**Software Metrics Directory**

**Prepared By**

Md Mynuddin

Year:3 Term :2

Roll: ASH1825007M

**Institute of Information Technology, NSTU**

**Measuring Internal Product Attributes**

**1. PROPERTIES OF SOFTWARE SIZE**

* **Code Size**

**1.Counting Lines of Code to Measure Code Size :**Program code is an integral component of software. Such code includes source code, intermediate code, byte code, and even executable code. We look at approaches for directly measuring code size. We must take great care to clarify what we are counting and how we are counting it. In particular, we must explain how each of the following is handled:

* + - Count of the number of physical lines (including blank lines).
    - Count of all lines except blank lines and comments
    - Count of all statements except comments (statements taking more than one line count as only one line)
    - Count of all lines except blank lines, comments, declarations and headings
    - Count of all statements except blank lines, comments, declarations and headings
    - Count of only the ESs (Executable Statement), not including exception condition.

Total size (LOC) = NCLOC(Effective Lines of Code) + CLOC.

Measures the density of comments in a program.

**Measuring Technique:**

**Automated Program:** I will develop a program that will take a text file as an input and calculate its code size measure.

**2.Halstead’s Approach**

Maurice Halstead made an early attempt to capture notions of size and complexity beyond the counting of LOCs. Although his work has had a lasting impact, Halstead’s software science measures provide an example of confused and inadequate measurement, particularly when used to capture attributes other than size Halstead’s software science attempted to capture attributes of a program that paralleled physical and psychological measurements in other disciplines. He began by defining a program P as a collection of tokens, classified as either operators or operands.

For a given problem, let:

μ1 = Number of unique operators

μ2 = Number of unique operands

N1 = Total occurrences of operators

N2 = Total occurrences of operands

From these numbers, several measures can be calculated:

Program Vocabulary : μ = μ1+μ2

Program Length : N = N1+N2

Calculated The Estimated Program Length: L = μ1 \* log2 \* μ1 + μ2 \* log2 \* μ2

If anyone want to find volume , Difficulty or Effort

Volume : V= N \* Log2 \* μ1

Difficulty : D = μ1/2 \* N2/μ2

Effort : E = D \* μ1

**Example:**

Consider the following [C](https://en.wikipedia.org/wiki/C_(programming_language)) program:

main()

{

int a, b, c, avg;

scanf("%d %d %d", &a, &b, &c);

avg = (a+b+c)/3;

printf("avg = %d", avg);

}

The distinct operators () are:main,(),{},int,scanf,&,=, +, /, printf,,, ;

The distinct operands () are:a,b,c,avg,"%d %d %d",3,"avg = %d"

Now Apply the laws to calculated the measurement.

**Measuring Technique:**

**Automated Program:** I will develop a program that will take a text file as an input and calculate its code size measure.

**3.Alternative Code Size Measures**

To define size differently, we have two other alternatives to explore, both of which are acceptable on measurement theory grounds as ratio measures :

1. We can measure size in terms of the number of bytes of computer storage required for the program text. This approach has the advantage of being on the same scale as the normal measure of size for object code. It is at least as well understood as LOC, and it is very easy to collect.

2. We can measure size in terms of the number of characters (CHAR) in the program text, which is another easily collected measure. For example, most modern word processors compute this count routinely for any text file. (Both the UNIX and Linux operating systems have the command wc <filename> to compute it.)

* **Design Size**

We can measure the size of a design in a manner similar to that used to measure code size. We will **count design elements rather than LOCs.** The elements that we count depend on the abstractions used to express the design, and the design aspects of interest. Thus, the appropriate size measure depends on the design methodology, the artifacts developed, and the level of abstraction.

To measure the size of a **procedural design**, you can **count the number of procedures and functions** at the lowest level of abstraction. You can also measure the size of the procedure and function interfaces in terms of the **number of arguments**. Such measurements can be taken without code, for example, by analyzing the APIs of a system. At higher levels of abstraction,

you can count the number of packages and subsystems. You can measure the size of a package or subsystem in terms of the number functions and procedures in the package.

Object-oriented designs add new abstraction mechanisms: objects, classes, interfaces, operations, methods, associations, inheritance, etc. Object-oriented design can also include realizations of design patterns (Gamma et al. 1994). When quantifying size, our focus is generally on the static entities rather than the links between entities, or run time entities. Thus, we will measure size in terms of packages, design patterns, classes, interfaces, abstract classes, operations, and methods.

**Thus, we will measure size in terms of packages, design patterns, classes,interfaces, abstract classes, operations, and methods.**

* **Packages:** Number of subpackages, number of classes, interfaces (Java), or abstract classes (C++)
* **Design patterns:**
  + Number of different design patterns used in a design
  + Number of design pattern realizations for each pattern type
  + Number of classes, interfaces, or abstract classes that play roles in each pattern realization
* **Classes, interfaces, or abstract classes**: Number of public methods or operations, number of attributes.
* **Methods or operations**: Number of parameters, number of overloaded versions of a method or operation.

