ABSTRACT

The purpose of this project is to demonstrate the concept and the principles involved in the designing, developing, and implementing a new programming control structure that can serve not only as a single line structural command but also as an actual on-line development tool in the designing, coding and implementation of program application logic.

The new program control structure is designed to serve as an instrumental tool in the separation of program structural logic code from application logic code. It is through the mutual separation of these two logic flows that a full top-down structural approach can be more easily and efficiently implemented in some third generation languages lacking program structural constructs conducive to a direct application of the top-down structural approach. Because the new control structure involves a semantic generalization of today's conventional constructs, a single statement using the new control structure will implement the functions of loops, conditionals and branches all within a pre-defined top-down designed structure.

A preprocessor, hereby known as the Action Designer was designed, coded, and implemented to primarily demonstrate the simulation of a new programming language containing such new control structure and concurrently to demonstrate the direct application of the new control structure in an actual on-line
processing environment. The Action Designer was designed to present the new single-line control structural command in some pre-defined syntactical notation and to generate the appropriate program structure and all the applicable constructs to successfully execute the defined top-down structure and its associated application logic.

Therefore, this project not only introduces the concept and principles involved in designing this new programming control structure, but it also requires the design and development of the Action Designer to demonstrate this project's conceptual theories or evaluation in an actual on-line environment.
# Table of Contents

## Chapter 1  Introduction

### 1.1 Theory

### 1.2 Design Logic

### 1.3 Implementation

## Chapter 2  The New Program Control Structure

### 2.1 Program Concepts

### 2.2 Logical Design

### 2.3 Implementation

## Chapter 3  The Active Designer

### 3.1 Program Evaluation

### 3.2 Evaluation Results

## Chapter 4  The New Control Structure In An Application

### 4.1 Implementation

### 4.2 Evaluation Results

## Chapter 5  Summary And Conclusions

### 5.1 Project Results

### 5.2 Project Evaluation

## Appendix A

### A.1 Screen Charts

### A.2 Flow Charts

### A.3 Source Code

### A.4 Test Results
Chapter 1

Introduction

Under research contributions of the Communications of ACM for August 1969, a generalized control structure encompassing the functionality of the proposed control structure is introduced as a very useful and desirable feature to have in a new programming language.

More Bura and Jacopini, in 1966, proved in their basic theory of structured programming, that any structured computer program can be written using only three logical constructs, it is apparent that any single control structure semantically implementing the functions of these constructs can be used to define a "high-level" structured program with all its respective control constructs.

In development of a new programming language having the features of the proposed control structure would be very beneficial and useful to the average programmer in the designing, coding and implementation of program application techniques.

The overall scope of this project is to analyze the possibilities of developing and recommending a new approach to program development and implementation by employing a new programming language structure. The basic theory of developing a program from
raper to minor functions (top-down) using tree diagrams to
visualize functional dependencies and relationships is analyzed
for ease of development, clarity and practicality.

Currently, logic generation languages use interdispersed
conventional constructs within the program logic to define
functional dependencies and thereby create the appropriate
program structure for the executable code. Originally, before the
acceptance and the principles of structured programming were
recognized, program structures were usually very loosely
organized, confusing, and untidy.

The theory and principles of structured programming was
introduced in the late 60's. This structured theory was first
initially rejected as just another trivial academic exercise.
Most programmers complained that structured program logic lead to
rigidation and was not flexible to allow for special situations.

It is through proper introduction and training, structured
programming was finally accepted because it led to better
organized and understandable program structures.

Even though many diagramming techniques have been
advocated for application logic representation, there has been
notable little enthusiasm to develop diagramming techniques to more
effectively represent the functional relationships within the program
structure while at the same time clearly representing the
structural application logic.
Instead, most programmers choose to separately represent the
functional components of their programs on a separate chart such
as a top-down design chart or some modular chart representation form. Even though, there is nothing inherently
wrong with these diagramming techniques, they still do not
clearly represent the relationships that serve as the logical
interface between the program's structural logic and the
program's application logic. A new program control structure
concept developed in this project is instrumental in the development of a
structural concept whose application does facilitate the
development of diagramming techniques to represent the functional
and application logic flows within a program. These concepts are
discussed in Chapter 2.

