DEPARTMENT OF COMPUTER SCIENCE
COLLEGE OF SCIENCES
OLD DOMINION UNIVERSITY
NORFOLK, VIRGINIA 23529

INFORMATION TECHNOLOGY: A TOOL TO CUT HEALTH CARE COSTS

By

Dr. Ravi Mukkamala, Principal Investigator

Final Report
For the period ended August 31, 1996

Prepared for
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

Under
Research Grant Number NAG-1-1690
Wayne H. Bryant, Technical Monitor

Submitted by the
Old Dominion University Research Foundation
P.O. Box 6369
Norfolk, VA 23508-0369

September 1996

DEPARTMENT OF COMPUTER SCIENCE
COLLEGE OF SCIENCES
OLD DOMINION UNIVERSITY
NORFOLK, VIRGINIA 23529

INFORMATION TECHNOLOGY: A TOOL TO CUT HEALTH CARE COSTS

By

Dr. Ravi Mukkamala, Principal Investigator

Final Report
For the period ended August 31, 1996

Prepared for
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-0001

Under
Research Grant Number NAG-1-1690
Wayne H. Bryant, Technical Monitor

Submitted by the
Old Dominion University Research Foundation
P.O. Box 6369
Norfolk, VA 23508-0369

September 1996

Information Technology: A Tool to Cut Health Care Costs

Final Report (NAG-1-1690)

R. Mukkamala K.J. Maly C. M. Overstreet E. C. Foudriat
Department of Computer Science
Old Dominion University
Norfolk, Virginia 23529.

Abstract

We report on the work done as part of the NASA LaRC grant NAG-1-1690. As part of this effort, we have designed and built a prototype for an integrated medical record system. MRS (Medical Record System) is written in Tcl/Tk. While the initial version of the prototype had patient information hard coded into the system, the later versions used an INGRES database for storing patient information. Currently, we have proposed an object-oriented model for implementing MRS.

As a result of the initial seed money by NASA LaRC, we have achieved a level of expertise in this area. As a consequence, we have recently been awarded two projects by EVMS (Eastern Virginia Medical School), CHKD (Children's Hospital of the King's Daughters) and CPR (Center for Pediatrics Research), all located in Norfolk, Virginia. These projects involve developing information systems for physicians and medical researchers to enhance their ability for improved treatment at reduced costs.

1 Introduction

Old Dominion University embarked on a project to see how current computer technology could be applied to reduce the cost/and or improve the efficiency of health care. Subsequent discussions with local hospital officials (Children's Hospital of King's daughters and Eastern Virginia Medical School) indicated that the health care industry was already well computerized and getting more so every day. The move to computerize patient records is also well underway, several standards exist for laboratory records [5] and several groups are working on standards for other portions of the patient record [1, 3].

Even though doctors realize that the move to electronic records is eventually going to happen in the near future, they do not relish the thought of giving up their paper records for the electronic variety. Paper records have many advantages over electronic records: the ability to display multiple formats; immediate access to required/applicable data; high resolution for X-rays and the like; unlimited display area; and reasonable confidentiality and security. To some degree these aspects of a paper record can be equaled by electronic records and when combined with its strengths such as access to information that is current and accessible anywhere a terminal exists (on a network, of course); support for a wide

variety of information formats (including audio/video); and an ability to massage data to provide a different view on a patient's record (e.g. trend analysis). The computerized record becomes not only a database but also an analysis tool.

One of the major complaints that doctors have about the current Computerized Patient Record systems (CPRs) is their inability to provide the same flexibility that a paper record system provides in scanning the documents pertaining to a patient. It is common practice for a doctor to quickly flip through a record to become acquainted with the patient's medical history and determine which facets are relevant to the patient's current problems. Computerized records may yet prevail over manual record keeping methods if they provide "one-stop-shopping." That is if it is possible to spare the physician from accessing information from several locations (chart pulls, phone calls, fax machines, etc.) they will be more inclined to use a computerized record [3, 6, 8, 10]. Most medical informatics software provide some capabilities to relate patient health documents together but none have the ability to graphically display the the entire patient record and the relationships between each document in a patient record [6, 7]. Because of this deficiency, it was decided to focus on this issue and develop an application that allowed complete and rapid access to a patient's electronic medical record.

Based on our earlier experience in designing decision support systems, it was believed that a similar problem-oriented organization of the patient record and graphical interface could be developed to give freer, quicker and more complete access to a patient record. With this intention, we have designed and prototyped our system, MRS or Medical Record System. It was decided to prototype MRS in an application in an interpreted language, Tcl/Tk, in order to easily accomodate requested changes and additions made by end users and other interested parties.

This report discusses the motivation behind the project, some aspect of the design and elaborates on the unique features of the application.

2 TCL/TK: The PROTOTYPING TOOL

The current prototype of MRS is implemented using Tcl/Tk, an interpretive language. Tcl or Tool Control Language is a scripting language that features string, list and array manipulation functions as well as the usual commands expected from a Unix shell. The Tcl shell or tclsh can be used like any other Unix shell (e.g., C-shell). An extension to Tcl which allows it access to GUI widgets and event-driven programming is the Tk toolkit. The combination of Tcl and Tk is embodied in the wish-shell which, like tclsh, interprets any commands sent to it. Commands are similar to Unix shell script commands with additional commands and capabilities that give a C-like flavor.

A unique feature of Tcl/Tk is that the interpreter can have its command set augmented either through pre-compiled extensions or through user defined procedures that have been written in C. For example, several extensions are available such as TkPixmap and TkEmacs. By integrating one or more of these extensions into an interpreter or wish-shell one can gain the ability to manipulate Pixmaps (TkPixmaps), or incorporate emacs into an application (TkEmacs). Tcl/Tk is very flexible in that C code can be written by the programmer to

provide new commands. Hence, if speed were required, compiled code could replace a long interpreted tcl procedure.

Like other X-window applications, resource files can be used to determine the properties of a Tcl/Tk application. Tcl/Tk also provides an auto-loading function that searches a user defined list of paths for files containing the procedure it needs. Tcl/Tk provides debugging support in addition to error control. Debugging and development is made easier simply by the fact that it is an interpreted language.

3 SYSTEM DESIGN

Initially, the objective was to provide a front-end viewer for an existing database that would enable doctors to have better access to an existing computerized patient record system (CPR). However, in order to successfully integrate the problem-oriented approach to the CPR, it was realized that an entire medical record system needed to be developed. This is because there is no way to get or generate problem information from existing records. Further, a standard means of entering test data would be faster and more accurate than editing data files. The problem oriented approach, it is hoped, would enable doctors to track actions taken to diagnose, treat, or maintain a particular problem, and to access all related documents quickly and to view them.

Our initial ideas for the MRS came from another ODU project called DHC. Essentially, DHC allowed the description of problems or tasks to be broken down into subproblems, which in turn could be described and broken down. The problems and subproblems were graphically displayed with each subproblem in a box and a link (an arrow) made between the parent problem and its subproblems. Applying this to an CPR would require defining a problem and subproblems then making links between related documents. One could display this graphically, giving the doctor an overview of several problems the patient had experienced in the past and the doctor would be able to zero in on the documents he wanted to see. Clicking on the icon representing a document would bring up the document text, picture or audio annotation.

The first version of MRS exposed unforeseen complications during the conceptual phase. For instance, the use of arrows to link related documents was found to be impractical as it is possible that a single document were related to several problems or sub-problems. It would not be long before the number of arrows on a screen would obscure other documents or make links unclear. Another problem encountered was with the links themselves; there was no common means of linking documents because the relationship between them varied due to the content and document type.

Discussions with potential end users (i.e., doctors) on how better to graphically represent the data resulted in several excellent ideas that were captured in the Zone Concept. As seen in Figure 4, for example, the Zone Concept can display documents by their type or temporal status and arrows have been eliminated in favor of using color to represent links (unfortunately, colors are not visible in Figure 4). Clicking on a problem would instantly colorize all those related to the particular problem. Links would be established using the single link definition given above. Dynamic linking by keywords can be gener-

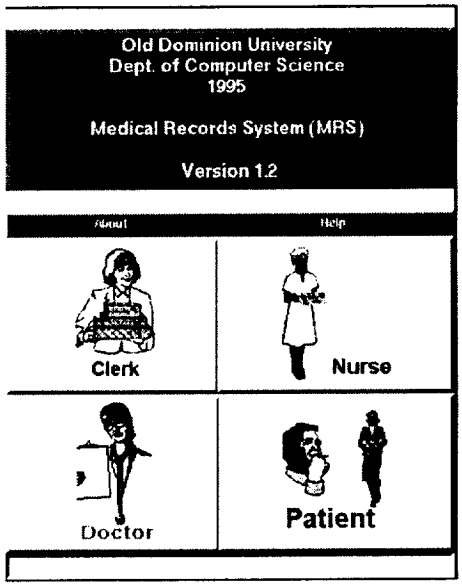


Figure 1: Roles supported by MRS

ated by searching all documents and colorizing those with positive hits. If the number of documents in a column cannot fit the available viewing area, they can be made to overlap. The zone concept as described above was used to create the graphical interface which is described below.

4 A Tour Through MRS

The current version of MRS implements the graphical document interface but also includes many of the functions of the average medical informatics program such as scheduling, billing, triage, and adding records. MRS was designed with the idea of limiting access to certain users and with built-in security [4]. Upon start-up, the user is presented with a screen with four large buttons representing the roles in the hospital supported by MRS (Figure 1). The user selects an appropriate button and then enters the name and password to gain access. As part of the security, different roles within the hospital/doctors office are given different degrees of access. The highest access goes to the doctor who can use all the functions. The lowest is the patient who can only access his own patient record, fill out forms (via Visit or First Visit) or check on his billing. Further, within each function, access to certain features is limited by what position the user occupies. Figure 2 shows the main menu and the associated choice menu for the doctor. We will now describe some of the options.

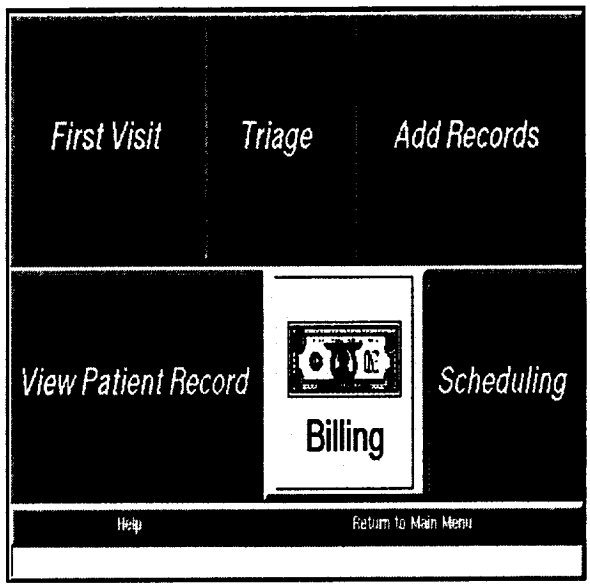


Figure 2: Doctor's Main Menu

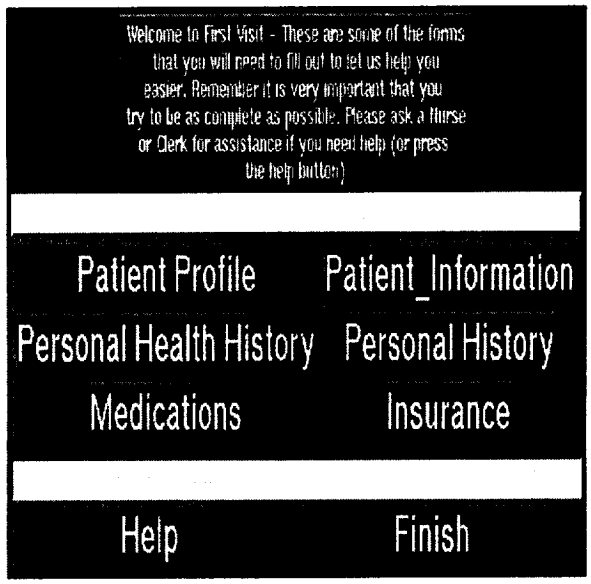


Figure 3: First Visit Options

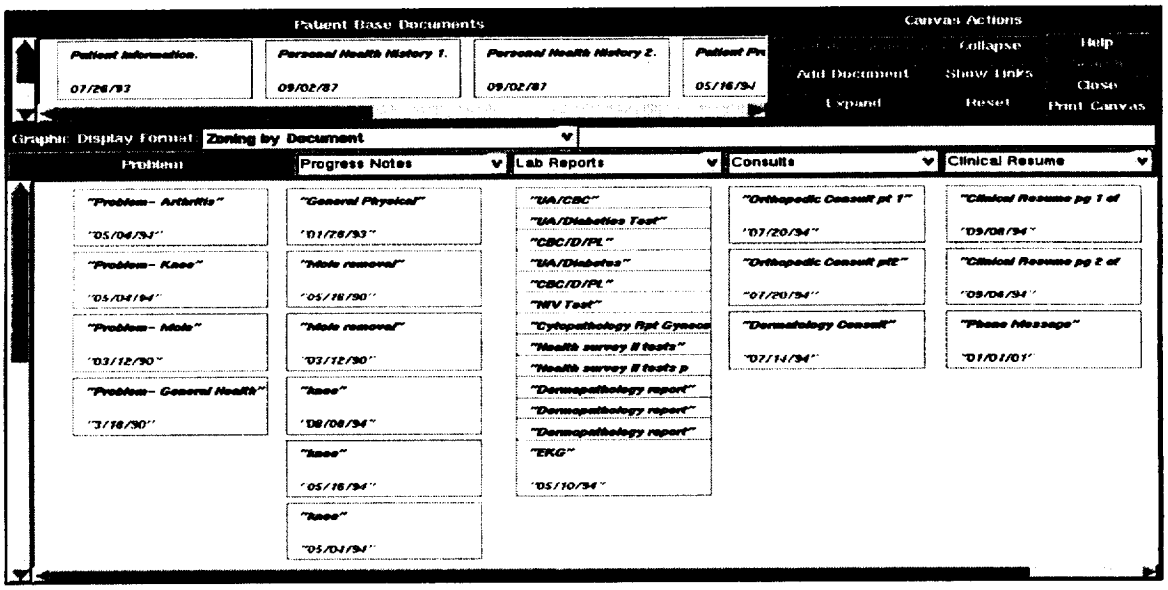


Figure 4: Patient Record Display

4.1 First Visit

A recent trend in medical informatics software is to allow the patient to fill out his/her own forms directly into the computer [2]. First Visit allows the patient or office personnel to fill out the forms needed to begin his medical record such as Personal Health History, and Patient Information (Figure 3). The doctor/nurse/clerk can interview the patient and type in the answers or the patient can do it himself. During this time the patient will receive a password so that he can see his medical record using the View Patient Record option, fill out any forms required for the visit (e.g., a form for describing complaints, symptoms) using the Visit option, and check on his outstanding balance using the Billing option. For patients who remember their password, the ability to enter data into forms allows them quicker service as their complaints enter the system quicker than those who must be interviewed by a nurse or doctor. Further, paperwork for the hospital or doctor's office is also reduced, which improves patient-care.

4.2 Triage

This option is used to collect the patient's vital signs and determine the patient's complaints, and history related to his current condition. As such this option is only available to the nurse and doctor. Currently, the triage option can be called when viewing the patient record or by selecting it from the intermediate menu. If selected from the intermediate menu, the doctor must fill in the patient's name first. After 3 letters, the computer looks

up the names of existing patients and fills in the remainder if a match exists. When called from the View Patient Record Iconbar, the name is automatically entered once the patient has been selected using the Select Patient option. Double clicking on any one of the vital signs (e.g. weight, temperature) will cause a trending chart to be displayed of vital sign vs time . Once the triage has been saved it can no longer be edited. However, the doctor can add to the triage by creating a progress note. The triage data is inserted into the progress note automatically.

4.3 Add Records

The user, most likely a nurse, would like to add documents to a patient record. The way it is added varies, however, on the form of the document. For instance, a scanned document such as an X-Ray or consult report can be added by specifying the filename, date initiated, date received, author, document type (e.g., Progress Note, Form 033) the associated problem and adding descriptive keywords. A document, such as the one created in the office, and hence available in electronic form, is added the same way but no keywords need be entered. During a search, the entire contents of the electronic document will be searched for keywords. Of course, a title for the file icon must also be entered in both cases.

4.4 View Patient Record

Once this option has been selected, an iconbar appears. The iconbar consists of an entry to hold the patient name and several buttons that activate various functions. Most functions such as triage, display patient record, forms and problems will not work until a patient is selected. Pressing the Select Patient button will bring up a Listbox holding patient names. Selecting a patient will fill in the *Patient:* entry of the iconbox. Clicking on triage brings up the triage screen. If it had been filled out previously by a nurse, the triage data would appear for the selected patient. This gives the doctor a synopsis of the patient's current condition and complaints. Clicking on *Form* brings up the Form Listbox with all the potential forms the doctor may fill out when examining the patient. Selecting a Form will bring up the corresponding screen such as a Progress Note or a Form 033 (A standard insurance billing form).

Selecting Display Patient Record brings up a graphical display of the entire patient record (Figure 4). Each document of the patient's record is represented as a file icon. A file icon consists of a rectangle with a descriptive title and a initiation date. Clicking the 2nd mouse button on the title will reveal the key words associated with the document; doing the same on the date will give the receive date for the document. Double-clicking with the first mouse button will cause the document to appear whether it be scanned or in text. General records which are useful to have available at all times, such as Patient Information, and Current and Temporary Medications are kept in the topmost canvas. Other records with linkages to problems appear in the lower canvas. The zone concept described previously was utilized to organize the patient record. Problems are in the first column(left most column) while, if in the Zone-By-Document mode, each zone contains any number of the same type of document. For example, in Figure 4, Progress Notes are

seen in the 2nd column (or zone). At this time, only the Zone-By-Document mode has been implemented but a Zone-By-Time option is planned.

