# Using Decision Measures to Verify Consistency Throughout the Software Life Cycle

by

Dean W. Craig

B.S., University of Illinois, 1980

---

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Approved by:

Major Professor

CONTENTS

LIST OF FIGURES

CHAPTER 1

# Introduction

Complexity measures such as McCabe's Cyclomatic number and Halstead's Software Science measures are usually calculated at the completion of the coding phase of the software life cycle. These measures need the code to be completed before calculating the measure. These complexity measures therefore give little indication if the system is on schedule or if the coded system is consistent with the requirements and design of the system.

A complexity measure that can be used at the completion of each phase could give an indication whether the system is progressing normally. The result of the complexity measure that is made at the end of each phase could be used as input for a effort estimation tool.

The use of the complexity measure at the end of both the requirements and design phases will also give an indication of whether the design of the system is consistent with the requirements of the system. An

example would be if the relationship between the requirement phase complexity measure and the design phase complexity measure is known, and if the relationship is not maintained, then this changing relationship might indicate the system design is not consistent with the requirements.

Complexity measures that can be used early in the software life cycle can be divided into two areas, those that can be applied at the end of the requirements phase and those that can be applied at the end of the design phase. The measures that are calculated at the completion of the requirements phase include Function Points [1], BANG [2], and Tsai's data structure complexity measure based on graph theory [3]. The measures that are calculated at the completion of the design phase include Design Weight [2] and data flow based measures [4]. These groups of measures however do not measure the same characteristic of the system at the end of the requirements and as at the end of the design phase. It would be difficult to relate the measures of the requirements phase to those of the design phase to tell if the requirements and design of the system were consistent. In addition, though each of these measures

would give an indication if the system is progressing normally, a different estimation model would probably be needed with each measure.

A measure that can be made against the same characteristic of the system throughout the software life cycle is therefore desirable so that an indication if the system is progressing normally could be given. In addition, such a measure is desirable so that the consistency of the system between phases of the software life cycle can be checked.

The portion of the system that is complete at the end of the requirements phase and is a part of the design is the data structure of the system. A complexity measure that is based on the data structure of a system therefore could be made at both the end of the requirement phase and at the end of the design phase. One such complexity measure is DeMarco's Design Weight. DeMarco, however, does not use the Design Weight to measure the complexity at the end of the requirements phase.

Extending DeMarco's Design Weight to measure the complexity of the data structure at the end of the

requirements phase along with the use of McCabe's Cyclomatic Number will provide a clear and consistent view of the system.

This study will calculate several complexity measures for projects taken from CMPSC 541 (SPRING 87). The measures will be calculated on documents from the end of the requirements phase, the design phase and the coding phase for each of the projects. The requirement specifications were written using the ERA model. The design specifications were written using a hierarchical model. The projects were coded using the C language.

CHAPTER 2

## Literature Search

Papers on software complexity measures can be divided into three categories: requirement phase measures, design phase measures and coding phase measures.

The first category of measures is those that can be calculated at the completion of the requirements phase of the life cycle. This set of measures include BANG, function points and Tsai's data structure measure.

The first requirement phase measure is called BANG [1]. The BANG measure is a measure of the functions that are to be performed by the software system. The system BANG is the summation of the BANG for each of the functions in the system.

The BANG of a function is calculated using the formula

BANG = ( I + O ) * FW

where I is the number of input items to the function, O

is the number of output items from the function and FW is the function weight.

DeMarco provides a table that gives a function weight for several different classes of functions. He indicates that the function weights may be changed by an organization to better suit the organization's needs. In addition, the table provided is not a complete list of classes of function, so additional classes may need to be developed.

Two algorithms are required for calculating BANG. The first algorithm is used for a function-strong system, while the second algorithm is for data-strong systems. DeMarco indicates that the BANG calculated for a function-strong system and for a data-strong system are not comparable.

The BANG measure is used as a prediction of the amount of effort required to develop the system. The BANG is also used as an input to a cost estimation model.

A second requirements phase measure is called function points [2]. The function point measure is another measure of the functions to be performed by the system.

The unadjusted function point of a system is a weighted sum of the number of inputs, the number of outputs, the number of data files, the number of inquiries, and the number of interfaces. The adjusted function point of a system is the unadjusted function point times a complexity factor. The determination of the complexity factor is subjective with the higher the factor, the greater the complexity.

