Chapter5. Software Metrics

5.1 Introduction

- Software measurement is the mapping of symbols to objects. The purpose is to quantify some attribute of the objects, for example, to measure the size of software projects. Additionally, a purpose may be to predict some other attribute that is not currently measurable, such as effort needed to develop a software project.

5.1 Introduction

- An important concern is the validation of metrics. However, validation is related to the use of the metric. An example is a person's height. Height is useful for predicting the ability of a person to pass through a doorway without hitting his or her head. Just having a high correlation between a measure and an attribute is not sufficient to validate a measure. For example, a person's shoe size is highly correlated to the person's height. However, shoe size is normally not acceptable as a measure of a person's height.

- The following are criteria for valid metrics
  1. A metric must allow different entities to be distinguished.
  2. A metric must obey a representation condition.
  3. Each unit of the attribute must contribute an equivalent amount to the metric.
  4. Different entities can have the same attribute value.

5.1 Introduction

- Many times, the attribute of interest is not directly measurable. In this case, an indirect measure is used. An indirect measure involves a measure and a prediction formula.
- In computer science, many of the ‘‘ilities’’ (maintainability, readability, testability, quality, complexity, etc.) cannot be measured directly, and indirect measures for these attributes are the goal of many metrics programs.
- The following are criteria for valid indirect metrics:
  1. The model must be explicitly defined.
  2. The model must be dimensionally consistent.
  3. There should be no unexpected discontinuities.
  4. Units and scale types must be correct.

5.2 Software Measurement Theory

- The representational theory of measurement involves an empirical relation system, a numerical relation system, and a relation-preserving mapping between the two systems.
- The empirical relation system (E, R) consists of two parts:
  - A set of entities, E
  - A set of relationships, R
- The numerical relation system (N, P) also consists of two parts:
  - A set of entities, N. Also called the ‘‘answer set,’’ this set is usually numbers—natural numbers, integers, or reals.
  - A set of relations, P. This set usually already exists and is often ‘‘less than’’ or ‘‘less than or equal.’’

5.2 Software Measurement Theory

- The relation-preserving mapping, M, maps (E, R) to (N, P). The important restriction on this mapping is called the representation condition. The most restrictive version says that if two entities are related in either system, then the images (or pre-images) in the other system are related:

$$x \text{ rel } y \text{ iff } \mathrm{M}(x) \text{ rel } \mathrm{M}(y)^{3}$$

- The less restrictive version says that if two entities are related in the empirical system, then the images of those two entities in the numerical system are related in the same way:

$$\mathrm{M}(x) \text{ rel } \mathrm{M}(y) \text{ if } x \text{ rel } y$$

5.2 Software Measurement Theory

EXAMPLE 5.1 HEIGHT OF PEOPLE

The classic example of mapping an empirical system to a numerical system is the height of people. In the empirical system, there is a well-understood height relationship among people. Given two people who are standing next to each other, everyone would agree about who is taller. This is the empirical system: people are the entities and the well-understood relation is ‘‘shorter or the same height.’’

The numerical system is the real number system (either metric or imperial units) with the standard relation of less than or equal.

The mapping is just the standard measured height of people. This is usually measured barefoot, standing straight against a wall.

The representation condition (either version) is satisfied, since if Fred is shorter than or equal to Bill, then Fred’s measured height is less than or equal to Bill’s measured height.

5.2 Software Measurement Theory

EXAMPLE 5.2

Develop a measure, BIG, for people that combines both weight and height. Empirically, if two people are the same height, the heavier is bigger, and if two people are the same weight, the taller is bigger. If we use this notion, we can have a partial order that most people would agree with. The only pair of persons that we would not order by this would be if one was heavier and the other was taller. Numerically, we can use a tuple, < height, weight >. Each part of the tuple would be a real number. Two tuples would be related if both parts were related in the same direction. That is, if x; y are tuples, than x is less than or equal to y in ‘‘bigness’’ if x_{height} =< y_{height} and x_{weight} =< y_{weight}. This is also a partial order, and both versions of the representation condition are satisfied.

