# A STUDY OF SOFTWARE METRICS

BY HILDA B. KLASKY

A thesis submitted to the

Graduate School-New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Professor Manish Parashar

and approved by

_____

_____

_____

New Brunswick, New Jersey

May, 2003

# ABSTRACT OF THE THESIS

A STUDY SOFTWARE METRICS

by Hilda B. Klasky

Thesis Director: Professor Manish Parashar

Software Engineering is defined in the IEEE Standard 610.12, as “The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.” Metrics are used by the software industry to quantify the development, operation and maintenance of software. The practice of applying software metrics to a software process and to a software product is a complex task that requires study and discipline and which brings knowledge of the status of the process and / or product of software in regards to the goals to achieve.

This thesis presents a study and implementation of different software metrics. We apply these metrics to a sample project, and evaluate the results. We find that that there are specific metrics for different stages of the software development cycle. When used properly, i.e., when a company uses the best software metric during each development phase, the quality of the software will dramatically increase. Therefore, we highly recommend using software metrics during all stages of the development process.

# Acknowledgements

It has been a privilege for me to study at the Department of Electrical and Computer Engineering at Rutgers University where students and professors are always eager to learn new things and to make constant improvements. I am especially grateful to Professor Manish Parashar for his constructive criticisms, patience and valuable comments and suggestions to my work. So, I would like to thank Professor Deborah Silver, Professor Ivan Marsic and Mrs. Barbara Sirman for their observations and remarks to improve my thesis.

There have been many people that have supported my carrier during my student time. I would like to thank Professor Ravi Samtaney, Professor Michael M. Bushnell, Professor Stanley Dunn and Mrs. Barbara Klimkiewicz. I would like to thank the management support and the learning experience that I have received at work from Shawn Wang, and K.C. Lee.

I have been very lucky to have the continuing support and encouragement from my family especially from my husband Scott, my parents Gregorio and Victoria, my brothers and sisters: Aida, Gustavo, Delia, Sandra, Eva, Nora and Jaime and my Parents in Law Charles and Mary. To all them many thanks.

# Dedication

To my husband Scott with love and respect.

# Table of Contents

# List of Tables

# List of Figures

# List of Equations

# Chapter 1 Introduction

This chapter presents an overview of the thesis, it describes the problem statement and continues with the contribution of this thesis, at the end, it presents the organization of the thesis.

## 1.1 Overview

Software Engineering like all other engineering professions has metrics. Software Engineering, as defined in IEEE Standard 610.12, is: “The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software”.

Metrics are used by the software industry to quantify the development, operation and maintenance of software. They give us knowledge of the status of an attribute of the software and help us to evaluate it in an objective way. The practice of applying software metrics to a software process and to a software product is a complex task that requires study and discipline, and which brings knowledge of the status of the process and / or product of software in regards to the goals to achieve.

By evaluating an attribute of the software, we can know its status. From there, we can then identify and classify what its situation is; which helps us to find opportunities of improvements in the software. This also helps us to latter make plans for modifications that need to be implemented in the future. In addition, to save the values obtained as history for further reference.

There is a broad range of software metrics. The software metrics depend on what software’s attributes we want to quantify or qualify. Generally, software metrics are

organized into two main different classes: metrics for the software process and metrics for the software product.

Metrics for the software process are related to the effort it takes to complete the project, the resources to spend on it, and the methodology to follow. For example: the time that will take to complete it, the people needed to develop it, the overall cost in money, and the methodology to follow.

Many types of attributes of software can be measured. The metric which one applies depends on the nature of the software product. For example, for the requirements, we might want to know how many requirements a project has, its specificity (lack of ambiguity) and completeness (if it covers all the functions needed). For the software product of a program, we might want to know the number of lines of code, its complexity, the functionality it covers, the number of potential defects (‘bugs’) it will have, and the number of test cases to verify that all requirements have been implemented. We also could measure the reliability of the software once delivered.

The Software Engineering community hasn’t agreed on a set of metrics universally accepted by the field. Therefore, many people have come up with different ways to measure the different attributes of software; yet, a lot of controversy has developed based on all these metrics. The Software Engineering community has still a long way to go in order to achieve a unified set of metrics to evaluate the software process and software products.

## 1.2 Problem Statement

Despite the fact that the Software Engineering field doesn’t have a unified set of metrics that the community has agreed to use, it is advised to use them. Often during the software

process, the members of the development team do not know if what they are doing is correct and they need a guide that could help them orientate further improvement and to objectively know if the improvement is being achieved. Software metrics are tools that help to track software improvement.

Most large companies dedicated to develop software, use metrics in a consistent way. Many companies have created their own standards of software measurement; so, the way that metrics are applied usually varies form one company to another one. Nevertheless, as they are used in a consistently way through different projects, the software groups get many benefits from them.

What to measure in regards of software process or product depends on the nature of the project, but in all cases, the customer satisfaction is the goal and measures should be taken to achieve that goal not only at delivery, but through the entire development process.

## 1.3 Contribution of the Thesis

There is no unique metric which works in the entire development phases. Keeping several metrics on one system helps to have a handy solution that can be used on different aspects of the software development process. The project developed in this thesis provides a solution for this need by implementing metrics that could be applied on several aspects of the Rational Unified Process. The Rational Unified Process is a Software Engineering Process that captures many of the best practices in modern software development in a form that is suitable for a wide range of projects and organizations.

The metrics covered in this thesis are the following:

- For the Requirements workflow, the Specificity and Completeness of Requirements
- For the Design: Cyclomatic Complexity, Function Points, Information Flow and the Bang Metric.
- For the Implementation: the Estimation of Number of Defects, the Lines of Code (LOC) and the Halstead Metrics.
- For the Test and Deployment: the Number of Defects is again mentioned, a metric to estimate the Reliability of the software and the estimation of the Number of Test Cases.

We have included metrics that support the workflows which estimate effort: they are the effort according to COCOMO II, the effort according to the Statistical Model and the effort according Halstead Metrics.

## 1.4 Organization of the Thesis

This thesis is organized into five chapters. Chapter 1 is the introductory chapter where the problem statement is described. Chapter 2 presents a classification of software metrics and the related work in software industry. Chapter 3 describes the system developed in this thesis. Chapter 4 describes the implementation and evaluation of this system taking metrics of it. Finally, Chapter 5 lists a number of improvements that can be done to this work.

# Chapter 2 Related Work

This chapter presents a classification of software metrics and summarizes the work that some companies and organizations have performed in this regard.

## 2.1 Classification of Software Metrics

Software Metrics are standards to determine the size of an attribute of a software product and a way to evaluate it. They can also be applied to the software process. Several books present different classification of software metrics, most of them agree on the following [26]:

a) Software product metrics: These metrics measure the software product at any stage of its development. They are often classified according with the size, complexity, quality and data dependency.

b) Software process metrics: These metrics measure the process in regards to the time that the project will take, cost, methodology followed and how the experience of the team members can affect these values. They can be classified as empirical, statistical, theory base and composite models.

This research project focuses on presenting software metrics as they could be applied on the different phases of the Rational Unified Process of Software Development.

The Rational Unified Process is a Software Engineering Process that captures many of the best practices in modern software development in a form that is suitable for a wide range of projects and organizations. The Rational Unified Process can be applied for small development teams as well as for large software teams. More over, the Rational Unified Process is a guide for how to effectively use the Unified Modeling Language

(UML). The UML is an industry-standard language that allows software organizations to clearly communicate requirements, architectures and designs [27].

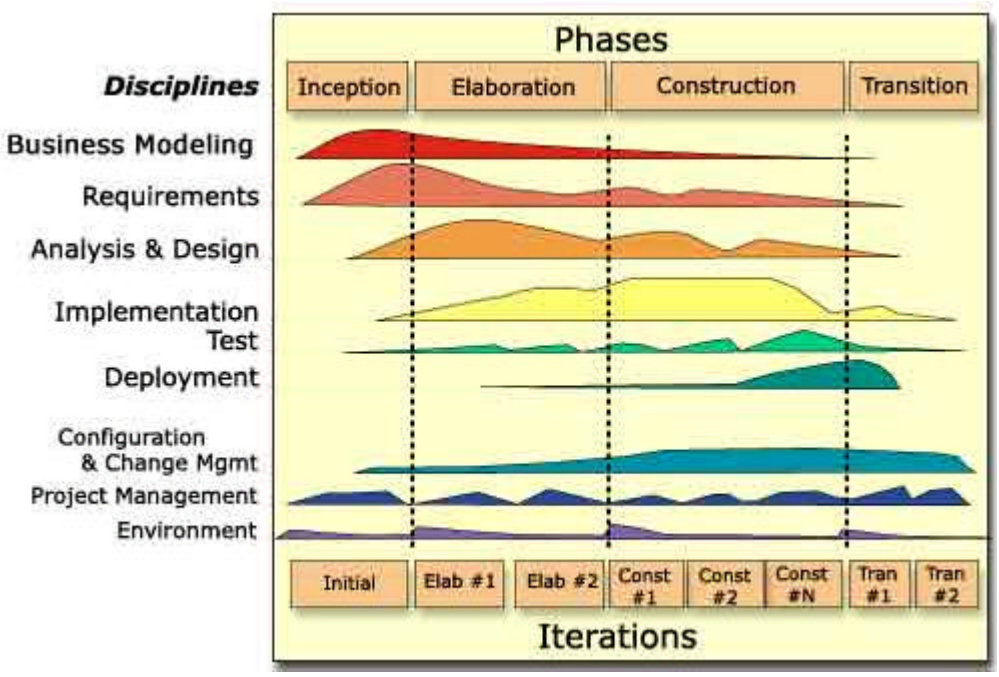


**Figure 1 Rational Unified Process**

The Rational Unified Process is shown in Figure 1. In the Rational Unified Process, we can distinguish two dimensions, one on the horizontal axis and another one on the vertical axis. The horizontal axis represents time and shows the dynamic aspect of the process expressed in terms of cycles, phases, iterations, and milestones. Those are the Inception phase, the Elaboration phase, the Construction phase and the Transition phase. Each phase ends with a well-defined major milestone. The vertical axis represents the static aspect of the process. A process describes who is doing what, how, and when. In the Rational Unified Process, the Workers are the ‘who’, the Activities are the ‘how’, the Artifacts are the ‘what’ and the workflows describe the ‘when’.

## 2.2 Examples of Software Metrics Systems

Many people have had the idea of concentrating software metrics that could be used during the software process, even though not all software metrics have been organized in the same manner. This section presents some examples of software metrics sets that present and implement different software metrics.

| | Company/Organization | Brief Comments |
|---|---|---|
| 1. | SMLab | Software Measurement Laboratory at the University of Magdeburg, Germany |
| 2. | ZD-MIS | Zuse /Drabe Measure-Information System private company. |
| 3. | Power Software | Private company. |
| 4. | Charismatek Software Metrics | Private company. |
| 5. | QSM | Quantitative Software Management is a private company. |
| 6. | CMM | Capability Maturity Model of Software of Software Engineering Institute at Carnegie Mellon University. |
| 7. | ISO 9000 | International Organization for Standardization. |
| 8. | Total Metrics | Consulting services. |
| 9. | David Consulting Group | Consulting services. |

**Table 1 Software Metrics Related Work**

### 2.2.1 Software Measurement Laboratory

There is a very extensive and comprehensive presentation of software metrics and tools at http://irb.cs.uni-magdeburg.de/sw-eng/us/index.shtml posted by The Software Measurement Laboratory (SMLab) at the University of Magdeburg, Germany. The SMLab's team Members led by Prof. Reiner R. Dumke. The SMLab’s team has done a very good job concentrating different useful community activities related to software metrics; those go from forums, conferences and workshops, to articles and applets for

metrics tools. They present a wide range of tools and topics related to software metrics and have created a large community of participants that comprises members all around the world.