One of the earliest object-oriented design size measures was the weighted methods per class (WMC) measure (Chidamber and Kemerer 1994). As proposed, WMC is **measured by summing the weights of the methods in a class**, where weights are unspecified complexity factors for each method.

* With a weight of 1, WMC becomes a size measure he number of methods in a class.
* Both the number of methods and the number of attributes can serve as class size measures.

**Measuring Technique:**

**Manual Human Inspection:** For counting number of classes, packages, design pattern, method, interface etc I will do manual inspection.

* **Requirements Analysis And Specification Size**

Requirements and specification documents generally combine text, graphs, and special mathematical diagrams and symbols. The nature of the presentation depends on the particular style, method, or notation used. When measuring code or design size, you can identify atomic entities to count (lines, statements, bytes, classes, and methods, for example). However, a requirements or specification document can consist of a mixture of text and diagrams. For example, a use case analysis may consist of a UML use case diagram along with a set of use case scenarios that may be expressed as either text or as UML activity diagrams. Because a requirements analysis often consists of a mix of document types, it is difficult to generate a single size measure.

There are obvious atomic elements in a variety of requirements and specification model types that can be counted:

* Use case diagrams: Number of use cases, actors, and relationships of various types
* Use case: Number of scenarios, size of scenarios in terms of steps, or activity diagram model elements
* Domain model (expressed as a UML class diagram): Number of classes, abstract classes, interfaces, roles, operatons, and attributes
* Data-flow diagrams used in structured analysis and design: Processes (bubbles nodes), external entities (box nodes), data-stores (line nodes) and data-flows (arcs)

**Measuring Technique:**

**Manual Human Inspection:** For counting number of use case, data flow, ER diagram etc I will do manual inspection.

* **STRUCTURAL MEASURES**

**Structural attributes: complexity, length, coupling, and cohesion**

The control flow measures are usually modeled with directed graphs, where each

node (or point) corresponds to a program statement or basic block (code that always executes

sequentially), and each arc (or directed edge) indicates the flow of control from one statement or

basic block to another. We call these directed graphs control flow graphs or flow graphs.

**1. McCabe’s Cyclomatic Complexity Measure**

McCabe proposed the cyclomatic number of a program’s flowgraph as a measure of program complexity.

For a program with flowgraph F, the cyclomatic number is calculated as

1. V(F) = e − n + 2

2.V(F) = 1 + d

Here, e = edges.

n = nodes.

d = predicate nodes or decision nodes.

**Measuring Technique:**

**Automated Program:** For calculating V(F)I will develop a program that will take a text file or class or method as an input and calculate cyclomatic complexity. This is done by tracking the number of decision nodes where decision nodes are if...else, do...while, while, for loops.

**Manual Human Inspection:**(Requirement : Control Flow graph)

* **DESIGN-LEVEL ATTRIBUTES**

**1. Models of Modularity and Information Flow**

A module is a contiguous sequence of program statements, bounded by boundary elements, having an aggregate identifier (Yourdon and Constantine 1979). This deliberately vague definition permits a liberal interpretation. Also, a module should be (at least theoretically) separately compilable. Thus, a module can be any object that, at a given level of abstraction, you wish to view as a single construct.or example, in C, a procedure or function may be considered a module. In Java, a class or interface is considered a module. An individual Java or C++ method is considered a program unit, but not a module.

To describe intermodular attributes, we build models to capture the nec- essary information about the relationships between modules. This type of model describes the information flow between modules; that is, it explains which variables are passed between modules. When measuring some attributes, we need not know the fine details of a design, so our models suppress some of them. For example, instead of examining variables, we may need to know only whether or not one module calls (or depends on) another module. In this case, we use a more abstract model of the design, a directed graph known as the module call-graph; a call- graph is not a flowgraph, as it has no circled start or stop node.

**2. Morphology**

They use the notion of morphology to refer to the “shape” of the overall system structure when expressed pictorially. Morphological characteristics such as width and depth can then be used to describe good and bad designs.

Many morphological characteristics are measurable directly, including the following:

• **Size:** Measured as number of nodes, number of edges, or a combination of these.

• **Depth:** Measured as the length of the longest path from the root node to a leaf node.

• **Width:** Measured as the maximum number of nodes at any one level.

• **Edge-to-node ratio:** Can be considered a connectivity density measure, since it increases as we add more connections among nodes.

**Measuring Technique:**

**Automated Program:** For calculating MorphologyI will develop a program that will take a text file or class or method as an input and calculate Morphology Characteristic. This is done by tracking the number of decision nodes where decision nodes are if...else, do...while, while, for loops.

**Manual Human Inspection:**(Requirement : Control Flow structure).

**3.Tree Impurity**

Examining the trees in a graph tells us much about the design. Thus, we seek to create a measure, called tree impurity, to tell us how far a given graph deviates from being a tree. In what follows, we restrict our attention to undirected graphs.

