American-Eurasian Journal of Scientific Research 10 (2): 73-87, 2015
ISSN 1818-6785

DOI: 10.5829/idosi.aejsr.2015.10.2.1149

Design and Implementation of a Combinational Logic Design Tutor

Umezinwa Chukwuebuka Ben

Department of Electrical/Electronics Engineering,
Imo State Polytechnics, Umuagwo Ohaji, Imo State, Nigeria

Abstract: Intelligent systems are nature-inspired, mathematically sound and computationally intensive problem solving tools and methodologies that have become extremely important for advancing the current trends in information technology. Artificially intelligent systems currently utilize computers to emulate various faculties of human intelligence and biological metaphors. They use a combination of symbolic and sub-symbolic systems capable of evolving human cognitive skills and intelligence, not just systems capable of doing things humans do not do well. In this thesis, an intelligent system program (software) developed with Microsoft Visual Basic 6.0 evaluates student's background knowledge on combinatorial logic design, simulates a customized training module based on the student's pervious knowledge of the course module and test the student's grasp of the course through feedback. This thesis will be useful to students, lecturers and all the lovers of combinatorial Logic design.

Key words: Logic design · Tutor · Information technology and software

INTRODUCTION

Computers have been employed to achieve a variety of educational goals since the early 1960s. Some of these goals include automated testing and routine drill and practice tasks that had been mechanized with earlier technologies, as far back as the thirties. Other computer assisted instructional programs engage the students in challenging and entertaining reasoning tasks and capitalize on multimedia capabilities to present information [1-4]. Computer- based instruction has successfully penetrated all education and training markets, home, schools, universities, business and government, but remains far from the educational experience. In the early 1970s a few researchers defined a new and ambitious goal for computer-based instruction. They adopted the human tutor as their educational model and sought to apply artificial intelligence techniques to realize this model in "intelligent" computer- based instruction. Personal human tutors provide a highly efficient learning environment and have been estimated to increase mean achievement outcomes by as much as two standard deviations [5-6].

The goal of intelligent tutoring systems (ITSs) would be to engage the students in sustained reasoning activity and to interact with the student based on a deep understanding of the students behavior. If such systems realize even half the impact of human tutors, the payoff for society promised to be substantial.

The first intelligent tutoring program, SCHOLAR merits special recognition and serves to exemplify this pattern [1-7]. This program attempted to engage the student in a mixed initiative dialogue on South American geography. The program and student communicated through a sequence of natural language questions and answers. The tutor could both ask and answer questions and keep track of the ongoing dialogue structure. This tutor was constructed around a semantic network model of domain knowledge. Such network models of conceptual knowledge were revolutionizing our understanding of question answering and inferential reasoning in cognitive science and remain the modal model of conceptual knowledge today [1-3]. However, the effort to sustain a dialogue revealed the importance of unexplored issues in dialogue structure and pragmatic reasoning. This fed into an interesting research program and successive dialogue tutors [8-9] but in the end the fascinating and challenging issues in natural language comprehension took precedence over research in how to deploy dialogue tutors effectively. While there has been a rich intellectual history in intelligent tutoring, we believe that for

Corresponding Author: Umezinwa Chukwuebuka Ben, Department of Electrical/Electronics Engineering,
Imo State Polytechnics, Umuagwo Ohaji, Imo State, Nigeria.

intelligent tutors to seriously penetrate the educational/training system, the evaluative focus must begin to shift to educational impact and away from artificial intelligence sufficiency. This has begun to happen, but at each of the two most recent International Conferences on Intelligent Tutoring Systems [9-11] only 25% of non-invited papers included empirical evaluations of any sort. Among these empirical studies only about one in ten (i.e., 2.5% of all papers) assessed the effectiveness of a computer-based learning environment by comparing SW- dent performance to other learning environments. We believe the emphasis on educational impact must permeate all stages of ITS development, deployment and assessment. After twenty five years we can frame many important questions in effective intelligent tutoring, but can provide only preliminary answers. However, usually such systems are more narrowly conceived of as artificial intelligence systems, more specifically expert systems made to simulate aspects of a human tutor. Intelligent Tutor Systems have been around since the late 1970s, but increased in popularity in the 1990s.

My interest in Intelligent Tutoring Systems (ITS) emerged from my work of using various instructional materials – textbooks and published papers for teaching introductory computer science to non-computer science students. The most recent incarnation of these efforts, Integrated Introduction to Computing drew upon the literature and experience of a number of specialists to create a course which was innovative, engaging and effective. Comparing student performance and instructor ratings for this course with its predecessor (indicates that we met many of our goals: students in this course perform better on standardized tests and a range of programming problems than did students in the previous course [12-16].

Although the average performance and student ratings of the course have improved over its predecessor, the dispersion of these measures is as great as or greater than it had been in the past. This implies that we are having problems reaching students who are well below the mean, losing a great number of the very students who most need this material to be successful in an information-rich world. In addition, we are seeing an increasingly large number of students who bring a richer set of backgrounds to the course than did previous students. These students do not find the course challenging and consequently believe that computer science is uninteresting.

Hence, we lose a number of students who might have had potential to become successful computer science majors.

With these factors in mind, we began to rethink the architecture of the course. i realized that a more modular, finer-grained curriculum for this course could have the potential to address many of these concerns by better targeting a wider range of student interests and abilities. Nonetheless, we had very real constraints on the size of the teaching staff. I knew that we would have to continue to use instructional technology to leverage faculty resources.

Instructional technology is the theory and practice of design, development, utilization, management and evaluation of processes and resources for learning.

Aims and Objectives: This Project is specifically geared towards improving learning of combinatorial logic circuit, reducing the cost of one on one teaching and customized learning. Due to the simulation nature of this project, it reduces the risk, over head cost and logistics required in training. These aims and objectives includes creating an Intelligent Tutoring System that would interact with students,

- Modularize the curriculum.
- Customize it for different student populations.
- Individualize the presentation and assessment of the content.
- Collect data which instructors could use to tutor and remediate students.