Since computer are sequential machines and act upon
sequential instructions, it is understandable why sequential
diagramming techniques are employed to represent program
structure. But as mentioned before, this particular program
structure leaves much to be desired.

The greatest and most serious weakness with this
conventional program structure is that it highlights or projects
the actual sequence of computer instructions at the expense of
not emphasizing the application logic and its various functional
dependencies and relationships. An analysis of a conventional
flowchart does not readily and clearly inform the analyst as to
what are the major functions and sub functions of the application being used within the program structure. Another weakness is that an analysis of any particular process box in a conventional flowchart does not reveal all the functional dependencies associated with that process. One has to use his logic and memory to signal all the conditionals that the process is dependent on by backtracking along the program structure. This can be a very time consuming and laborious task if one is working with a very long source list or program chart. Also the conventional structure, because of its sequential nature, does not allow for easy maneuvering of the processes within the structure.

If the application processes could be developed independently of program structure and based just on their functional dependencies and relationships then this could be the ground for a very more efficient and more informative program structure than the conventional sequential flowchart.

Currently the author of this paper does not know of any computer languages that have a structural command like the one proposed in this project. That is a command to define a hierarchical program structure. Currently, the conventional sequential command is defined to the computer by the sequential order of the language instructions and the placing of the application constructs within the structure. The absence of
A structural command is the reason why the conventional sequential flowchart, despite all its weaknesses, is still widely used.

This project includes the design, development, and implementation of a structural command to be used in defining a process structure for developing, coding, and executing application logic, and at the same time providing online developments features to the application programmer.

The new control structure introduced in this project will take the designing and developing of functional logic according to tree structures, thereby making the application logic more readable and more understandable and also more manageable.

Edward Yourdon, a renowned computer expert in the field of program development, actually advocates such tree structured designing techniques in program development in his text, TECHNIQUES OF PROGRAM DEVELOPMENT. In chapter 2 of this text (p. 60), Yourdon identifies the use of tree structure diagramming techniques as development of functional logic as the relational approach. The relational approach is analogous to top-down structured programming design and implementation because its nature (by process or tree structure) occurs automatically as a result of categorization and hierarchical consideration being given to all the proper functional dependencies and relationships inherent in that particular application. Therefore, Yourdon's
relational approach to program development can be used to develop and accurately represent functional logic in tree structured format.

Yet, unfortunately, even though application programmers fully understand and appreciate the philosophy and principles of using a relational approach to program development, many of them still suffer from a vigorous and consistent use of it because of a lack of a structural command in the third generation programming languages that they use.

Due to the lack of this structural command, the programmer must implement the functional logic of a tree structure manual through the process of converting his tree structure to a conventional structured programming diagram so that he can properly code his program and insert the applicable constructs in the right sequential fashion and therefore goes through the process of intermittently undoing what he has already structurally designed and developed. Yourdon’s relational approach is still not being fully utilized and in some cases totally ignored, because of having to convert from one diagramming technique to another due to the lack of a structural command. This is the major problem that this project attempts to solve by analyzing, learning, developing and implementing an adequate structural
order to solve this problem, a single-line structural
command is introduced whereby an application programmer can
define a particular tree structure and all its applicable
constructs, thereby alleviating the application programmer from
having to restructure from one diagramming technique to another
every time he uses Bourdon’s relational approach to program
development.

A solution to this dilemma is demonstrated by simulating a
new programming language having a structural command that
facilitates the implementation of tree-structured diagrams into
the coding, negotiating, testing and executing of programs.
Developed using Bourdon’s relational approach to program
development, this simulation includes a preprocessor that
includes and evaluates a tree structure according to specifications
of a structural command.