The doctor can quickly locate the related documents to a problem by activating the Show Links function which is activated when the Show Links button is pressed. Once activated, the cursor can be put over a file icon and mouse button one clicked. If the file icon was one of a document then the problem(s) associated with the document change color. If it was a problem then all the documents related to that problem change color. The doctor knows instantly which documents are relevant and should be reviewed. Hence in a matter of 2 button clicks the doctor has narrowed considerably the scope of his search. He may then inspect the keywords associated with the document or use the type of document to guide him as to whether or not to view the document. This is considerably faster and more efficient than browsing a paper record.

To show a patient record better, the expand and compress functions allow the user to spread the file icons out or overlap them, respectively. Also the user has the option of deciding which documents go in what zone. Each zone has a pull down menu that is used to select the document type (Zone-By-Document mode) or date (Zone-By-Time mode) that the zone will represent. Instantly, the zones change to reflect the user's preferences.

4.5 Scheduling

This option is available to the doctor, nurse and clerk but in varying degrees. For instance, only the doctor has the ability to block off sections of time that he wishes not to see any patients. All can make appointments for patients, although this will generally fall to the nurse or clerk. All can examine the schedule either on a daily monthly or yearly basis. When making an appointment, the user has the ability to visually inspect the schedule and insert the appointment or he can conduct a search for all the schedulable time slots using length of time needed and limiting the search to the timeframe that the appointment must be made.

4.6 Billing

Billing functions given in this option are very simplistic since it is not our intention to develop the administrative portion of a medical informatics application. If required several commercial add-on programs can be used. This option is primarily for the clerk; however, the doctor may wish to inspect/modify the billing data. Data from various entries found in forms (e.g, the form 033) will be used to compile billing data for each patient. Clerks can accept payment and subtract from the outstanding balance. As mentioned earlier, the patient may also access his own billing records in a read-only mode.

5 CONCLUSION

Old Dominion University has implemented a Medical Informatics application called the Medical Record System (MRS) that introduces a new paradigm to the Computerized Pa-

tient Record. A patient's problems are used as the basis to organize and link the documents in a medical record. Through the use of a graphical interface the patient's medical record can be browsed much easier than could the corresponding paper record thus removing a major drawback to CPR's. This work is not yet complete, some options still need to be implemented; however, in its current state, MRS has proven the problem-oriented concept and the associated graphical interface to be very effective. Further development is planned and progress should be rapid given the exceptional ease that the Tcl/Tk language can be used to implement supporting data structures and graphical interfaces.

References

[1] Cowey, H. D. and Bernstein, R. (1994). Data Models, Standards and Nomenclatures for Primary Care: The Data Standards Project. 1994 HIMSS Proceedings, Vol 4, 49-59.

[2] Cushing, M. (1995) The data entry conundrum. M.D. Computing, 12, (4), 259-261.

[3] Dicks, R.S. and Steen, E.B. (Editors) (1991). Computer-based patient record: An essential technology for health care. National Academy Press, Washington, D.C.

[4] Donaldson, M.S. and Lohr, K.L. (Editors) (1994). Health data in the information age: use, disclosure, and privacy. National Academy Press, Washington, D.C.

[5] Hammond, W. E., et al. (1994). Computer Standards: Their Future Within Health Care Reform. 1994 HIMSS Proceedings, Vol 1, 33-52.

[6] Henkind, S. J. (1994). Physician Involvement in Information Systems: An Overview of the Issues. 1994 HIMSS Proceedings, Vol 2, 33-41.

[7] Kahn, M.G. (1993). The computer-based patient record and Robert Fulghum's 16 principles. M.D. Computing, 10, (2), 253-261.

[8] Shortliffe, E.H., Perreault, L.E., Widerhold, G., and Fagan, L.M. (1990). Medical Informatics: Computer Applications in Health care. Addison-Wesley.

[9] Walsh, B. (1995). Using Tcl/Tk. Linux Journal. Feb. 1995, 26-33.

[10] Weed, L.L. (1993). Medical records that guide and teach. M.D. Computing, 10, (2), 100-114.