Block Diagram of Project: Since this research is a software based one, the block diagram will represent the program modules of the software.

Intelligent Systems for Training: Many characteristics of intelligent systems are readily applicable to training. Although training entails many different areas of aerospace engineering, our experience has been with pilot training. Figure 2.0 presents an implementation of an automated hover training system. This system was implemented in a fixed-base simulation facility and was shown to provide basic hover training skills with no human intervention. The neural network based intelligent system adapts the helicopter dynamics to the student pilot and automatically changes the dynamics of the helicopter as learning progresses.

Methodology & System Analysis

Methodology Used: Therefore in this project the methodology defines the who, what, where, when of then Intelligent combinatorial logic design trainer.

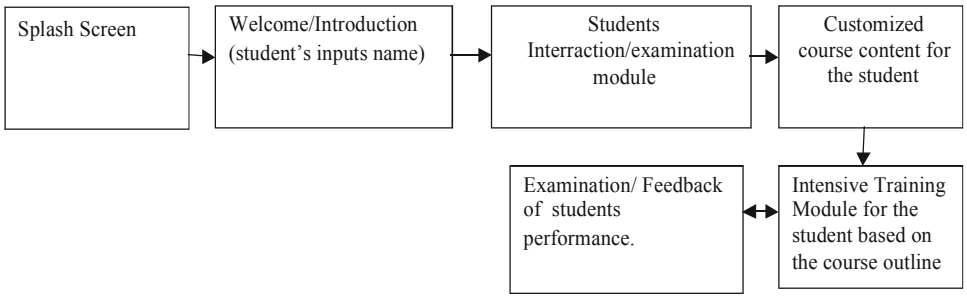

Fig. 1.0: Block Diagram of the Project

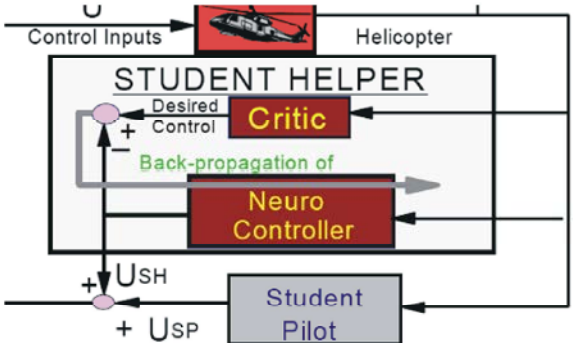

Fig. 2.0: An automated hover training system

WHO: This identifies the target users for this software project, which is the student or any body that wants to learn combinatorial logic design.

WHAT: This explains the software that is being designed, that is the Intelligent combinatorial design tutor developed with Microsoft Visual Basic 6.0

WHERE: The software project is a stand alone project that is platform independent. It runs on any computer system

WHEN: This software can be used any time and any place by any persons

WHY: This part of the methodology outlines the benefits or merits of this project:

- It reduces the cost of one on one teaching
- It increases the learning ability and speed of the students because it involves the students.
- It improves student's eagerness to learn since it is tailored to their abilities.

Structured Analysis and Design: Structured Analysis and Design Technique (SADT) is a diagrammatic notation designed specifically to help people describe and understand systems. It offers building blocks to represent entities and activities and a variety of arrows to relate boxes. These boxes and arrows have an associated informal semantics. SADT can be used as a functional analysis tool of a given process, using successive levels of details. The SADT method allows the definition of user needs for IT developments, which is very used in the industrial Information Systems, but also to explain and to present an activity's manufacturing processes, procedures.

The SADT supplies a specific functional view of any enterprise by describing the functions and their relationships in a company. These functions fulfill the objectives of a company, such as sales, order planning, product design, part manufacturing and human resource management. The SADT can depict simple functional relationships here and can reflect data and control flow relationships between different functions [17-20].

SADT has been developed and field-tested during the period of 1969 to 1973 by Douglas T. Ross and SofTech, Inc.. The methodology was used in the MIT Automatic Programming Tool (APT) project. It received extensive use starting in 1973 by the US Air Force Integrated Computer Aided Manufacturing program [21, 22].

According to Levitt (2000) [22] "it is part of a series of structured methods, that represent a collection of analysis, design and programming techniques that were developed in response to the problems facing the software world from the 1960s to the 1980s. In this timeframe most commercial programming was done in Cobol and Fortran, then C and BASIC. There was little guidance on "good" design and programming techniques and there were no standard techniques for documenting requirements and designs. Systems where getting larger and more complex and the information system development became harder and harder to do so. As a way to help manage large and complex software. Since the end 1960 multiple Structured Methods emerged".

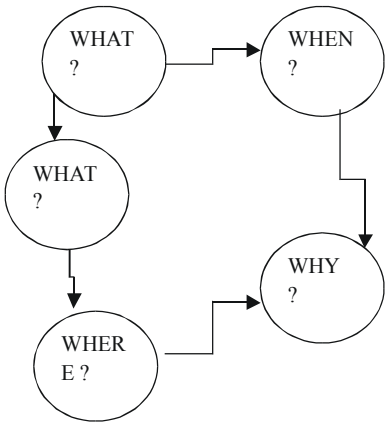

Fig. 3.0: Methodology Circle

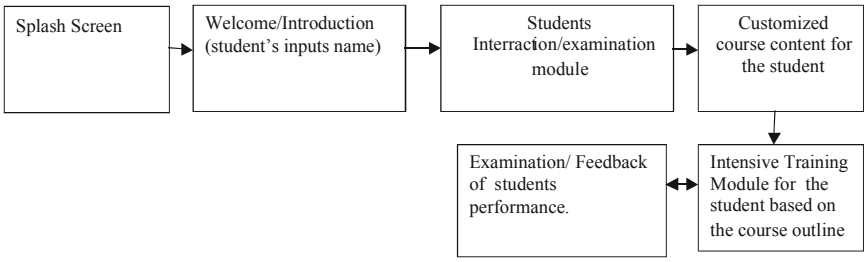

Fig. 4: Structured Diagram of Envisaged System

Top Down Design: Top-down and bottom-up are strategies of information processing and knowledge ordering, mostly involving software, but also other humanistic and scientific theories (see systemics). In practice, they can be seen as a style of thinking and teaching. In many cases *top-down* is used as a synonym of *analysis* or *decomposition* and *bottom-up* of *synthesis*.