Studies have shown that as a prediction of effort that the function point metric is good for information processing systems [5]. The function point measure has not been as good for real time or embedded systems. An extension of function points called feature points [5] was developed to be used with real time and embedded systems.

A third requirements phase measure is Tsai's data structure complexity measure [3]. The calculation of this data structure complexity measure is based on graph theory. The data structure to be measured is taken from the data dictionary that is present at the end of the requirement phase.

Tsai [3] provides reasons for selecting the data structure of the system for calculating a complexity measure prior to describing the method for calculating the measure [3].

The first reason for selecting a data structure measure is that most of the current software methodologies require that a data dictionary defining the data for the system be completed at the end of the requirements phase.

The second reason for selecting a data structure measure is that the authors believe that the complexity of the program is a direct result of the complexity of the data structure. That is, the more complex the data structure the more complex the program will be.

The data structure complexity measure is given in terms of a polynomial of one variable. The polynomial is calculated from a graph that is constructed from the structure of the data.

There is one node in the graph for each data structure that is defined in the data dictionary. A node is included in the graph for each of the atomic data

structure types (integer, character, etc.) that is used in the data dictionary.

The edges of the graph show the reference from one data structure to another data structure. An example would be if "A" is defined as a integer, then there would be a edge from the node labeled "A" to the node labeled integer.

The graph is then modified, eliminating nodes and edges for auxiliary definitions, and the graph is split into strongly connected components.

A single term polynomial is constructed for each strongly connected component of the graph. The coefficient of the term is the number of loops in the strongly connected component. The multipler of the term is the number of nodes and edges contained in the strongly connected component. The single term polynomial is then used to calculate the complexity of each node in the graph. Every node in a strongly connected component of the graph will have the same complexity.

The overall data structure complexity is calculated from the addition of each of the complexities calculated for the nodes. The final data structure complexity is still in the form of a polynomial of one variable.

The second category of measures is those that can be calculated at the completion of the design phase of the life cycle. This set of measures include Henry and Kafura's information flow complexity measure and DeMarco's design weight.

The first design phase measure is Henry and Kafura's information flow of the system [4]. The flow of information is studied at the boundaries of the modules that make up the system.

Henry and Kafura define two terms that are necessary to calculate the complexity measures. The terms are fan-in and fan-out. Fan-in is defined as the number of data items entering the module plus the number of data structures that the module will retrieve data from. Fan-out is defined as the number of data items leaving a module plus the number of data structures that are updated by the module.

The information flow complexity measure is calculated using the formula

COMPLEXITY = LENGTH * (FAN-IN * FAN-OUT) ** 2

Henry and Kafura use a length measure of lines of code, but indicate that a alternate length measure can be used. Alternate length measures being the length measure of software science or the cyclomatic number.

A second design phase measure is DeMarco's design weight [2]. The design weight of a system is a prediction of the number of decisions that are to be made by the system. Thus, the design weight is a prediction of McCabe's cyclomatic number.

The design weight of the system is the summation of the design weight of each of the modules of the system. The design weight of the module is calculated by using the data dictionary definitions for each of the data items that are entering a module. The data items entering a module may be either data that is passed to the module from a calling module or data that is returned from a called module.

DeMarco also uses the design weight in a design compliance measure. Since the design weight is a prediction of the number of decisions, it is compared to the actual number of decisions to give a design compliance measure.

The third category of measures is those that can be calculated at the completion of the coding phase of the life cycle. This set of measures include software science [6], cyclomatic number [7] and Ramamurthy and Melton's measure based on the combination of software science and the cyclomatic number [8].

The first coding phase measure is the software science measures [6]. Software Science, which was first developed by Halstead, contains measures for program length, the volume of a program, the program level, and an effort measure. The program level measure is an indication of the complexity of the system.

Each of the software science measures is calculated from the number of operators, the number of unique operators, the number of operands and the number of unique operands in the system.

The second coding phase measure is McCabe's cyclomatic number [7]. The cyclomatic number is based on graph theory. A graph is constructed to show the control flow of the program. The cyclomatic number is calculated from the number of nodes and edges of the control flow graph. The result is that the cyclomatic number is one plus the number of decisions that are made in the program.