### 2.2.2 ZD-MIS

ZD-MIS stands for Zuse / Drabe Measure-Information-System, http://home.t-online.de/home/horst.zuse/zdmis.html, which provides a 'comprehensive software test framework'. The project was initiated by Horst Zuse and Karin Drave. It comprises a large set of software metrics and a book with the fundamentals. This project is currently offered as a product that can be purchased.

### 2.2.3 Power Software

Power Software is a company (http://www.powersoftware.com/) that provides different tools of software metrics. The tools that Power Software provides go from counting lines of code to Cyclomatic Complexity, Halstead product metrics, and Object Oriented metrics. The company also provides tools to measure the effort and project management metrics.

### 2.2.4 Charismatek Software Metrics

Charismatek is a company (http://www.charismatek.com.au/) that provides Metrics Software Tools and consulting services, they have developed a Function Point Tool: 'WORKBENCH™' which has been receiving good ratings by a user satisfaction survey. This company also has other software programs to aid in the software management process.

### 2.2.5 QSM

Quantitative Software Management (QSM) can be found at http://www.qsm.com/. QSM also specializes on developing software metric tools for project management and they have developed SLIM-Metrics and SLIM-Data Manager software tools that graphically allow users to see resources spent and estimation of quality for the project. Both have a database system integrated to track changes and see the history of the project across the time.

### 2.2.6 Other Quality Models

There are other Quality Models provided by different organizations that give guidelines of software product/process improvement. One notable work is the Capability Maturity Model of Software (CMM) of the Software Engineering Institute (SEI) at Carnegie Mellon University) (www.sei.cmu.edu). The CMM suggests a software company evolution improvement that goes from an Initial Level of software development process, in which there is no organization; to an Optimizing Level, in which there is a continuously improving process. In the Optimizing level, the software development process and products are constantly monitored and the results are predictable.

Another important work is the International Organization for Standardization, which has a standard for quality management systems (http://www.iso.ch/iso/en/iso9000-14000/). Many companies follow does standards to achieve certifications.

Other companies that provide documents and consult services for software metrics application and improvement are: Total Metrics provides consultancy services to improve software metrics practices (http://www.totalmetrics.com/) and the David Consulting

Group (http://www.davidconsultinggroup.com/) which also has a large set of articles written and a vast experience in software metrics implementation.

## 2.3 Summary

This chapter has presented a classification of software metrics. Software Metrics give us knowledge of the status of an attribute of the software and help us to evaluate this software attribute in an objective way. Software Metrics also helps us to latter make plans for modifications that need to be implemented in the future. In addition, to save the values obtained as history for further reference. What motivates this study of Software Metrics is to develop a software metrics framework that has a set of metrics that could be used as a stand alone metric as needed, or as part of the pipeline of the phases through all the development process. In the next chapter, we will see how one can use software metrics in the early stages of development in order to improve the software product.

# Chapter 3 A Software Metrics Framework

In this chapter, we will describe a series of metrics that are implemented in this thesis, and give a mathematical definition of these metrics. Several programs, with Graphic User Interfaces (GUI), have been created to calculate these metrics. These programs can be used as a stand-alone system to determine the quality or quantity of the software attribute measured. We conclude this chapter with a discussion of the metrics, which we have implemented, and the relationship of the metrics in the different phases of the software development cycle.

## 3.1 Characteristics of the Software Metrics System

The study of software metrics developed on this thesis includes an implementation of those metrics, which can be applied through the Rational Unified Process of Software Development. The core workflows covered in this thesis are the following: Requirements, Design, Coding, Testing, and Maintenance. We have also included another set of metrics to evaluate some aspects of the software process metrics such as effort and productivity.

We wrote programs for the following metrics:

- Specificity and Completeness of Requirements.
- Cyclomatic Complexity, Function Points, Information Flow, and the Bang metric.
- The estimation of Number of Defects, the Lines of Code (LOC) and the Halstead metrics.

- A metric to estimate the Reliability of the software and the estimation of the number of Test Cases.

We have also implemented other metrics that estimate effort: they are the effort according to COCOMO II [5], the effort according to the Statistical Model [30] and the effort according Halstead's metrics [13]. In the following sections a brief overview of the metrics implemented for each workflow is given.

## 3.2 Requirements

This section presents software metrics that measure specificity and completeness of the requirements. As the name states, these metrics should be used during the inception phase, which was defined in chapter 2. They can also be used during the elaboration and construction phases when the business model is being analyzed.

The quality of software requirements specification [30] can be measured by determining two values. The first value is the specificity of the requirement. By specificity, we mean the lack of ambiguity. We use: 'Q1' to refer to the specificity of the requirements. The second value is the completeness. By completeness, we mean how well they cover all functions to be implemented. We use 'Q2' to refer to this completeness. Note that this value doesn't include non-functional requirements. To include the non-functional requirements we use another value denoted by 'Q3', which will be explained below, as described by Davis [7]. Figure 2 shows the implementation of the requirement metrics. To determine the specificity (lack of ambiguity) of requirements we use equation 1.

$$Q1 = \frac{nui}{nr}$$

**Equation 1 Specificity of Requirements**

Where:

Q1 ≡ Specificity of requirements.

nui ≡ number of requirements for which all reviewers had identical interpretations.

nr ≡ Total number of requirements, it is given by:

nf ≡ number of functional requirements

nnf ≡ number of non-functional requirements

nr = nf + nnf

The optimal value of Q1 is one, thus, if our requirements are not ambiguous, we need to get a value closer to one. The lower the value, the more ambiguous the requirements are; and this will bring problems in the latter phases. Therefore, the requirements need to be constantly reviewed and discussed by all team members until they all understand the requirements and agree to adhere to these requirements.

The completeness of functional requirements is given by equation 2:

$$\mathbf{Q}2 = \frac{nu}{ni * ns}$$

**Equation 2 Completeness of Functional Requirements**

Where:

Q2 = completeness of functional requirements only. This ratio measures the percentage of necessary functions that have been specified for a system, but doesn't address nonfunctional requirements.

nu ≡ number of unique functional requirements,

ni ≡ number of inputs (all data inputs) defined or implied by the specification document.

ns ≡ number of scenarios and states in the specification.

We also need to consider the degree to which requirements have been validated. Equation 3 presents this value.

$$\mathbf{Q}3' = \frac{nc}{nc + nnv}$$

**Equation 3 Degree to which the Requirements Have Been Validated**

Where:

Q3' ≡ degree to which the requirements have been validated.

nc ≡ number of requirements that have been validated as corrected.

nnv ≡ number or requirements that have not yet been validated

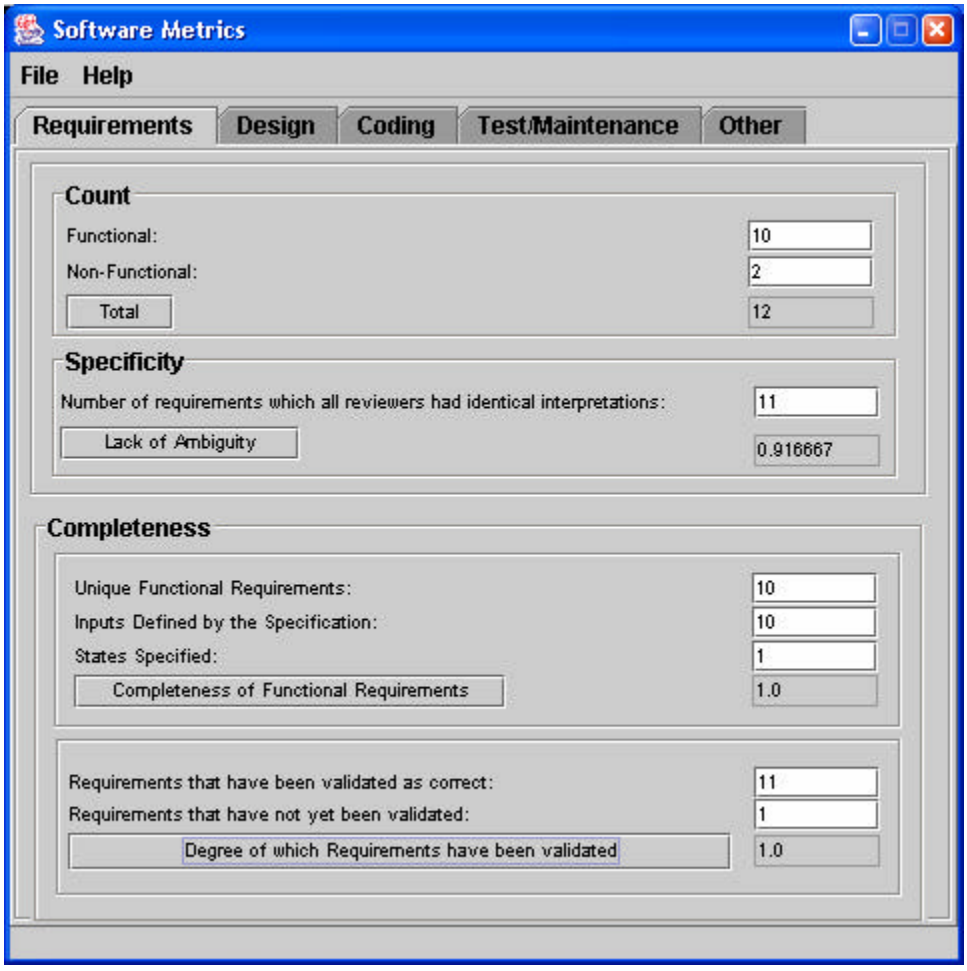


**Figure 2 Requirement Metrics**

## 3.3 Design

This section presents software metrics which can be used during the design phase: McCabe's Cyclomatic number, Information Flow (fan in/out), Function Points and the Bang metric by Tom DeMarco.

### 3.3.1 Cyclomatic Complexity – v (G)

The software metric known as Cyclomatic Complexity was proposed by McCabe [22]. McCabe suggest seeing the program as a graph, and then finding out the number of different paths through it. One example of this graph is called a control graph. Our implementation is shown in Figure 3.

When programs become very large in length and complexity, the number of paths cannot be counted in a short period of time. Therefore, McCabe suggests counting the number of basic paths (all paths composed of basic paths). This is known as the "Cyclomatic Number"; which is the number of basic paths [30] from one point on the graph to another, and we refer to the Cyclomatic Number by v(G). This is shown in equation 4,

$$v(G) = E - N + 2P$$

**Equation 4 Cyclomatic Complexity**

Where:

v (G) ≡ Cyclomatic Complexity

E ≡ number of edges

N ≡ number of nodes

P ≡ number of connected components or parts

We do not have to construct a program control graph to compute v (G) for programs containing only binary decision nodes. We can just count the number of predicates (binary nodes), and add one to this, as shown in equation 5.

$$v(G) = p + 1$$

**Equation 5 Cyclomatic Complexity for Binary Decision Nodes**

Where:

v (G) ≡ Cyclomatic Complexity

p ≡ number of binary nodes or predicates.

**McCabe's rules for counting edges and nodes:**

- **if / while statements:** The action depends on a binary node containing a conjunction or disjunction of conditions. Each condition counts as one binary decision node. For example, in the following sentence, we see that there are two binary decision nodes: `'if (count < m || count = MAX) then...'`
- **do / for statements**: iteration statements count as one binary decision node.
- **case / switch statement**: Either n or n-1 nodes ( n= number of alternatives in the statement). Depends on language semantics of the language and the form of the statement. Examples: C switch statement contributes either n or n-1. Ada case statement contributes n-1.