A top-down approach is essentially the breaking down of a system to gain insight into its compositional sub-systems. In a top-down approach an overview of the system is first formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. A top-down model is often specified with the assistance of "black boxes", these make it easier to manipulate. However, black boxes may fail to elucidate elementary mechanisms or be detailed enough to realistically validate the model.

Top-down approaches emphasize planning and a complete understanding of the system. It is inherent that no coding can begin until a sufficient level of detail has been reached in the design of at least some part of the system. The Top-Down Approach is done by attaching the stubs in place of the module. This, however, delays testing of the ultimate functional units of a system until significant design is complete.

Top-down design was promoted in the 1970s by IBM researcher Harlan Mills and Niklaus Wirth. Mills developed structured programming concepts for practical use and tested them in a 1969 project to automate the *New York Times* morgue index. The engineering and management success of this project led to the spread of the top-down approach through IBM and the rest of the computer industry. Among other achievements, Niklaus Wirth, the developer of Pascal programming language, wrote the influential paper *Program Development by Stepwise Refinement*. Since Niklaus Wirth went on to develop languages such as Modula and Oberon (where one could define a module before knowing about the entire program specification), one can infer that top down programming was not strictly what he promoted. Top-down methods were favored in software engineering until the late 1980s and object-oriented programming assisted in demonstrating the idea that both aspects of top-down and bottom-up programming could be utilized.

Modern software design approaches usually combine both top-down and bottom-up approaches. Although an understanding of the complete system is usually considered necessary for good design, leading theoretically to a top-down approach, most software projects attempt to make use of existing code to some degree. Pre-existing modules give designs a bottom-up flavor. Some design approaches also use an approach

where a partially-functional system is designed and coded to completion and this system is then expanded to fulfill all the requirements for the project.

Top-Down Approach: Practicing top-down programming has several advantages:

- Separating the low level work from the higher level abstractions leads to a modular design.
- Modular design means development can be self contained.
- Having "skeleton" code illustrates clearly how low level modules integrate.
- Fewer operations errors (to reduce errors, because each module has to be processed separately, so programmers get large amount of time for processing).
- Much less time consuming (each programmer is only involved in a part of the big project).
- Very optimized way of processing (each programmer has to apply their own knowledge and experience to their parts (modules), so the project will become an optimized one).
- Easy to maintain (if an error occurs in the output, it is easy to identify the errors generated from which module of the entire program).

Bottom up Design: A bottom-up approach is the piecing together of systems to give rise to grander systems, thus making the original systems sub-systems of the emergent system. In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. This strategy often resembles a "seed" model, whereby the beginnings are small but eventually grow in complexity and completeness. However, "organic strategies" may result in a tangle of elements and subsystems, developed in isolation and subject to local optimization as opposed to meeting a global purpose.

Bottom-up emphasizes coding and early testing, which can begin as soon as the first module has been specified. This approach, however, runs the risk that modules may be coded without having a clear idea of how they link to other parts of the system and that such linking may not be as easy as first thought. Re-usability of code is one of the main benefits of the bottom-up approach.

In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. This strategy often resembles a "seed" model, whereby the beginnings are small, but eventually grow in complexity and completeness.

Object-oriented programming (OOP) is a programming paradigm that uses "objects" to design applications and computer programs.

In Mechanical Engineering with software programs such as Pro/ENGINEER, Solidworks and Autodesk Inventor users can design products as pieces not part of the whole and later add those pieces together to form assemblies like building LEGO. Engineers call this piece part design.

This bottom-up approach has one weakness. We need to use a lot of intuition to decide the functionality that is to be provided by the module. If a system is to be built from existing system, this approach is more suitable as it starts from some existing modules.

Pro/ENGINEER (as well as other commercial Computer Aided Design (CAD) programs) does however hold the possibility to do Top-Down design by the use of so-called *skeletons*. They are generic structures that hold information on the overall layout of the product. Parts can inherit interfaces and parameters from this generic structure. Like parts, skeletons can be put into a hierarchy. Thus, it is possible to build the overall layout of a product before the parts are designed.

Parsing: Parsing is the process of analyzing an input sequence (such as that read from a file or a keyboard) in order to determine its grammatical structure. This method is used in the analysis of both natural languages and computer languages, as in a compiler.

Bottom-up parsing is a strategy for analyzing unknown data relationships that attempts to identify the most fundamental units first and then to infer higher-order structures from them. Top-down parsers, on the other hand, hypothesize general parse tree structures and then consider whether the known fundamental structures are compatible with the hypothesis.

Choice Design Approach: Due to the complexity of this software project, I was careful to choose the kind of design approach to use in other to present my modules simple enough for the user to use.

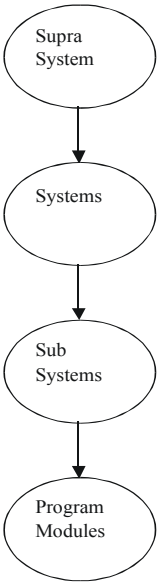

Fig. 4: Representing the Top down design method

Therefore I chose to use the *top down design method* which is mostly used in software development because of its simplicity and because an overview of the system is first formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. The graphical diagram of a Top down Design method is shown below. From the diagram it shows that the system starts from the supra system down to the program module.

Using the top down design method in my project, the project was divided into the modules,

- The Flash Screen module
- Welcome and name input module

Information Gathering: The expert information or data used to prepare the training materials for this project was obtained through.

- Through text books and materials used as references
- Through internet surfing and research (also stated on the reference).

Limitations of the Existing System: In this project, the existing system which is the manual or physical training of students has so many limitations that this project is out to tackle. Such limitations include:

- Low teaching efficiency because of difference in students ability
- Poor documentation of course outline for learning.
- Poor or absence of tracking student's performance and improving on it.
- Teachers may get tired of after a long teaching period.