The project includes developing the proper techniques and
methodology for constructing a sound tree structure in the development
of application logic as well as how to properly identify all the
applicable functional dependencies during the developmental
phase. Tree structures are easier to implement, because the
programmer does not have to build a particular program structure
in a design as application logic. Rather, he goes first to
specifying the functional logic according to functional
relationships and then as he does this, the proper tree
structure will evolve by definition.
During the first stage of application development, the programmer should identify and establish the hierarchical relationships that exist between the application's functions. The relational approach stresses that an accurate identification be made for each function to be developed and that appropriate relationships be developed based on some given or preconceived functional dependencies or relationships. This process leads to some structured diagrams.
Chapter 2
The New Program Control Structure

II. Theory

The program control structure as designed and developed in this project revolves around three basic theoretical concepts. This author does not claim the authentic development of the conceptual theory to be discussed. Indeed, for the most part these basic conceptual theories have been discussed and applied in various different perspectives in the formulation and advancement of programming techniques. The basic conceptual theories as applied to the development of the new program control structure are as follows:

There are three unique logic flow paths in the formulation of any application program.

a. Functional logic - The proper logical placement of all identifiable functions in respect to their relationships and functional dependencies.

b. Directional logic - The logical placement of all program control constructs within a functional tree structure.

c. Application logic - the logical placement of all the instructional commands within their proper functions.

These structures can be mathematically expressed in terms of sets and relations. In the next section, it is shown how this concept is used to develop the
synthetical process for defining the new control structure.

A general control structure can lead to programs that are at least as clear and as efficient as programs that are written with embedded program structural elements.

The Action Designer is designed to implement the new program control structure. It applies these conceptual theories in actual use as is discussed in chapter 3.

Before going into the designing details of the new program control structure, certain terminology should be covered at this time.

From structure - A top-down designed functional structure identifying the hierarchical consideration and functional dependencies prevalent in an application program.

Function block - In the Action Designer, a function block is referred to as an action block. An action block or function block contains only the application logic necessary to do that function and may also contain a functional key as a guard. The application logic within the block is only carried out if the functional key is satisfied.

Action Designer - A preprocessor specifically designed and developed for this project to demonstrate the actual use of the new program control structure developed for this project. The Action Designer simulates a new programming language having such a new program control
structure are also on-line development capabilities.

The design tactic of the new control structure uses basic set theory. In tracing the tree structure of Fig. 2A, it's apparent that each level has a finite number of nodes (x) and each node has a finite number of branches (y). Therefore, it is established through logical deduction that each level containing one or more nodes has a finite set of pairs (x, y) consisting of one pair per each node within that level. This may be written in the notation as follows:

\[ x = \{1, 2, 3, \ldots\} \]
\[ y = \{(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), \ldots\} \]
\[ (x, y) = \{(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), \ldots\} \]

---

Fig. 2A
The universal tree structure of all the levels in fig 2A is:

L1 = L2 + L4 =
(4,0), (4,2), (4,3), (4,0), (5,0), (5,2), (5,0), (5,0), (5,0), (5,0), (2,0), (2,0),
(2,0).

In order to develop a syntactical process to define a universal set of levels, the symbol "+" is used to identify the break between levels. Therefore, the universal set can be stated as:
L1 = L2 + L4 =
(4,0), (4,2), (4,3), (4,0), (5,0), (5,2), (5,0), (5,0), (5,0), (5,0), (2,0), (2,0),
(2,0).

Since the first element of each set pair within each level is unique, then it follows that the universal set can be stated for syntactical purposes as:
L1 = L2 + L4 = (4,0), (4,2), (4,3), (5,0), (5,2), (5,0), (5,0), (5,0), (2,0), (2,0).

Note: the preprocessor is designed to recognize all ending strings of zeros within each level, the syntactical formula to define a tree becomes even shorter as:
L1 = (4,0), (4,2), (4,3), (5,0), (2,0).