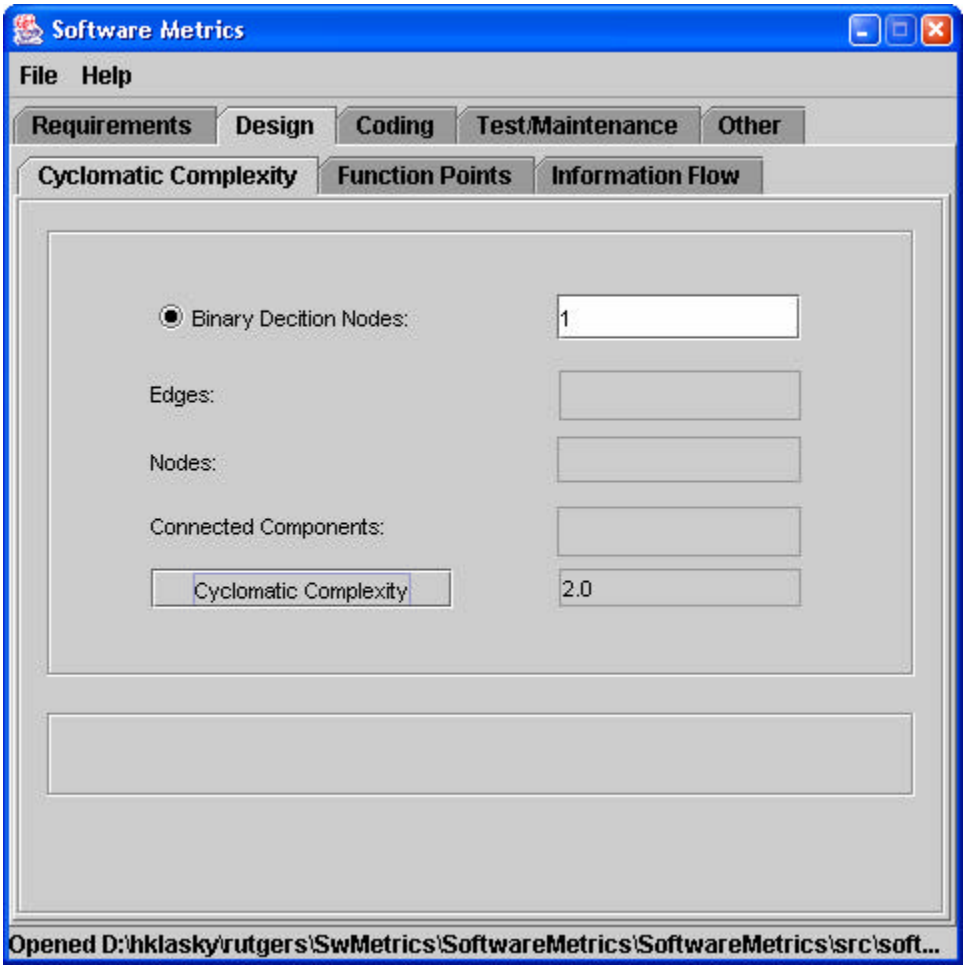


**Figure 3 Cyclomatic Complexity**

Cyclomatic complexity can help in the following situations:

a) Identify software that may need inspection or redesign. We need to redesign all modules with v (G) > 10.

b) Allocate resources for evaluation and test: Test all modules with v (G) > 5 first.

c) Define test cases (basis path test case design): Define one test case for each independent path.

Industry experience shows that Cyclomatic Complexity is more helpful for defining test cases (c), but doesn't help that much on identifying software that may need inspection or

redesign (a) and allocating resources for evaluation and test (b) as McCabe suggests. [31] Studies [32] have also found that the ideal v (G) = 7, which proves to be easier to understand for novices and experts.

### 3.3.2 Functions Points (FP):

This metric was developed by Albrecht [1]. It focuses on measuring the functionality of the software product according to the following parameters: user inputs, user outputs, user inquiries number of files and the number of external interfaces.

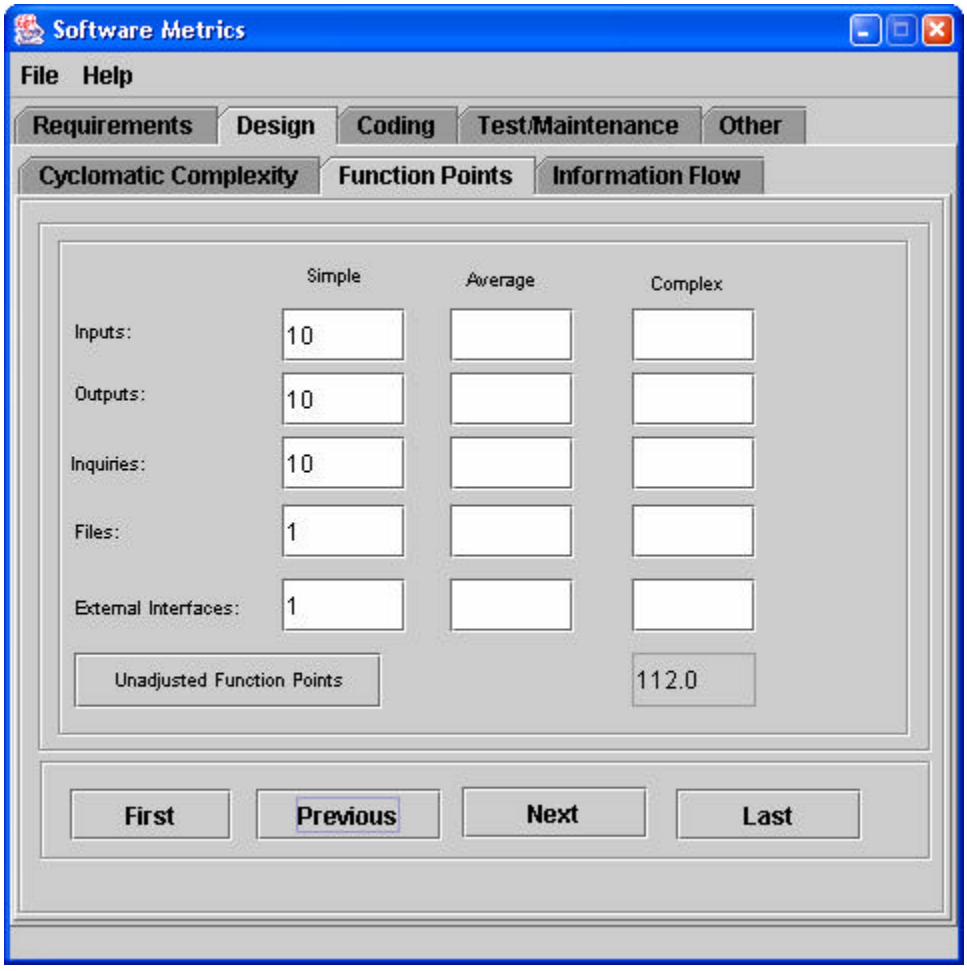


**Figure 4 Unadjusted Function Points**

Once the parameters are counted, a complexity (simple, average and complex) value is associated to each parameter. Figure 4 shows the implementation provided for unadjusted FP. A complexity adjustment value is added to the previous count (see Figure 7). This value is obtained from the response to 14 questions related to reliability of the software product. The implementation of the reliability questions is shown in figure 5 and Figure 6. Equation 6 presents the estimation of Function Points.

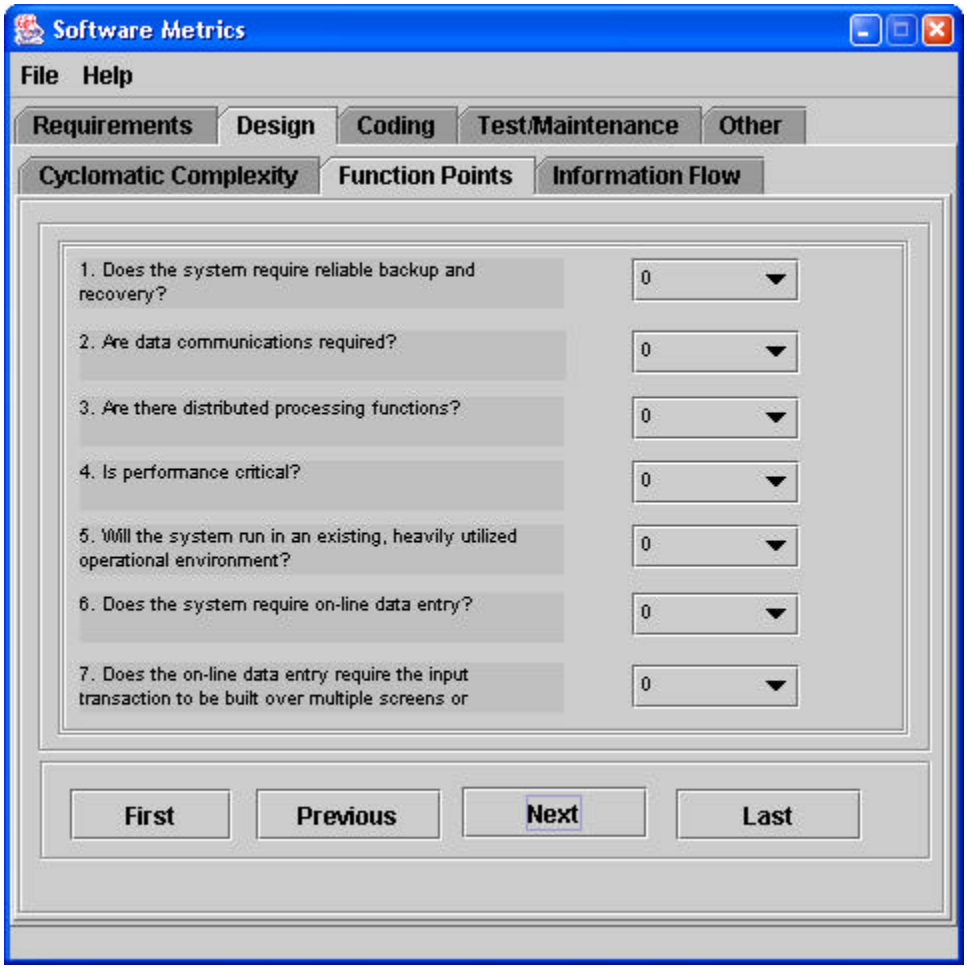


**Figure 5 FP Reliability Questions Part I**

$$FP = count\ total * [0.65 + 0.01 * \sum Fi]$$

**Equation 6 Function Points**

Where:

FP = Total number of adjusted function points

count total ≡ the sum of all user inputs, outputs, inquiries, files and external interfaces to which have been applied the weighting factor.

Fi ≡ a complexity adjustment value from the response to the 14 reliability questions.