Overview of the Envisaged System: The project overcomes the limitations of the existing system of one on one teaching, the above block diagram identifies the new system.

The first block in the diagram shows the welcome screen, that welcomes the user to the software.

Block 2: The user is asked to input his/her name so as to access the examination module.

Block 3: The assessment/ examination module, here the user is examined to enable the software design his/her course outline.

Block 4: The course outline is listed out for the student

Block 5: Training takes place here, in this module.

Block 6: Experiment and examination are conducted here.

The new system

- It increases teaching efficiency.
- It increases the learning ability and speed of the students because it involves the students.
- It improves student's eagerness to learn since it is tailored to their abilities.

System Design

Software Subsystem Implementation: The software of this project was developed with Microsoft Visual Basic; Visual Basic (VB) is the third-generation event-driven programming language and integrated development environment (IDE) from Microsoft for its COM programming model. VB is also considered a relatively easy to learn and use programming language, because of its graphical development features and BASIC heritage.

Visual Basic was derived from BASIC and enables the rapid application development (RAD) of graphical user interface (GUI) applications, access to databases using Data Access Objects, Remote Data Objects, or ActiveX Data Objects and creation of ActiveX controls and objects. Scripting languages such as VBA and VBScript are syntactically similar to Visual Basic, but perform differently.

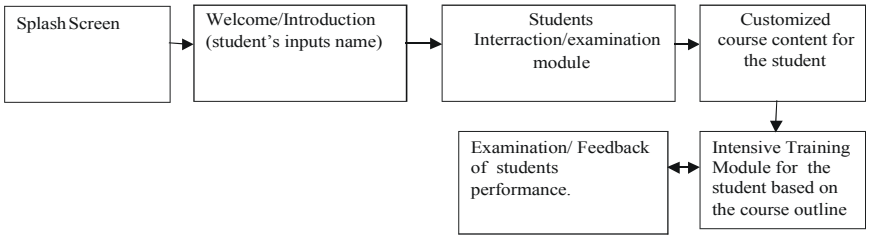

Fig. 5: Overview of Envisaged System

A programmer can put together an application using the components provided with Visual Basic itself. Programs written in Visual Basic can also use the Windows API, but doing so requires external function declarations.

The final release was version 6 in 1998. Microsoft's extended support ended in March 2008 and the designated successor was Visual Basic. NET (now known simply as Visual Basic).

Language features Like the BASIC programming language, Visual Basic was designed to be easily learned and used by beginner programmers. The language not only allows programmers to create simple GUI applications, but can also develop complex applications. Programming in VB is a combination of visually arranging components or controls on a form, specifying attributes and actions of those components and writing additional lines of code for more functionality. Since default attributes and actions are defined for the components, a simple program can be created without the programmer having to write many lines of code. Performance problems were experienced by earlier versions, but with faster computers and native code compilation this has become less of an issue.

Although programs can be compiled into native code executables from version 5 onwards, they still require the presence of runtime libraries of approximately 1 MB in size. This runtime is included by default in Windows 2000 and later, but for earlier versions of Windows like 95/98/NT it must be distributed together with the executable.

Forms are created using drag-and-drop techniques. A tool is used to place controls (e.g., text boxes, buttons, etc.) on the form (window). Controls have attributes and event handlers associated with them. Default values are provided when the control is created, but may be changed by the programmer. Many attribute values can be modified during run time based on user actions or changes in the environment, providing a dynamic application. For example, code can be inserted into the form resize event handler to reposition a control so that it remains centered on the form, expands to fill up the form, etc. By inserting code into the event handler for a keypress in a text box, the program can automatically translate the case of the text being entered, or even prevent certain characters from being inserted.

Visual Basic can create executables (EXE files), ActiveX controls, or DLL files, but is primarily used to develop Windows applications and to interface database systems. Dialog boxes with less functionality can be used to provide pop-up capabilities. Controls provide the basic functionality of the application, while programmers can insert additional logic within the appropriate event handlers. For example, a drop-down combination box will automatically display its list and allow the user to select any element. An event handler is called when an item is selected, which can then execute additional code created by the programmer to perform some action based on which element was selected, such as populating a related list.

Alternatively, a Visual Basic component can have no user interface and instead provide ActiveX objects to other programs via Component Object Model (COM). This allows for server-side processing or an add-in module.

Component Object Model (COM): Is a binary-interface standard for software componentry introduced by Microsoft in 1993. It is used to enable inter-process communication and dynamic object creation in a large range of programming languages. The term *COM* is often used in the Microsoft software development industry as an umbrella term that encompasses the OLE, OLE Automation, ActiveX, COM+ and DCOM technologies.

The essence of COM is a language-neutral way of implementing objects that can be used in environments different from the one in which they were created, even across machine boundaries. For well-authored components, COM allows reuse of objects with no knowledge of their internal implementation, as it forces component implementers to provide well-defined interfaces that are separate from the implementation. The different allocation semantics of languages are accommodated by making objects responsible for their

own creation and destruction through reference-counting. Casting between different interfaces of an object is achieved through the Query Interface function. The preferred method of inheritance within COM is the creation of sub-objects to which method calls are delegated.

The language is garbage collected using reference counting, has a large library of utility objects and has basic object oriented support. Since the more common components are included in the default project template, the programmer seldom needs to specify additional libraries. Unlike many other programming languages, Visual Basic is generally not case sensitive, although it will transform keywords into a standard case configuration and force the case of variable names to conform to the case of the entry within the symbol table. String comparisons are case sensitive by default, but can be made case insensitive if so desired.

The Visual Basic compiler is shared with other Visual Studio languages (C, C++), but restrictions in the IDE do not allow the creation of some targets (Windows model DLLs) and threading models.