A third coding phase measure is Ramamurthy and Melton's combination of the software science measures and the cyclomatic number [8]. Ramamurthy and Melton point out that the software science measures are better for some programs, while the cyclomatic number is better for other programs. Their combination of the software science measure and the cyclomatic number is to provide a measure that is beneficial for both types of programs indicated above.

The combination of the software science measures and the cyclomatic number is done when counting the number of operators and operands. A weight is assigned based on the use of the operator or operand in the control flow of the program.

The application of the combined measure, the software science measures and the cyclomatic number was done and is presented to show that the combination measure resolves the conflicts between the software science measures and the cyclomatic number.

CHAPTER 3

## Applying the Measures

The measures, Design Weight, BANG and Cyclomatic Number, were selected to be applied to the software projects. In addition, an extension to the Design Weight measure was made so that the design weight measure could be applied at the completion of the requirements phase. The final measure applied was the number of lines of code. Each of the measures that were applied to the projects are described in greater detail in this chapter.

Design Weight, Cyclomatic Number and the Design Weight Extension were selected since each of these measures provides a count of the number of decisions to be made by the software system at the conclusion of each of the phases of software development. The BANG and lines of code measures were selected to provide a comparison with the other measures.

### Design Weight

DeMarco introduces a measure for the complexity of a

system based on the data structure at the end of the design phase called Design Weight. The Design Weight is a prediction of the number of decisions that will be made by the system.

The Design Weight of a module is calculated from the data that is arriving at the boundary of the module. This data may be data from an external source, data passed to the module from a calling module or data returned from a called module.

The Design Weight of a module is calculated as follows:

1. The initial Design Weight of the module is set to zero.

2. Each data item that is arriving at the boundary of the module must be fully defined in a data dictionary.

3. The Design Weight of the module is incremented by one for each occurrence of an iteration, selection or option in the data dictionary descriptions of the data arriving at the module boundary.

The Design Weight of the system is the summation of the Design Weight of each of the modules of the system.

An extension to the Design Weight measure will be made so that the measure may be calculated at the completion of the requirements phase. The Design Weight will be calculated for each activity in the data flow diagram. The input data flows to each activity will be used to calculate the Design Weight. The same rules that were used to calculate the Design Weight after the design phase will be used for the calculation after the requirements phase.

The systems design weight for the requirements will be calculated from the design weight of the activities in the data flow diagram that do not have any subactivities.

The calculation of the design weight at the requirements phase is shown using the ERA specification and data dictionary items that are in Figure 3-1. The data dictionary entries for those data items that are arriving at the boundary of the module are also shown. Following the rules for calculating design weight given above, the design weight of the module would be 7. The data item

"selection" will add 4 to the design weight. The iteration of this item would add 1 while the selection will add 3. The data item "trec" will add 3 to the design weight. The iteration of "preds" will add one while the selection in "v_status" will add 2, giving a Design Weight of 7.

```
ERA SPECIFICATION

PROCEDURE: Add_Edit

     Parameter In: selection

     External In: id

     External Out: id_prompt

     Calls : find_task

          passes down: id, fflag

          returned: fflag, recnum, trec

DATA DICTIONARY

selection = [A|E|D|Q]
fflag = integer
recnum = integer
trec = record
record = name_of_task + v-id + [preds] + v_time
         + v_manpower + task_description + v_status
         + v_percent + e_time + e_manpower
preds = v_id
v_id = string of char
v_status = N|C|P
v_time = integer
v_manpower = integer
task_description = string of char
v_percent = real (0-100)
e_time = integer
e_manpower = integer
```

Figure 3-1

McCabe's Cyclomatic Number

McCabe's cyclomatic number is the number of decisions to be made in a module plus one. Since the design weight is

a prediction of the number of decisions, it would be expected that the design weight of a module is one less than the cyclomatic number of the module. The Cyclomatic Number is calculated from the source code of the module.

The occurrence of an if-then-else statement, do-while statement, while statement or a for statement in the code will increment the Cyclomatic Number of the module by one. An occurrence of a case statement in the code will increment the Cyclomatic Number of the module by the number of selections in the case statement.