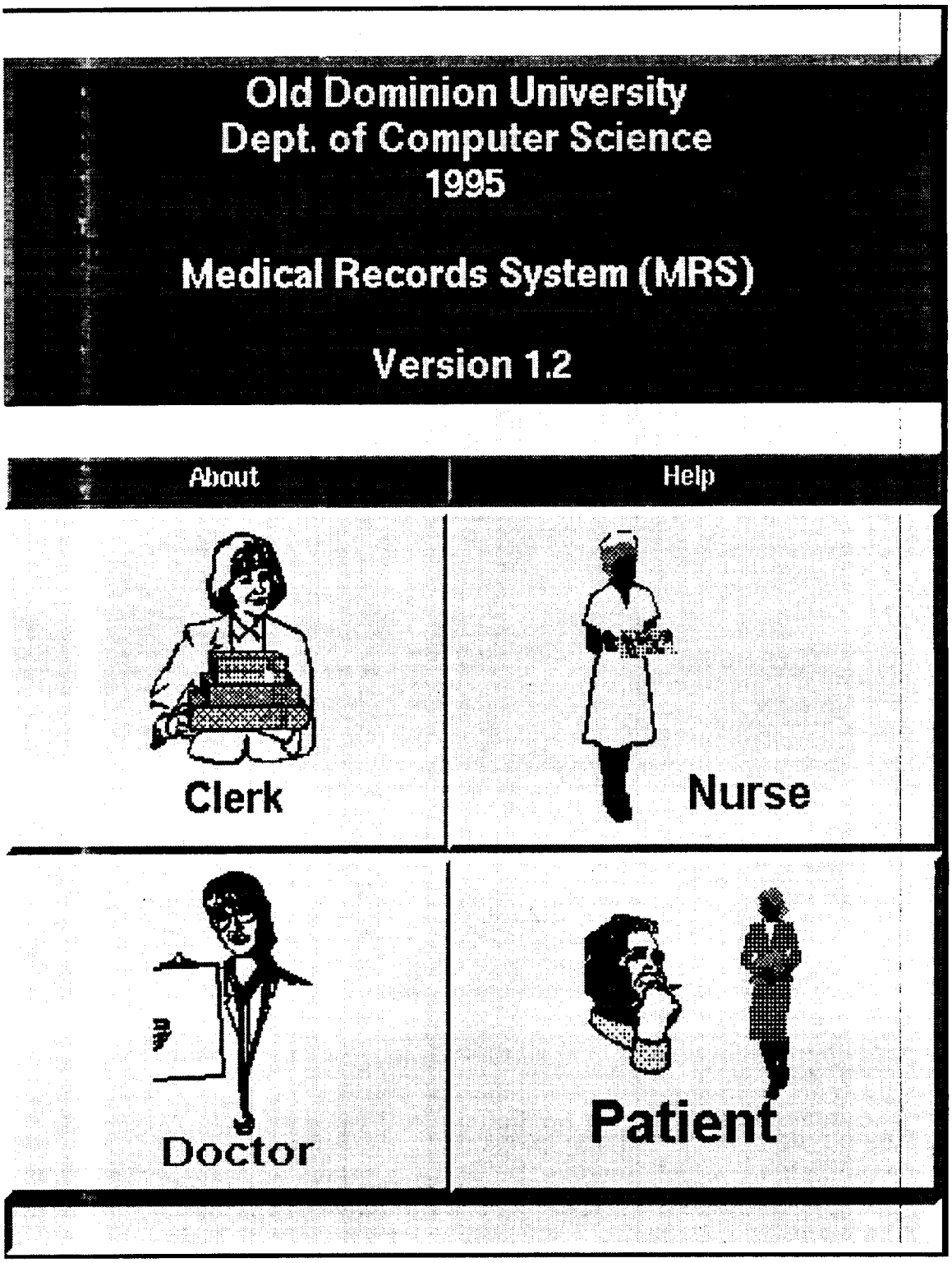


Figure 1: Roles supported by MRS

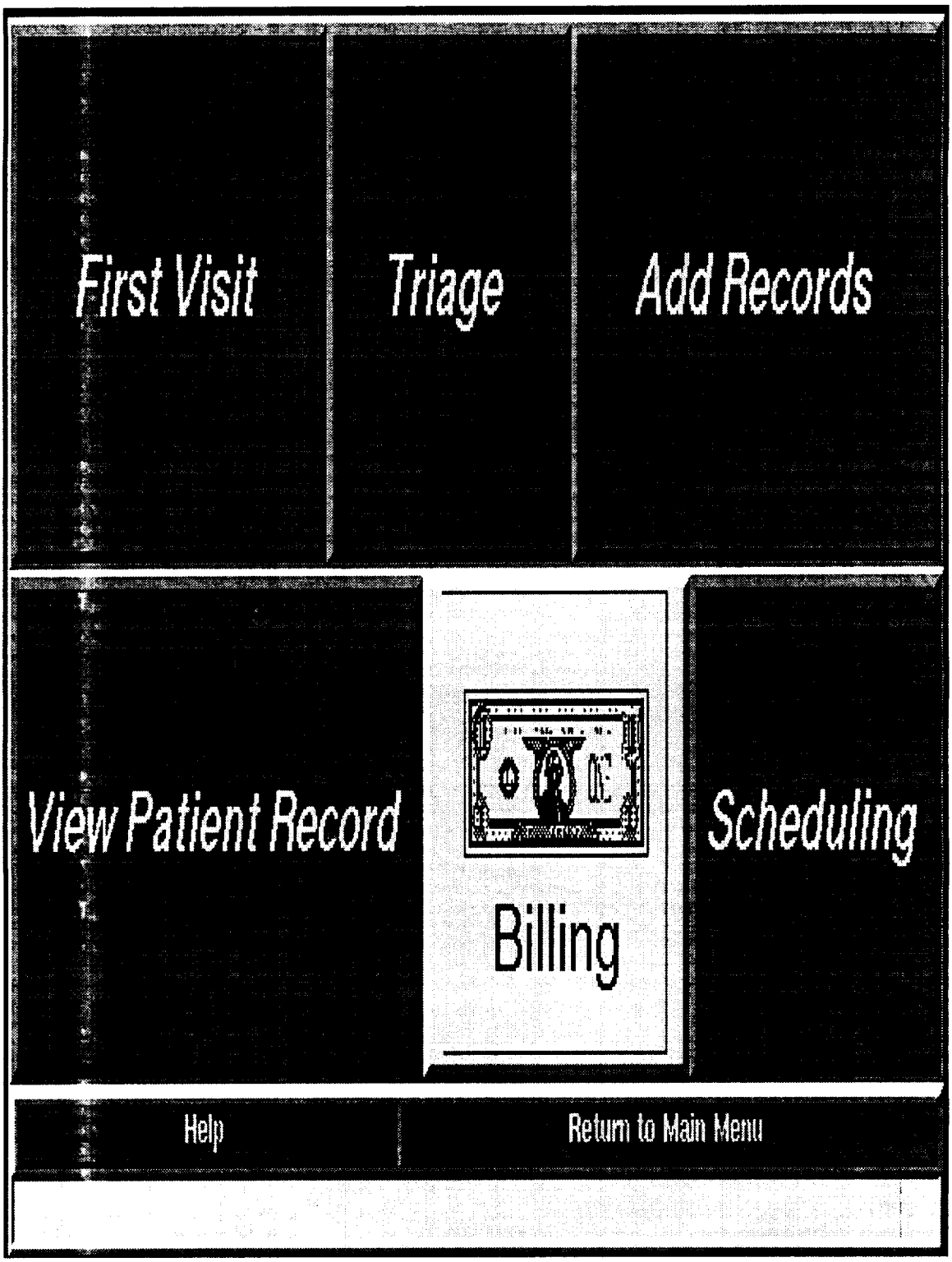


Figure 2: Doctor's Main Menu

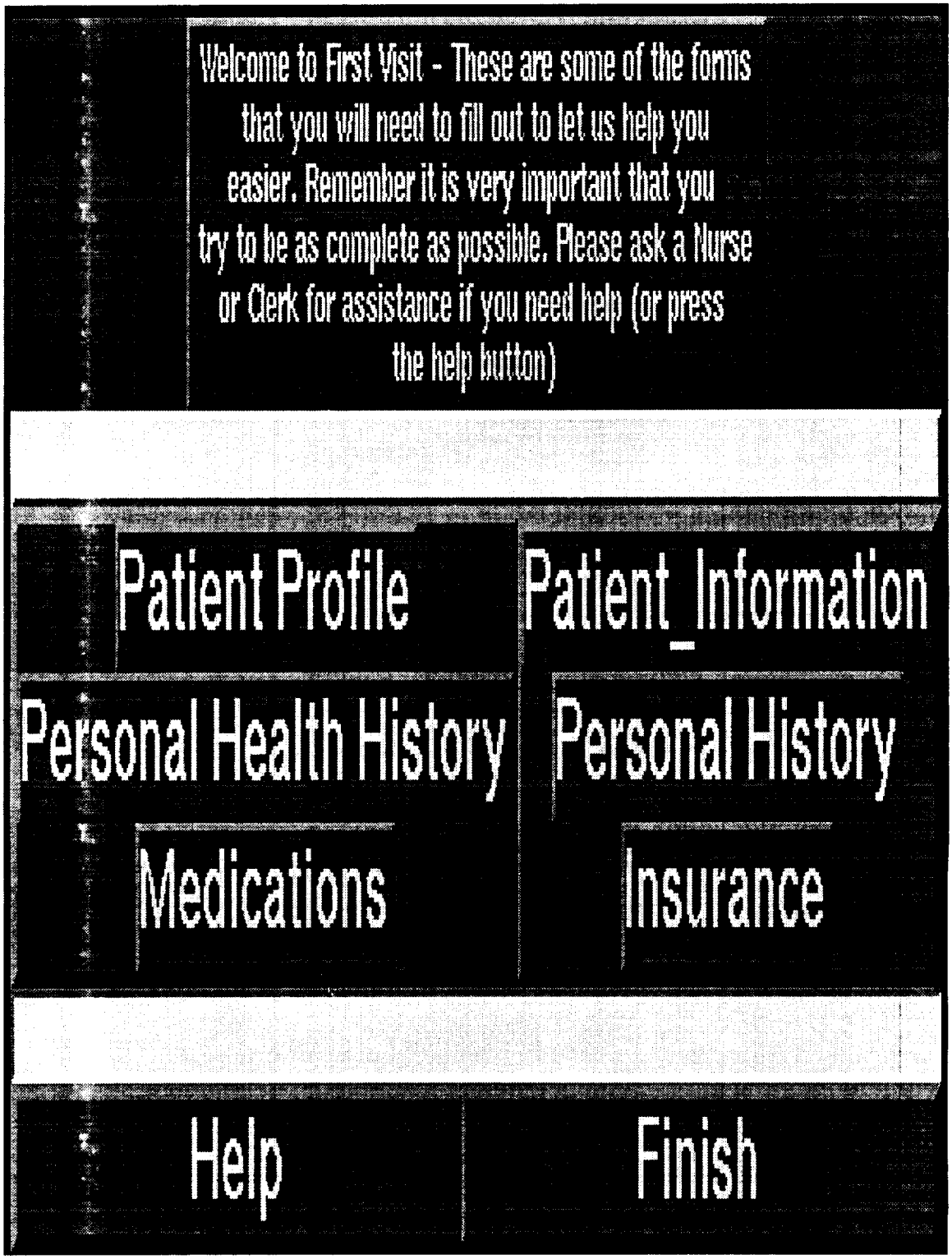


Figure 3: First Visit Options

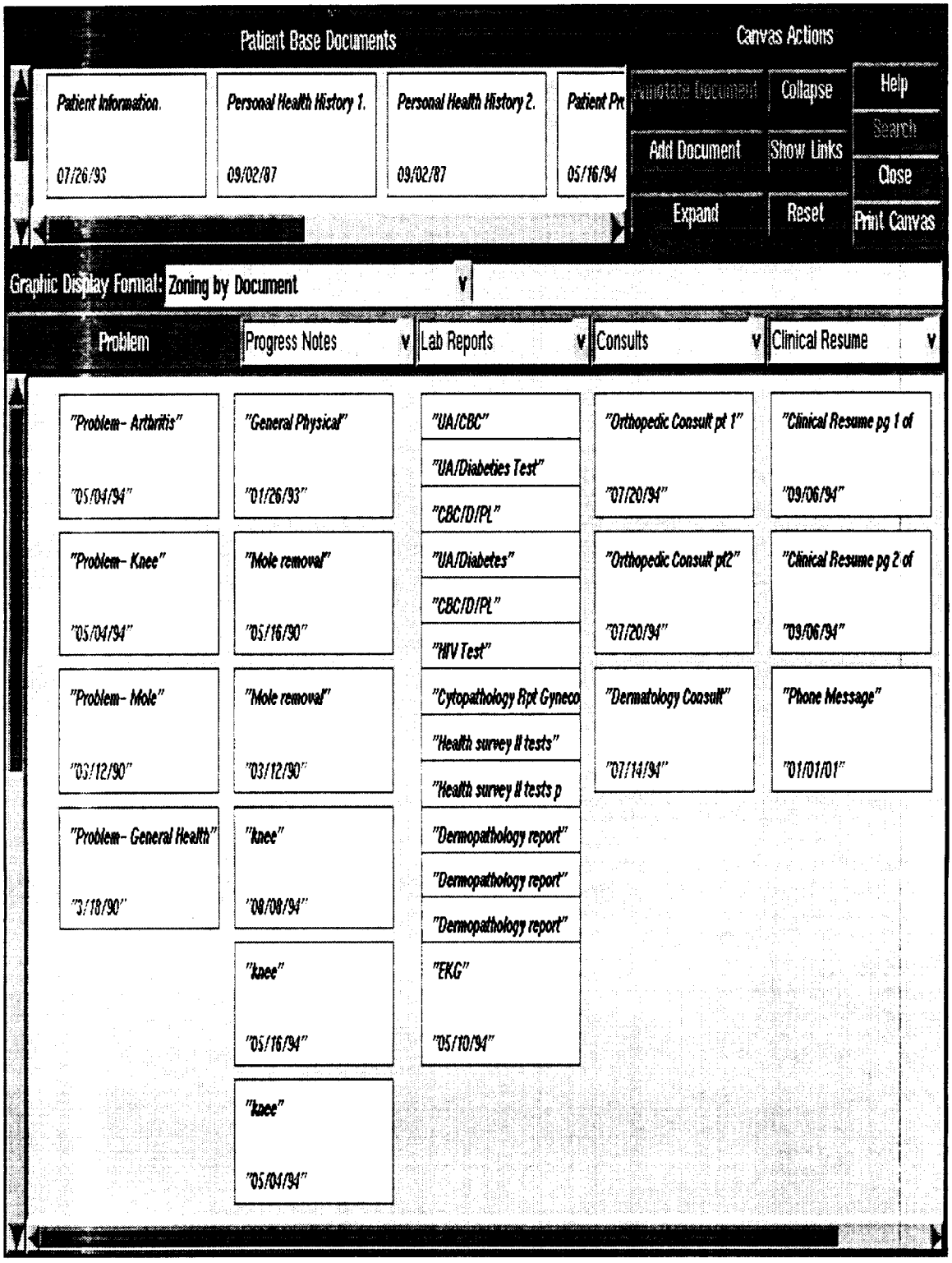


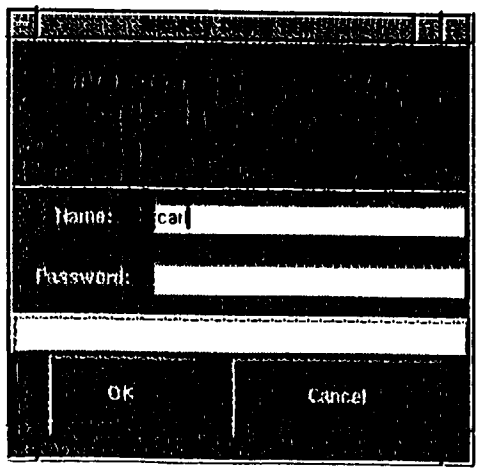
Figure 4: Patient Record Display

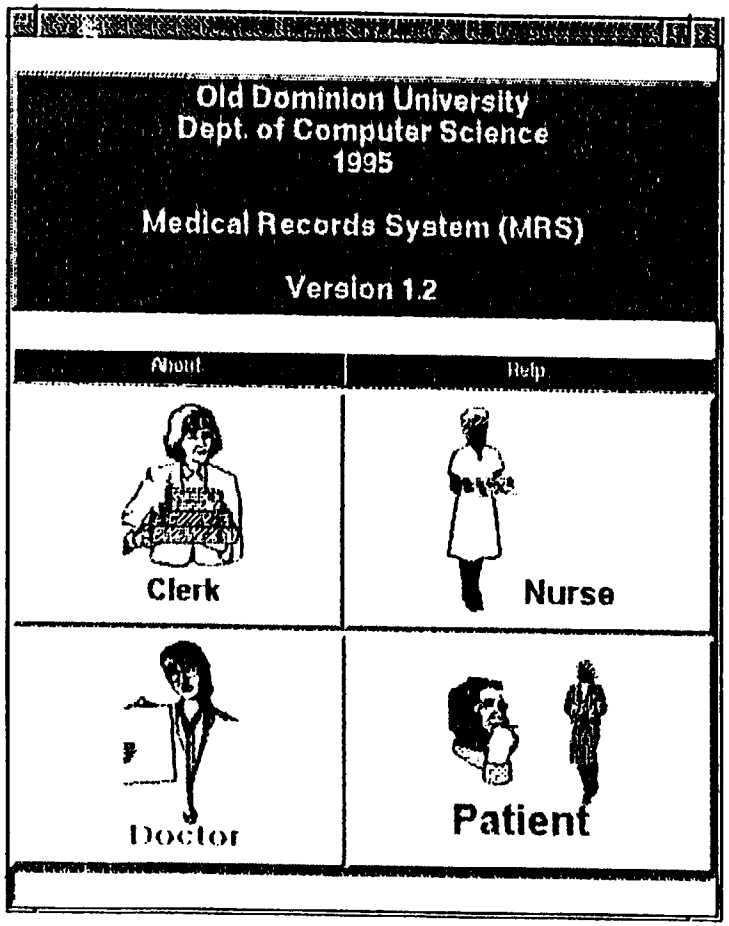
Decision-Based Support System For Medical Professionals

OBJECTIVE: Improve Effectiveness in Time & Quality

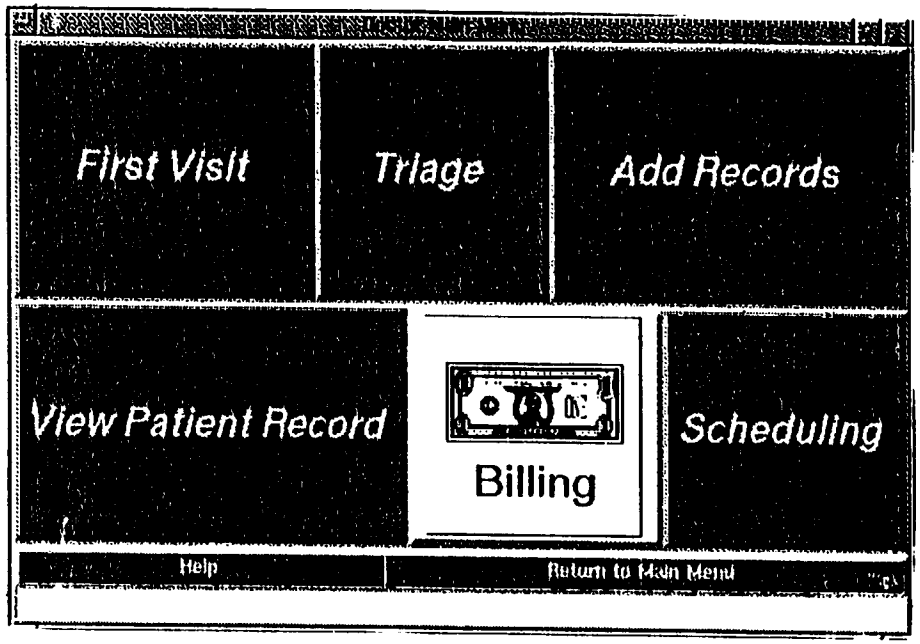
- **Consistent, Fully-Informed Decision Making**
- **Customization of Interface**
- **Integrates with other Tools/Databases**
- **Billing and Insurance**
- **Malpractice**