5.2 Software Measurement Theory

5.2.1 MONOTONICITY

- An important characteristic of a measure is ***monotonicity***. It means that the value of the measure of an attribute does not change direction as the attribute increases in the object.

EXAMPLE 5.3

A linear function is monotonic, since it always goes in the same direction. A quadratic function is usually not monotonic. For example, $y = 5x - x^2$ is not monotonic in the range $x = 0$ to $x = 10$. From $x = 0$ to $x = 5$, y increases. From $x = 5$ to $x = 10$, y decreases.

5.2 Software Measurement Theory

5.2.2 MEASUREMENT SCALES

- There are five different scale types: nominal, ordinal, interval, ratio, and absolute.
- The least restrictive measurement is using the **nominal** scale type. This type basically assigns numbers or symbols without regard to any quantity. The classic example for a nominal scale measure is the numbers on sports uniforms. We do not think that one player is better than another just because the number on one uniform is bigger or smaller than the number on the other uniform. There is no formula for converting from one nominal scale measure to another nominal scale measure.
- In an ***ordinal*** scale measure, there is an implied ordering of the entities by the numbers assigned to the entity. The classic example is class rank. However, we never assume that the numerical difference in rank is significant. If a student is ranked first, her performance has been better than a student who is ranked second, or third, or any other number greater than 1. However, we never assume that the numerical difference in rank is significant. That is, we don’t assume that the difference between the first and second student is the same as the difference between the 100th and 101st student. Any formula that converts from one ordinal scale measure to another ordinal scale measure for the same entity must preserve the ordering.

5.2 Software Measurement Theory

5.2.2 MEASUREMENT SCALES

- In an ***interval*** scale measure, the amount of the difference is constant. An example is temperature. There are two instances of temperature measures that are commonly used, Fahrenheit and Celsius. The formula for converting from a Celsius scale measure to a Fahrenheit scale measure is 9/5 * x + 32. With any two interval scale measures for the same attribute, the formula for conversion must be of the form a * x + b.
  Zero degree does not mean that there is no temperature.

  10 degree and 20 degree do not mean that 10 degree is 2 times hotter than 20 degree.
- In a ***ratio*** scale measure, the amount of the difference is constant and there is a well-understood zero that any scale measure would use. For example, money, length, and height are measurements using ratio scales. These measurements have well-understood notions of zero: zero money, zero height, and zero length. Any formula for converting from one set of units to another—from centimeters to inches, for example—would just use a multiplicative constant.
- The ***absolute*** is a counting scale measure. Counting marbles is an example of an absolute scale measure.

5.2 Software Measurement Theory

5.2.3 STATISTICS

- Not all statistics are appropriate for all scales. The following indicates which common statistical methods are appropriate:

  Nominal scale: Only mode, median, and percentiles

  Ordinal scale: The above and Spearman correlations

  Interval scale: The above and mean, standard deviation, and Pearson correlations

  Ratio scale: All statistics

  Absolute scale: All statistics

  EXAMPLE 5.4 AVERAGES

  Temperature is an interval scale measure. Thus, it makes statistical sense to give an average temperature. However, the numbers on baseball players' uniforms are a nominal scale measure. It does not make sense to give the average of the numbers on a team's uniforms. Similarly, the average ranking of the students in a class or the average of a student's rankings in a number of classes is not appropriate.

5.3 Product Metrics

- Product metrics are metrics that can be calculated from the document independent of how it was produced. Generally, these are concerned with the structure of the source code. Product metrics could be defined for other documents. For example, the number of paragraphs in a requirements specification would be a product metric.

EXAMPLE 5.5 LINES OF CODE

The most basic metric for size is the lines of code metric. There are many different ways to count lines of code. The definition may be a simple as the number of NEW LINE characters in the file. Often comments are excluded from the count of lines. Sometimes blank lines or lines with only delimiters are excluded. Sometimes statements are counted instead of lines.