DeMarco's BANG

DeMarco uses a measure at the end of the requirements phase called BANG. The BANG of an activity is a measure of the functionality of the activity. The BANG is represented as a weighted number based on the count of data items associated with the activity and the function being performed by the activity. Bang is calculated only for those activities of the data flow diagram which do not have any subactivities.

The BANG is calculated for an activity by counting the number of data items to and from the activity. The type of function that the activity is performing is determined and a weight is associated with the type of function. DeMarco provides a list of types of functions and their associated weights. The activity's BANG is then calculated by using the following formula:

BANG = NDI * log2(NDI) * AW

where NDI is the Number of Data Items of the activity and AW is the Activity Weight.

The BANG for the system is the summation of the BANG for each of the individual activities.

The final measure that will be calculated for the projects is the number of lines of code. The following rules were used to count lines of code:

- One statement per line
- Lines containing only comments will not be counted.

- Data Declarations will be counted.

CHAPTER 4

# OBSERVATIONS

The results of applying the BANG and the extended Design Weight to the requirements phase are given in Figure 4-2. The results of applying the Design Weight, Cyclomatic Number, and lines of code measures to the design phase and coding phase are given in Figure 4-3.

Since the extended Design Weight, Design Weight and Cyclomatic Number measures are providing the same quantity at the end of the phases of the software development life cycle, the results of applying the measures were reviewed looking for relationships between the measures. In addition, the results of using the extended Design Weight measure were reviewed for relationships between extended design weight calculated for an activity and its subactivities.

| ACTIVITY | EXTENDED DESIGN WEIGHT | | BANG |
|---|---|---|---|
| | INITIAL | CORRECTED | |
| 1 | 0 | 5 | 2.4 |
| 2 | 0 | 0 | 0 |
| 3 | 6 | 35 | |
| 3.1 | 0 | 0 | 0 |
| 3.2 | 6 | 35 | |
| 3.2.1 | 2 | 4 | 1.7 |
| 3.2.2 | 4 | 4 | 1.7 |
| 3.2.3 | 4 | 4 | 1.8 |
| 3.2.4 | 3 | 27 | 1.9 |
| 3.2.5 | 3 | 3 | 1.0 |
| 3.2.6 | 7 | 7 | |
| 3.2.6.1 | 3 | 3 | 2.9 |
| 3.2.6.2 | 4 | 4 | 1.9 |
| 3.2.6.3 | 0 | 0 | 1.9 |
| 3.2.6.4 | 5 | 5 | 1.9 |
| 3.2.6.5 | 6 | 6 | 3.2 |
| 3.3 | 0 | 4 | |
| 3.3.1 | 0 | 0 | 0.7 |
| 3.3.2 | 4 | 4 | 1.7 |
| 3.3.3 | 0 | 0 | 1.0 |
| 3.3.4 | 0 | 4 | 4.8 |
| 4 | 0 | 0 | 1.0 |
| 5 | 0 | 9 | |
| 5.1 | 0 | 5 | 1.0 |
| 5.2 | 4 | 4 | 5.8 |
| 5.3 | 4 | 4 | |
| 5.3.1 | 4 | 4 | 1.7 |
| 5.3.2 | 4 | 4 | 1.7 |
| 5.3.3 | 4 | 4 | 1.7 |
| 5.3.4 | 0 | 0 | 2.4 |
| 5.4 | 4 | 4 | |
| 5.4.1 | 4 | 4 | 1.5 |
| 5.4.2 | 5 | 2 | 0 |
| 5.4.3 | 5 | 2 | 0 |
| 5.5 | 0 | 0 | 1.0 |
| 5.6 | 2 | 2 | 2.0 |
| 5.7 | 2 | 2 | 2.0 |