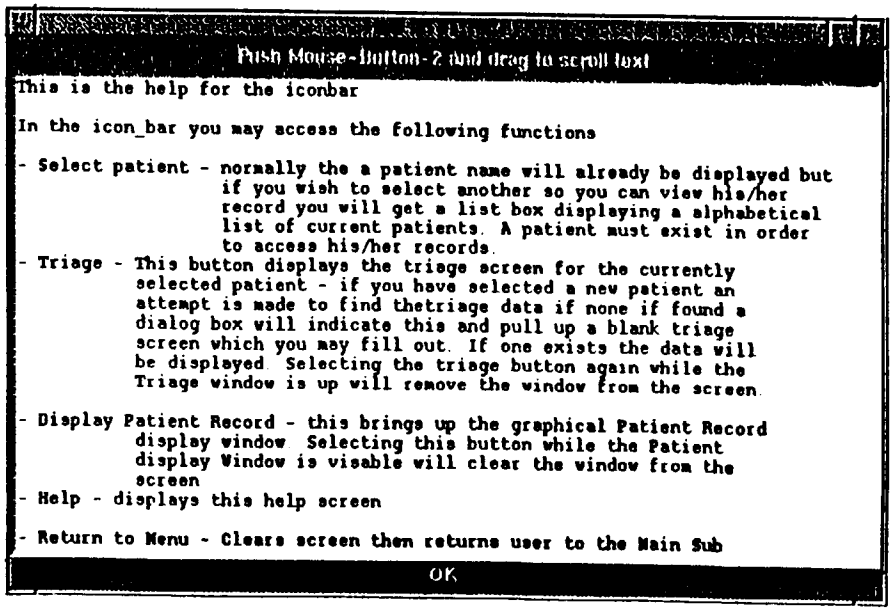
Medical Record System



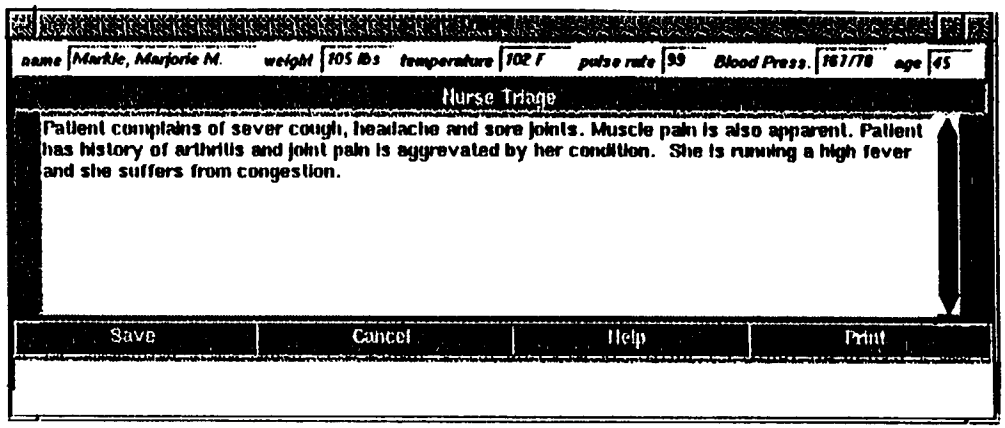


Medical Record System

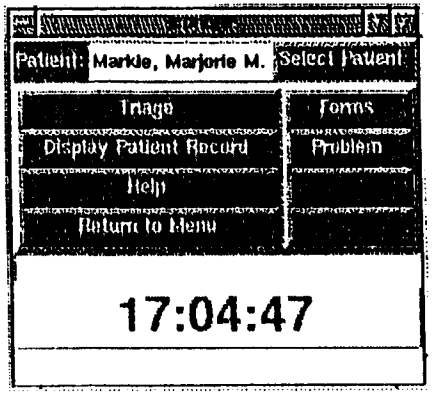


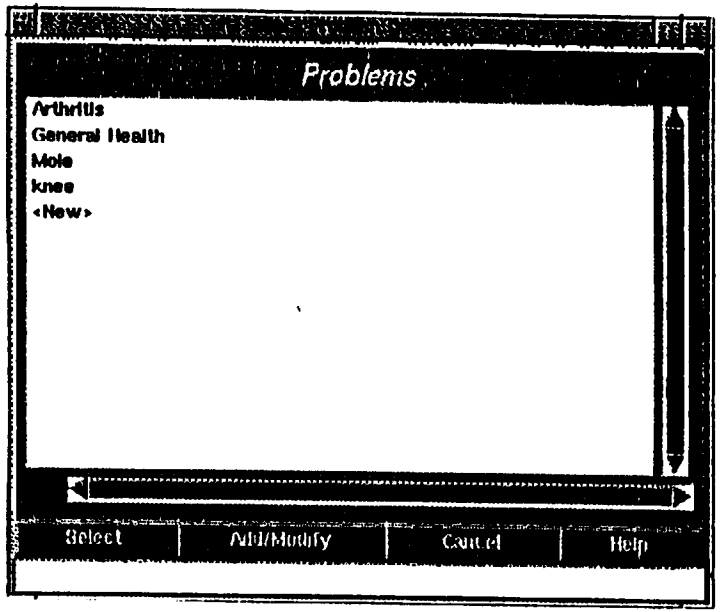


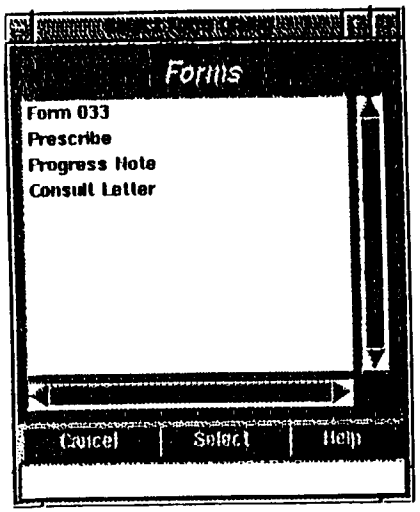
Medical Record System

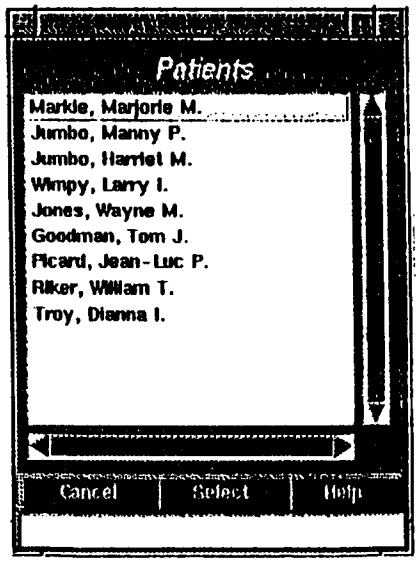


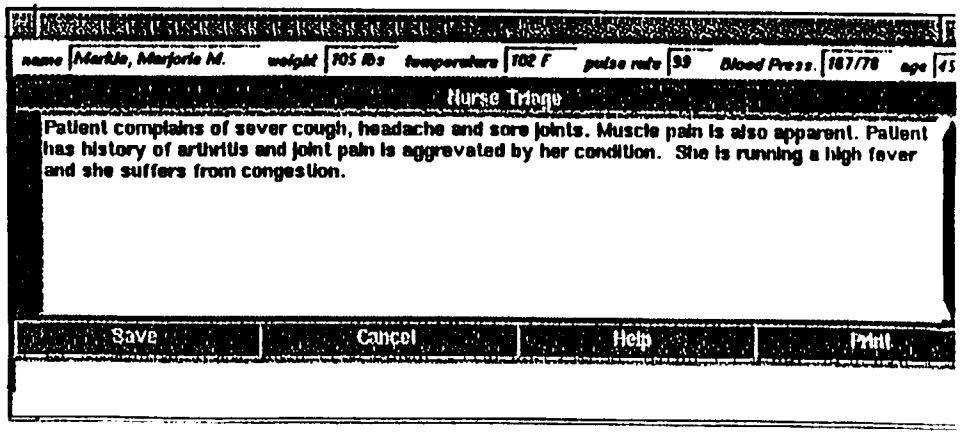
Medical Record System



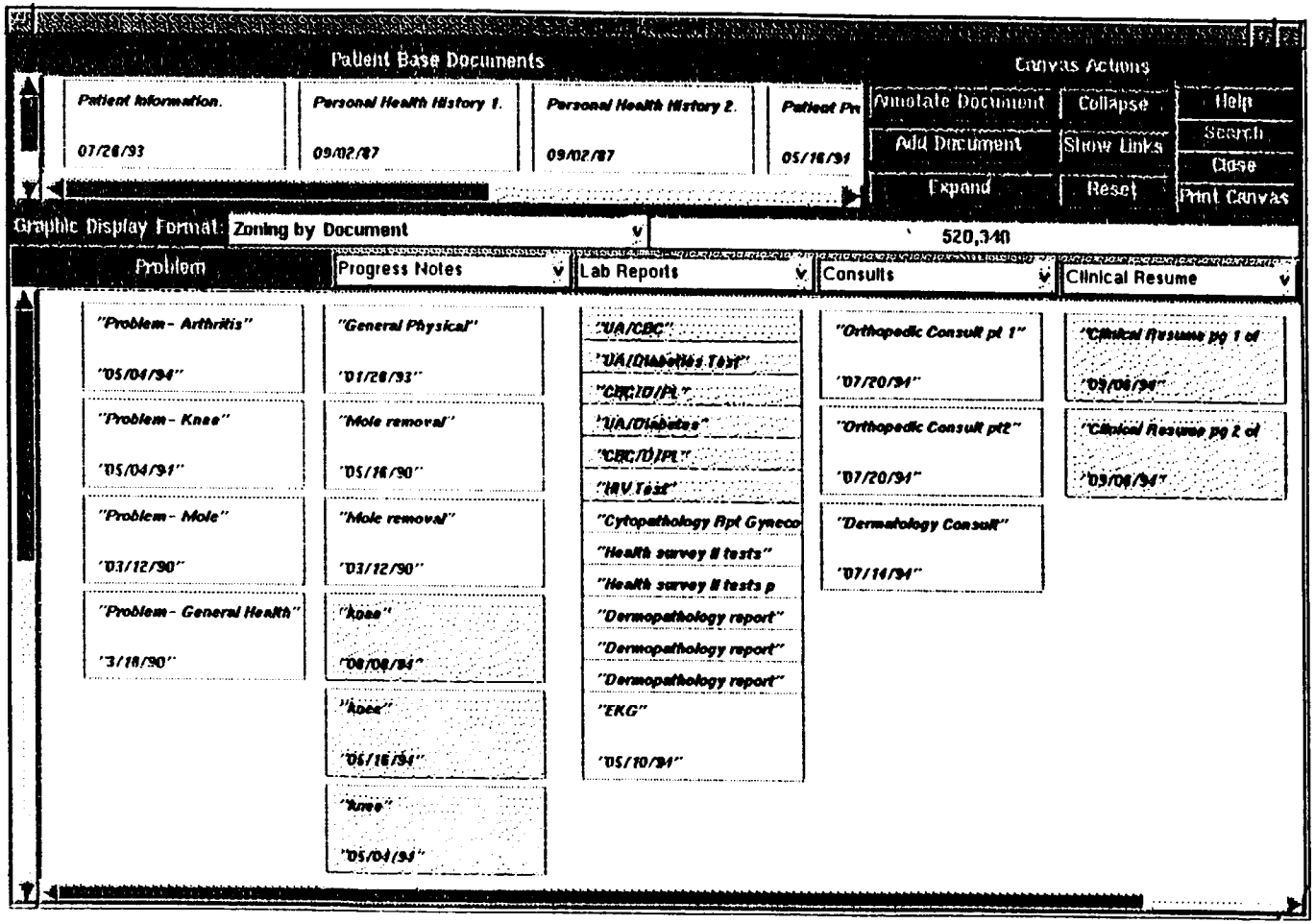




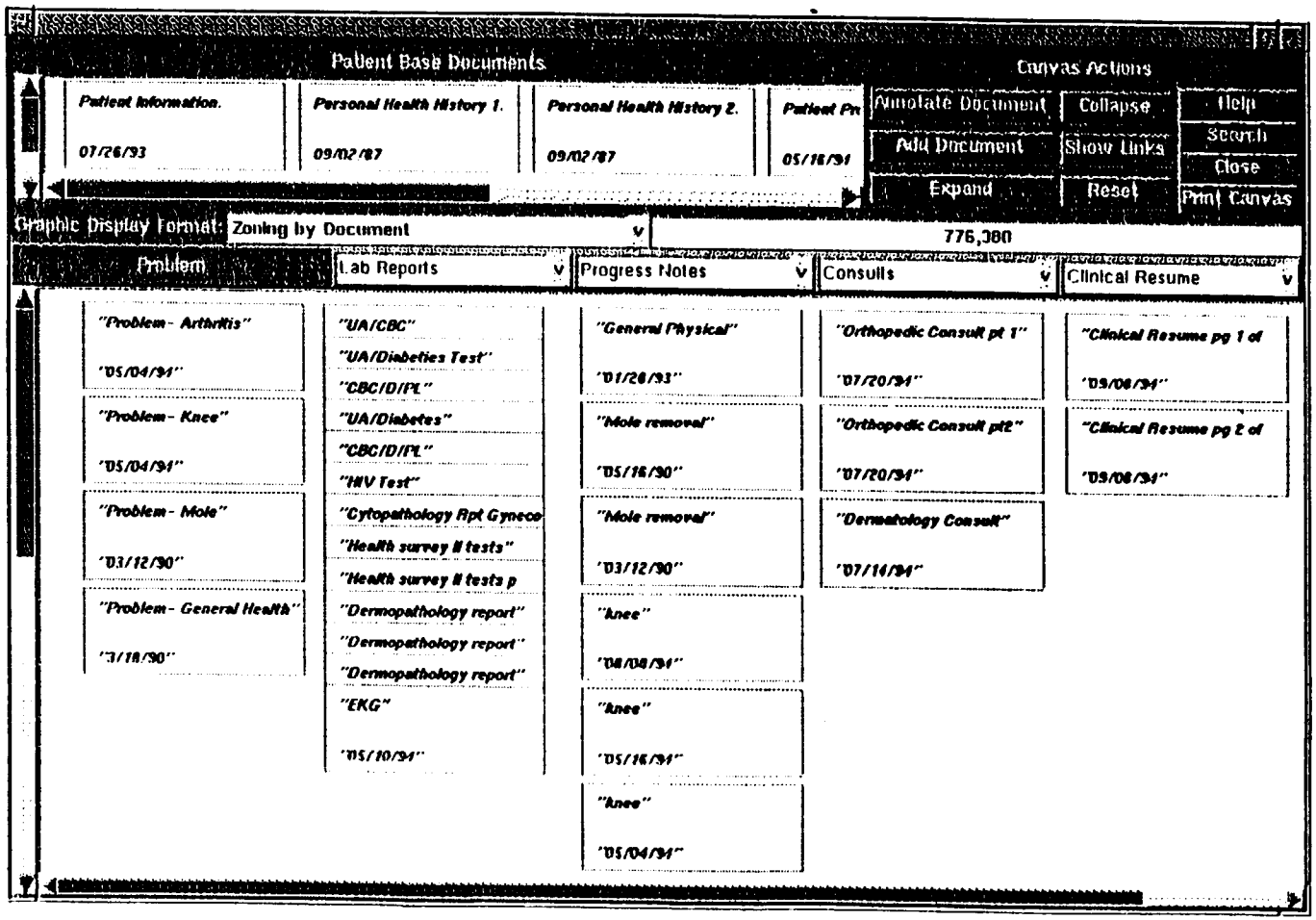




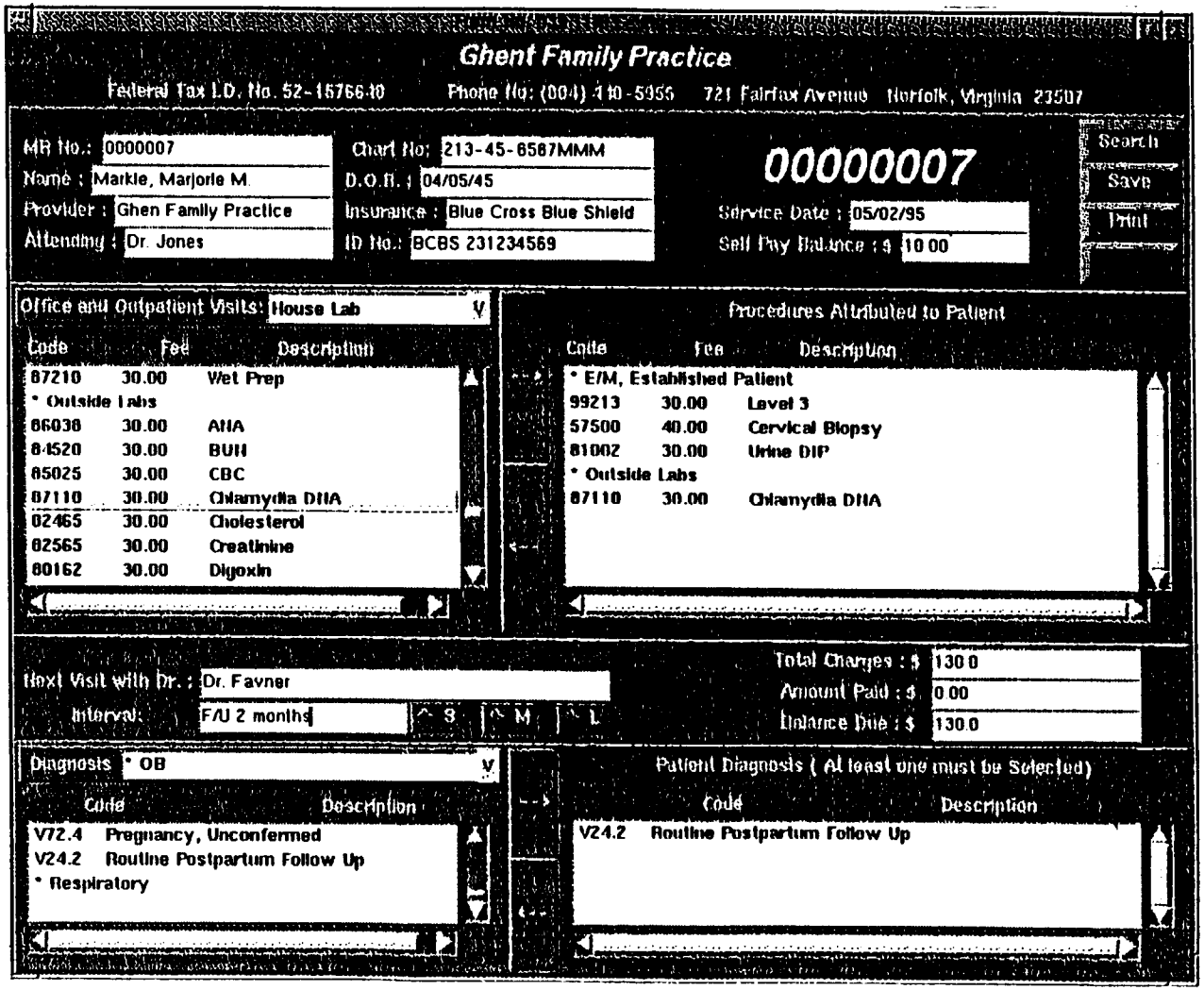
Medical Record System



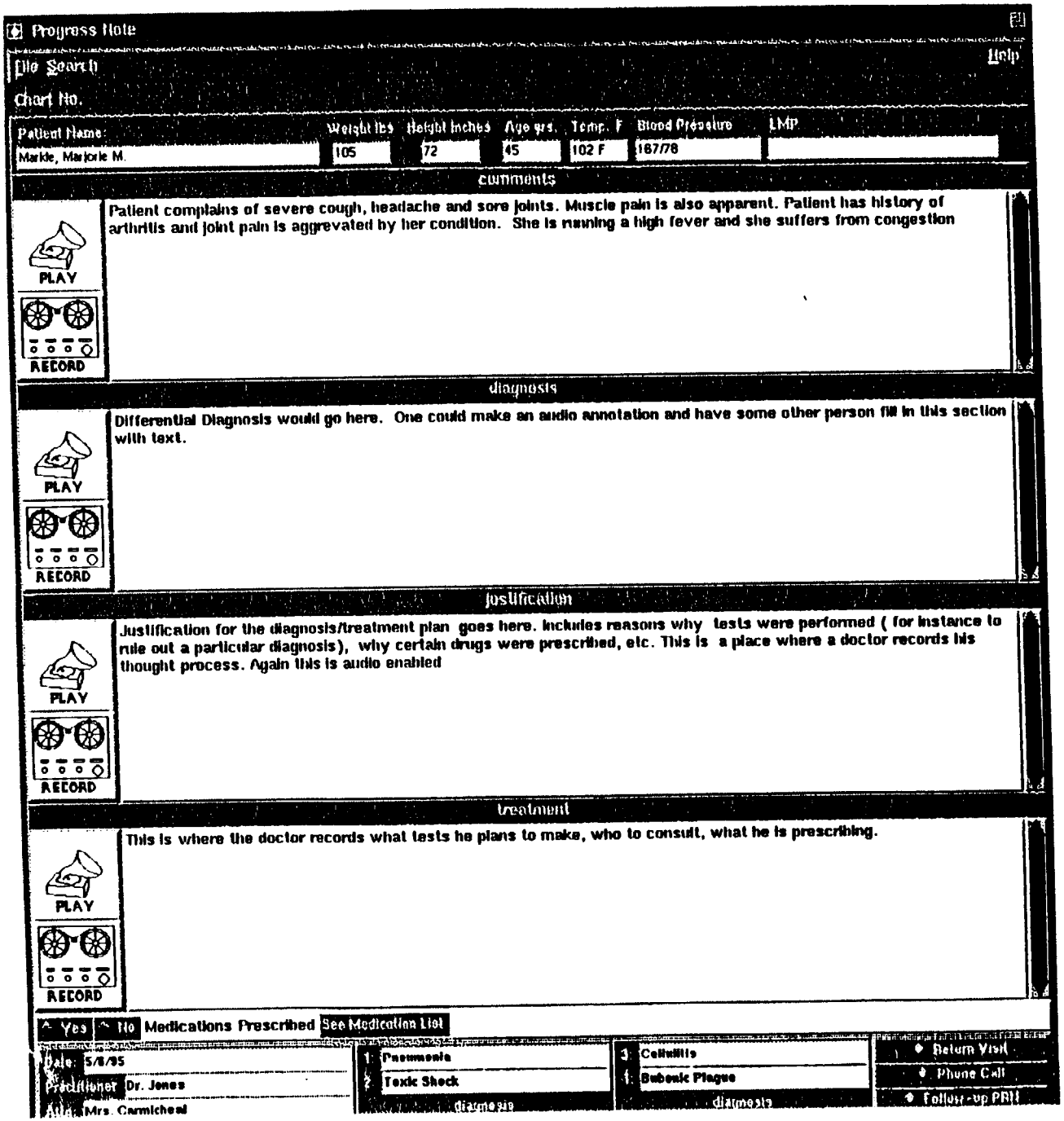
Medical Record System



Medical Record System



Medical Record System



Masters Project Report

The Medical Information System GUI Design Project

By William E. Ward, III

Abstract: My master's project consisted primarily of work in the design and development of a prototype Graphical User Interface (GUI) for the Medical Information System (MIS). The work consisted of new implementation; correction of logic, code and performance errors; new methods of source creation; version coherence management; and incorporation of new code by other students into the already existing system. Major efforts included code error fixes, appearance and handling enhancements, porting the existing work from BSD Unix to Solaris (due to a change in the operating system), and basic research into the requirements for any future versions of the system, including important considerations for internal data representation, database management and upkeep, functionality and improvement concerns, distributed implementation and data issues, and the need to implement expert systems into the MIS to increase the performance enhancement aspects that would compel doctors to use system.

Background: Information Technology is rapidly advancing as the field matures and the benefits of utilizing computers for data storage are realized. The advanced features inherent in these machines allow data to be shared by users. The advantages are a more cost effective system and superior service due to the replacement of the redundancy and inaccuracy inherent to traditional paper and pen

methods. With soaring costs in Health Care becoming an important concern in our modern society, the need to implement a new form of computer-based medical record system is crucial. The advantages are many. The doctor, nurse or other health care professional would be able to search the patient's record for similar occurrences, for previously seen symptoms of medical problems, or search other patients' records for similar cases. Medical records could be passed automatically through expert system filters to diagnose conditions with which the doctor is not familiar. Billing, a time consuming and costly event, could be streamlined greatly. Unambiguous medical records could be kept. Researchers could use a standardized medical record system on a computer database to look at large populations. Specialized medical tracking could be conducted more efficiently, such as follow-up immunization, psychological counseling, etc. Insurance payments could be done online, with instant updates. New medical procedures could be worked into additional expert systems, standardizing the level of care by allowing step-by-step patterns for the health professionals to follow. Medical records could be shared more efficiently with specialists during referrals. Reminders and prompts could be given to the physician of medical test results. Long term trends could be instantly pinpointed simply by allowing the computer to plot them graphically, such as weight fluctuations, blood pressure, etc.

With obvious advantages such as these, it is imperative that the health physicians begin using this type of system. However, while computers give

tremendous advantages, they also have obvious disadvantages: computer interfaces tend to be more difficult and time consuming for the physician to use during actual examinations; medical records are not currently in a format that would provide any of the benefits listed previously; physicians may feel that the expert systems embedded in these systems would usurp their own expertise; and a current lack of standard recording methods for the medical community. The problem of a community-wide recording standard is one that could be readily corrected if a suitable system were developed. The concern that the system would usurp the physicians' expertise is obviously a psychological problem beyond the scope of any pure Computer Science-based research. The problem of format is significant, but solvable, given the standards of the last concern. Since the inherent difficulty of use in many computer information systems is the area most likely to provide real impact on usability, we decided to emphasize our research there.

The difficulties with using a computer system for record keeping primarily revolve around two key issues: one, the system's interface design is "different" from the previously used system, requiring training before one becomes proficient; and two, the system's perceived value-to-effort ratio is low. At Old Dominion University, both of these issues were examined, and it was felt that the development of a suitable Graphical User Interface might hold the key to acceptance. We felt that the acceptance issue would be lessened considerably if the interface could be created such that: 1.) the amount of retraining was small; 2.) the actual look and feel could

be customized to match the health care professional's previous methods; and 3.) the interface offered real performance enhancement through the standard features that the system would deliver, (or through hooks for, as yet, undeveloped modules.) Carl Jolly, a graduate student at ODU, was originally assigned to develop a prototype of a GUI that will be able to offer a lower learning curve, the ability to customize, and a powerful feature environment. This prototype was not to be truly functional; instead, it was designed to show some of the features that could be incorporated into a fully functional system, as well as to assess changes by incorporating feedback from the medical community. In May 1995, Carl Jolly graduated, leaving the project with most areas of the GUI design set and in place. In April 1995, I was asked to take over for Carl: refining, debugging, and expanding the existing prototype, while studying areas where the future "Mach 2" may need to be changed substantially from the existing design.

Work: As I began the project, the most pressing need was for me to learn Tcl/Tk, the language in which Carl Jolly had written the project. Since other graduate students would be requesting assistance from me in using and learning Tcl themselves, I needed to become an expert relatively quickly. My next major project was Xf, a GUI interface prototyping tool for Tcl/Tk. After studying the systems and reading the reference manual and tutorials, I decided that until I was sure that my knowledge

was correct, my best approach was to construct a series of small, dummy applications, rather than experimenting on the actual MIS source code. This phase lasted one week, at the end of which I was confident that I could safely expand and debug the existing code. Consequently, I began a rapid process of system testing, exercising any and all functions and modules that then existed in the system. During this time, I began to isolate individual errors and areas where improvements could be performed. I chose, through this phase, to begin debugging some of the source code errors in light of my understanding of the system performance. The first significant software bug that I identified during this stage was an error in the handling of text documents in the patient medical record module. This bug caused the system to handle all documents as if they were graphical documents, using XV as a viewer. Since we had a mixed document system, with some graphical and some textual documents, this was a severe error. An example of the correct code in use can be seen by opening any Problem description in the Patient Medical Record interface.

At this point, I began expanding the help files for a number of the interface modules. During discussions with the faculty representing the project, we reached a consensus that the Physician's Triage interface was inappropriate and that the audio buttons then in use were unsuitable. I removed the module from the system and began modification work in Xf toward improving the interface. These improvements included an additional option to view the Nurse's textual triage report, as well as hooks for the actual audio library, replacing the titles of the various

windows to bring them in line with the system used by the physicians with which we were working, creation of a fourth window for the doctor to use, and correction of an existing error that caused a system crash when an attempt was made to save the doctor's triage. Immediately after, I replaced the audio icons then in use with new icons of my creation, giving a more professional look to the entire interface. During this phase, I determined that some of the documentation for the original source was incorrect, and a great deal of documentation was simply insufficient for the needs of maintaining the code. From this time on, I carefully checked the documentation, correcting errors when discovered, and updating the sparsely, and sometimes cryptically documented sections.