5.3 Product Metrics

5.3.1 McCABE’S CYCLOMATIC NUMBER

- McCabe’s cyclomatic number, introduced in 1976, is, after lines of code, one of the most commonly used metrics in software development. Also called ‘‘McCabe’s complexity measure’’ from the title of the original journal article, it is based on graph theory’s cyclomatic number. McCabe tries to measure the complexity of a program. The premise is that complexity is related to the control flow of the program. Graph theory uses a formula, $C = e - n + 1$ to calculate the cyclomatic number. McCabe uses the slightly modified formula:

$$C = e - n + 2p$$

where:

e = Number of edges

n = Number of nodes

p = Number of strongly connected components (which is normally 1)

5.3 Product Metrics

EXAMPLE 5.6

Determine the cyclomatic number from the control flow graph shown in Fig. 5-1.

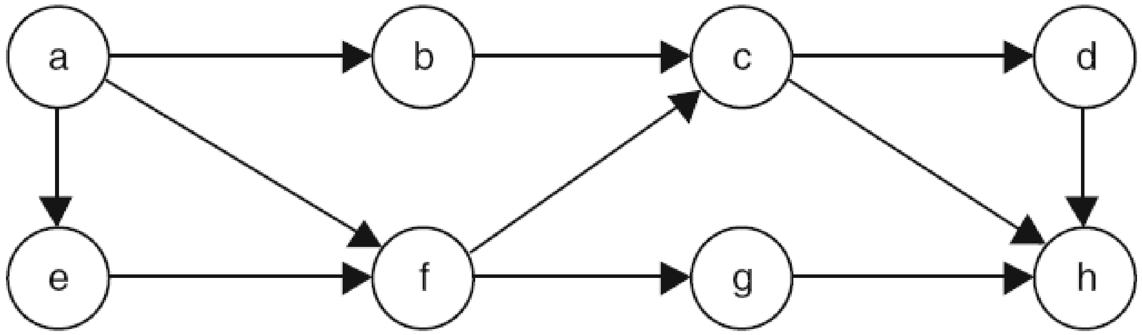

Fig. 5-1. Control flow graph.

There are 8 nodes, so n = 8. There are 11 arcs, so e = 11. The cyclomatic number is C = 11 ─ 8 + 2 = 5.

5.3 Product Metrics

- A ***planar graph*** is a graph that can be drawn without lines crossing. The Swiss mathematician Leonhard Euler (1707–1783) proved for planar graphs that $2 = n - e + r$, where r = number of regions, e = number of edges, and n = number of nodes. A region is an area enclosed (or defined) by arcs. Using algebra, this can be converted to $r = e - n + 2$. The number of regions on a planar graph equals the cyclomatic number.

EXAMPLE 5.7

Label the regions in the control flow graph from Example 5.6 with Roman numerals.

As shown in Fig. 5-2, there are five regions. Region I is the outside of the graph.

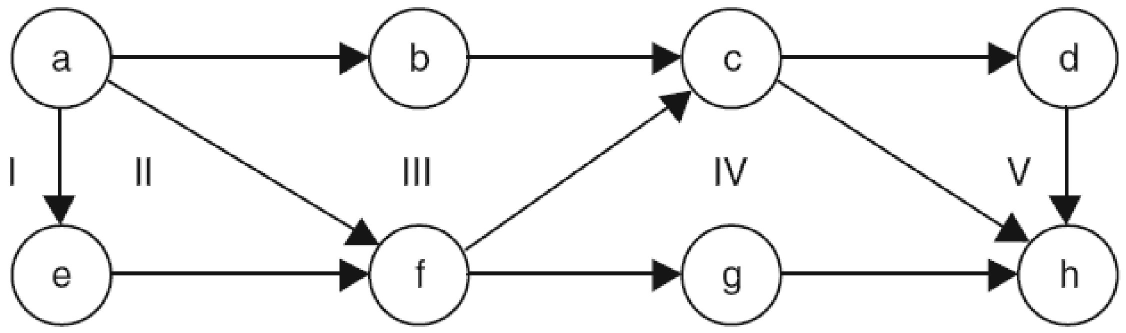

Fig. 5-2. Control flow graph with roman numerals.