Visual Basic has the following traits which differ from C-derived languages:

Multiple assignment available in C language is not possible. A = B = C does not imply that the values of A, B and C are equal. The boolean result of "Is B = C?" is stored in A. The result stored in A could therefore be false(0) or true(-1) Boolean constant True has numeric value -1.[3] This is because the Boolean data type is stored as a 16-bit signed integer. In this construct -1 evaluates to 16 binary 1s (the Boolean value True) and 0 as 16 0s (the Boolean value False). This is apparent when performing a Not operation on a 16 bit signed integer value 0 which will return the integer value -1, in other words, True = Not False. This inherent functionality becomes especially useful when performing logical operations on the individual bits of an integer such as And, Or, Xor and Not. [4]. This definition of True is also consistent with BASIC since the early 1970s Microsoft BASIC implementation and is also related to the characteristics of CPU instructions at the time.

Logical and bitwise operators are unified. This is unlike some C-derived languages (such as Perl), which have separate logical and bitwise operators. This again is a traditional feature of BASIC.

Variable Array Base: Arrays are declared by specifying the upper and lower bounds in a way similar to Pascal and Fortran. It is also possible to use the Option Base statement to set the default lower bound. Use of the Option Base statement can lead to confusion when reading Visual Basic code and is best avoided by always explicitly specifying the lower bound of the array. This lower bound is not limited to 0 or 1, because it can also be set by declaration. In this way, both the lower and upper bounds are programmable. In more subscript-limited languages, the lower bound of the array is not variable. This uncommon trait does exist in Visual Basic NET but not in VBScript.

OPTION BASE was introduced by ANSI, with the standard for ANSI Minimal BASIC in the late 1970s.

Relatively strong integration with the Windows operating system and the Component Object Model. The native types for strings and arrays are the dedicated COM types, BSTR and SAFEARRAY

Banker's rounding as the default behavior when converting real numbers to integers with the Round function. [5] ? Round(2.5, 0) gives 2, ? Round(3.5, 0) gives 4.

Integers are automatically promoted to reals in expressions involving the normal division operator (/) so that division of an odd integer by an even integer produces the intuitively correct result. There is a specific integer divide operator (\) which does truncate.

By default, if a variable has not been declared or if no type declaration character is specified, the variable is of type Variant. However this can be changed with Deftype statements such as DefInt, DefBool, DefVar, DefObj, DefStr. There are 12 Deftype statements in total offered by Visual Basic 6.0. The default type may be overridden for a specific declaration by using a special suffix character on the variable name (# for Double, ! for Single, & for Long, % for Integer, $ for String and @ for Currency) or using the key phrase As (type). VB can also be set in a mode that only explicitly declared variables can be used with the command Option Explicitly.

Determining Data to Be Stored: In a majority of cases, a person who is doing the design of a database is a person with expertise in the area of database design, rather than expertise in the domain from which the data to be stored is drawn e.g. financial information, biological information etc. Therefore the data to be stored in the database must be determined in cooperation with a person who does have expertise in that domain and who is aware of what data must be stored within the system.

This process is one which is generally considered part of requirements analysis and requires skill on the part of the database designer to elicit the needed information from those with the domain knowledge. This is because those with the necessary domain knowledge frequently cannot express clearly what their system requirements for the database are as they are unaccustomed to thinking in terms of the discrete data elements which must be stored. Data to be stored can be determined by Requirement Specification.

Normalization: The process of applying the rules to your database design is called normalizing the database, or just normalization. Normalization is most useful after you have represented all of the information items and have arrived at a preliminary design. The idea is to help you ensure that you have divided your information items into the appropriate tables. What normalization cannot do is ensure that you have all the correct data items to begin with. You apply the rules in succession, at each step ensuring that your design arrives at one of what is known as the "normal forms." Five normal forms are widely accepted-the first normal form through the fifth normal form. This article expands on the first three, because they are all that is required for the majority of database designs.

First Normal Form: First normal form states that at every row and column intersection in the table there, exists a single value and never a list of values. For example, you cannot have a field named Price in which you place more than one Price. If you think of each intersection of rows and columns as a cell, each cell can hold only one value.

Second Normal Form: Second normal form requires that each non-key column be fully dependent on the entire primary key, not on just part of the key. This rule applies when you have a primary key that consists of more than one column. For example, suppose you have a table containing the following columns, where Order ID and Product ID form the primary key:

Order ID (primary key)
Product ID (primary key)
Product Name

This design violates second normal form, because Product Name is dependent on Product ID, but not on Order ID, so it is not dependent on the entire primary key. You must remove Product Name from the table. It belongs in a different table (Products).

Third Normal Form: Third normal form requires that not only every non-key column be dependent on the entire primary key, but that non-key columns be independent of each other. Another way of saying this is that each non-key column must be dependent on the primary key and nothing but the primary key. For example, suppose you have a table containing the following columns:

Product ID (primary key)
Name
SRP
Discount

Assume that Discount depends on the suggested retail price (SRP). This table violates third normal form because a non-key column, Discount, depends on another non-key column, SRP. Column independence means that you should be able to change any non-key column without affecting any other column. If you change a value in the SRP field, the Discount would change accordingly, thus violating that rule. In this case Discount should be moved to another table that is keyed on SRP.

Input/output Arrangement: In computing, input/output, or I/O, refers to the communication between an information processing system (such as a computer) and the outside world possibly a human, or another information processing system. Inputs are the signals or data received by the system and outputs are the signals or data sent from it. The term can also be used as part of an action; to "perform I/O" is to perform an input or output operation. I/O devices are used by a person (or other system) to communicate with a computer. For instance, a keyboard or a mouse may be an input device for a computer, while monitors and printers are considered output devices for a computer. Devices for communication between computers, such as modems and network cards, typically serve for both input and output. Note that the designation of a device as either input or output depends on the perspective. Mouse and keyboards take as input physical movement that the human user outputs and convert it into signals that a computer can understand. The output from these devices is input for the computer. Similarly, printers and monitors take as input signals that a computer outputs. They then convert these signals into representations that human users can see or read. For a human user the process of reading or seeing these representations is receiving input. These interactions between computers and humans is studied in a field called human–computer interaction.