At this time, the project split into two parts. One part, the data base version, was to be extensively modified by Mark Carter. He was to incorporate a data base instead of the flat file structure we had previously been using. The other version would be based on the original flat file structure. Due to the inherent instability of the modifications that Mark was making, the frequent and chronic problems with Ingress, and the difficulty of maintaining integrity of code, it was felt that a dual version approach, with a reconciliation phase after completion of the database, was preferable. Consequently, I worked primarily on the flat file version. This decision led to an unfortunate situation; after Mark finished the database module, he needed time to study for the diagnostic exam. Afterwards, he and I were never able to completely reconcile the two versions, although most of the important bug fixes were

moved into the database version by November 1995, with some few additions in December and January. During this phase, I detected another major bug, again in the Patient Medical Record Interface, specifically in the exact method in which the patient's Permanent Medical File (the Patient Info, Personal Health History, Temporary and Permanent Medications, etc.) was handled. Although this sub-module did not work at that time (a bug in itself), when the sub-module was activated in a certain method, the read-only attributes of the files and of the source code were changed, causing a corruption in the module which loads the original patient information. This was not an obvious effect, but it led to significant problems. I was fortunate at this phase to have a duplicate copy in the database version of this section of code with which to replace the corrupted code. Tracking down the cause of the corruption, however, was not a trivial task, and I actually discovered the cause by a combination of luck, knowledge of the code, thoughtful contemplation of the actual step-by-step execution of that module, determination, and serendipity (since I gave up for a few days and decided to correct the bug causing the Permanent documents retrieval failure, which led to the module causing the disastrous code violations.)

Around this period of time, the inadequacies of the internal data representation of the existing code began to cause serious problems in increasing functionality. In conference with the faculty members in charge of the project, I repeatedly made the assertion that the internal data structure was not adequate for further expansion.

The faculty advisors decided that since we were only going to prototype an interface, actual functionality could be sacrificed for the impression of functionality. The internal data structure as it exists now, is a hodgepodge of arrays and trees, designed to hold multiple types of information. It is not object oriented, but would greatly benefit from an object oriented approach. If we create each document as a discrete object, we could use object oriented data base for storage of the documents. By combining these objects, we could create a complete patient object. Databases of this type are widely available and would be more useful than a pure SQL database such as the one used on this project. By combining an object oriented approach and an object oriented database, we would have the ability to expand the domain of the documents nearly endlessly, removing the constraints set by the initial parameters the original programmer created, allowing us to adapt to new techniques and technology. Internally, we would need to represent the absolute type of document, along with some method of viewing or interpreting the data contained therein. Ideally, for the types of documents with which we have been working, a good example would be to represent each document as either a scanned image, data set collection, or textual document, accessed by a graphic viewer, data interface, or text viewer, respectively, as the methods associated with the corresponding objects. Building on these simple document type objects, we could represent the patient's historical data similarly. By using special expert systems methods tailored to the specific document type (i.e., graphic, data set, or text), we

could then associate a particular attribute set for each document. By combining the attribute sets, we would be able to do associative sets of related documents, allowing search by subject without regard to document type. This feature could be extended throughout the patient object, allowing the patient problem logs to be kept and updated automatically. Patient objects could also be queried using the patient methods, with higher level objects being built of groups of patient objects. This method would alleviate many of the problems that are experienced with the existing data structure, since each module added would build stability, compared to the existing structure, where each new data item requires an extension of the existing data structures format, forcing occasional wholesale code changes.

Mark Carter's Masters Project was an attempt to integrate the MIS project with a database, allowing a more flexible, searchable system of data storage. During the late summer of 1995, I was asked by Mark to help integrate his work into the existing internal data structure. Unfortunately, the internal structure's poor design forced Mark to make some decisions that affected the usefulness of the database. The original design had a flat file structure, with a series of special purpose files acting as indexes for the entire structure. These indexes included detailed information on how to display each document's icon, information about what kind of data the document represented, the document's author, creation date, active date, a hard-coded list of relation links, the file path to the document, and a few other miscellaneous pieces of information. This file structure was deeply embedded into

the actual code of the project as a whole, as well. This forced Mark to substitute the database for the index file, a poor implementation of a database system for this purpose. Due to the nature of the files, the individual documents were not affected, and instead remained in the original location, with the only significant change being the handling of the index file. Realizing this was unsatisfactory, Mark choose to rebuild the patient information files, which consist of basic information such as name, address, date of birth, medical insurance, etc., and integrate that file directly into the database. At this point, Mark asked me to assist in the creation of a series of utilities that would be integrated into the database version that would allow actual input of this kind of information, so we would have this data stored internally in the database, versus the addition of another flat, read-only file. I chose to help design the interface, using my familiarity with the internal data representation, while Mark concentrated on reworking the database to accept the input. The initial attempt proved to be satisfactory from my viewpoint, although Mark later slightly reworked the interface to make data input somewhat easier.

Shortly after that, Kim Johnson asked me how the data she was preparing would be indexed in the system so that she could write routines that would allow us to utilize the data. I mentioned to her that the existing data representation was soon to be outdated, and that we would be forced to either write a routine that could parse that data representation into the database format or modify and insert the data by hand. I suggested that Kim create a parsible index to her data that would be

convenient for her while remaining straightforward enough that no special work would have to be done to translate her data into the new format. She and I decided on the format her index would take and I relayed that information to Mark. I then began the modifications that would allow multiple patients to be used in the actual program. Previously, we simulated multiple patients by allowing the user to select from a patient list but using the same patient record file for each patient, with the net result being that all patient cases were identical. The most important modification performed here was a precedence modification, which allowed the default condition to remain, namely that the identical patient record would be utilized when no patient record was available for the patient selected. Although this is not, in general, a good practice, it allowed us to avoid the null patient case where the patient record did not exist, but the patient record was requested to be displayed which was causing a system crash. However, when a patient record did exist, the new system would bring up that patient's record instead of using the default condition. Kim's data was now utilizable, and was incorporated into the system.

I began a series of much smaller look and feel modifications to the existing design. Specifically addressed were a number of issues, such as phone-in visits, help files, medical diagnosis, nurse's triage, referrals, and consultations, etc. With the major exception of the nurse's triage, most of these were simply simulations of functionality. The nurse's triage required no change in the actual internal operation but was unsatisfactory in the GUI interface presented. I created three different

versions of the nurse's triage interface in an attempt to find the one most suitable. Ultimately, I found a suitable interface that seems powerful enough and efficient enough for the needs of the project. The smaller look and feel simulations were relatively easy since I was able to use Xf to prototype a GUI, insert dummy data into the GUI, and capture an image of it. I then inserted the image into the file structure in place of existing documents.

Porting the existing code from a BSD Unix environment over into a Solaris environment was necessary at this time, as well. Although this required little total code change, it did require a careful revision of the default parameters loaded into the system during startup. For example, the change to Solaris resulted in a few irregularities in the default startup positions for some of the existing interface windows. By changing a series of runtime constants, I was able to shift the windows into the appropriate locations, resulting in a return to the neat, crisp effect we had previously achieved. Further efforts at that time included efforts to enlarge the default size of the Patient Medical History interface, in an attempt to increase the number of columns displayed to include Billing (unfortunately, the effect was too crowded, and exceeded the default display size of the monitor.)

In a series of project meetings, the issue was raised by a faculty member on the committee that there was a need to test the reactions of the existing prototype with the medical community. We contacted Dr. VanWalkenton, and set up a meeting with him to ascertain his thoughts on the requirements the system must fulfill. After

the meeting, the consensus was that the system must offer expert system diagnosis and treatment methodology in order to achieve its maximum usefulness. Although this was outside of the domain of the MACH 1 prototype we were working with, it is a serious design consideration for the MACH 2 and would be more than suitable for a series of Masters level projects.

Design Proposal for the Mach II: Based on the lessons learned from the existing Mach I prototype, a strawman Mach II design proposal can be advanced. While not attempting to be totally complete in all design aspects, the strawman proposal easily allows for a basic framework on which to base further design. Due to the nature of the actual data, and the ease with which the data can be idealized into groups of documents or sets of groups of documents, an Object Oriented design is obviously the logical initial starting point.

On the lowest level, we would have an object type defined as a document object. Internally, each document object is quite different, with various internal formats and methods associated with the specific document. In order to avoid confusion, we will call each actual medical document a datadoc. The actual datadocs at this time would consist of three main types, with subtypes of each: a textual datadoc, a graphical datadoc, and a data set datadoc. Externally, we would wish to be able to use common interfaces on all of the document types, with the

exception that a few operations which should only be performed once for each datadoc ever, such as the relationship matching.

The textual datadoc document objects can be subdivided into a number of significant different types, including, but not limited to, Nurse's Triage reports, Consultation Letters, Physician's Progress Notes, Phone messages, Patient Health History, etc. Each of these would require an internal method set different from each other, but similar as a group, with the ability to reuse many (but not all!) methods completely. The external methods of interaction with the document are straightforward, however, and would be the same for all document objects. We would expect that one method would be (view <object>), which would display the selected object regardless of what internal format the object itself used. For a textual document, this would mean a viewer that would display the text formatted appropriately for the actual contents, i.e., a Nurse's Triage viewer for the Nurse's Triage document, a Progress Note viewer for the Progress Notes, etc., while a graphical document object would obviously use some form of graphic viewer appropriate for the exact type of graphic, and a dataset document object would use some data-type specific plotting method (a simple X-Y graph plotter, for instance, for something such as a weight vs. time dataset).

Additionally, all document objects would be required to supply information on the document itself, such as creation date, author, document category (not to be confused with document type, a document category would be what the document

object should be classed as, such as Progress Note, Billing Information, Lab Report, etc., without regard to the actual content format), and any associated links that may exist for the document. By using a standardized format, we need only treat each encapsulated document object identically, after its initial creation, in order to properly use the datadoc. This has obvious merits in simplifying the main program code, as well as to aid debugging of that system, since only the methods associated with a particular datadoc could cause display or compatibility problems for that datadoc.

If we pin down the architecture, we would have the following external methods for interacting with a previously created object (where asterisks denote multiple pointers or documents):

- (fetch_author <object> <doc_author>),
- (fetch_cdate <object> <doc_dat>),
- (fetch_category <object> <doc_cat>),
- (fetch_links <object> <*doc_link_ptr>),
- (fetch_rel_docs <object> <*linked_list_of_related_document_names>),
- (display_descript <object>), and
- (display_doc <object>).

Additionally, we need only two, initially, special external methods:

- (update_rel_docs <object> <*additional_related_documents_ptr>) and
- (delete_doc <object>).

All document objects, regardless of the actual datadoc format, would be required to support these methods. During the creation of a document object, we would additionally use two single use methods,

- (create_document <object>) and
- (create_links <object>).

This may sound like a complicated set of methods, but it is actually a relatively simple set of commands, necessary for minimal functionality. They can be grouped into four categories: Document Creation, Document Deletion, Document Information Retrieval, and Document Presentation. Document Deletion would be rarely, if ever used, with the only major document type that would be deleted regularly being for reminder notices. Notice that no provision is made for document modification other than the insertion of new links to related documents after the initial creation phase. To all practical purposes, a document is read-only once completed and added into the patient record.

The actual syntax and use of the various methods could be defined in the following ways: **fetch_author** is designed to get the name of the actual author of the document in question. **fetch_cdate** would be used to retrieve the actual creation date of a document. **fetch_category** would be used to retrieve a quick summary of the type of content of the document (not the FORMAT of the document, but rather, what the contents represent). **fetch_links** would be used to retrieve the list of

document categories the particular document may be related to, such as "General Health," "Asthma," etc. Similarly, **fetch_rel_docs** would retrieve a list of all documents, regardless of category, the document might be related to. Note that a key difference between these two is that **fetch_links** could be used in principle to fetch documents that are not in the same group, i.e., documents that are for a different patient, or that may apply to many patients, such as a medical journal entry in the database, or a similar symptom group for a different patient. **display_descript** would be a singular method designed to simply give a quick overview of the document. This may be one of the more difficult hand inserted elements, since it is basically a summary of the document, and must be done manually during document creation. **display_doc** is, of course, simply a method designed to display the native object in a format useable to the physician, be that format a text viewer, special graphics viewer, or some form of plotting program for information such as a cardiogram. **update_rel_docs** is a special command that allows a document that has just been created to modify an existing document (the object which the **update_rel_docs** method is being used on) to point to the newly created document, or when a document is deleted, to modify the referenced documents such that no pointers refer to the delete object. Note that not all related documents should return a pointer back, such as medical journal entries, drug dosage information, etc, that is appropriate for MANY documents. It is the

responsibility of the document creation methods **create_document** and **create_links** and the document deletion method **delete_doc** to use this method, and in general it should not be used by any other method or interaction. **create_document** is used to create a new document object of the correct type. This method is inherent in the object type, and not in the object itself, and thus is not capable of being used by the object itself. Likewise, **create_links** is inherent in the **create_document** method, and may only (normally) be called by **create_document**. **delete_doc** would be used simply to remove and destroy an object document, along with updating all of the associated pointers referencing it. Note that some document types, regardless of the fact that they should have this method available, should never be actually deletable, such as medical journal listings and articles, etc., that may be pointed at by some documents, but not return the pointer to that document. Methods could be used which allow for a listing of the NUMBER of documents pointing to it, however, which may allow the document to be removed after no documents have pointed to it for some lengthy period of time. This is simply a refinement of the **delete_doc** method for those types of documents, however.

Sets of related document objects could easily be sorted by a number of methods. The most intuitive would be something similar to the method utilized in the MACH 1 prototype, where documents are catergorized by what the document

content meaning represents, ie., Nurse's Triage, Billing, Consultations, Lab Reports, etc. However, the ability to organize a particular record based on the set of related documents should not be ignored. Whereas we can search using the document links for documents containing any information on a specific subject, by utilizing the related document method to interact with these documents, we would be able to categorize by medical problems, similar to the method used in Mach 1. Unfortunately, as in the Mach 1, we have a significant difficulty in identifying related documents; whereas simple keyword searches could be used (on textual documents, at least) to develop the document_links, only an expert or an expert system would be completely able to categorize ALL related documents. It is obvious that the Nurse's Triage and the Doctor's Progress Note for a single visit would be related documents; however, when the doctor orders a test, the system would be required to generate some new document object immediately in order that the fact a test was ordered could be documented into the Progress Note. Until the actual Lab Report Document was returned, the existing object could be used to mark that a test is pending, and the related PN, Nurse's Triage, etc. However, once a lab report is returned, the Pending Lab Report Document Object would need to be used to find out what documents are related to the new document. Those objects could be updated to the new Lab Report Document, the related_link to the Pending object could be removed, and the Pending Object could eventually be deleted, with the new Lab Report in it's place. An interesting point occurs here: do we update the

relate_links of old document objects when a new object is created that has a related_link pointer to the old object? Obviously, this could require extensive updates. The answer must be yes, however. Since every document has different criteria as to what it's related documents are (a document may contain information on more than one problem, which are not related to each other), any document should be able to be chosen as the source for a related document search. If we include a special category of document objects that we designate Root Problem, we could use those objects to determine only the documents pertaining to a problem. Yet if we do not perform a backward propogation of updating the pointers, we will be forced to use the most recent objects to find all related documents. The back propigation of updates allows us to avoid this by updating older documents so that they can reference newer documents.

All documents relating to a particular patient would make up a superset object called the Patient Object. Individual Patient Objects represent individual patients. This allows us to again group the datadocs in an orderly, understandable manner. This approach does have one potential disadvantage, however, in that care must be taken to design a series of methods to interact with multiple Patient Objects on a piecemeal basis if we wish to examine similar situations on separate patients. For example, if we determine that patient J has an illness that was previously seen in patient B, we may wish to use information learned from patient B to help in the treatment of patient J. However, patient B's medical record as a whole is not useful;

only the information pertaining to the particular illness is valuable and useful. Therefore, we must remove part of the patient record to use for comparison purposes, with a directed search through patient B's record. Obviously, this will not be a simple task, yet it is one of the areas where the MRS can be most useful for general upgrading of patient care. Also, how will we determine that both patient B and patient J have similar illnesses? Some form of relational search index would be useful, a form of Meta-index of all patient records. This brings up the next level of objects in our schema, a Meta-patient object for each physcian's database to allow for an index of related patient objects. However, do we limit it to individual physcians, or do we expand the Meta-patient object to include all patients in the database? One approach, limiting the Meta-patient object to just an individual physcian's patients, has the advantage of securing privacy much more adequately; however, it limits the number of related patients a physcian can examine for relationships, as well as only ensuring a standard quality of care for a particular phycian's patients. A new intern, therefore, would have a very limited database to work on, and the care from physcian to physcian would differ, even as a particular physcian's care became more uniform. This may not be desirable, since this could lead to substandard care becoming the norm for some physcians, instead of expanding the quality of care to a uniform level for ALL physcians. Current privacy law may limit the ability to have multiple physcians in the same Meta-patient object however.

Further Research: The MACH 1 prototype has reached the end of its useful existence as an expandable tool. A MACH 2 prototype has been proposed and will doubtless be begun if funding is available. Some of the lessons that were learned from the MACH 1 prototype are directly applicable to the MACH 2. Specific emphasis should be placed on the data structure internal to the program, the data base storage requirements, inter-networking, and the refinement of a somewhat more intuitive interface. These four major issues coupled with the expert system issue previously mentioned, should drastically effect the design of the MACH 2.

The GUI interface look and feel is suitable for scaling up into a MACH 2 design. Further refinements would be necessary, but the basic design seems workable. Major emphasis would have to be placed on the internal representation, however. As it exists at this time, the current internal data representation is almost completely unworkable; it is unsuitable for expansion, refinement, or actual use, and serves only as a presentation structure. As mentioned previously on pages 8 and 13, an object oriented approach would be superior. The object oriented approach would allow new data types to be integrated seamlessly into the existing system, more rapid construction of data types, higher quality of data handling internal to the program, and higher confidence of accuracy in the programming code while allowing the program to be fitted into an object oriented database easily. If the object oriented database is set up to deliver documents only on demand, but deliver

document summaries automatically, then it would be possible to lessen, and possible cure, any inter-networking problems we may experience.