5.3 Product Metrics

- Calculating the cyclomatic number from control flow graphs is time-consuming. Constructing a control flow graph from a large program would be prohibitively time-consuming. McCabe found a more direct method of calculating his measure. He found that the number of regions is usually equal to one more than the number of decisions in a program, $C = \pi + 1$, where π is the number of decisions.
- In source code, an IF statement, a WHILE loop, or a FOR loop is considered one decision. A CASE statement or other multiple branch is counted as one less decision than the number of possible branches.
- Control flow graphs are required to have a distinct starting node and a distinct stopping node. If this is violated, the number of decisions will not be one less than the number of regions.

5.3 Product Metrics

EXAMPLE 5.8

Label the decisions in the control flow graph of Example 5.6 with lowercase letters. As shown in Fig. 5-3, from node a, there are three arcs, so there must be two decisions, labeled a and b. From nodes c and f, there are two arcs and so one decision each. The other nodes have at most one exit and so no decisions. There are four decisions, so C = 4 + 1 = 5.

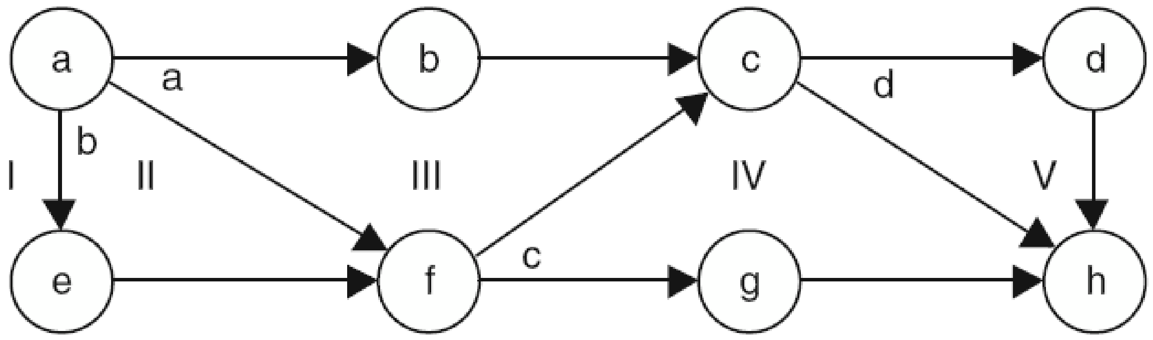

Fig. 5-3. Control flow graph with lowercase letters.

5.3 Product Metrics

EXAMPLE 5.9

Calculate the cyclomatic number using the invalid control flow graph shown in Fig. 5-4.

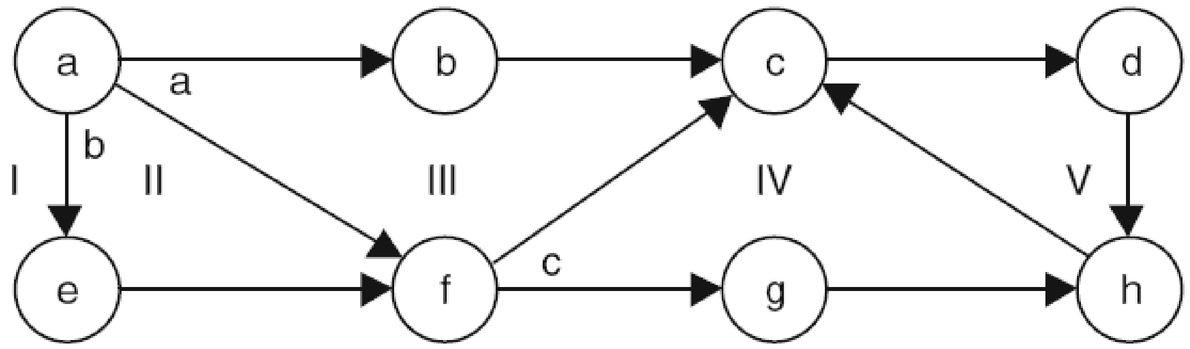

Fig. 5-4. Invalid control flow graph.