To define tree impurity, we first describe several properties of graphs and trees. A tree with n nodes always has n − 1 edges. For every connected graph G, we can find at least one subgraph that is a tree built on exactly the same nodes as G; such a tree is called a spanning subtree.

We can define a measure of tree impurity that satisfies all four properties:

m(G)= (Number of edges more than the spanning tree/ Maximal number of edges more than the spanning tree)

The actual number of edges more than the spanning subtree must be e − n + 1, so our formal equation for the tree impurity measure is

m(G)=

**Measuring Technique:**

**Automated Program:** For calculating MorphologyI will develop a program that will take a text file or class or method as an input and calculate Morphology Characteristic. This is done by tracking the number of decision nodes where decision nodes are if...else, do...while, while, for loops.

**Manual Human Inspection:** (Requirement : Control Flow Graph).

**4.Internal Reuse**

Internal reuse the extent to which modules within a product are used multiple times within the same product.Yin and Winchester have proposed a simple measure of internal reuse, r, Calling the system design measure, it is defined by.

r(G) = e − n + 1

where G has e edges and n nodes. Thus, the design measure is equal to the number of edges additional to the spanning subtree of G. The reuse measure is crude; not only does it take no account of possible different calls from the same module; it also takes no account of the size of the reused components. However, it gives an idea of the general level of internal reuse in a system design.

**Measuring Technique: Manually**(Requirement : Control Flow structure).

**5.Information Flow**

The total level of information flow through a system, where the modules are viewed as the atomic components (an intermodular attribute). The total level of information flow between individual modules andthe rest of the system (an intramodular attribute). Information flow measure, a well known approach to measuring the total level of information flow between individual modules and the rest of a system. To understand the measurement, consider the way in which data move through a system. We say a local direct flow exists if either,

1. A module invokes a second module and passes information to it, or

2. The invoked module returns a result to the caller

A global flow exists if information flows from one module to another via a global data structure

Using these notions, we can describe two particular attributes of the information flow. The fan-in of a module M is the number of local flows that terminate at M, plus the number of data structures from which information is retrieved by M. Similarly, the fan-out of a module M is the number of local flows that emanate from M, plus the number of data structures that are updated by M. Based on these concepts, we measure information flow “complexity” as

**Information flow complexity(M) = length(M) × ((fan-in(M) × (fan-out(M))^2**

**Measuring Technique:**

**Manual Human Inspection**(Requirement : Flow structure).

* **OBJECT-ORIENTED STRUCTURAL ATTRIBUTES AND MEASURES**

**1. Measuring Coupling in Object-Oriented Systems**

Many object oriented connections tend to be persistent—the connected entities can remain connected between method invocations, and may persist over the lifetime of an object. Persistent connections include objects coupled through associations that are implemented using instance or state variables. Generalization/specialization relations that are implemented using inheritance are persistent. There are also persistent connections between language-defined types, user-defined types, and classes.

In addition to the coupling properties, several orthogonal coupling properties can help to evaluate coupling measures:

* Type: What kinds of entities are coupled?
* Strength: How many connections of a particular kind?
* Import or export: Are the connections import and/or export?
* Indirect: Is indirect coupling measured?
* Inheritance: Are connections to or from inherited entities counted?
* Domain: Are the measures used to indicate the coupling of individual attributes, methods, classes, sets of classes (e.g., packages), or the system as a whole

**Coupling in Object-Oriented Systems:** connections between elements *from one module to others*. Instability metric,

I = Ce / Ca + Ce.

Here,

Ce = Efferent coupling. [Fan-in],

Ca = Afferent coupling. [Fan-out].

**Measuring Technique:**

**Manual Human Inspection:** Counting number of inheritance relationship , import export files, strength of connection.

**2.Cohesion in Object-Oriented Systems** : connection between elements in a *individual module.*

TCC(C) = NDC(C)/NP(C)

LCC(C) = (NDC(C) + NIC(C))/NP(C)

RC(P) = (R(P) + 1)/N(P)

Here,

TCC = Tight Class Cohesion

LCC = Loose Class Cohesion

NDC = Number of Direct Cohesion

NIC = Number of Indirect Connection

NP = Number of Possible Connections

RC = Relational Cohesion

R(P) = number of relations between classes and interfaces

N(P) = number of classes and interfaces in the package

Depending on the level of abstraction of interest, a module may be a method, class, or package. Method cohesion is conceptually the same as the cohesion of an individual function or procedure. Class cohesion is an intraclass attribute. It reflects the degree to which the parts of a class—methods, method calls, fields, and attributes belong together. A class with high cohesion has parts that belong together because they contribute to a unified purpose. Most of the proposed cohesion metrics are class-level metrics.

**Measuring Technique :** Manual Human Inspection