Note: the preprocessor is intended solely for demonstrational purposes. It is presently designed to just recognize single digit parameters. Therefore, the syntactical definition of the universal tree structure did not require use of the parenthesis as is that the use structural command to define the tree can be omitted.

L1 = (4,0), (4,2), (4,3), (5,0), (2,0).
Implementation of the new structural command developed in the preceding section is dependent on the development of a preprocessor to do all the syntactical validation in addition to a special notation. The implementation required further expansion of the syntactical notation. In order to convey to the preprocessor certain nodal properties, the structural command is extended even further by adding the symbol "\( \rangle \)" to the end. If node 4 is determined to be a terminating node then this notation is applied at the end of the structural command as:

\[
X = 420 \langle \rangle, \quad 2004
\]

A terminating node is hereby defined to be a node whose successful execution leads to a normal termination of the process. This property can be arbitrarily set by the application programmer and be conveyed to the preprocessor through the use of the structural command as shown above.

Another nodal property which the preprocessor is designed to accept is that of a loop node. A loop node is defined to be a node whose successful execution is followed by a branch back to the immediate owner. The intent of the loop node is to give any expressions developed on the preprocessor limited looping capabilities for purposes of demonstration. If for example, node 5 is determined to be a loop node, then the symbol "\( \rangle \)" is
attached to the end of the structural command after the terminating code "="

For: 14420315424704 06

Three more symbols "!" are attached to the end of the structural command to allow for future expansion in order to transmit certain nodal properties to the preprocessor as the situation may warrant. The final syntactical form for the structural command is: the tree structure of fig.2A with the nodal properties and material properties.

For: 14420315424704 0633
Chapter 3

The Action Designer

1. Program Concepts

The Action Designer processes the new structural command according to the syntactical notation developed in the preceding chapter. The Action Designer is designed to process action COBOL files; action COBOL files are designated by the file extension .act. An action COBOL file is much like a regular COBOL file except in place of a procedure division, it instead has a display division and an action division. All the other items in an action COBOL file are coded just like in a regular COBOL file (.cob).

The primary reason the action COBOL file does not have a procedure division is because in an action COBOL file the application programmer is relieved from the task of having to write a program structure as he codes his application logic. And this is due to the fact that the Action Designer has a structural command that provides the application programmer with the ability to specify his program structure separately from his application code. Indeed, a structural command is a very powerful feature because it actually provides the programmer with the capability to develop his structural logic in a concise, logical and controlled environment, rather than that provided by
The procedure division. In a procedure division, an application programmer has the responsibility of constantly having to stay aware of the structural logic at the same time he is trying to develop the application code. Again, the structural command provides the application programmer with the capability to state the program structure and all its key structural constructs in one single unique command.

In the development of an action COBOL file, a programmer has to choose to design and use a top-down approach in developing his application. The basic reason for this restriction is due to the fact that the structural command can exercise its features as shown in chapter 2 from the specifics of a tree structure. This restriction forces the application programmer to identify, as accurately as possible, not only all the individual components of his application but also all the subroutines, subroutines and dependencies that may prevail. It is very much a fact that an application programmer can construct a single functional tree structure. Once the functional tree structure is constructed, then the programmer proceeds to decompose any applicable structural constructs out of the functional tree structure. Again, all this is done using the unique single line structural command:

```
  (single line structural command)
```
The logical design of the Action Designer can be divided into two main programs or parts. The first main program is called Act 1 and the second main program is called Act 2.

The first main program is basically responsible for all of the action rules and the on-line system. The logical design of this rule interesting in that it introduces a rather unique concept in a software development system.

The concept revolves around what this author calls a sourcebase. The sourcebase is just basically a file holding source code. The source code is stored in modular form with each source code being assigned a unique node identification number. This allows for a novel arrangement of source code within a file. Anyone who used in conjunction with the single-line structural command in an on-line environment, the application programmer can insert and very quickly reconfigure his application into whatever the structure he deems appropriate for his application. Since the sourcebase file stores the source code and not the structural elements, the programmer can execute many different configurations against the sourcebase.