A major issue at this stage would be the design and incorporation of expert systems tools into the MACH 2. Since the MACH 2 would have to be designed to allow these expert systems to be added as they are created, some form of representation would have to be created during the software architecture creation phase. It is obvious that some of the expert systems would have to be integrated into the objects as methods. What is not obvious is how we would integrate the output from these various methods when the methods themselves, are different in similar document types such as the methods used in a textual document vs. The methods used in a graphical document. This is probably not insolvable in general, and is solvable in specific for this problem, if the architecture design is inclusive of a common interface for the output. We could then treat each document type identically, with only the internals of the object requiring information about the actual data organization. This approach would also allow a modularity in the addition of new document types as new students become available, with each student creating the entire object, including the expert system analyzer for the object, as part of a project. The possibility exists that commercial expert medical systems could be tailored for this task if desired as well. This approach could conceivably allow the medical establishment to expand their existing tools as opposed to abandoning the tools they have already begun using.

An SQL interface for a Tcl/Tk Medical Records System Prototype

Submitted by
Basil P. Stieffen

CS697 Independent Study
Summer 1995
Dr. Ravi Mukkamala

This paper is a representation of the work I performed to create an, Ingres based, SQL interface to a Tcl/Tk based Medical Records System prototype. The work evolved in stages and this presentation will be given as a chronological depiction of the day by day achievements on the project. Each day work was performed will be outlined as well as the number of hours worked and the accomplishments of that day. Any references to code that was developed will be accompanied by a copy of that code. Finally the total number of hours spent on the project will be tallied and a short conclusion will be given.

06/29 - .5 hrs - E-mailed Dr. Mukkamala to set up our first meeting on my participation in the ongoing MRS project.

06/30 - .5 hrs - Dr. Mukkamala responded and we set up the meeting for Tuesday July 11 at 1:30pm.

07/07 - .5 hrs - E-mailed Mark Carter and set up a meeting with him concerning MRS for Monday July 10 at 2:00pm.

07/10 - 6.5 hrs - Met with Mark Carter today to discuss our approach to the integration of Ingres SQL and MRS. He has already done quite a bit of work regarding the development of what will be our Ingres relations. It appears as if there is a Tcl/Tk extension which will enable us to interface directly with Ingres, but since Ingres was down it was impossible for him to show me any of the actual commands. I decided to first get the current system working on my home Linux system so I could run certain tests etc. and by the end of this day I have almost ported the entire system over.

07/11 - 1 hr - Met with Mark, Kim, Bill and Dr. Mukkamala about the course and goals of the project. Because of my August graduation it was decided that my goal would be to get the database functionality of the system completed by August 15th. Bill will look at all of the current flat file reads this week to determine the scope of the database implementation. Kim is going to create additional patient information so we can have more than one patient active in the system, and Mark will help her with this. I have agreed to clean up the system, which was a somewhat disorganized mix of directories and files, into a more concise and straightforward format (Figure 1).

07/14, 15, 16 - 3 hrs - Cleaned up directories and files, trying to make the system more navigable. This involved creating new directories and placing files into these as necessary and changing links within the code to point to my new hierarchy.

07/18 - 3 hrs - Had a meeting today with Mark and Bill. I installed my cleaned up version of the system. Ingres is still down so they decided to work on application code and I decided to attempt to find a freeware version of Ingres.

07/19, 20 - 3 hrs - Found a freeware version of Ingres and installed it on my Linux system. Everything went well until I discovered that it does not support SQL but something called EQUEL which will not suit our purpose.

07/21, 22, 23, 24 - 6 hrs - Found a freeware version of SQL called msql or mini-sql and installed it on my Linux system. It uses a subset of ANSI SQL but it should be sufficient for our purposes. I have also installed a version of Tcl/Tk which works with msql and have them up and working with one another, however, Mark has just informed me that Ingres is now up so I'll put my msql activities on hold for now.

07/25 - 8 hrs - I met with Mark today to begin working with tclsql. We made excellent progress but shortly found that there are serious problems with the Tk interface. Certain events do not process (e.g. buttons don't push, menus don't pulldown) which leaves it unusable. We spent quite a while seeing if it had been compiled incorrectly or some other explainable error but we didn't have much luck. So we sent mail to the guy who developed it in hopes of getting some answers. There is no way the Tk portion could have even been tested, as the errors are so obvious. Mark and I decided to meet again on the 27th.

07/27 - 7 hrs - I met with Mark today to continue to investigate our problems. I decided to go ahead and install msql and tclmsql on the Suns just in case we can't get tclsql working properly. Once that was done I decided to try and debug tclsql and maybe see if I could find the problem. This involved recompiling tclsql, sqlwish, and both Tcl and Tk using the -g option, no small task. This was very productive in that I did discovered that while some events (e.g. scrollbars) worked fine, others did not. I discussed all this with

Mark and he gave me some additional hints about the event handler. I ended the day with a good grasp of how all the packages compile, but still no solution to our problem.

07/28 - 10 hrs - Today I went to school to attempt a step by step investigation of tclsql to try to determine the cause of our problems. Mark and I traded ideas on how to go about this. I decided to progressively eliminate objects which makes tclsql and sqlwish different from the standard tclsh and wish. At one point in this process I had eliminated every object which made the two sql based programs unique, in essence I created tclsh and wish from tclsql and sqlwish, and the problems still existed. However, this brought to light the fact that sqlwish linked in tclAppInit.o while wish linked in tkAppInit.o. We decided to change the Makefile and link in tkAppInit.o for sqlwish as well, when I did this everything started working except for the SQL stuff. Since tclsql was not even supplied with tkAppInit.c I had to grab a copy from the original Tk release and add the SQL code to make everything work. Once this was done everything started working great :-).

07/30 - 1 hr - I attempted to do some remote work from home but I did not have grant authority to the health database so I sent Mark some e-mail to allow me to use it.

08/01 - 7 hrs - Now that sqlwish works properly I can finally get down to actually doing some actual coding. First, however there are two things which I have to do. One, is to create my own testing environment. This includes copying the entire system to my testing directory and creating the necessary links for it to run there. Two, there seems to be a great deal of writes to standard out in the current code. I counted approximately 2,500 lines of output in the 14,000 or so lines of source code. Unfortunately all these writes have a tendency to mask any of my own debugging output. So I have had to try to eliminate as much of these writes as possible. It was almost nine o'clock before I reduced the output enough where it will not interfere with my own output. Now I can start to really test the current system to see where changes need to be made.

08/02 - 6 hrs - Today I created several small test scripts to simulate some of the current flat file processing which takes place in the system. The format currently used is so pervasive throughout the system that, in my opinion, it would be foolhardy to change the data format too drastically. My next step

was to create a new database relation based on the current patient medical information. Once created my next step was to create some test scripts to access that data and format it in such a way that the system could understand it and use it. It was 9pm by the time I reached this point so I decided to leave it until tomorrow.

08/03 - 5 hrs - Today I initially ran some tests at home on Linux simulating the current system and what will be the Ingres interface. After many different tests I was able to create something I felt would work well. I then signed on to school and uploaded my tests so I could use them with Ingres. Unfortunately Ingres was down again, so I mailed root and asked if someone could look at the problem. I kept checking periodically throughout the day but Ingres never came back up so I decided not to come into school, but would check on things tomorrow.

08/04 - 4 hrs - Ingres was down most of today but finally came up in the late afternoon. This enabled me to test those test scripts I had for yesterday. They worked great with only minor modifications. It was late in the day by this time so I decided to come in tomorrow and apply the test code to an actual MRS program and see if it will work.

08/05 - 6 hrs - I went to school today to apply my script SQL code to the first two indicated areas in MRS (Figure 2). With a small amount of tweaking I got the code working. Next I found another call that simply requests the names of the column headers, this fit great into the Ingres methodology since Ingres makes the column headers the first row of any relation. So far I have found these three references but I know there are more because the system doesn't work when I rename the flat file, meaning that it is still in use. It was 4pm by this time so I decided to go home and research those additional file references.

08/06 - 6 hrs - Last night I found the remaining references to the patient file and came to school this morning to integrate them into the code. They differ from the other accesses in that these process the file one line at a time rather than the entire file (Figure 3). So I wrote several test scripts to experiment with how the "cursor" format works within tclsql. My tests went well and I ended up with code which works very well. The lab was closing by now so I left to come back tomorrow.

08/07 - 6 hrs - Today I implemented my "cursor" code and it went well except for a few idiosyncrasies in how this access works. It appears as if the "SQLfetchCursor" command has to always be used before a row is used, even though the data is actually fetched beforehand. Dr. Mukkamala came by the lab today to get an update on the project. He stated my primary and final goal was to get the system working with one patient relation which contains all of the patient data. I feel that if I rebuild the patient relation with an ssn key then I could get this working in perhaps one or two days.

08/08 - 5 hrs - I started making the changes today to implement the one patient relation scenario. I had to make changes to all of my code so far and also set up an additional patient's information for testing purposes. I had to make changes to the patient.index file to use the ssns instead of the original patient code. To my amazement as soon as the changes were put into place the code worked the first time. All that remains now is to put additional patient information in the relation. This should be done by someone who will be staying with the project, since it requires a greater understanding of how the entire system works.

I really enjoyed working on the project. It enabled me to get a much greater understanding of Tcl/Tk and see it's almost unlimited possibilities. Mark and I had some large hurtles to overcome, especially with tclsql, but we did overcome them and I think the system has improved immensely.

The total amount of hours spent on this project was:	95
The total amount of hours spent on this paper was:	10
For a total of:	105 hours.

I reduced the following mix of directories and files:

```
/research/health/tcltk [27] >la
#book.examples#*    first_run/          pcsi_srch*          shitfile*
#book.examples#~*   firstvisit.sh*      phonesystem.sh*     sorted_Globals*
2ndrun/             firstvisit.sh~*     phonesystem.sh~*    square*
CIF/                forms/              phonesystem.tcl*    stinky.sh*
Unique_Globals*     help/               phonesystem.tcl~*   stinky.sh~*
XF.notes*           lbtest.sh*          plans*              taputils.tcl*
all_globals*        lbtest.sh~*         prochelp.tcl*       tclGlobal*
bitmap/             lbtest.tcl*         prochelp.tcl~*      tclIndex*
book.examples*      lbtest.tcl~*        replace*            tclsourcefiles/
c-interface/        makefile*           replace~*           temp/
data/               multiform.sh*       repository/         temp.tcl*
data_structures/    multiform.sh~*      saveproc.tcl*       temp140604*
dhc.tcl*            oldtcl/             saveproc.tcl~*      tkman.tcl*
dhc.tcl~*           paper/              searchbox.tcl*      todo.txt*
dhctop.tcl*         patidx.tcl*         securdialog.tcl*    trainer/
dhctop.tcl~*        patient.idx*        securdialog.tcl~*   triage.tcl*
dialogtest.tcl*     patient_data/       security/           triage.tcl~*
dialogtest.tcl~*    patientinfo.sh*     security.sh*        unique_Globals*
examples/           patientinfo.sh~*    security.sh~*       uniqueglobals*
file.test           patientinfo.tcl*    security.tcl*       view/
filestruct.fig*     patientinfo.tcl~*   security.tcl~*      wishtest*
```

Into these more concise and meaningful directories:

```
/research/health/mrs [30] >la
Medical_Records_System/  forms/           pictures/
bitmap/                  help/            tcl-tk-xf_info/
build_info/              medical_images/  tclsourcefiles/
data/                    non_vital_info/
```

Figure (1)

These are the first two database calls integrated in the MRS code:

This is the call from Medical_Records_System/canvasops.tcl:

```
#################################################################
# The following starts some new database code using Ingres SQL
# to access patient data.  The database name is currently
# doc_1_index.
#################################################################

    set size [Lookup_All_Matches_Index problem {"BLANK"}
$patient(patient_num) patient_med_info docarray ]

    set format [Lookup_Index_Format patient_med_info ]
```

And this is the first called procedure in tclsourcefiles/index.tcl:

```
proc Lookup_All_Matches_Index { column key patient_num relation array }
{
   upvar $array a

#################################################################
# The following starts the new database code using Ingres SQL
# to access patient data.
#################################################################

if  { [catch { SQLopendb health } f ] != 0 } { puts $f; return 0 }

#Note '"BLANK"', ingres requires '' to process character data correctly.
SQLselect -noheaders atmp * from $relation where $column='$key' \
    and pssn=$patient_num
SQLclosedb

set size [array size atmp]
for {set i 1; set k 0 } { $i <= [ expr $size/12 ] } { incr i; incr k } {
    for {set j 1 } { $j < 12 } { incr j } {
      if { $atmp($i,$j) != "" } {
          append a($k) $atmp($i,$j)
            if { $j < 11 } {
                append a($k) ":"
            }
      }
     }
}
set found $k
return $found
}
```

And this is the second called procedure in tclsourcefiles/index.tcl:

```
proc Lookup_Index_Format { relation } {

#####################################################################
# The following starts the new database code using Ingres SQL
# to access patient data.
#####################################################################

if  { [catch { SQLopendb health } f ] != 0 } { puts $f; return 0 }

SQLselectRow 0 as * from $relation
SQLclosedb

set size [array size as]

# 11 = 0 - 10 total fields always
for {set i 0; set k 0 } { $i < [ expr $size/12 ] } { incr i; incr k } {
    for {set j 1 } { $j < 12 } { incr j } {
      if { $as($i,$j) != "" } {
          append b($k) $as($i,$j)
            if { $j < 11 } {
                append b($k) ":"
            }
      }
     }
}

return $b(0)

}
```

Figure (2)

This is the called procedure (the calling procedure was unchanged) in Medical_Records_System/canvasops.tcl:

```
proc Scan_Doc_Index_And_Tag { } {
global gdirlist doc patient debug
eval global $gdirlist

      .
      .
      .
      .

set relation patient_med_info

#####################################################################
# The following starts some new database code using Ingres SQL
# to access patient data.  The database name is currently
# doc_1_index.
#####################################################################

if { [catch { SQLopendb health } f ] != 0 } {
     puts "$f: Scan_Doc_Index_And_Tag"
     return 0
}

SQLdeclareCursor cur
SQLprepareCursor cur select * from $relation where
pssn=$patient(patient_num)
SQLopenCursor cur
SQLdescribeCursor cur dbarray
SQLfetchCursor cur larray; #Get rid of the column names

set dbline 0
set i 0
while { [ SQLmoreRows ] != 0 } {
    unset dbline
    SQLfetchCursor cur dbarray
    for {set j 1 } { $j < 12 } { incr j } {
       if { $dbarray($j) != "" } {
        append dbline $dbarray($j)
            if { $j < 11 } {
                append dbline ":"
            }
       }
    }

set line $dbline

      .
      .
      .
      .
      .
```

```
      .

SQLcloseCursor cur
SQLclosedb

# Build problem array ( links to relations)
 Build_Problem_Array $relation
}
```

And this is the other called procedure (called from the above statement) Medical_Records_System/canvasops.tcl

```
proc Build_Problem_Array { relation } {
global gdirlist doc patient problem
eval global $gdirlist

       .
       .
       .
       .

#####################################################################
# The following starts some new database code using Ingres SQL
# to access patient data.  The database name is currently
# doc_1_index.
#####################################################################

if { [catch { SQLopendb health } f ] != 0 } {
     puts "$f: Build_Problem_Array"
     return 0
}

SQLdeclareCursor curl
SQLprepareCursor curl select * from $relation where
pssn=$patient(patient_num)
SQLopenCursor curl
SQLdescribeCursor curl larray
SQLfetchCursor curl larray; #Get rid of the column names

set dbline 0
set i 0
while { [ SQLmoreRows ] != 0 } {
    unset dbline
    SQLfetchCursor curl larray
    for {set j 1 } { $j < 12 } { incr j } {
       if { $larray($j) != "" } {
        append dbline $larray($j)
            if { $j < 11 } {
                append dbline ":"
            }
       }
    }
```

```
set line $dbline

        .
        .
        .
        .

}
SQLcloseCursor curl
SQLclosedb
}
```

Figure (3)

Master's Project Report

Creating a Relational Database Interface for the Health Care Project using Tcl/Tk as the host language.

Mark C. Carter
Advisor: Dr. Mukkamala
Department of Computer Science
Old Dominion University
Norfolk, VA 23508-0162

August 21, 1995

Abstract:

After creating a working prototype of MRS (Medical Record System) for the health care project it became apparent that this information could be better represented by the use of a database instead of a traditional file system approach. The database was created for two main reasons. First of all, it was believed that the use of database concepts will expedite the future development of the MRS prototype. Secondly, by demonstrating an SQL interface it would show potential customers that MRS does indeed have the capability of supporting a "real world" interface. This paper discusses some of the rationale for implementing the database along with its advantages. The graphical interfaces that were created for the database along with some examples of using the SQL commands from within Tcl/Tk are presented. Some potential problems with the current implementation will also be discussed.

1. Introduction

1.1 Major Reasons for using a Relational Database over a traditional file processing approach:

With a traditional file system approach, each programmer would define and implement the files needed to complete a specific task. Several problems arise when the same data is keep in multiple files. First of all, this redundancy wastes storage space by having to store the same data repeatedly, which can be a real problem for large databases. Secondly, it requires more effort to maintain multiple copies and to insure that this common data is up-to-date in each file. If this is not done then we have inconsistent data within our system. For example, when a new patient enters there are currently several files which maintains the patients name and this information must be included in each of these files. However, in a database approach, there will be only one repository for the data. This data, can then be accessed by multiple users and can be used to create multiple views of this data.

In a typical file processing application using PASCAL, COBOL , Tcl/Tk etc... the file structure and sometimes the exact location of an attribute within a record are coded into the program that accesses this data. Some actually require the position and the size of the data within the record.

In traditional file processing, the actual structure of the files are embedded into the program, so any changes to the structure of a file will require or may require changing all programs that access this file. For example, a each file may define its record length, the number of characters(bytes) in each record, and each item may specify the starting byte within a record and its length in bytes. In a relational database system, the actual access is done independent of any specific file in other words all that changes is the description of our relation. This is a form of a "data model" which provides the user with a "conceptual representation" of the data that does not include how the data is stored (i.e. hides it from the user). This is accomplished by keeping the detailed structure and organization of each file in a catalog. The database user will refer to this conceptual representation or data model for interpreting information and the actual database management system extracts the details from the catalog when needed.