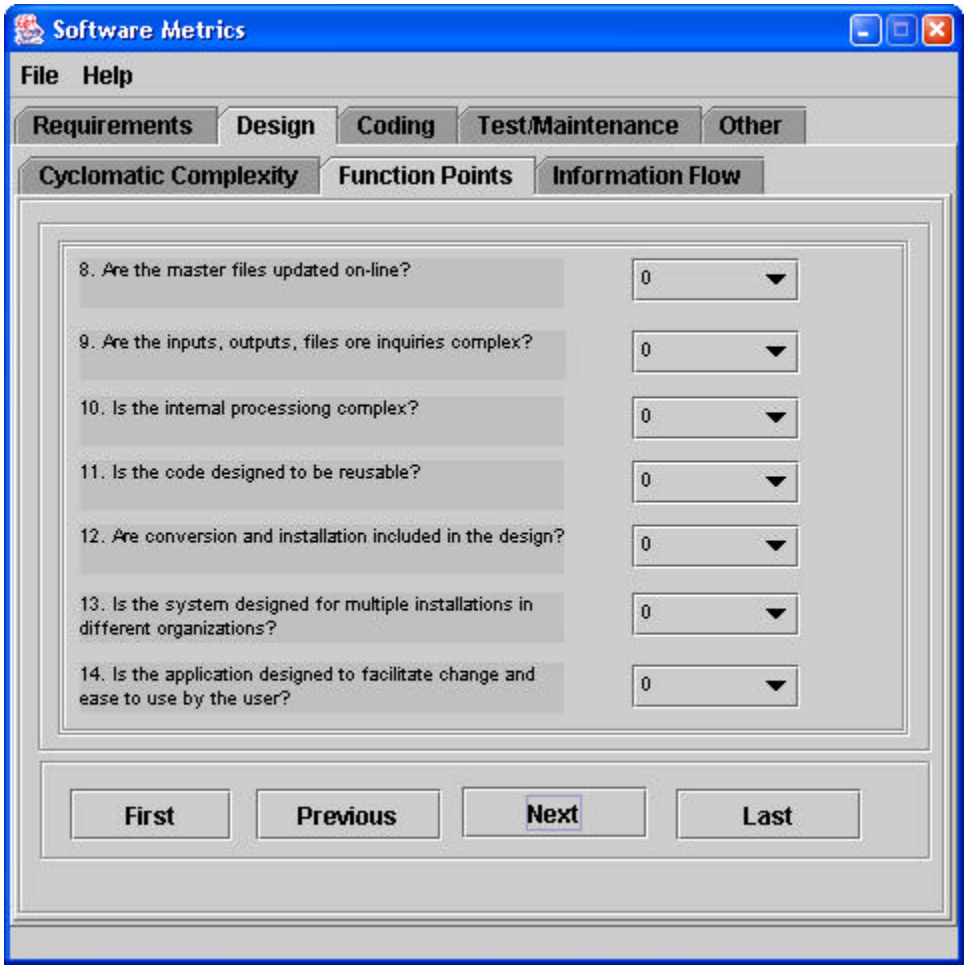


**Figure 6 FP Reliability Questions Part II**

The FP metric is difficult to use because one must identify all of the parameters of the software product. Somehow, this is subjective, and different organizations could interpret the definitions differently. Moreover, interpretations could be different from one project to another in the same organization, and this could be different from one software release to another.

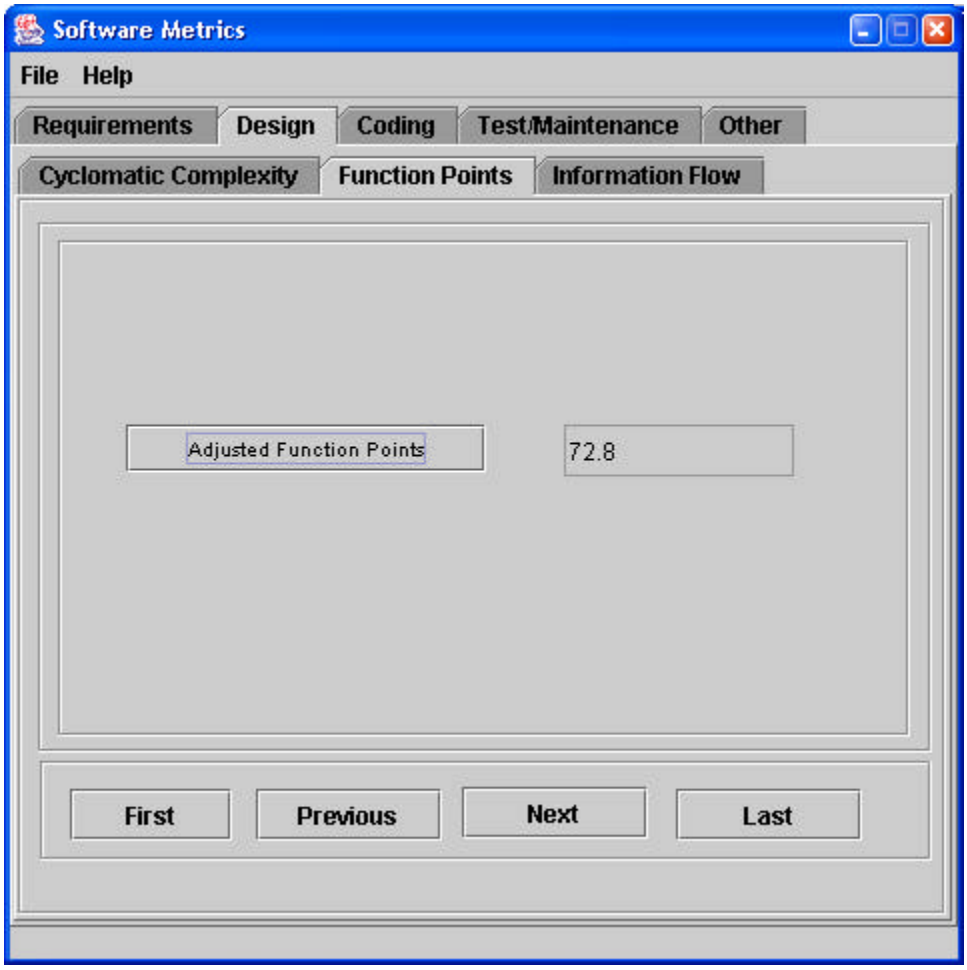


**Figure 7 Adjusted Function Points**

### 3.3.3 Information Flow

Information Flow is a metric to measure the complexity of a software module. This metric was first proposed by Kafura and Henry [19]. The technique suggests identifying the number of calls to a module (i.e. the flows of local information entering: fan-in) and identifying the number of calls from a module (i.e. the flows of local information leaving: fan-out). Figure 8 shows the implementation of Information Flow.

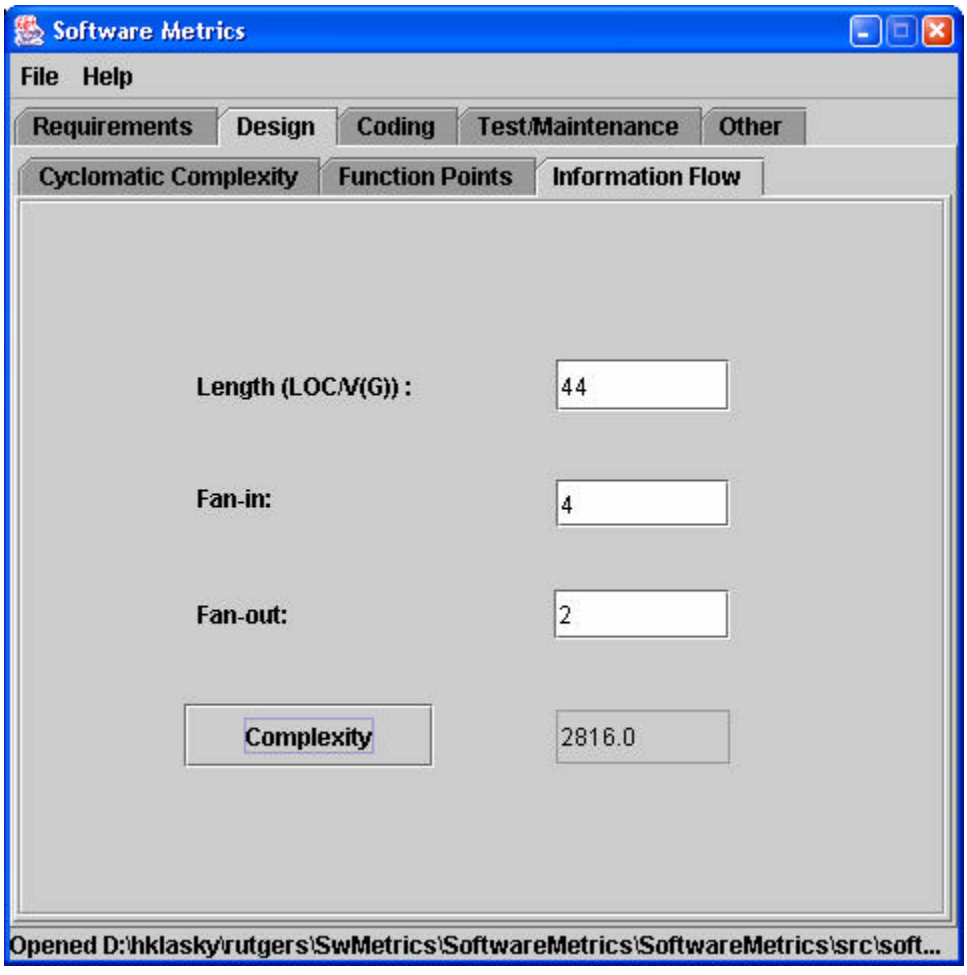


**Figure 8 Information Flow**

The complexity in Information Flow metric is determined by equation 7.

$$c = [procedure\ length] * [fan - in * fan - out]^2$$

**Equation 7 Information Flow**

Where:

c ≡complexity of the module

procedure length ≡ a length of the module, this value could be given in LOC or by using v(G) ≡Cyclomatic Complexity.

fan-in ≡The number of calls to the module.

fan-out ≡ The number of calls from the module.

### 3.3.4 The Bang Metric

The Bang metric was first described by DeMarco [9, 10]. This metric can attempt to measure the size of the project based on the functionality of the system detected during the time of design. The diagrams generated during the design give the functional entities to count. The design diagrams to consider for the Bang metric are: data dictionary, entity relationship, data flow and state transition. Other design diagrams such as the Business Model, Use Cases, Sequence and Collaboration diagrams of the Unified Modeling Language (UML) can also be used to find the functional entities. The Bang metric can also be combined with other measures to estimate the cost of the project.

To estimate the Bang metric, we classify the project in three major categories: function strong, data strong, or hybrid. We first divide the number of relationships between the numbers of functional primitives. If the ratio is less than 0.7 then the system is function strong. If the ratio is greater than 1.5 then the system is data strong. Otherwise, the system is hybrid.

For Function strong systems, we first need to identify each primitive from the design diagrams. Then for each primitive, we identify how many tokens each primitive contains. We assign a corrected Functional Primitive Increment (CFPI) value according to the number of data tokens. At the same time, the primitive will be identified with one class, out of 16 classes. Each primate will also be assigned a value for the increment. Afterward, the CFPI is multiplied by the increment and the resulting value is assigned to the Bang value. This process is repeated for each primitive and the values added all together to compute the final Bang for Function strong systems.

For Hybrid and Data strong systems, we first identify the number of objects. Then, for each object, we count the number of relationships and then record the Corrected Object Increment (COBI) accordingly. The resulting value will be added to the values of each object and the total will be assigned to the final Bang computation for Hybrid and Data strong systems.

## 3.4 Coding

This section presents the software metrics appropriate to use during the implementation phase of the software design. The metrics presented in this section are: Defect Metrics, Lines of Code (LOC), and the Halstead product metric.