Therefore, Act 2 not only introduces the sourcebase concept but also introduces an on-line environment in which to demonstrate the use of the single-line command in conjunction with the sourcebase concept.
The second main program in the Action Designer is called Act 2. This program is responsible for doing the actual compilation of the action COBOL file into a regular COBOL file. Act 2 builds a logical status word for each node in the tree structure according to the information supplied in the single-line command. This logical status word is basically a field containing the proper transfer flow that is to be taken by the generated COBOL program every time that a node gets called for execution within the generated program. Therefore, Act 2 builds a concise table containing a logical status word for each node in the single-line command. Act 2 in turn writes this table of logical status words into the working-storage section of the generated program. When the generated program is executed, the first thing it does is to start execution of the nodes with the first logical status word at the start of the table. Each succeeding logical status word calls another logical status word until the whole generated program terminates execution.

In the Action Designer, Act 1 and Act 2 are chained programs. Act 1 executes first because it sets up the menu screen through which all input is originated. Act 1 then chains to Act 2 so that Act 2 can do all the necessary file processing and generate the appropriate COBOL program. If Act 2 successfully generates a COBOL program, it terminates with "ALL FILES HAVE BEEN SUCCESSFULLY PROCESSED ".

22
The implementation of the Action Designer requires the use of various different file extensions. The file extensions involved in the implementation of the Action Designer are as follows:

**Example (using fictitious filename)**

**flowroll.CDP** - a line sequential file that contains the action CDP2 program. This is the program with the structural and input divisions instead of the procedure division.

**flowroll.DOB** - a line sequential file that contains the generated DOB2 program produced by the Action Designer.

**flowroll.SBA** - an index sequential file containing the sourcebase discussed in the preceding section. This sourcebase contains the action blocks or nodes that may be potentially used in a given configuration depending on the programmer's specified configuration. The SBA file may be created and deleted on-line. The programmer uses the SBA file to actually code various functions of the application on-line. While on-line the programmer may incorporate his action blocks into whatever tree configuration he desires to test.

**Conclusion** - This line sequential file contains the configuration that may support a particular application. For instance, if a programmer had another configuration he could use against the sourcebase he would probably identify it as flowroll.SBA.
This file is a system file generated by the MS-DOS operating system in response to the creation of the index sequential file M4.

Outline: 1) This file basically holds a copy of the input data that was passed to the Act 2 program within the Action Designer.

All of the above file types are used in the processing of applications using the Action Designer. The Action Designer has a well-developed screen system whereby the programmer can easily provide all the necessary file identifications and other parameters needed in the Action Designer in the processing of all applications.

The screen system starts with the welcoming screen:
The Introduction screen lets the user be aware that in the
private mode he can build his own structural and action
sequences through the use of the sourcebase. Press enter key and
the file parameter screen appears:

FILE PARAMETERS

1. ACT
2. CAR
3. LES

OPERATION MODE
1 BATCH 2 ON-LINE
In the file parameter screen, the programmer enters the filename of the action COBOL file he developed and the filename of the OTF file to be generated by the Action Designer. The programmer also enters the filename of the list file (.lis) that is to contain a list of the action COBOL program.

After selecting the file specification on the file parameter screen, the programmer then proceeds to select the appropriate operational mode he desires. Under the batch mode, the structural and action divisions must already be included in the action COBOL file.

If the programmer selects the batch mode option, then the processing screen appears as shown below. While the processing screen is on display, the Action Designer is busy converting the action COBOL file into a generated COBOL program. After successful conversion, the message "All Files Successfully Processed" will appear on the processing screen.
If the programmer had instead selected the on-line mode then the on-line menu screen would appear:

```
ON-LINE MENU

F1: CREATE SBA  F2: CREATE ACT BLK
F3: UPDATE ACT BLK  F4: DELETE ACT BLK
F5: CREATE .CON  F6: SPECIFY .CON
F7: PROCESS SBA  F8: QUIT
```

The first thing the programmer does on the on-line menu is to specify his sourcebase file for the proper creation and editing of action blocks. The next thing he does is to specify the configuration file so that he may create a tree configuration and store that configuration as a structural command.