1.1.2 Additional Advantages of the Relational database

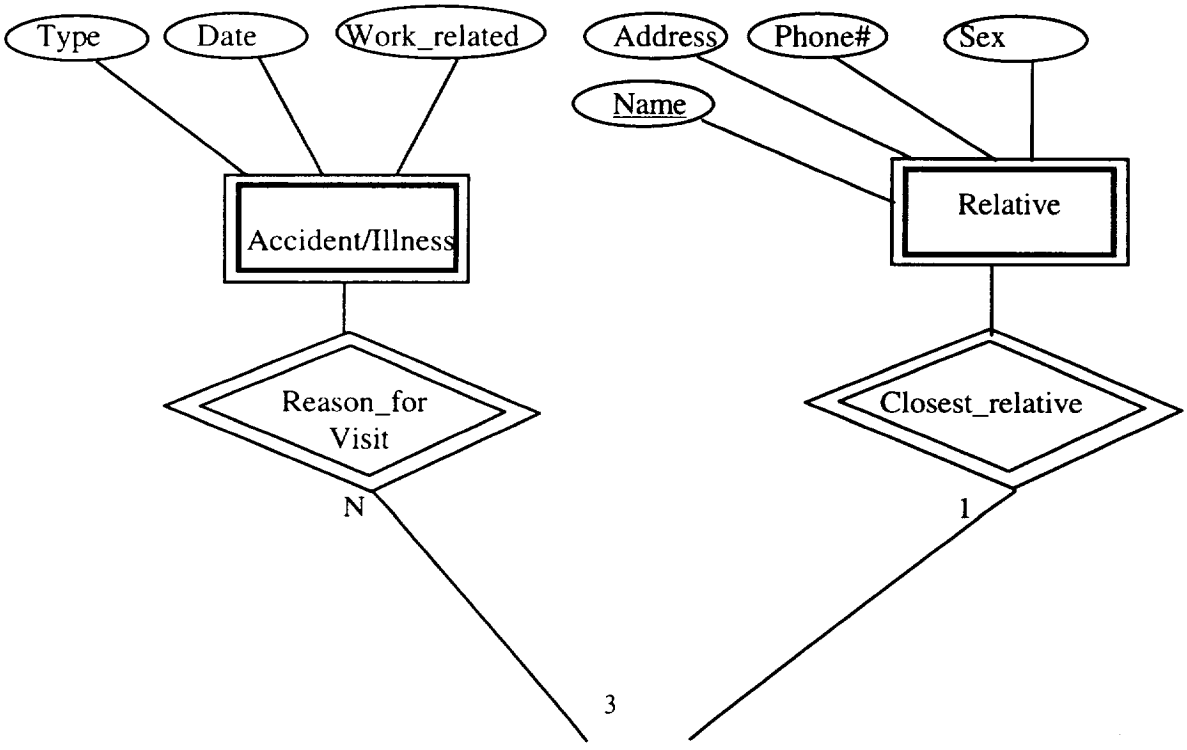
It will be easier to make changes to a database if more information is needed to be kept about a particular relation only the definition of the relation is changed not the existing application code, which would have to be modified in a traditional file processing application.

It will usually require less time to develop a new applications using a database approach than it would with a traditional file processing approach assuming the database is up and running due mainly to the easy to use relational interface.

With a database the information is available to all users and therefore once an update is applied to the database, all other users will have access to this updated information. This prevents inconsistency in our data .

2 Entity-Relationship (ER) Diagrams

An Entity-Relationship model , which is a high level conceptual data model, was used in the initial design of the database. It allowed us to get an understanding of what data was required and the relationships that exist among this data. However, no formal requirements specification was given and therefore it soon became evident that the data would change and the ER diagram could only be used as a starting point and would not be a unique or definitive solution throughout the development of this project . For example, this preliminary ER diagram was created using an existing patient record form and analyzing the information that this form is used to gather; however, this form changed dramatically in a short period of time based apparently on some doctors preference. Again, this is not the case with most databases systems which are usually stable , this is the information is known in advance and is relatively constant throughout the design and implementation phases. Figure 1. shows the Entity-Relationship diagram used for the health care project. Due to the large number of attributes and a limited page size only the most important attributes are shown for each entity.



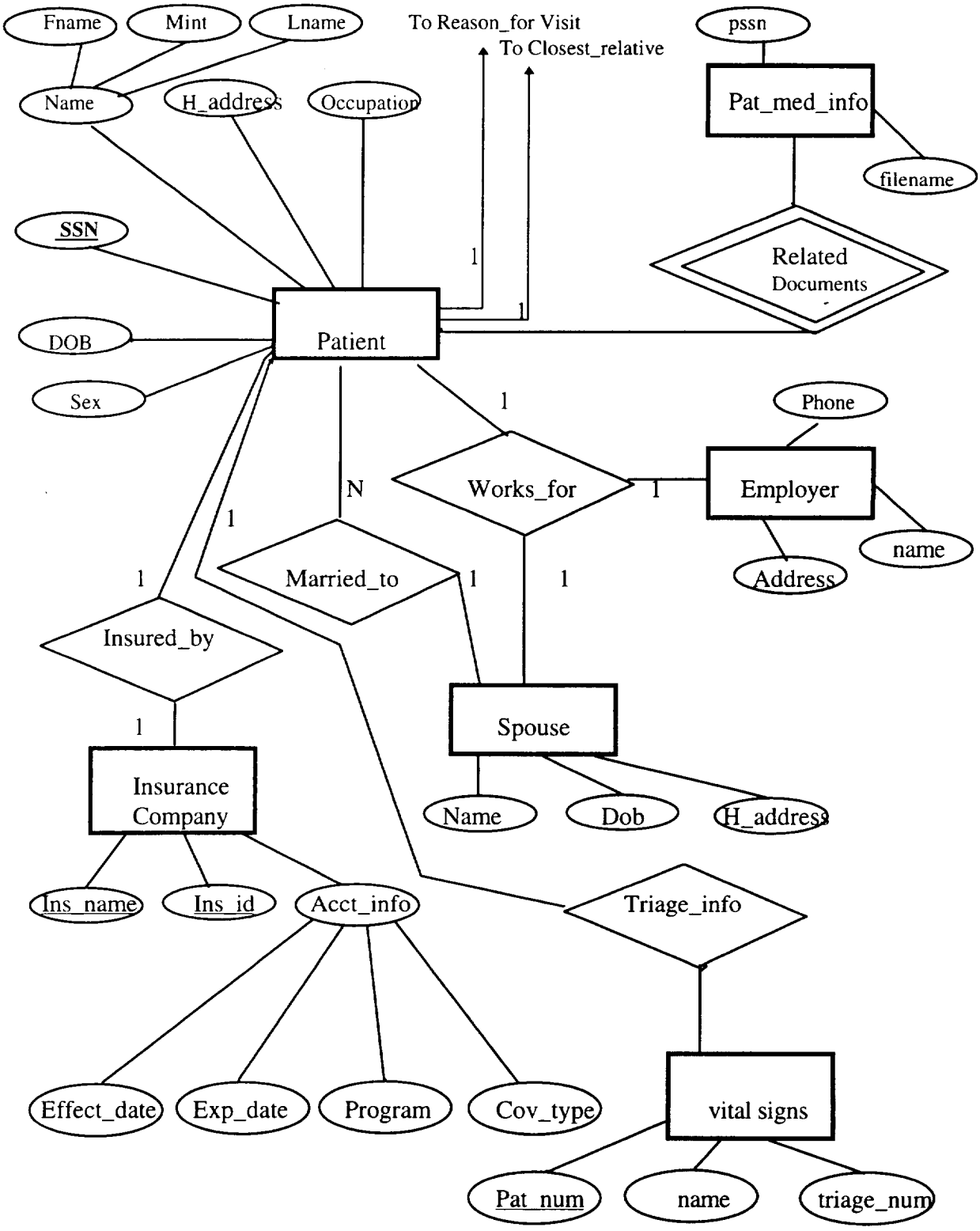


Figure 1. Entity-Relationship diagram.

Notes

Need to include "participation" after confirming the "weak entities" , keys, etc...

not enough room to show all the attributes!

3 Relational Database Schema

A relational database schema usually contains a set of relations and their corresponding integrity constraints. We have shown the key constraints, which are keys that must be unique for every record in any relation instance for that particular schema. We have enforced the entity integrity constraint; which states that no primary key can be null by specifying the appropriate INGRES commands while creating the relations. Referential integrity constraint was not implemented, due mainly to the fact that it's our belief that INGRES does not support referential integrity. Referential integrity constraint is used to maintain the consistency among tuples of any two relations. In other, words if a tuple of one relation refers to another relation then that reference must be to an existing tuple in that relation. The relational database schema for the health care project is shown in figure 2.

PATIENT_MED_INFO

pssn	filename	title	date_init	date_recv	author	problem	type	doctype	index	links	keyword

PATIENT

ssn	pname	dob	sex	establish	paddr	pcity	pstate	pzip	hm_phone	marital_stat	poccupation

SPOUSE

pssn	sname	saddr	scity	sstate	szip	shm_phone	soccupation

RELATIVE

pssn	rname	raddr	rcity	rstate	rzip	rhm_phone	relation

EMPLOYER

pssn	ename	eaddr	ecity	estate	ezip	wk_phone

INSURED_BY

pssn	sub_name	insur_id	group_id	type	rel_to_sub	eff_date	exp_date	iphone	ins_address

REASON_FOR_VISIT

pssn	accident	accid_time	accid_date	work_related	ref_physician

TRIAGE_INFO

name	pssn	date	weight	temp	pulse	blood_pressure	age	triage_text

Figure 2. Relational database schema

Some of the previous tables require some further explanation. Others are self explanatory and any further details can be obtained by viewing the status window from within the graphical interfaces that already exist for inserting information into these tables

Patient_Med_Info is an arcroynm for patient medical information and its attributes are described below in Table 1.

PATIENT_MED_INFO

pssn	filename	title	date_init	date_recv	author	problem	type	doctype	index	links	keywords

Attribute Name	Description of Attribute	Typical Value
pssn	Stands for patient social security number.	123456789
filename	The name of an external file, the type varies	ptrec1.gif
title	A short description of the file being referenced by "filename". Should be less than 30 characters in order to be displayed properly.	Patient Information
date_init	The date the file was created. This is the date that actually gets displayed in the "main canvas".	07/26/93
date_recv	This is the date that the document is received by the hospital.	07/26/93
author	Creator of the Document.	C. Riley
problem	This is a user defined description of what information is contained within the file. If the word "BLANK" is used this will indicate that the file should be placed in area entitled Patient Base Document, otherwise is goes into it respective area (i.e. lab report, consult, clinical resume etc...)	knee or "BLANK"
type	Represents a type of record or report that is being maintained by MRS. A list of acronyms used for this attribute follow. **CM** - **C**urrent **M**edications **TM** - **T**emporary **M**edications **PN** - **P**rogress **N**ote **PINFO** - **P**atient **INFO**rmation **PP** - **P**atient **P**rofile **PHH** - **P**ersonal **H**ealth **H**istory **PROBLEM** - Problem Types **CONSULT** - **Consult**ant **LR** - **L**ab **R**eports **OTHER** - other type of forms	**PINFO**

	AN - **AN**notation **CR** - **C**linical **R**esume **033** - Billing Information	
doctype	The type of document being referred too (e.g. a scanned image or text file etc...) The current types are being employed. SCANNED, TEXT, 033, PN, TR, AUDIO	**SCANNED**
index_file	This field is always labeled as **"BLANK"** for unknown reasons.	**"BLANK"**
links	Are problems that are related or associated with this current "problem".	knee or "BLANK"
keywords	Are those words that are redeemed as important in the current file.	patient information knee

Table 1. A Complete Description of table Patient_Med_Info.

The following code will create **all** tables described above. The easiest method is to start INGRES while logged onto the host machine "tsunami" as follows. It is assumed that the initiator of the following code is a valid INGRES user and has access to a database or has created a database called "health"

sql health to start INGRES.

\read <a file name> This file should the contain following code as a minimum and tables can be added if desired.

```
create table patient_med_info
     (pssn          varchar(15)    not null not default,
     filename       varchar(20),
     title          varchar(35),
     date_init      varchar(15),
     date_recv      varchar(15),
     author         varchar(20),
     problem        varchar(20),
     type           varchar(20),
     doctype        varchar(20),
     index_file     varchar(20),
     links          varchar(20),
     keywords       varchar(456)
)
```

```
create table PATIENT
    (ssn            varchar(15)    not null not default,
    pname           varchar(64),
    dob             varchar(15),
    sex             varchar(6),
    establish       varchar(15),
    paddr           varchar(64),
    pcity           varchar(34),
    pstate          char(4),
    pzip            varchar(15),
    phm_phone       varchar(15),
    marital_stat    varchar(15),
    poccupation     varchar(34)
)
\p\g

create table SPOUSE
    (pssn           varchar(15)    not null not default,
    sname           varchar(84)    not null not default,
    saddr           varchar(84),
    scity           varchar(34),
    sstate          char(4),
    szip            varchar(15),
    shm_phone       varchar(15),
    soccupation     varchar(34)
)
\p\g

create table RELATIVE
    (pssn           varchar(15)    not null not default,
    rname           varchar(84)    not null not default,
    raddr           varchar(84),
    rcity           varchar(34),
    rstate          char(4),
    rzip            varchar(15),
    rhm_phone       varchar(15),
    relationship    varchar(34)
)
\p\g

create table EMPLOYER
    (pssn       varchar(15)    not null not default,
    ename       varchar(84)    not null not default,
    eaddr       varchar(84),
```

```
    ecity          varchar(34),
    estate         char(4),
    ezip           varchar(15),
    ephone         varchar(15)
)
\p\g

create table INSURED_BY
    (pssn          varchar(15)    not null not default,
    sub_name       varchar(34),
    insur_id       varchar(34)    not null not default,
    group_id       varchar(34),
    type           varchar(34),
    rel_to_sub     varchar(34),
    eff_date       varchar(15),
    exp_date       varchar(15),
    iphone         varchar(15),
    ins_address    varchar(84)
)
\p\g

create table REASON_FOR_VISIT
    (pssn          varchar(15)    not null not default,
     accident      varchar(25),
     accid_time    varchar(15),
     accid_date    varchar(15),
     work_related  varchar(15),
     ref_physcian  varchar(32)
)
\p\g

create table TRIAGE_INFO
    (pssn          varchar(15)    not null not default,
    name           varchar(34),
    date           varchar(18),
    weight         varchar(15),
    temperature    varchar(10),
    pulse          varchar(7),
    blood_pressure varchar(15),
    age            varchar(7),
    triage_text    varchar(258)
)
\p\g
```

4 Tcl (Tool Command Language) interface to SQL (Structured Query Language).

There were several methods of providing an SQL interface to Tcl/Tk for use with the health care database. For example, there is existing support already for mini-SQL which is a stand alone database for employing an SQL commands. Mini-SQL had several major limitations, for example, only several commands were available for inserting and retrieving information most of these were very primitive and awkward to use. Most commands used a type of iterator, which required traversing each record step by step. Again, most of the SQL commands did not appear to be standard or at least familiar to me so this method for supporting an SQL interface was abandoned. The application msqltcl was reviewed but it suffers the same problems as was discovered with the mini-SQL this was actually expected because msqltcl was only a Tcl/Tk interface to the mini-SQL database.

Next, we examined the idea of creating our on interface to INGRES. This idea was indeed feasible , but it would require an in-depth knowledge of Tcl/Tk and the low-level calls necessary to access INGRES assuming this is possible. In the process of implementing this idea we discovered an application which provided the necessary interface between Tcl/Tk and SQL (see reference [4]).

After installation this application provided two executables . The first was called tclsql which is a version of "Tcl" that also supports the SQL commands. The second executable is called sqlwish which is a version of "wish" or "Tk" that also supports the SQL extensions. This application was ideal because it provided numerous commands as described below in a standard SQL format. After testing these commands it was evident that this was an excellent extension to Tcl/Tk. However, even though the SQL commands seemed to work flawlessly it was later determined that this application was not supporting other scripts that had previously been written in wish or tclsh . After several days of debugging and testing it was determined that there where several source code errors and even makefile errors.

There are three types of commands: general commands, non-cursor commands and cursor commands.

The general commands are used for opening and closing the database. For example, the following general commands will open and close a database called **health** from inside a Tcl/Tk script.

SQLopendb **health**
SQLclosedb

Most non-cursor commands return values in a two dimensional Tcl array. These commands are very similar to the INGRES SQL commands with the exception that the programmer must specify an array name to hold the returned data. For example, the following simple commands (**Ex.1**) will retrieve all attributes from a table called **"patient"**

where the social security number is equal to 123456789 and places them into an array called keep_it. This example assumes the database is already opened.

Ex.1
SQLselect keep_it * from **Patient** where pssn= "123456789"

Ex.2 will only insert two values into a relation called patient. This is accomplished by specifying the attribute names (ssn, pname) along with their corresponding values. **Ex.2** can be expanded to include any number of attributes up to the maximum for the given relation by specifying additional attributes names and their corresponding values.

Ex.2
SQLopendb health
SQLinsert patient (ssn, pname) values ('"111111111"','"ZOO"')
SQLcommit

Ex.3
SQLselect -noheaders keep_values * from patient where ssn='"111111111"'

The previous "selection process" will return all attributes for the patient with the ssn value of "111111111".

The following is a print out for **Ex.3** .

array name = attribute values
keep_values(1,0) = "111111111"
keep_values(1,1) = "ZOO"
keep_values(1,10) = N/A
keep_values(1,11) = N/A
keep_values(1,2) = N/A
keep_values(1,3) = N/A
keep_values(1,4) = N/A
keep_values(1,5) = N/A
keep_values(1,6) = N/A
keep_values(1,7) = N/A
keep_values(1,8) = N/A
keep_values(1,9) = N/A

N/A has been assigned to all values that were not give a specific values during our previous insertion. This is accomplished automatically by INGRES if no default value is supplied.