### 3.4.1 Estimation of Number of Defects

During the1998 IFPUG conference, Capers Jones [16] gave a rule of the thumb to get an estimation of the number of defects based on the Function Points of the system. This rule is defined by equation 8.

$$Potential\ Number\ of\ Defects = FP^{1.25}$$

**Equation 8 Potential Number of Defects**

Where:

FP ≡ Function Points

Equation 8 is based on the counting rules specified in the Function Point Counting Practices Manual 4.0[16] (http://www.ifpug.org/) and it is currently in effect. Figure 9 shows the implementation of this metric.

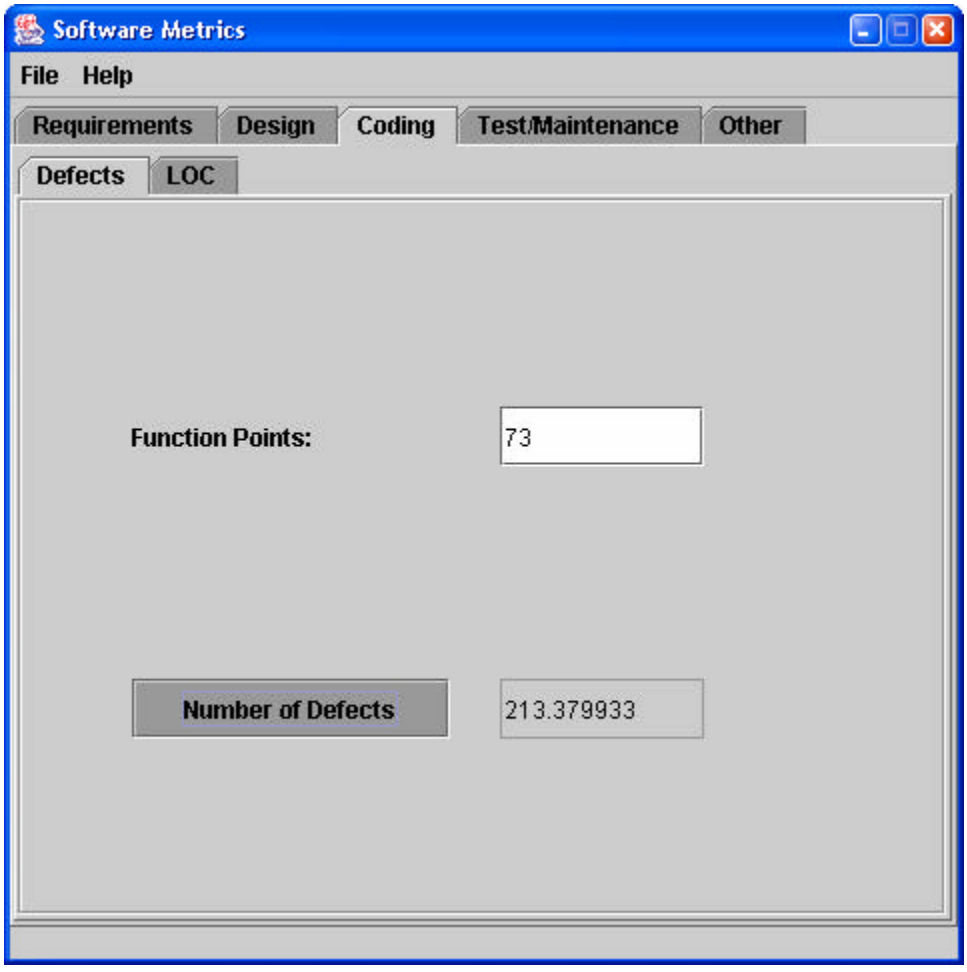


**Figure 9 Estimating the Potential Number of Defects**

### 3.4.2 Lines of Code

The Lines Of Code (LOC) metric specifies the number of lines that the code has. The comments and blank lines are ignored during this measurement. The LOC metric is often presented on thousands of lines of code (KLOC) or source lines of code (SLOC).

LOC is often used during the testing and maintenance phases, not only to specify the size of the software product but also it is used in conjunction with other metrics to analyze other aspects of its quality and cost. Figure 10 shows the implementation of LOC.

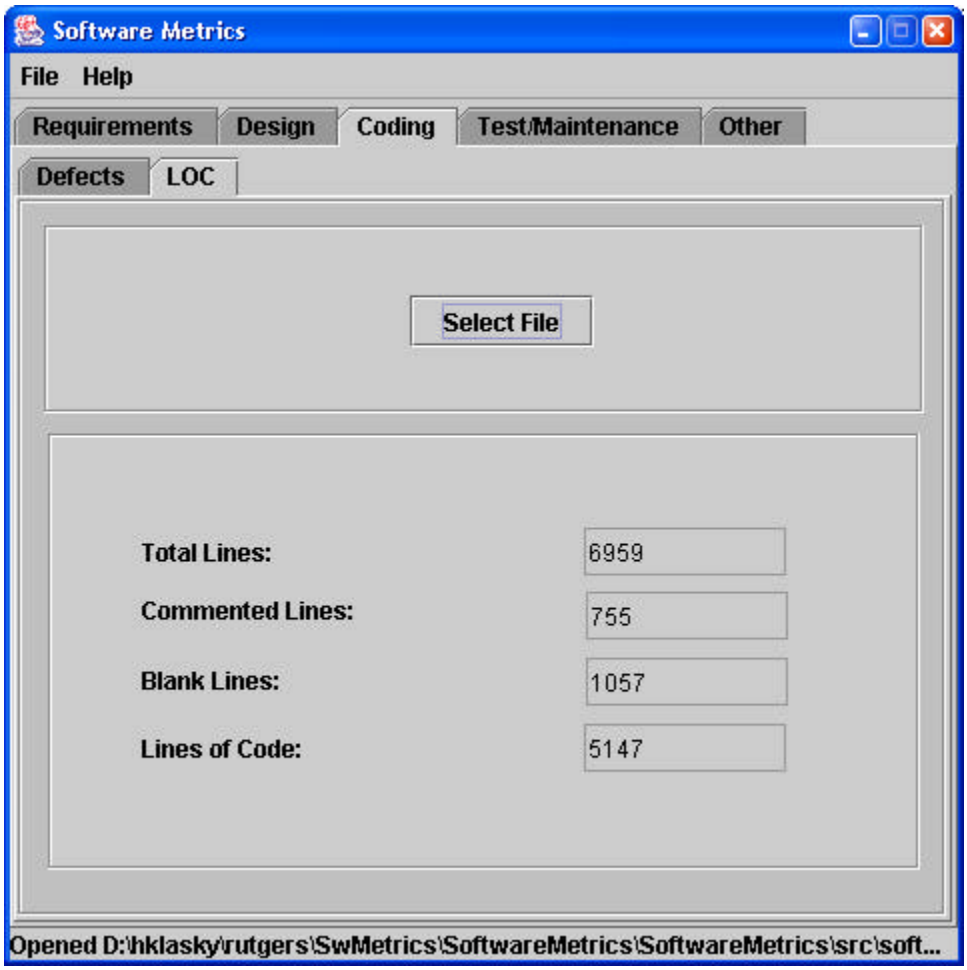


**Figure 10 Lines of Code (LOC)**

Several LOC tools are enhanced to recognize the number of lines of code that have been modified or deleted from one version to another. Usually, modified lines of code are taken into account to verify software quality, comparing the number of defects found to the modified lines of code.

Other LOC tools are also used to recognize the lines of code generated by software tools. Often these lines of code are not taken into account in final count from the quality point of view since they tend to overflow the number. However, those lines of code are taken into account from the developer's performance measurement point of view.

### 3.4.3 Product Metrics of Halstead

Halstead [13] was the first to write a scientific formulation of software metrics. Halstead stated that any software program could be measured by counting the number of operators and operands, and from them, he defined a series of formulas to calculate the vocabulary, the length and the volume of the software program. Halstead extends this analysis to also determine effort and time.

a) **Program Vocabulary***:* is the number of unique operators plus the number of unique operands as defined by equation 9.

$$n = n1 + n2$$

**Equation 9 Halstead's Program Vocabulary**

Where:

n ≡ program vocabulary

n1 ≡ number of unique operators

n2 ≡ number of unique operands

b) **Program Length***:* the length N is defined as the total usage of ‘all’ operators appearing in the implementation plus the total usage of ‘all’ operands appearing in the implementation. This definition defined by equation 10.

$$N = N1 + N2$$

**Equation 10 Halstead's Program Length**

Where:

N ≡ program length

N1 ≡ ‘all’ operators appearing in the implementation

N2 ≡ ‘all’ operands appearing in the implementation

c) **Program Volume***:* Here volume refers to size of the program and it is defined as the Program Length times the Logarithm base 2 of the Program Vocabulary. This definition is defined by equation 11.

$$V = N \log_2 n$$

**Equation 11 Halstead's Program Volume**

Where:

V ≡ program volume

N ≡ program length (see equation 10)

n ≡ program vocabulary (see equation 9)

## 3.5 Testing / Maintenance

This section presents software metrics appropriate to use during the testing/maintenance phase of the software. The metrics mentioned follow: Defect Metrics, that consist on the

number of design changes, the number of errors detected during system testing, and the number of code changes required; and the Reliability metrics.

### 3.5.1 Defect Metrics

Defect metrics help to follow up the number of defects found in order to have a record of the number of design changes, the number of errors detected by code inspections, the number of errors detected during integration testing, the number of code changes required, and the number of enhancements suggested to the system. Defect metrics' records from past projects can be saved as history for further reference. Those records can be used latter during the development process or in new projects. Companies usually keep track of the defects on database systems specially designed for this purpose.

### 3.5.2 Software Reliability

Software reliability metrics use statistical methods applied to information obtained during the development or maintenance phases of software. In this way, it can estimate and predict the reliability of the software product. Figure 11 shows how the software reliability behaves during the different development phases.

Availability metrics [24] are also related to the reliability of the system. Sometimes, there is confusion in regards to the software availability and the software reliability; a definition of both terms is given below.

**Software availability**: The period of time in which software works satisfactorily.

**Software reliability***:* The probability that a system will not fail in a period of time.

To calculate Software Availability, we need data for the failure time/rates and for the restoration time/rates. Therefore, the Availability is defined by equation 12.

$$Availability = 1 - unavailability$$

**Equation 12 Software Availability Metric**

Note: unavailability here refers to both hardware and software.

To calculate Software Reliability, we need to collect data from the system in regards to the failures experienced during operation in a period of time. A software failure is represented with by λ; where λ represents the failure rate per unit time (per hour, month, etc.). $\lambda(\tau)$ is a function of the average total number of failures until time t, denoted by $\mu(\tau)$ .The failure rate decreases as the number of bugs is fixed and no new bugs are discovered.