After he has created his sourcebase and configuration, then he can start creating action blocks by pressing the function key **F7**. This will cause the action block creation screen to appear as shown on the top of the next page.
After creating his action blocks, the programmer can always go back and reexamine or update any code in his action block through the use of the update screen. By pressing F3, the programmer causes the update screen to appear.
After the programmer has specified his sourcebase, configuration, and his action blocks, then he is ready to process the current sourcebase against the current configuration. To process the sourcebase the programmer presses F7. The message "Processing The Sourcebase" will appear and then the processing screen will appear displaying the message "Processing All Files 30% completed." After the successful generation of the appropriate COBOL program, the operating system prompt (A>) will appear indicating the termination of the Action Designer. Now, the programmer is ready to compile his generated COBOL program.
Chapter 4

The New Control Structure in An Application

1. Sample Evaluation

The new control structure developed in this project was applied to an actual application that had been previously designed and coded using conventional COBOL. By applying the new control structure to an already existing and running application, the usefulness and theoretical implications of the new control structure may be better analyzed for further development.

The particular application used for evaluation in this project was derived from the textbook PROGRAMMING THE IBM MAINFRAME COMPUTER COBOL. The purpose of the application is to enable it to read a payroll file, calculate wages and print a simple report. Since an action COBOL program and a regular COBOL program are the same except that the action COBOL program has no structure decision, not all the divisions needed analysis. Therefore, the six divisions evaluated in this report are the structural and action divisions of the action COBOL versus the structural division of conventional COBOL.

The first step in this sample evaluation is to decipher or to identify all the functional components and dependencies in the existing application program.
From this process of functional identification, a functional tree structure was constructed as shown below.

```
PRINT PAYROLL

GET NET PAY
ASSIGN INITIAL VALUES
PRINT PAYROLL
PRINT TOTALS
PRINT PAYROLL
PRINT DETAIL
```

As can be seen from the functional tree structure above, there is no problem in developing a structural command for such a simple structure. To develop the appropriate structural command the programmer must start with level one. Since level 1 has a root of one node and four branches are emitting from that node the structural command takes the form:

```
STD LEVEL 1 - 5 in total of 4 nodes the command takes the form: PRINT NET PAY
```

But, since the first 2 nodes of level 2 have zero branches emitting, the command takes the form:

```
STD LEVEL 2 - 2 in total of 2 nodes the command takes the form: PRINT INITIAL VALUES
```
The top-level node of level 2 has 4 emitting branches.

Therefore, the command then takes the form:

```
LA 44002
```

Since the preprocessor automatically assumes zero emitting branches for ending nodes, there is no need to specify the fourth node at level 2.

By the third level, since all the nodes have zero emitting branches, all that is needed is just the total number of nodes at this level. Therefore, the command takes the form:

```
1EA 30002
```

To finish the structural command, the programmer needs to identify the appropriate structural constructs that cause the non-occurrence in the functional logic path. Since in our case it is evaluation, we plan to terminate the program after the final node of level 2. We need to identify node 9 as a terminating node. Therefore, the command takes the form:

```
1EA 90002
```

Since we plan to process a payroll record until no more input, we need to have node 5 a looping node in our structural command. Therefore, the command takes the final form:

```
1EA 50002 90002
```

Now, that we have formed the single-line structural command for the loop application, we can load the Action Designer into the computer and switch to the on-line mode. Once on the on-line
and we can specify a sourcebase in which to load our application code and we can also specify a configuration file in which to contain our single-line structural command. In this sample evaluation, we have specified PAYROLL.SBA and PAYROLL.COM respectively for the above files.