The cursor commands allow a user to efficiently step through the results of a select statement one row at a time. The cursor commands that return a value will place the result into a one dimensional array. **Ex.4** is an example which prints all attributes of a table called patient one row at a time or in other words one record at a time.

Ex.4

```
SQLopendb health
SQLdeclareCursor cur
SQLprepareCursor cur  select *  from patient
SQLdescribeCursor cur patient_header
```

The following will print the header of the table patient.
parray patient_header -----> means print array named patient_header

The headers for the patient relation as obtained from the preceding print out follows.

array name **column names of table patient**

```
patient_header(0)  = ssn
patient_header(1)  = pname
patient_header(10) = marital_stat
patient_header(11) = poccupation
patient_header(2)  = dob
patient_header(3)  = sex
patient_header(4)  = establish
patient_header(5)  = paddr
patient_header(6)  = pcity
patient_header(7)  = pstate
patient_header(8)  = pzip
patient_header(9)  = phm_phone
```

The following will print each row within the table called **patient**

```
SQLopenCursor cur
while {[SQLmoreRows]} {
   SQLfetchCursor cur one_row
   parray one_row
}
SQLclosedb;
```

The following is a print out for **Ex.4**.

array name = attribute values

```
one_row(0)  = 123456789
one_row(1)  = Mark Carter
one_row(10) = single
one_row(11) = student
one_row(2)  = 12/12/66
```

```
one_row(3) = male
one_row(4) =
one_row(5) = 1321 Sussex Place
one_row(6) = Norfolk
one_row(7) = VA
one_row(8) = 23508
one_row(9) = 804-489-2997
```

The following example (**Ex.5**) code provides a second method for printing all of the attributes of the patient relation one record at a time.

Ex.5

```
SQLopendb health
SQLdeclareCursor cur
SQLprepareCursor cur select * from patient
SQLdescribeCursor cur find_size

The following code will print each row within the table called patient
SQLopenCursor cur
set size [array size find_size]
SQLfetchCursor cur one_row
while {[SQLmoreRows]} {
   for { set i 0} { $i < $size } { incr i } {
   puts "$find_size($i) = $one_row($i)"
    }
SQLfetchCursor cur one_row
}
SQLclosedb;
```

The following is a print out for **Ex.5**.

attribute name = attribute value

```
ssn = 123456789
pname = Mark Carter
dob = 12/12/66
sex = male
establish =
paddr = 1321 Sussex Place
pcity = Norfolk
pstate = VA
pzip = 23508
phm_phone = 804-489-2997
marital_stat = single
poccupation = student
```

The following is a complete list of SQL commands currently available. For more detailed information on each command use the on line help as follows "man" <command name>. For example, man SQLfetchCursor. The following list was taken from reference [4].

GENERAL TCLSQL COMMANDS

SQLopendb database_name
: Open a database inside tcl

SQLdbName
: Return the name of the currently open database.

SQLcommit
: Commit SQL transactions to the database.

SQLclosedb
: Close the open database inside tcl.

NON-CURSOR TCLSQL COMMANDS

Non-cursor tclsql commands use an internal SQL Data Area (SQLDA) structure. The SQLDA structure is created and initialized with the first call to SQLexec or SQLselect*. The SQLDA structure is freed when SQLclosedb is called.

SQLexec result_array sql_statement
: Execute an SQL statement inside tcl.

SQLselect [-noheaders] [-ld] result_array select_statement
: Execute an SQL select statement inside tcl.

SQLselectRow [-ld] row_number result_array select_statement
: Execute a select statement, get a specific row.

SQLcolNamesSelect [-ld] result_array select_statement
: Execute a select statement, get the column names.

SQLcolTypesSelect [-ld] result_array select_statement
: Execute a select statement, get the column types.

SQLimmediate sql_statement
: Execute and SQL command.

SQLdelete sql_delete_statement
: Delete a row from a table.

SQLinsert sql_insert_statement
: Insert a row into a table.

SQLupdate sql_update_statement
: Update columns in a table.

CURSOR TCLSQL COMMANDS

SQLdeclareCursor cursor
: Declare an SQL cursor.

SQLprepareCursor cursor sql_statement

Setup an internal SQL statement for access from tcl.

SQLdescribeCursor cursor [result_array] [row_indice]

Setup the SQL result area, get the column headers.

SQLopenCursor cursor

Open an SQL cursor for fetching.

SQLisACursor cursor

Return 1 if the cursor exists.

SQLlistCursors result_array [row_indice]

Get the names of all the cursors.

SQLcloseCursor cursor

Close an SQL cursor.

SQLfetchCursor cursor result_array [row_indice]

Fetch a row from an SQL cursor.

SQLmoreRows

Return 1 if there are more rows to fetch from an SQL cursor.

SQLcolNamesCursor cursor result_array [row_indice]

Get the column names of the query associated with a cursor.

SQLcolTypesCursor cursor result_array [row_indice]

Get the column types of the query associated with a cursor.

SQLdeleteCursor table_name cursor

Delete a row from a table.

5 Adding Patient Data

During the patient's first visit to a Dr. he/she will select the icon labeled "First visit" which will bring up another menu with several options one of these options is entitled patient information. This interface will allow a user to input all necessary patient information into an existing database called health. The information requested includes social security number, name , address, physician who referred you, type of injury and the patient's responsible party information. One functionality that was added to ease the task of inputting information was to automatically place the cursor into the adjacent columns or "entry widget" once the user pressed the enter or tab keys. Another useful feature is that of a "status window" which will inform the user of the desired action for each entry. The Figure 3. is a captured image of the interface that will be used by the patient. Even though this interface was originally designed to be used only by the developers of the health care project for inputting test data we thought it was actually aesthetically pleasing enough to be incorporated into the Medical Record System (MRS)

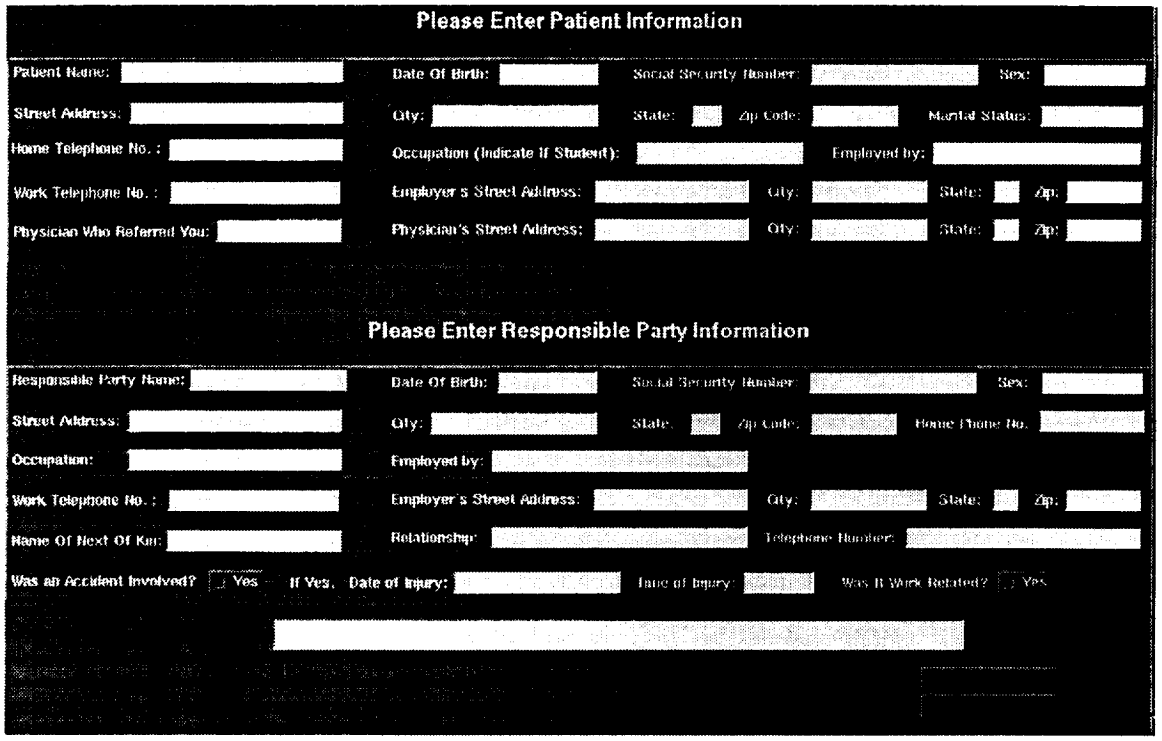


Figure 3. Patient Information Interface

6 Requesting Insurance Information

During the patient's first visit he/she will also need to fill out an insurance form indicating how they will pay for this visit. This is accomplished by selecting the "First Visit" icon which will bring up another menu with several options one of these options is entitled insurance information. This interface will allow a user to input all necessary insurance information into an existing database called health. The information requested includes subscribers name and whether it's their primary, secondary or other insurer. The social security number is obtained from the patient information that will already have been inputted into the database through the patient data interface. This interface has similar features as the interface for patient information mainly automatically placing the cursor into the adjacent columns or "entry widget" once the user has pressed the enter or tab key and providing a "status window". Figure 4. is a captured image of the interface that will be used by the patient to input their insurance information.

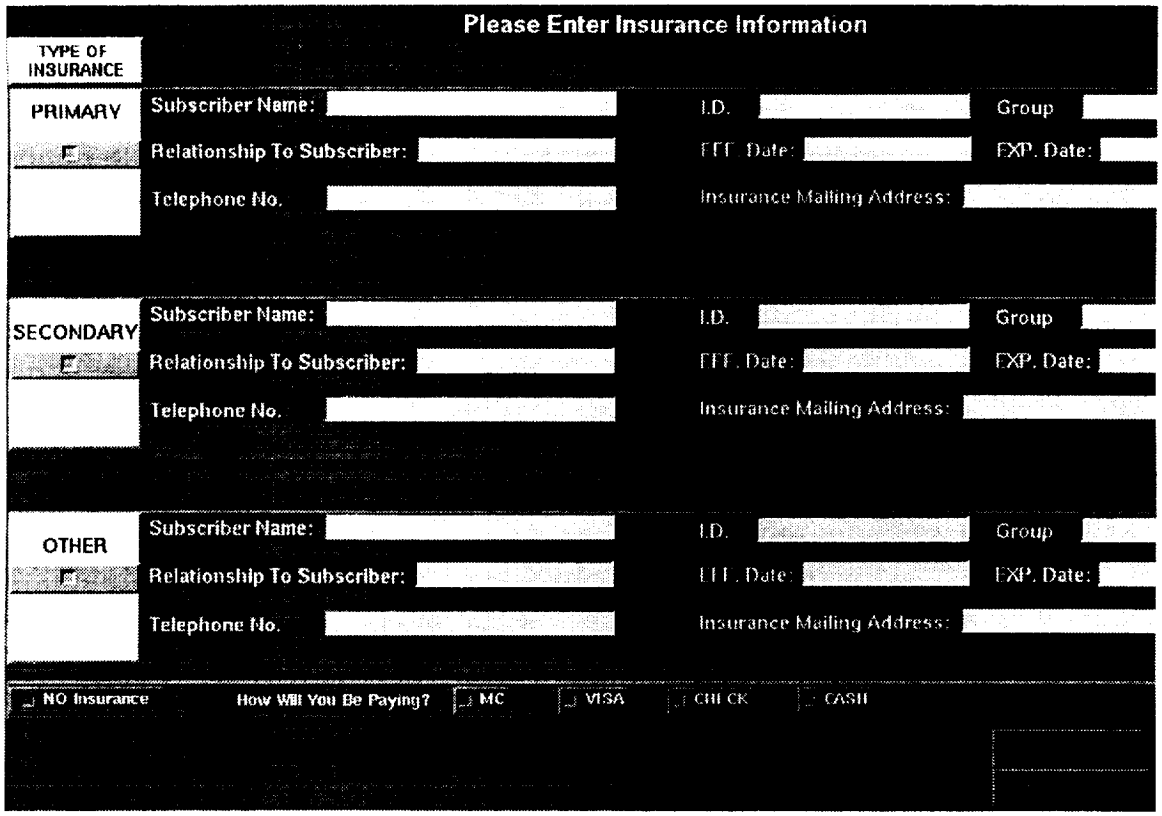


Figure 4. Insurance Information form.

7 Triage Information

This interface was redesigned to include such information as data, social security number and name which will allow for individual triage records for a particular patient to be retrieved based on a particular date. Features such as "partial key" completion is now being provided through the database. The partial key feature is implemented as follows after three letters have been typed a database query for all existing names is done and then a comparison is made between all these names and the partial key if a unique match occurs then the remainder of the word is filled in automatically. One small problem about efficiency, should be mentioned here. It requires some time to retrieve the information from the database, however, it should be possible to create an index of names. An index would provide a quick look-up mechanism for frequently used data. This would solve this problems related to slow data retrieval. Once the information is loaded in to a data structure within the program quick access can be obtained. The interface for the triage information is shown in figure 5.

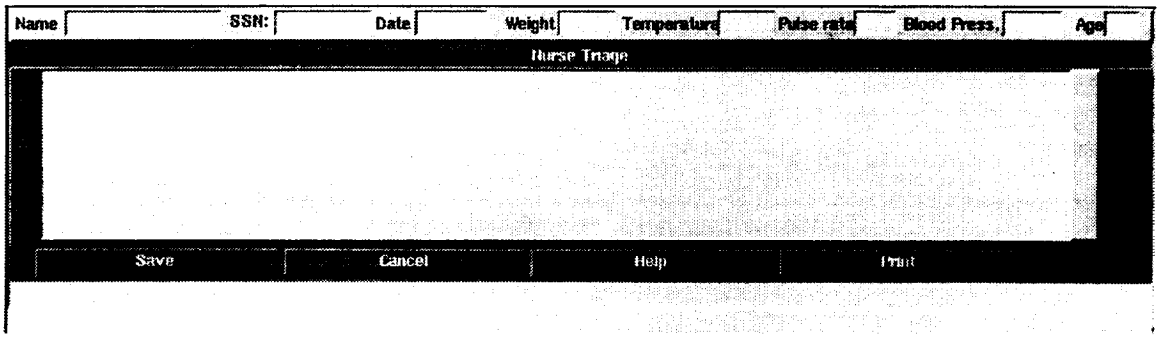


Figure 5. Triage Input Form.

8 Adding Records

Adding records option can be seen after selecting "View patient info" and then selecting Display data. This interface was created to aide in the inclusion of related documents such as scanned images, charts, x-ray images etc... This interface was originally to be used under "view Patient Information Display data" only because there was a reserved "button" called add records. However, it is our belief that this is not the best place to be adding records because the title implies that you are only able to view data. This interface was also designed to add additional patient records to the database but it requires a rather detailed knowledge of the existing code and the relationships that exist among the data that is to be entered through this interface. Because of these requirements this interface was not incorporated into MRS. Solutions to this problem are discussed under the section entitled Problems and Needed Features.

9 Problems and Needed features

It soon became apparent that a problem would occur when new records are added. For example, the original MRS had all the necessary information for establishing links hard coded into the program. This was a satisfactory solution for a prototype or demo of a single patient; however, it was far from being useful in a "real world" application. First, it appears that some of this information may be redundant or even unnecessary. Information such as dates and doctor's name may be obtained from other relations. Next, there needs to be a "selection process" that will enable the user to identify the problem or injury in a systematic way. For example, if a patient enters the hospital with a knee injury the nurse or doctor should be able to identify this problem by selecting it from a "list box" containing common ailments and associated codes. This would prevent the nurse from naming this injury a "leg" problem and the doctor referring to it as a "knee" problem which would cause two separate links to be created one for leg and one for the knee.

Furthermore , this entire concept may require re-thinking with this type of selection process in mind. Another problem is automatically selecting keywords from the triage text.. There may be an algorithm that will determine keywords based on grammar, but I'm not aware of any that can accomplish this task. This area requires further research.

After creating the patient information interface we tested it by adding data into each of the fields that we had created. It became apparent that this interface was requesting too much information, even though, this form had already been reduced substantial from the original form that we was given. Some of the trivial items could be removed. For example, the address of your next of kin or employers complete address. At this time we do not see why a doctor would need this information. Some inquiries into why this information is required is needed

Before this prototype is released to a beta test site it would require making it more user friendly by checking data types and bounds on data etc... This will require some time and was not concerned necessary at this stage in our prototype.

A nice feature would be to include Audio messages for the status windows in addition to the message actually being displayed. There are numerous other places that Audio can and should be used for example the progress notes already contain a section requiring audio to be recorded. AK is an extension to Tcl/Tk which provides audio capabilities to to a Tcl/Tk application. For more information about AK please see reference [1]

10 Conclusions

The MRS prototype now has a working database that can be accessed through Tcl/Tk. There are numerous interfaces to the database already. Most of the graphical interfaces deal with adding to the database. While the data can be retrieved from the database it is now necessary to think of situations where this is likely or necessary to happen. The SQL interface provided through Tcl/Tk is simple enough to be used and it supports most of the SQL commands generally expected or required by any database employing SQL commands. Several problems and missing features with the existing prototype were mentioned.

References

1. Ousterhout, John K., "Tcl and the Tk Toolkit". Addision-Wesley publishing company 1994.

2. Elmasri, Ramez and Navathe B, Shamkant, "Fundamentals of Database Systems. Benjamin/Cummings, Redwood City, CA 1989.

3. Sven Delmas, paper, "Design and Implementation of a Programming Environment for Interactive Construction of Graphical User Interfaces. Berlin, 19 March 1993.

4. Hylands, Christopher. www distribution, tclsql version 1.1 - tcl/SQL interface package, e-mail cxh@eecs.berkeley.edu.

5. Jolly, Carl, Numerous personnel interviews and e-mail correspondences, Summer 1995.