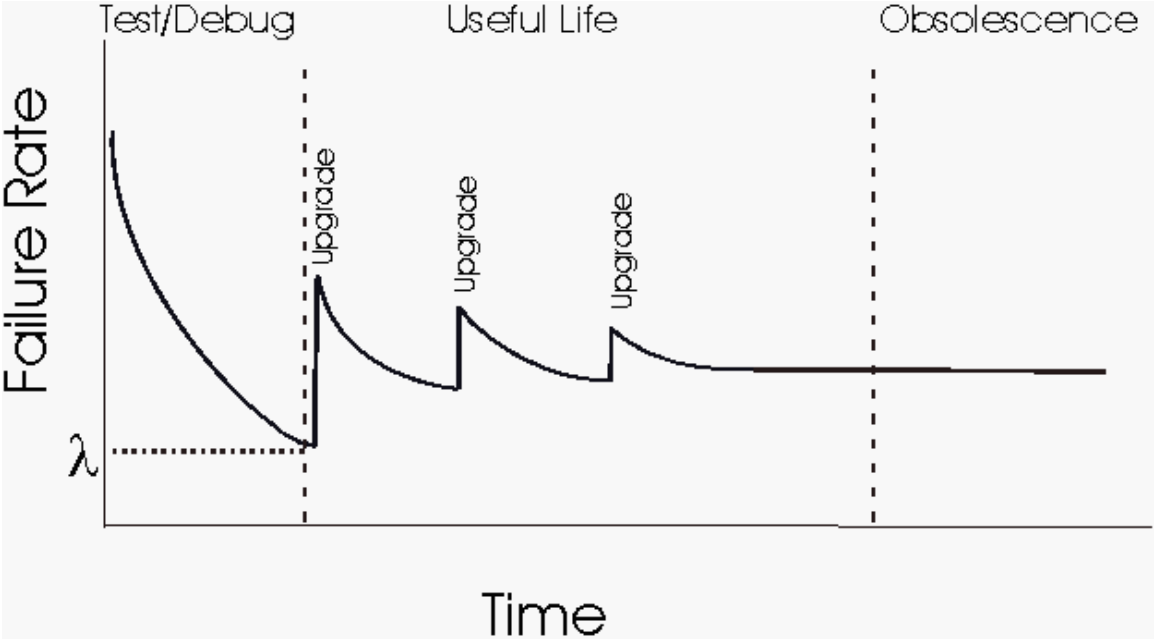


**Figure 11 Reliability Metrics**

We used the Software Reliability Growth (SRG) Logarithmic Model developed by Musa [24]. In this model, the function Lambda (t) is defined by equation 13.

$$l(t) = l_0 * \exp^{(-bt)}$$

**Equation 13 Software Reliability Growth Logarithmic Model**

Where:

$l_0 = E * b$

E ≡ the number of defects in the software at t = 0 (i.e., at the time of delivery/start of operation)

b is the Mean Time To Failure (MTTF) which is given by: $b = \frac{1}{t_p}$

$t_p$ ≡ the time constant, is the time at which 0.63*E defects will have been discovered.

### 3.5.3 Estimation of Number Test Cases

On 1998, in the IFPUG conference Capers Jones [16] gave a rule of the thumb to estimate the number of test cases based on the number Function Points of the system. Figure 12 shows the implementation of this metric. This rule is described by equation 14.

$$Number\ of\ test\ cases = FP^{1.2}$$

**Equation 14 Number of Test Cases**

Where:

FP = Function Points.

The formula is based on the counting rules specified in the Function Point Counting Practices Manual 4.0 [16], [http://www.ifpug.org/]. Equation 14 for estimating the potential number of software defects is currently in effect.

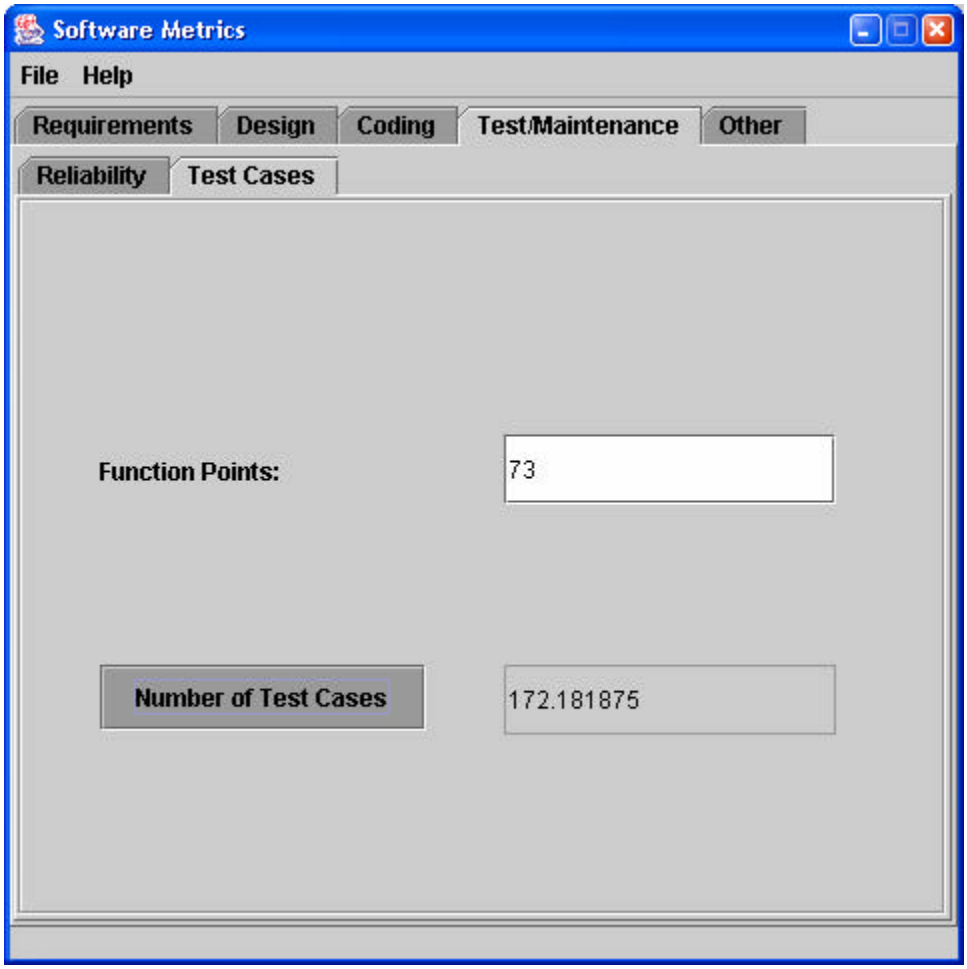


**Figure 12 Estimating Number of Test Cases**

## 3.6 Other

This section presents software metrics that could help to know the quality of the system during all phases of the project in regards to the Effort, Time and Cost of it:

### 3.6.1 COCOMO II

COCOMO II is the new version of the Constructive Cost Model for software effort, cost and schedule estimation (COCOMO) [5]. COCOMO II adds the capability of estimating the cost of business software, OO software, software created via evolutionary or spiral

model and other new trends (for example commercial off the shelf applications –COTS) [4].

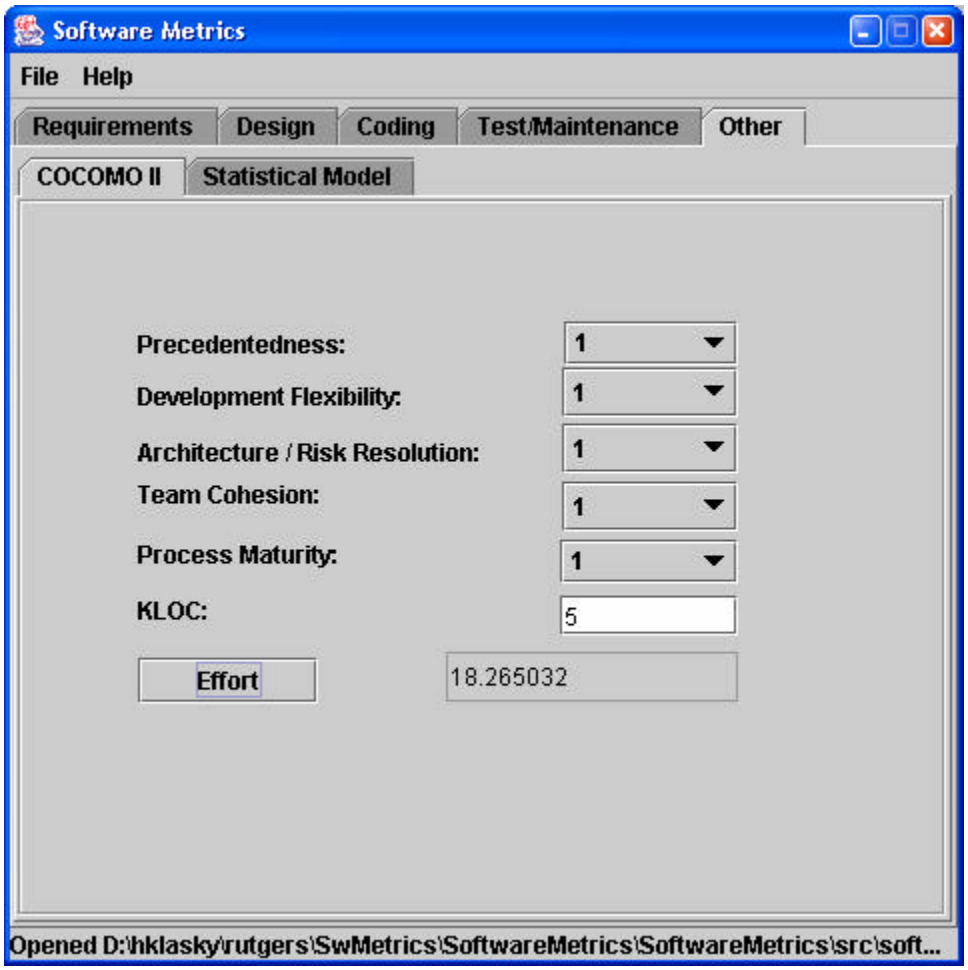


**Figure 13 COCOMO II Effort**

The market sectors, which benefit the most from COCOMO II, are: 1) the 'Application Generators' (Microsoft, Lotus, Novell, Borland, etc.), 2) the 'Application Composition' sector that focuses on developing applications which are too specific to be handled by prepackaged solutions, and 3) the 'Systems Integration' sector, which works with large scale, highly embedded systems (for example: EDS and Andersen Consulting, etc.).

The COCOMO II model for the Application Composition sector is based on Object Points. Object Points count the number of screens and then it will report back on the

application a weighted which can be three levels of complexity: simple, medium and difficult. Figure 13 shows this implementation.

The COCOMO II model for the Application Generator and System Integrator sectors is based on the Application Composition model (for the early prototyping phase), and on the Early Design (analysis of different architectures) model and finally on the Post-Architecture model (development and maintenance phase).

In COCOMO II metric, the effort is expressed in Person Months (PM). Person Months (also known as man-months) is a measure that determines the amount of time one person spends working in the software development project for a month.

The number of PM is different from the time the project will take to complete. For example, a project may be estimated to have 10 PM but have a schedule of two months. Equation 15 defines the COCOMOII effort metric.

$$E = A * (KLOC)^{B}$$

**Equation 15 COMOMO II Effort**

Where:

E ≡Effort

A ≡ A constant with value 2.45

KLOC ≡Thousands of Lines of Code

B ≡ factor of economies or diseconomies (costs increases), given by equation 16.

$$B = 1.01 + (0.01) * \left(\sum SFi\right)$$

**Equation 16 COCOMO II Factor of Economies and Diseconomies**

Where:

? SFi is the weight of the following Scale Factors (SF): Precedentedness (how new is the program to the development group), Development Flexibility, Architecture/Risk Resolution, Team Cohesion, and Process Maturity (CMM KPA's).

### 3.6.2 Statistical Model

Another approach to determine software effort has been provided by C. E. Walston and C.P. Felix of IBM [30]. They used equation 17 to define the effort in its statistical model.

$$E = a * L^{b}$$

**Equation 17 Statistical Model**

Where:

E ≡ Effort

a ≡ a constant with value: 5.2

L ≡ Length in KLOC

b ≡ a constant with value: 0.91

Walston and Felix also calculated the Nominal programming productivity in LOC per person-month as defined by equation 18.

$$P = \frac{L}{E}$$

**Equation 18 Nominal Programming Productivity**

Where

P ≡ Nominal programming productivity in LOC per person-month

L ≡ Length in KLOC

E ≡ Effort from equation 17.