Project Block Diagram

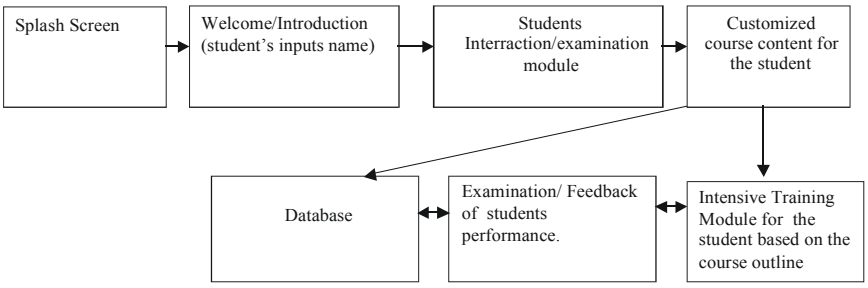

Fig. 6: Project Block Diagram

In computer architecture, the combination of the CPU and main memory (i.e. memory that the CPU can read and write to directly, with individual instructions) is considered the brain of a computer and from that point of view any transfer of information from or to that combination, for example to or from a disk drive, is considered I/O. The figure below represents the project block diagram with the database well represented. The database holds the customized course content for students, text questions and the feedback response for students [23-25].

System Testing: System testing of software or hardware is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of black box testing and as such, should require no knowledge of the inner design of the code or logic.

As a rule, system testing takes, as its input, all of the "integrated" software components that have successfully passed integration testing and also the software system itself integrated with any applicable hardware system(s). The purpose of integration testing is to detect any inconsistencies between the software units that are integrated together (called *assemblages*) or between any of the *assemblages* and the hardware. System testing is a more limiting type of testing; it seeks to detect defects both within the "inter-assemblages" and also within the system as a whole.

System testing is performed on the entire system in the context of a Functional Requirement Specification(s) (FRS) and/or a System Requirement Specification (SRS). System testing is an *investigatory* testing phase, where the focus is to have almost a destructive attitude[l]and tests not only the design, but also the behaviour and even the believed expectations of the customer. It is also intended to test up to and beyond the bounds defined in the software/hardware requirements specification(s).

Types of Tests Done on the Project Software: The following examples are different types of testing that I performed during System testing:

- GUI software testing
- Usability testing
- Performance testing
- Compatibility testing
- Error handling testing
- Reliability Testing.

GUI Software Testing: In computer science, GUI software testing is the process of testing a product that uses a graphical user interface, to ensure it meets its written specifications. This is normally done through the use of a variety of test cases. To generate a 'good' set of test cases, the test designer must be certain that their suite covers all the functionality of the system and also has to be sure that the suite fully exercises the GUI itself. The difficulty in accomplishing this task is twofold: one has to deal with domain size and then one has to deal with sequences. In addition, the tester faces more difficulty when they have to do regression testing [4-6].

The size problem can be easily illustrated. Unlike a CLI (command line interface) system, a GUI has many operations that need to be tested. A very small program such as Microsoft WordPad has 325 possible GUI operations[1]. In a large program, the number of operations can easily be an order of magnitude larger.

The second problem is the sequencing problem. Some functionality of the system may only be accomplishable by following some complex sequence of GUI events. For example, to open a file a user may have to click on the File Menu and then select the Open operation and then use a dialog box to specify the file name and then focus the application on the newly opened window.

Obviously, increasing the number of possible operations increases the sequencing problem exponentially. This can become a serious issue when the tester is creating test cases manually.

Usability Testing: This is a technique used to evaluate a product by testing it on users. This can be seen as an irreplaceable usability practice, since it gives direct input on how real users use the system.[1] This is in contrast with usability inspection methods where experts use different methods to evaluate a user interface without involving users.

Usability testing focuses on measuring a human-made product's capacity to meet its intended purpose. Examples of products that commonly benefit from usability testing are foods, consumer products, web sites or web applications, computer interfaces, documents and devices. Usability testing measures the usability, or ease of use, of a specific object or set of objects, whereas general human-computer interaction studies attempt to formulate universal principles

Usability testing is a black-box testing technique. The aim is to observe people using the product to discover errors and areas of improvement. Usability testing generally involves measuring how well test subjects respond in four areas: efficiency, accuracy, recall and emotional response. The results of the first test can be treated as a baseline or control measurement; all subsequent tests can then be compared to the baseline to indicate improvement.

- *Performance* -- How much time and how many steps, are required for people to complete basic tasks? (For example, find something to buy, create a new account and order the item.)
- *Accuracy* -- How many mistakes did people make? (And were they fatal or recoverable with the right information?)
- *Recall* -- How much does the person remember afterwards or after periods of non-use?
- *Emotional response* -- How does the person feel about the tasks completed? Is the person confident, stressed? Would the user recommend this system to a friend?

Methods of Usability Test: Setting up a usability test involves carefully creating a scenario, or realistic situation, wherein the person performs a list of tasks using the product being tested while observers watch and take notes. Several other test instruments such as scripted instructions, paper prototypes and pre- and post-test questionnaires are also used to gather feedback on the product being tested. For example, to test the attachment function of an e-mail program, a scenario would describe a situation where a person needs to send an e-mail attachment and ask him or her to undertake this task. The aim is to observe how people function in a realistic manner, so that developers can see problem areas and what people like. Techniques popularly used to gather data during a usability test include think aloud protocol and eye tracking.

Hallway Testing: Hallway testing (or Hall Intercept Testing) is a general methodology of usability testing. Rather than using an in-house, trained group of testers, just five to six random people, indicative of a cross-section of end users, are brought in to test the product, or service. The name of the technique refers to the fact that the testers should be random people who pass by in the hallway

Remote Testing: Remote usability testing (also known as unmoderated or asynchronous usability testing) involves the use of a specially modified online survey, allowing the quantification of user testing studies by providing the ability to generate large sample sizes. Similar to an in-lab study, a remote usability test is task-based and the platforms allow you to capture clicks and task times. Hence, for many large companies this allows you to understand the WHY behind the visitors intents when visiting a website or mobile site. Additionally, this style of user testing also provides an opportunity to segment feedback by demographic, attitudinal and behavioural type.