Figure 4-2

| MODULE | DESIGN WEIGHT INITIAL | CORRECTED | CYCLOMATIC NUMBER | LOC |
|---|---|---|---|---|
| 1 | 0 | 5 | 6 | 99 |
| 2 | 2 | 4 | 4 | 43 |
| 3 | 5 | 10 | 18 | 190 |
| 4 | 4 | 4 | 6 | 87 |
| 5 | 6 | 30 | 93 | 615 |
| 6 | 3 | 3 | 3 | 40 |
| 7 | 4 | 4 | 6 | 50 |
| 8 | 5 | 5 | 11 | 90 |
| 9 | 6 | 6 | 17 | 112 |
| 10 | 0 | 0 | 9 | 54 |
| 11 | 2 | 5 | 9 | 81 |
| 12 | 4 | 4 | 22 | 163 |
| 13 | 0 | 0 | 1 | 61 |
| 14 | 0 | 0 | 0 | 42 |
| 15 | 4 | 4 | 4 | 64 |
| 16 | 4 | 4 | 2 | 36 |
| 17 | 4 | 4 | 2 | 42 |
| 18 | 2 | 2 | 7 | 100 |
| 19 | 2 | 2 | 6 | 85 |
| 20 | 0 | 0 | 0 | 31 |
| 21 | 4 | 4 | 17 | 173 |
| 22 | 2 | 2 | 5 | 68 |
| 23 | 2 | 2 | 7 | 81 |
| 24 | 3 | 3 | 10 | 80 |
| 25 | 3 | 3 | 4 | 42 |
| 26 | 4 | 4 | 7 | 52 |
| 27 | 3 | 3 | 0 | 15 |
| 28 | 0 | 0 | 1 | 49 |
| 29 | 0 | 0 | 0 | 16 |

**Figure 4-3**

It was expected that the design weight of the module would be one less than the cyclomatic number of the module. It can be seen in Figure 4-3 that this was not true. There were three major reasons for the differences between the design weight of a module and its cyclomatic

number.

The first reason for the difference between the design weight of the module and its cyclomatic was due to errors in the data dictionary. The most common error was data was not defined to a detailed enough level. An example is the user of the system is presented with a menu to make a selection, with the user response to be stored in a data item called selection. The correct responses for selection are an "A", an "E", an "D", or a "Q", however the data dictionary defines selection as a character. The difference in the data dictionary definition and the possible values for the data item selection causes a difference in the calculation of the design weight.

Another common error was that data items that were to function as boolean variables were defined as integers. The correct definition should have limited the possible values to 0 or 1.

The errors in the data dictionary caused the design weight of the module to be lower than what it should have been. The errors were corrected in the data dictionary based on how the data item was used in the source code.

The design weight was recalculated after correcting the data dictionary errors. This corrected design weight is shown under the column labeled "CORRECTED" in Figure 4-2 and Figure 4-3. It now can be seen from Figure 4-3 that the design weight of the module is one less than the cyclomatic number in several instances. It should be noted that the correction of the error in the data dictionary may not have been the proper correction. The data dictionary could be correct and the use of the data item in the source code could be incorrect.

A second reason for the the difference into the design weight of a module and its cyclomatic number is that the design weight does not take into account decisions that need to be made for error checking of the data items. This reason is the primary cause of the difference between the design weight and cyclomatic number for module number 5.

The first type of error check would be for boundary conditions on the domain of the data items. Consider the data item selection that was described above; an error check should be in the system to check for an input value of something other than an "A", "E", "D", or "Q". This

error check will add one to the number of expected decisions of the module.

The algorithm for calculating design weight does not take into account this extra level of selection of possible values for the input.

A second type of error check that is to be performed is one that is required in the statement of work of the system. This type of error check is much harder to specify in a data dictionary definition than the boundary condition.

A third reason for the difference between the design weight of a module and the cyclomatic number of a module is for modules that format output reports. This is the cause for the difference in design weight and cyclomatic number in module numbers 12, 18, and 19.

The cause of this difference is that the design weight is calculated from the structure of the input data and not from the structure of the output data. For modules 12, 18, and 19, the input data structure was fairly simple but the output structure was complex; therefore, the

design weight did not predict the number of decisions required to format the output reports correctly.

As the data dictionary is constructed and maintained there is a need not only for checking for syntatic errors in the data dictionary but also for semantic errors. It is the semantics of the data that will lead to a calculation of design weight that will include the number of decisions to be made for error checking.

A second observation made is that two rules exist that define the relationship between the design weight of an activity and the design weights of the subactivities. The rules for the relationship of design weight between an activity and its subactivities are given in Figure 4-4.

1. The design weight of an activity A will be greater than or equal to the design weights of each of the subactivities of A.
2. The design weight of an activity A will less than or equal to the sum of the design weights of the subactivities of A.

Figure 4-4

The first rule implies that there is no new data

introduced at the subactivity level. This follows from the fact that the subactivities are created from the decomposition of the activity.