The implementation of this metric is shown in figure 14.

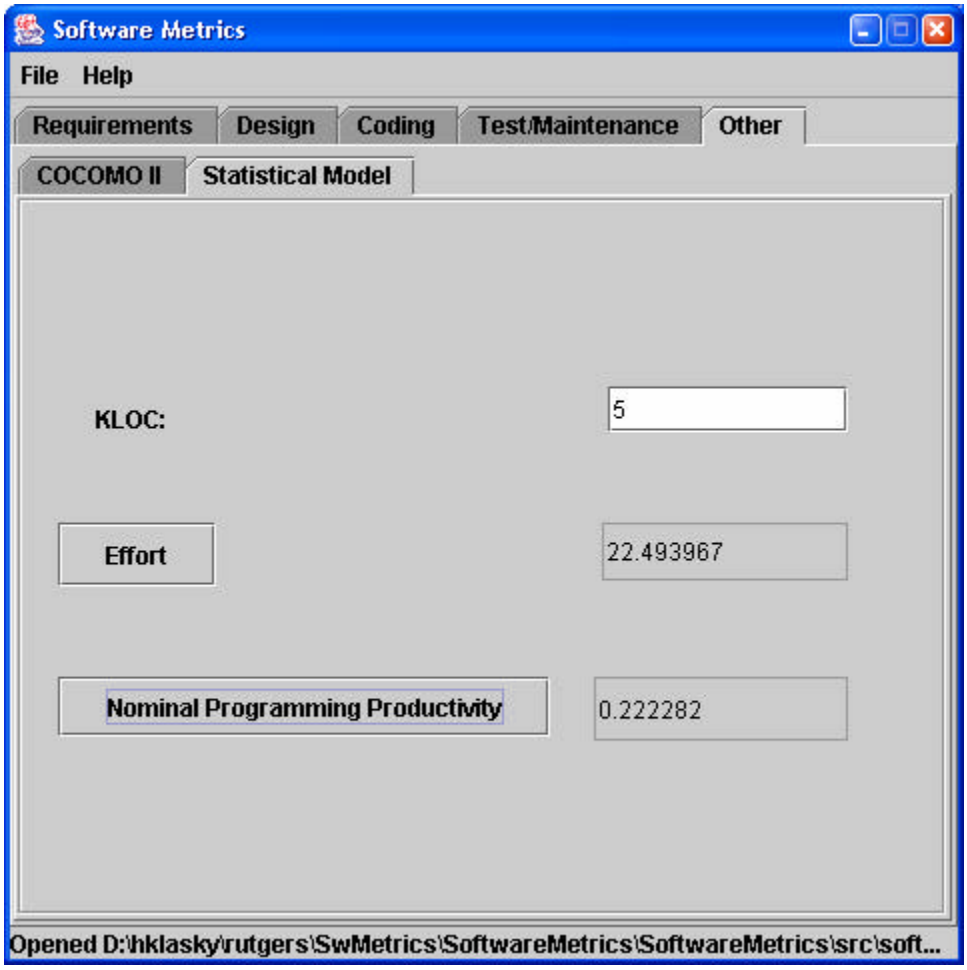


**Figure 14 Statistical Model Effort**

### 3.6.3 Halstead Metric for Effort

Halstead has proposed an effort metric to determine the effort and time which is given by equations (19), (11), (20) and (21):

$$E = \frac{V}{L}$$

**Equation 19 Halstead's Effort Metric**

Where:

E ≡ Effort in mental discriminations needed to implement the program.

$V = N \log_2 n$ (See equation 11)

Where:

V ≡program volume

N ≡program length (see equation 10)

n ≡program vocabulary (see equation 9)

$$L = \frac{V^*}{V}$$

**Equation 20 Halstead's Program Level**

Where

L ≡program Level

V*≡ potential volume given by equation 21.

$$V^* = \left(N1^* + N2^*\right) * \log_2 \left(n1^* + n2^*\right)$$

**Equation 21 Halstead's Potential Volume**

Halstead's metrics include one of the most complete sets of mathematical combinations of software metrics and even though most people recognizes that they are interesting not many companies follow them due that is hard to convince people about change solely on the basis of these numbers [12].

## 3.7 Discussion

In this chapter, we have presented the following metrics:

- Specificity and Completeness of Requirements,

- Cyclomatic Complexity, Function Points, Information Flow and The Bang Metric,
- An Estimation of the Number of Defects, The Lines of Code (LOC), The Halstead Metrics,
- The Reliability metric, An Estimation of the Number of Test Cases,
- The Estimation of Effort According to COCOMO II, The Effort According to the Statistical Model, The Effort According to Halstead's Metrics.

We have seen the importance of properly defining the requirements. If the metrics are properly defined, we can avoid problems that will be more expensive to correct during the latter phases of development. Function Points can be used to estimate the potential number of defects and the number of test cases to be written to test the software. LOC can also give us an estimation of the Function Point according to the language used. It is useful to validate a value such as the effort, with different metrics to see how close are of each other.

# Chapter 4 Implementation and Evaluation

Chapter 4 provides an experimental evaluation of the software metrics implemented in this study. The chapter begins with the hardware and software requirements, which are needed to run the program. Then, the design of the system is outlined. The design is followed by the software metrics values obtained by applying the software metrics to the program development of this study.

## 4.1 Hardware and Software Requirements

The Software Metrics framework has been tested and run on machines with the following software and hardware requirements:

- **Minimal hardware Requirements**

  Processor: X86 Processor (Pentium II)

  RAM: 256 MB

  Hard Drive: 4 GB free

- **Software Requirements:**

  Windows 2000 Professional Service Pack 3

  Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.1_02-b06)

  IMPORTANT NOTE: Older versions of java don’t work to run this software.

## 4.2 How To Run The Program

This section provides instructions about how to run the program. First, the jar file needs to be saved in a directory on the local hard drive. Then type in the following command in

the command line (assumed that you are positioned on the current directory where you have saved the jar file):

```
java  –jar  SoftwareMetrics.jar
```

NOTE: The system could take 10 to 20 seconds to display on the screen.

## 4.3 Development Experience

The software was developed as follows. All metrics were developed first as their own classes. Then, the GUI was developed by using JBuilder 8.0 Enterprise Trial. The code of the metrics is completely independent from the GUI and other GUI could be used to visualize the results. The reason why we chose JBuilder for the GUI is to facilitate the work by dragging and dropping swing components in panels. This Java Swing GUI generator was helpful during the development of the system prototype.

## 4.4 Main Classes of the Software Metrics System

This section describes the design of the Software Metrics framework implemented on this thesis. First, a list of the main classes is presented. Then a diagram of the interaction of the classes is provided.

### 4.4.1 Classes Implemented

Table 2 presents a list of the classes implemented. On Table 2, the classes have been organized alphabetically. One java file and class has been implemented per metric. Some classes required the creation of private classes; those cases are identified when it is appropriate to mention them. The software metrics framework has the following main classes:

| Class/File Name | Brief Description |
|---|---|
| Bang.java | Implements the Bang metric. |
| Fup | Bang's private class that consist of an array of Functional Primitives for function strong systems. |
| OB | Bang's private class that consist of an array of objects for hybrid and data strong systems. |
| CocomoII.java | Implements the COCOMO II effort metric. |
| Factor | Private class of COCOMO II that is used to keep the weights of each factor to compute the effort. |
| CycloComplex.java | Implements the Cyclomatic Complexity metric. |
| Defects.java | Implements the potential number of defects. |
| FP.java | Implements the Function Points metric. |
| Halstead.java | Implements the Halstead's metrics. |
| Operator | Halstead's private class that consist of an array of operators. |
| Operand | Halstead's private class that consist of an array of operands, they are used to get the vocabulary of the system. |
| InfoFlow.java | Implements the Information Flow metric. |
| LOC.java | Implements the counter of lines of code of a file. |
| Reliability.java | Implements the reliability metric. |
| reqMetrics.java | Implements the requirement metrics. |
| Statistic.java | Implements the statistic metric. |
| TestCases.java | Implements an estimation of the number of test cases. |

**Table 2 Software Metrics Classes Implemented**

Table 3 shows, between other classes, the classes implemented for the Java Swing GUI. First, Table 3 lists the main class that calls all other classes, then the class that generates the frame, and then, the class that implements the about dialog. Finally, there is a class

that implements common functions. The tool class implements generic functions that are used by all other classes.

| Class/File Name | Brief Description |
|---|---|
| SoftwareMetricsClass.java | Contains the main method. |
| SoftwareMetricsFrame.java | Implements the frame Java Swing GUI. |
| SoftwareMetricsFrame_AboutBox.java | Implements the about dialog. |
| Tools.java | Implements generic tools functions used by all classes. |

**Table 3 Java Swing GUI Classes and other Tools implemented**

The diagram in figure 15 shows the interaction of the main classes implemented in the software metrics system developed on this thesis. Figure 15 depicts how classes are called from one to another one. Initially the SoftwareMetricsClass is invoked, it has the main method. Then the SoftwareMetricsFrame appears. The frame presents different tabs. There is one tab per workflow. The workflows are: Requirements, Design, Coding, Testing and Maintenance, and Other for the Effort Metrics. Each workflow has at the same time several tabs, one per metric implemented. Some classes have private classes, on those cases the private classes are indicated on the diagram.

