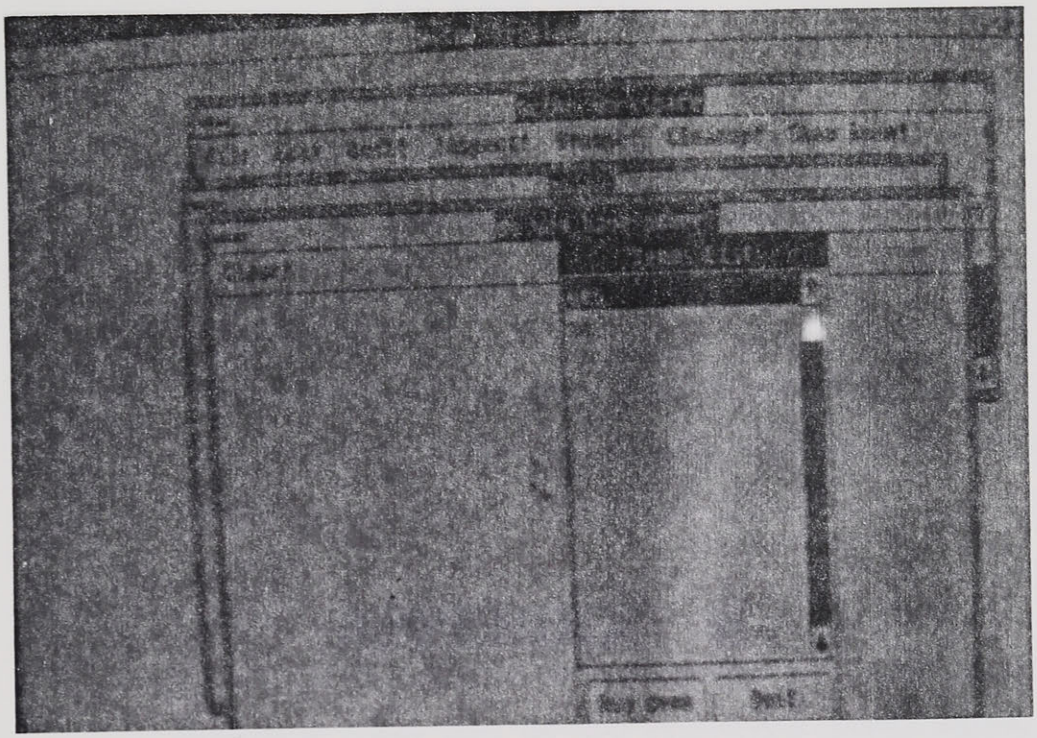
APPENDIX D

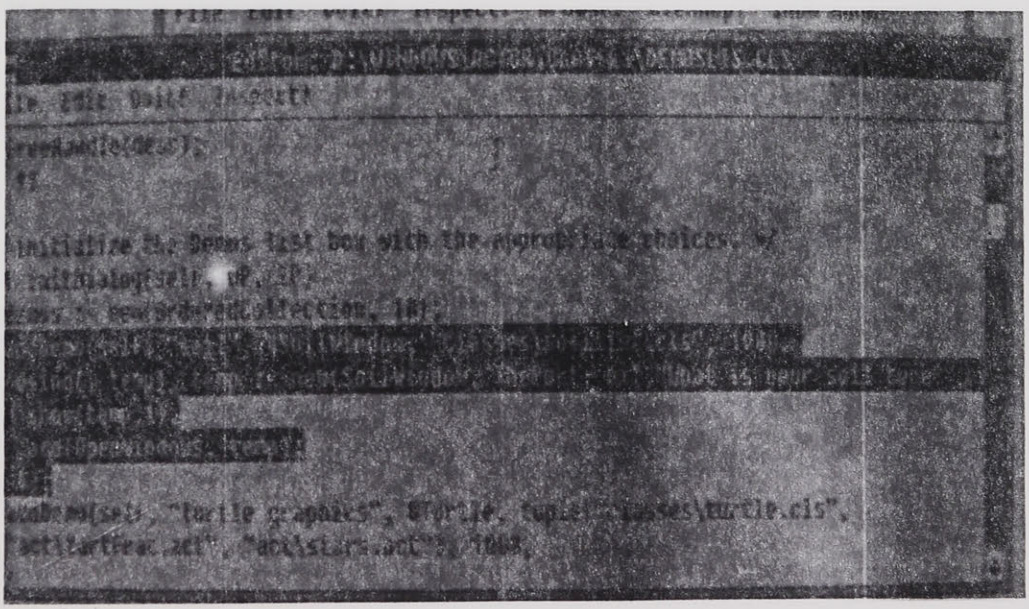












APPENDIX E

GLOSSARY

ALGORITHM

Step by step representation of method solving process.

ANCESTORS

More general classes from which more specialized classes descend, while inheriting their instance variables and methods. Classes have only one immediate ancestor.

CLASS

A category of objects that all have the same functionality and data format. All instances of a class share the same methods and instance variables. By convention, class names begin with a capital letter.

DECISION TREE

Structural form representation of tough process.

DESCENDANTS

Classes created as offshoots from or more specialized versions of another class, or ancestor. The descendant inherits from its ancestor all of the methods and instance variables, but can redefine them. A descendant has only one immediate ancestor, but an ancestor cam have many descendants..

DYNAMIC BINDING

Dynamic binding is a loosely coupled collections where the consumer's code cannot predict the type of data to be operated on until the code is being run.

ENCAPSULATION

From a consumer's point of view, is a seamless capsule that offers a number of services, with no visibility as to how these services are implemented.

INFERENCE ENGINE

This is the part that governs the execution, the knowledge base access and the certainty factor of an expert system.

INHERITANCE

A hierarchical scheme that relates the classes. The higher, or ancestral classes, are more general and the lower, or descendant classes, are more specialized. A class inherits methods and instance variables from its ancestors. The class can then modify the methods or add new ones to become specialized. Actor utilizes single inheritance, meaning that each class has only one immediate ancestor.

LIBRARY

Collection of unattached object. Exactly as books in a library shelf.

LINKING

Combining several object module at compile time in a tightly coupled collection. All desired code must be present at this early stage.

MODULARITY

Relating to module which standardized units for flexibility and variety in use.

OBJECT

An object is some private data and a set of operations that can access that data. An object is requested to perform one of its operations by sending it a message telling the object what to do.

OBJECT-ORIENTED LANGUAGES

Languages that treat data structures as objects belonging to classes. The classes define methods and inherit methods from other classes according to a hierarchical inheritance scheme.

OPERAND

Procedural code that composes the static defense of the object's private part.

RUN TIME

A given function's run time occurs when the function is being executed. One function's run time can be another's compile time.

STATIC DEFENCE

Wall of code which govern the exchange between the world and the objects Private Part.