The second rule implies that all of the data items of the activity are used by the subactivities. That is, as the activity is decompose into subactivities that no data item is lost.

A third observation made is that of the relationship between the design weight of the requirements phase and the design weight of the design phase. The mapping of activities of the requirements to the modules of the design is required to understand the relationship between the design weight of the requirements phase and the design weight of the design phase.

The mapping of activities to modules is divided into three cases. The three cases are

1. One activity maps to one module.

2. Two or more activities map to one module.

3. One activity maps to two or more modules.

The rules for the three cases of mapping of an activity to a module are given in Figure 4-5.

Case 1: The design weight of the activity is equal to the design weight of the module.

Case 2: The design weight of the module is less than or equal to the sum of the design weights of the activities.

Case 3: No relationship exists between the design weight of the activities and the modules.

Figure 4-5

If a difference exists in the design weights for case 2 of the activity to module mapping relationships, then the difference is equal to the design weights of the common input items of the activities.

The first two cases are supported by the idea that no new data items are introduced or data items lost in the transition from the requirement phase to the design phase.

The third case of mapping activities to modules did not

give a relationship between the design weights. However, a mapping of one activity to two or more modules may indicate that the activity could be further decomposed.

CHAPTER 5

## Conclusions and Extensions

The measures, extended design weight, design weight, and cyclomatic number provide a set of measures that are measuring the same characteristic of the system at the end of each phase of the software lifecycle. In measuring the number of decisions to be made by the system, this set of measures allows for comparison of the system at the various points in the life cycle.

The rules for comparing the various measures at the end of the phases provides a means for checking to see if the design follows the requirements and if the code follows the design. This checking will allow for the detection of errors earlier in the life cycle.

The use of this set of measures show the importance of the data dictionary in software development. Methods to ensure the correctness of the data dictionary are required. Also methods that would allow the data dictionary to capture more of the semantic meaning of the data would be beneficial.

The application of the extended design weight must be done on a wider scope to see if the extended design weight can be used as a predictor of effort following the requirements phase.

BIBLIOGRAPHY

[1] Albrect, A. J. and Gaffney, J. E., Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation, IEEE Transaction on Software Engineering, November 1983, Volume SE-9.

[2] DeMarco, T., Controlling Software Projects: Management, Measurement and Estimation, Yourdon Press, 1982.

[3] Tsai, W. T., Lopez, M. A., Rodriguez, V. and Volovik, D., An Approach to Measuring Data Structure Complexity, Proceedings of Compsac 1986.

[4] Henry, S. and Kafura, D., Software Structure Metrics Based on Information Flow, IEEE Transactions on Software Engineering, September 1981, Volume SE-7.

[5] Jones, T. C., Optimizing Software Production and Quality, Technology Transfer Institute, 1987.

[6] Shen, V. Y., Conte, S. D., and Dunsmore, H. E., Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support, IEEE Transaction on Software Engineering, March 1983, Volume SE-9.

[7] McCabe, T. J., A Complexity Measure, IEEE Transactions on Software Engineering, December 1976, Volume SE-2.

[8] Ramamurthy, B. and Melton, A., A Synthesis of Software Science Metrics and the Cyclomatic Number, Proceedings of Compsac 1986.

# Using Decision Measures to Verify Consistency Throughout the Software Life Cycle

by

Dean W. Craig

B.S., University of Illinois, 1980

---

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

ABSTRACT

Complexity measures such as McCabe's Cyclomatic Number and Halstead's Software Science measures are calculated at the completion of the coding phase of the software lifecycle. These complexity measures therefore give little indication if the system is on schedule or if the coded system is consistent with the requirements or the design of the system.

Complexity measures that can be used at the completion of each phase could give an indication whether the system is progressing normally. The use of a complexity measure at the end of the requirement phase and design phase will also give an indication whether the design of the system is consistent with the requirements of the system.

A complexity measure which measures the same characteristic of the system would be beneficial for determining if the system is progressing normally and if the system is consistent from one phase to the next. This allows for easier comparison of the phases of the software life cycle.

A study of a set of complexity measures that measures the same characteristic of the system at the completion of the phases of the software life cycle was done. The study provided a set of relationships between the measures that were calculated at the end of each phase of the software life cycle.