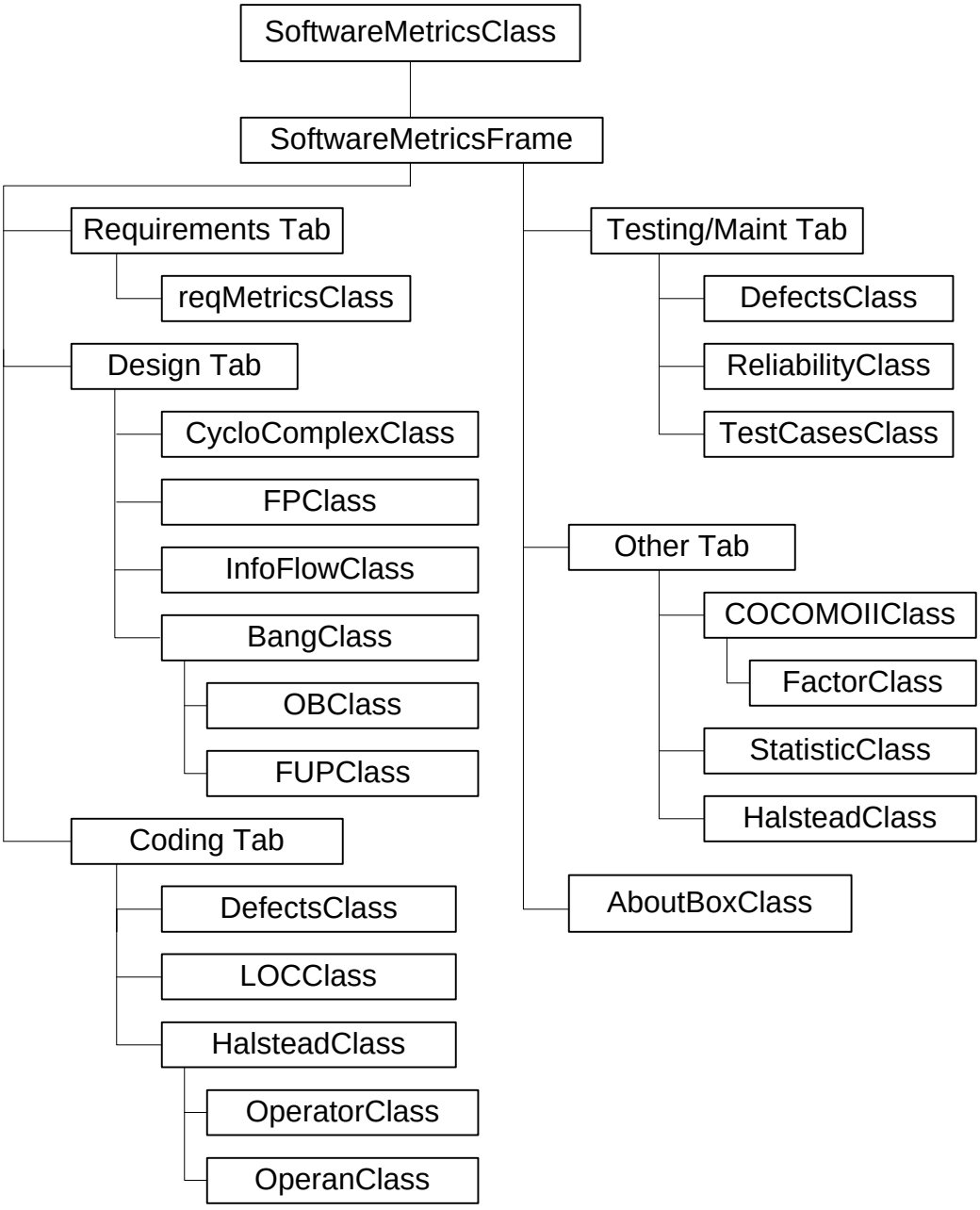


**Figure 15 Software Metrics Design Diagram**

## 4.5 Software Metrics Obtained

For the Software Metrics project, to ensure the lack of ambiguity on the requirements it is important that all members that will be involved on it have the same interpretation of them. We are looking for a lack of ambiguity of one or close to one. See Figure 2.

The Cyclomatic Complexity measure was applied to the CycloComplex.java class, this class has one if statement that counts as 1 binary decision nodes, (if it has or not binary decision nodes), so the Cyclomatic Complexity v (G) is 2. It is important to remember that it is recommended to re-design the modules with v (G) greater than 10. In this case, the complexity has a low value so it is good. (See Figure 3.)

The LOC for the Software Metrics developed in this thesis is 5,147 LOC. The SoftwareMetricsFrame.java is the file that has the most lines of code (it has 3,847 LOC). These values exclude comment lines and blank lines (about 1800 commented and blank lines in this project). See Figure 10.

Figure 4 shows the estimation of Unadjusted Function Points for this Software Metrics system. All inputs, outputs and inquiries are simple; so, they are considered as one value per metric implemented. Because the complexity adjustment values are minimal for this system, the number of Adjusted Function Points is: 72.8. For this project, the 14 questions of reliability were not important, so no weight was assigned to them (see Figure 5 and Figure 6). This gives us a total of 72.8 Adjusted Function Points (Figure 7), or 112 Un-adjusted Function Points (Figure 4).

We can validate if the number of Function Points is correct, by using the table of Function Points per LOC according with the language used (http://www.qsm.com/FPGearing.html). In the java programming language, there is an

average of 62 lines of code per Function Point which gives a value of 83 Function Points for this system that has 5,147 LOC. Counting other function points for the exit function and the About dialog would give a very close approximation.

The Information Flow metric was applied to the InfoFlow.java class, this class has 44 lines of code, it has 4 fan-in (calls in the module) and 2 fan-out (calls from the module) this gives a complexity value of 2816. See Figure 8.

For 73 Function Points the potential number of defects is 213, see Figure 8. For 213 defects on 6 months of development the reliability of the system is 1.51, see Figure 11. For 73 Function Points the number of test cases that need to be written and run is: 172. See Figure 12.

The COCOMO II effort obtained is 18 man/months (see Figure 13). We can compare the same value with the Statistic metric, which gives a value of 22 man/months (see Figure 14). The difference in the effort given by those two metrics show that the results are not conclusive.

# Chapter 5 Conclusions and Future Work

Software Metrics are tools that help us to qualify or quantify software attributes in an objective way. They can also be applied to the software development process. Software Metrics provide value to the software development process by giving information about the status of the project and product of software. Software Metrics also provide a way to know if the development goals are being achieved. Software Metrics from past projects can be used as reference on further projects.

Software Metrics that are too complex or require many data entry parameters could be difficult to understand. When Software Metrics are too complex, the values obtained from them might be ambiguous, which could make the team members abandon the practice.

Applying software metrics is good practice that can bring a lot of value to a project, but it requires time, work and money. By using software metrics in a consistent manner, software developers will see improvement in the software and on the use of the metrics. Lack of consistency while using software metrics could lead to ambiguous results.

No unique metric works during all of the development phases. Using several metrics for one system helps to have a handy solution that can be used during different aspects of the process of software development. The project developed in this thesis provides a solution for this need by implementing metrics that can be applied on the several aspects of the Rational Unified Process.

The metrics covered in this thesis are the following: For the Requirements workflow, the Specificity and Completeness of Requirements. For the Design: Cyclomatic Complexity, Function Points, Information Flow and the Bang Metric. For the Implementation: the

Estimation of Number of Defects, the Lines Of Code (LOC) and the Halstead Metrics. For the Test and Deployment: the Number of Defects is again mentioned, a metric to estimate the Reliability of the software and the estimation of the Number of Test Cases. We have included metrics that support the workflows which estimate effort: they are the effort according to COCOMO II, the effort according to the Statistical Model and the effort according Halstead Metrics.

The toolkit developed in this work could be enhanced by providing a LOC counter that can work with several files. It could also be improved by parsing projects with multiple files, to use as input for the metrics implemented in this thesis. Other future work could include adding a database so a better history of the metrics of the system could be tracked. Finally, adding a metric that graphically shows the expected time when the system is supposed to be “bug” free or stable will be very helpful (statistics and probability curves).

# References

[1] Albrecht, A. J. and J. E. Gaffney. Jr. *"Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation"*, IEEE Trans. Software Eng. SE-9, 6, Nov. 1983, pp. 639-648.

[2] Arthur, L. J. *"Measuring Programmer Productivity and Software Quality"*, New York, John Wiley, 1985.

[3] Austin, Robert D., Lister, Timothy R., Demarco, T. *"Measuring & Managing Performance in Organizations"*, New York, Dorset House, June, 1996.

[4] Boehm, B. W. *"Software Engineering Economics"*, Englewood Cliffs, New Jersey, Prentice-Hall, 1981.

[5] USC-CSE, 1999, *"COCOMO II Definition Manual"*, Computer Science Department, University of Southern California, Center for Software Engineering, Los Angeles, CA, 1999.

[6] Curtis, B., S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love. *"Measuring the Psychological Complexity of Software Maintenance tasks with the Halstead and McCabe Metrics"*, IEEE Trans. Software Eng. SE-5, 2 (March 1979), pp. 96-104.

[7] Davis, A., et al. *"Identifying and Measuring Quality in a Software Requirements Specification"*, Proc. First Intl. Software Metrics Symposium, IEEE, Baltimore, MD, May 1993, pp. 141-152.

[8] Dekkers, C., *"Function Points and Use Cases – Where's the Fit?"* IT Metrics Strategies, January 1999, pp. 34-36.

[9] DeMarco, T. *"Controlling Software Projects: Management, Measurement & Estimation"*, New York, Yourdon Press, 1982, pp. 184-192.

[10] DeMarco, Tom and Boehm, Barry W. *"Controlling Software Projects: Management, Measurement, and Estimates"*, Prentice Hall PTR/Sun Microsystems Press, March 1998, pp. 80-91.

[11] Florac, W.A., Carleton A.D. *"Measuring the Software Process"*, SEI Series in Software Engineering. Addison-Wesley. Second printing, Canada, November 2001.

[12] Grady, Robert B. *"Practical Software Metrics for Project Management and Process Improvement"*, Hewlett-Packard Professional Books, Prentice Hall, New Jersey, 1992.

[13] Halstead, M. H. *"Elements of Software Science"*, New York: Elsevier North-Holland, 1977.

[14] Henry, S., D. Kafura, and K. Harris. *"On the Relationships Among Three Software Metrics"*, Performance Eval. Rev. 10, 1 (Spring 1981), pp. 81-88.

[15] Herron, D. *"A Measure of Success"*, Silicon India, July 1998, pp.1-5.

[16] International Function Point Users Group Staff. *"Function Point Counting Practices Manual"*, International Function Point Users Group, Release 4.1.1, Princeton, NJ, 2001.

[17] Jones, T. C. *"Programming Productivity"*, New York, McGraw-Hill, 1986.

[18] Jones, C. *"Software Assessments, Benchmarks, and Best Practices"*, Addison-Wesley, Boston MA, April 2000.

[19] Kafura, D. and G. R. Reddy. *"The Use of Software Complexity Metrics in Software Maintenance"*, IEEE Trans. Software Eng. SE-13.3 (March 1987), pp. 335-343.

[20] Kemerer, C. F. *"An Empirical Validation of Software Cost Estimation Models"*, Comm. ACM 30, 5 (May 1987), pp. 416-429.

[21] Kemerer, C.F. and Porter, B.S. *"Improving the Reliability of Function Point Measurement: An Empirical Study"*, IEEE Transactions on Software Engineering, Vol. SE-18, No. 11, Nov. 1992, pp. 1011-1024.

[22] McCabe, T. J. *"A Complexity Measure"*, IEEE Trans. Software Eng. SE-2 (4) (Dec. 1976), pp. 308-320.

[23] Mellis E. *"Software Metrics. SEI Curriculum Module"*, SEI-CM-12-1.1, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, December, 1998.

[24] Musa, J. D., A. Iannino, and K. Okumoto, *"Software Reliability: Measurement, Prediction, Application"*, New York, McGraw-Hill, 1987.

[25] Myers, G. J. *"An Extension to the Cyclomatic Measure of Program Complexity"*, ACM SIGPLAN Notices 12, 10 (Oct. 1977), pp. 61-64.

[26] Pressman, R. S. *"Software Engineering A Practitioner's Approach"*, 5th Edition. New York, McGraw Hill, 2000.

[27] Rational Software Staff. *"Rational Unified Process. Best Practices for Software Development Teams"*, Rational Software White Paper. TP026B, Rev 11/01, Rational Software, 2001.

[28] Shepper, M. *A. "Critique of Cyclomatic Complexity as a Software Metric"*, Software Engineering Journal, vol. 3 (March 1988), pp. 30-36.

[29] Stetter, F. *"A Measure of Program Complexity"*, Computer Languages 9, (3-4) (1984), pp. 203-208.

[30] Walston, C. E. and C.P. Felix, *"A Method of Programming Measurement and Estimation"*, IBM Systems Journal, 16, (1), (1977), pp. 54-73.

[31] Bail, W., and G. Vecellio. *"Difficulties in Using Cyclomatic Complexity on Software with Error Handling"*, The MITRE Corporation, Software Eng. Center, Bedford, MA, March 1998. [http://www.mitre.org/support/swee/html/60_bail/sld001.htm].

[32] Ashish, Woldeit O., Zeron, L. *"Experiment for the Correlation Between Cyclomatic Complexity and Comprehension of Program"*, Oregon State University, Corvallis, OR, 1998. [http://cs.oregonstate.edu/~ashish/FinalReport1.html]