2. Evaluation Results

After loading our sample application into the sourcebase and the single-line structural command, we processed the sourcebase against the configuration file and we obtained an action COBOL generated program and a complete action COBOL file.

A review of the of the conventional COBOL program versus the same program written in action COBOL reveals that the action COBOL program is a lot easier to read and to identify the applicable structural constructs in effect. The reader can follow the results by referring to the appendix under test results. The action COBOL program provides another unique advantage in that all the application code is in line unlike the conventional COBOL program with embedded perform untils that go in and out of line code. Still another advantage when in the action COBOL program is that application code is in sequential order whereas in the conventional COBOL program the sequential order is not at all observed since it is practically impossible, given that the "perform until" generates out of line code.
Chapter 5

As originally stated, this project's purpose is to introduce and demonstrate the concept and main principles involved in the designing, developing, and implementing a new programming control structure that can serve not only as a convenient structural command but also as an actual on-line development tool in the designing, coding and implementation of new application logic.

The designing, development and implementation of the Action Language in this project has proven to be very innovative, enlightening, informative and applicable in demonstrating and applying the concepts involved in the designing of a new program control structure. Again, this author does not claim the basic development of these concepts but rather seeks to demonstrate the new concepts, worthiness and applicativeness of these concepts in developing new ideas and approaches in the development of new programming languages. The author believes that the new programming control structure as developed in this project demonstrates the usefulness of incorporating a similar single-line structural command in the development of future compilers.

Consequently, the results of this project as demonstrated and reported in the sample evaluation provides a successful and
for each weapon in projecting the usefulness of the single-
lane structural command in developing better top-down design and
more readable line code than what is now attainable in some of
modern conventional programming compilers.

3.2 Project Evaluation

The overall project evaluation is based on the fivefold
purpose of the action Designer.

1. To demonstrate and apply the concepts required in the
development of a new program control structure.

2. To demonstrate an actual simulation of a new
programming language having such a new control
structure.

3. To demonstrate that programs using such new control
structure are at least as or more readable than
programs lacking the single-line structural command.

4. To demonstrate how the new control structure can be
implemented to create an on-line environment for
facilitating the development and coding of
applications.

5. To demonstrate the ease and flexibility of using a
single-line command in developing applications through
the use of the top-down design approach.

It is the author's sincere conviction and defense that
the action Designer as designed and developed in this project has
satisfactorily demonstrated the aforementioned five-fold purpose
of the Action Designer.

Therefore, an objective evaluation of this project's
overall objectives and results has revealed that this project's
objectives have been met and satisfactorily demonstrated by the
project's results.

5.4 Recommendations

In the December 10, 1984, issue of Computerworld, it was
reported that the independent program design development software
world would grow from $215 million in 1985 to $1,949 million in
1990. It is this growth in program design software that is
currently and constantly introducing new perspectives and
approaches to program development. The basic fundamental building
blocks of program development will never be obliterated by these
new perspectives or concepts. Instead, new approaches intend to
build up and apply these fundamental principles with greater ease
and speed. The introduction of the single-line structural
command developed in this project is a classic example of an
improvement to current existing fundamental theory.

Therefore, any further development of the Action Designer
with the single-line structural command would be fully dependent
on a thorough study and appreciation of the fundamental principles
developed by programming theorists like Dijkstra, Wurden, Bohm,
Codd, and others.
Nevertheless, this author visualizes many areas where the subject of the single command could be appropriately applied. The next developer in this project could be used to develop graphical software to define the single-line structural command. Thereby the programmer could graphically develop his program structure and its applicable structural constructs without ever entering a single line of code. Another further recommendation is the development of better tree structure diagramming techniques that can create more application code flowcharts themselves instead of needing to write separate flowcharts for the application code.

Therefore, in this project conclusion, the author wishes to acknowledge the usefulness, worthiness and potential growth of the single line command as developed in this project. In closing, this author also desires to acknowledge the usefulness and potential of all the academical training and assistance that has been rendered by the faculty and staff in his pursuit for better learning.
Bibliography