The tests are carried out in the user's own environment (rather than labs) helping further simulate real-life scenario testing. This approach also provides a vehicle to easily solicit feedback from users in remote areas.

Performance Testing: This covers a broad range of engineering or functional evaluations where a material, product, system, or person is not specified by detailed material or component specifications: rather, emphasis is on the final measurable performance characteristics.

Performance testing can refer to the assessment of the performance of a human examinee. For example, a behind-the-wheel driving test is a performance test of whether a person is able to perform the functions of a competent driver of an automobile.

In the computer industry, software performance testing is used to determine the speed or effectiveness of a computer, network, software program or device. This process can involve quantitative tests done in a lab, such as measuring the response time or the number of MIPS (millions of instructions per second) at which a system functions. Qualitative attributes such as reliability, scalability and interoperability may also be evaluated. Performance testing is often done in conjunction with stress testing.

Compatibility Testing: This is a part of software non-functional tests, is testing conducted on the application to evaluate the application's compatibility with the computing environment. Computing environment may contain some or all of the below mentioned elements:

- Computing capacity of Hardware Platform (IBM 360, HP 9000, etc.)..
- Bandwidth handling capacity of networking hardware
- Compatibility of peripherals (Printer, DVD drive, etc.)
- Operating systems (MVS, UNIX, Windows, etc.)
- Database (Oracle, Sybase, DB2, etc.)
- Other System Software (Web server, networking/ messaging tool, etc.)
- Browser compatibility (Firefox, Netscape, Internet Explorer, Safari, etc.)

Browser compatibility testing, can be more appropriately referred to as user experience testing. This requires that the web applications are tested on different web browsers, to ensure the following:

- Users have the same visual experience irrespective of the browsers through which they view the web application.
- In terms of functionality, the application must behave and respond the same way across different browsers.

Exception Handling: This is a programming language construct or computer hardware mechanism designed to handle the occurrence of exceptions, special conditions that change the normal flow of program execution.

Programming languages differ considerably in their support for exception handling (as distinct from error checking, which is normal program flow that codes for responses to adverse contingencies such as invalid state changes or the unsuccessful termination of invoked operations). The degree to which such explicit validation and error checking is necessary is in contrast to exception handling support provided by any given programming environment. Hardware exception handling differs somewhat from the support provided by software tools, but similar concepts and terminology are prevalent.

In general, an exception is *handled* (resolved) by saving the current state of execution in a predefined place and switching the execution to a specific subroutine known as an *exception handler*. Depending on the situation, the handler may later resume the execution at the original location using the saved information. For example, a page fault will usually allow the program to be resumed, while a division by zero might not be resolvable transparently.

From the processing point of view, hardware interrupts are similar to resume-able exceptions, though they are typically unrelated to the user's program flow.

From the point of view of the author of a routine, raising an exception is a useful way to signal that a routine could not execute normally. For example, when an input argument is invalid (e.g. a zero denominator in division) or when a resource it relies on is unavailable (like a missing file, or a hard disk error). In systems without exceptions, routines would need to return some special error code. However, this is sometimes complicated by the semipredicate problem, in which users of the routine need to write extra code to distinguish normal return values from erroneous ones.

In runtime engine environments such as Java or NET, there exist tools that attach to the runtime engine and every time that an exception of interest occurs, they record debugging information that existed in memory at the time the exception was thrown (call stack and heap values). These tools are called automated exception handling or error interception tools and provide 'root-cause' information for exceptions.

Contemporary applications face many design challenges when considering exception handling strategies. Particularly in modern enterprise level applications, exceptions must often cross process boundaries and machine boundaries. Part of designing a solid exception handling strategy is recognizing when a process has failed to the point where it cannot be economically handled by the software portion of the process.

Reliability Testing: This was developed apart from the mainstream of probability and statistics. It was originally a tool to help nineteenth century maritime insurance and life insurance companies compute profitable rates to charge their customers. Even today, the terms "failure rate" and "hazard rate" are often used interchangeably.

The failure of mechanical devices such as ships, trains and cars, is similar in many ways to the life or death of biological organisms. Statistical models appropriate for any of these topics are generically called "time-to-event" models. Death or failure is called an "event" and the goal is to project or forecast the rate of events for a given population or the probability of an event for an individual.

When reliability is considered from the perspective of the consumer of a technology or service, actual reliability measures may differ dramatically from perceived reliability. One bad experience can be magnified in the mind of the customer, inflating the perceived unreliability of the product. One plane crash where hundreds of passengers die will immediately instill fear in a large percentage of the flying consumer population, regardless of actual reliability data about the safety of air travel.

Reliability period of any object is measured within the durability period of that object.

Test Plan: Software testing can be approached in different ways for example one may decide to proceed in a top-down fashion in which case.

- The main system driver is tested first using dommies to replace sub-systems.
- The sub-system driver for each of the sub-system is tested.
- Finally, for each sub-system the software modules are tested one after the other to complete then top down test plan.

An alternative to top-down test plan is the bottom-up test plan. In this:

- The program module comprising each sub-system are tested one after the other
- The sub-system driver for each of the sub-system is tested to ensure that control is transferred to the appropriate program modules.
- Finally the main system driver is tested.

Because the lower levels are available before any top level is tested. There are other test approaches but the two mentioned above lie within the scope of this project.

Software Sub-System Testing: In the software sub-system testing, i used the bottom-up test plan, which includes the types of test I listed above on the system testing module.

I generated adequate test data for each level of testing namely:

- System Driver
- Sub-systems
- Program modules

An adequate test data set at each level can be defined as that data that exercises all or most of the alternative branches in result for that data item.

Performance Evaluation: A performance appraisal, performance review, or (career) development discussion is a method by which the performance of a software is evaluated (generally in terms of quality, quantity, cost and time) typically by the corresponding manager or supervisor. A performance appraisal is a part of guiding and managing software development. It is the process of obtaining, analyzing and recording information about the relative worth of a software to an organization. Performance appraisal is an analysis of a software's recent successes and failures, strengths and weaknesses and suitability for upgrade or debugging. It is also the judgement of a software's performance in a job based on considerations other than productivity alone.

Project Packaging: Since this is a software based project, it can only be packaged in converting the program codes into an executable file. This can be done from Microsoft Visual basic through these steps

- Click on file menu
- Click on make executable
- Select the location to save the file

After that the file becomes an executable compiled file that can run on any computer or platform. The file can now be burnt into a cd-rom for distribution.

Project Costing: The cost of this project is broken down into.

- Visual Basic studio cd-rom ₦1000
- Project typing ₦3000
- Total cost: ₦4000

Deployment: The software will be deployed for user, after it is burnt into the cd-roms provided.

DISCUSSION

This Project has specifically improved learning of combinatorial logic circuit, reducing the cost of one on one teaching and customized learning. Due to the

simulation nature of this project, it has reduced the risk, over head cost and logistics required in one to one training. This software was used by the Department of Electrical/ Electronic Engineering (Federal University of Technology Owerri) to teach a course called Introduction to Digital electronics (EEE 204) and these are the achievements of this project after the classroom simulation.

- To Interact with students,
- Modularize the curriculum.
- Customize it for different student populations.
- Individualize the presentation and assessment of the content.
- Collect data which instructors could use to tutor and remediate students.

Problems Encountered and Solutions: The problems I encountered while undertaking this project includes:

- Limited power supply to finish the software coding and typing of report.
- Scarcity of original Microsoft Visual Basic Cd-rom.
- Limited Funds

Recommendations/ Suggestions for Further Improvement: Due to the importance and significance of this project, I recommend the following for future researchers:

- The program software should be design to run on the internet for wider use
- Multimedia (videos and sounds) should be included in the training materials for more interesting learning experience.
- The interactive aspects of this software should be increased.
- For more over all completion, a hardware should be interfaced to this project.

CONCLUSION

Personal human tutors provide a highly efficient learning environment and have been estimated to increase mean achievement outcomes by as much as two standard deviations. The goal of intelligent tutoring systems (ITSs) would be to engage the students in sustained reasoning activity and to interact with the student based on a deep understanding of the students behavior. If such systems realize even half the impact of human tutors, the payoff for society promised to be substantial.

REFERENCES

1. Anderson, J.R., 1983. The Architecture of Cognition. Cambridge, Massachusetts: Harvard University Press, pp: 4.
2. Mayer, R.E., 1988. Teaching and Learning Computer Programming: Multiple Research Perspectives. Hillsdale, New Jersey: Lawrence Erlbaum Associates, pp: 7-9.
3. Olson, G.M., S. Sheppard and E. Soloway, 1987. Empirical Studies of Programmers: Second Workshop. Norwood, New Jersey: Ablex Publishing Corporation, pp: 9-13.
4. Anderson, J.R. and G.H. Bower, 1973. Human associative memory. Washington, DC: V.H Winston and Sons., pp: 12.
5. Anderson, J.R., 1983. The architecture of cognition. Cambridge, MA: Harvard University Press, pp: 3-4.
6. Bloom, B.S., 1984. The 2 sigma problem: The search for methods of group instruction as effective as one-to- one tutoring. Educational Researcher, 13: 4-16.
7. Carbonell, 1970. Al in CA!: An artificial intelligence approach to computer-assisted instruction. Washington, DC: V.H Winston and Sons, pp: 12.
8. IEEE Transactions on Man-Machine Systems, 11,190-202.
9. Cohen, P.A., J.A. Kulik and C.C. Kulik, 1982. Educational outcomes of tutoring: A meta analysis of findings. American Educational Research Journal, 19: 237-248.
10. Collins, A.M. and E.F. Loftus, 1975. A spreading-activation theory of semantic processing. Psychology cal Review, 82: 407-428.
11. Frasson, C., G. Gauthier and A. Lesgold, 1996. Intelligent tutoring systems: Third International Conference, ITS'96. New York: Springer-Verlag.
12. Lesgold, A., S. Lajoie, M. Bunzo and G. Eggan, 1992. SHERLOCK: A coached practice environment for electronics troubleshooting job. In J. Larkin, pp: 6.
13. Weinshank, D.J., M. Urban-Lurain and T. Danieli, 1988. Introduction to Computing: Telecourse with Structured BASIC. (2nd ed.). Dubuque, Iowa: Kendall/Hunt Publishing Company, pp: 5.
14. Weinshank, D.J., M. Urban-Lurain, T. Danieli and G. McCuaig, 1995. Integrated Introduction to Computing. (updated first edition, revised and enlarged ed.). Dubuque, Iowa: Kendall/Hunt Publishing Company, pp: 2-5.
15. Larkin and Chabay, 1992. Computer Assisted Learning Program guidelines, Washington, DC: V.H Winston and Sons, pp: 12.

16. Steven and Collins, 1977. Artificial learning a practical guide, Washington, DC: V.H Winston and Sons, pp: 1-5.
17. Collins and Steven, 1991. Natural Learning System, Washington, DC: V.H Winston and Sons, pp: 9-12.
18. Wolf and Mc Donald, 1984. Research Program Tutoring system, V.H Winston and Sons, pp: 3-7.
19. Mayer, 1988. Computer aided learning schemes, Washington, DC: V.H Winston and Sons, pp: 8.
20. Olson, Sheppard and Soloway, 1987. The Computer teaches, California Press: V.H Winston and Sons, pp: 8-12.
21. Anderson and Corbett, 1993. Artificial learning systems, using Robots, V.H Winston and Sons, pp: 12.
22. Levitt, 2000. Combinational Digital Systems, New York Times Press, NY, pp: 2-6.
23. Douglas, T. and Soft Tech, 2001. Artificial learning Technologies, New York Times Press, NY, pp: 8-16.
24. Harlan Mills and Nikklaus, 1970. Parallel Structured Programming Concepts, New York Times Press, NY, pp: 2-6.
25. Boolos and Jeffrey, 1974, 1999. Introduction to final reading, New York Times Press, NY, pp: 2-6.