The cfg is the same as earlier examples, except that the c-h arc has been replaced by an h-c arc. This will not change the counts of nodes, edges, or regions. Thus, the first two methods of counting the cyclomatic number will not change. However, decision d has been eliminated, so the third method will not give the same answer. However, this is not a valid cfg, since there is now no stopping node.

5.3 Product Metrics

Threshold Value

- An important aspect of a metric is guidance about when the values are reasonable and when the values are not reasonable. McCabe analyzed a large project and discovered that for modules with cyclomatic number over 10, the modules had histories of many more errors and many more difficulties in maintenance. Thus, 10 has been accepted as the threshold value for the cyclomatic number in a module. If the cyclomatic number is greater than 10, efforts should be made to reduce the value or to split the module.

5.3 Product Metrics

5.3.2 HALSTEAD’S SOFTWARE SCIENCE

Basic Entities—Operators and Operands

- The basic approach that gave Halstead good results was to consider any program to be a collection of tokens, which he classified as either operators or operands. ***Operands*** were tokens that had a value. Typically, variables and constants were operands. Everything else was considered an operator. Thus, commas, parentheses, arithmetic operators, brackets, and so forth were all considered operators.
- Halstead also was concerned about algorithms and not about declarations, i/o statements, and so on. Thus, he did not count declarations, input or output statements, or comments.

5.3 Product Metrics

Basic Measures—η_1 and η_2

- The count of unique operators in a program is η_1 (pronounced ‘‘eta one’’), and the count of unique operands in a program is η_2 (pronounced ‘‘eta two’’). The total count of unique tokens is $\eta = \eta_1 + \eta_2$. This is the basic measure of the size of the program.

EXAMPLE 5.10

Identify the unique operators and operands in the following code that does multiplication by repeated addition.

```
Z = 0;
while X > 0
        Z = Z + Y ;
        X = X-1 ;
end-while ;
print(Z) ;
```

operators

```
= ; while-endwhile > + - print()
```

operands

```
Z 0 X Y 1
```

thus, $\eta_1 = 8$ and $\eta_2 = 5$

5.3 Product Metrics

Potential Operands, η_2^*

- Halstead wanted to consider and compare different implementations of algorithms. He developed the concept of potential operands that represents the minimal set of values needed for any implementation of the given algorithm. This is usually calculated by counting all the values that are not initially set within the algorithm. It will include values read in, parameters passed in, and global values accessed within the algorithm.

Length, N

- The next basic measure is total count of operators, N_1, and the total count of operands, N_2. These are summed to get the length of the program in tokens:

$$N = N_1 + N_2$$

5.3 Product Metrics

EXAMPLE 5.11

Calculate Halstead’s length for the code of Example 5.10.

operators

`=`	3
`;`	5
`while-endwhile`	1
`>`	1
`+`	1
`-`	1
`print`	1
`()`	1

operands

`Z`	4
`0`	2
`X`	3
`Y`	2
`1`	1

There are 14 occurrences of operators, so N_1 is 14. Similarly, N_2 is 12.

$N = N_1 + N_2 = 14 + 12 = 26.$

5.3 Product Metrics

Estimate of the Length (est N or N_hat)

- The estimate of length is the most basic of Halstead's prediction formulas. Based on just an estimate of the number of operators and operands that will be used in a program, this formula allows an estimate of the actual size of the program in terms of tokens:

$$\text{est N} = \eta_1 * \log_2 \eta_1 + \eta_2 * \log_2 \eta_2$$

EXAMPLE 5.12

Calculate the estimated length for the code of Example 5.10.

The $\log_2$ of x is the exponent to which 2 must be raised to give a result equal to x. So, $\log_2$ of 2 is 1, $\log_2$ of 4 is 2, of 8 is 3, of 16 is 4:

$$\log_2 \text{ of } \eta_1 = \log_2 8 = 3$$

$$\log_2 \text{ of } \eta_2 = \log_2 5 = 2.32$$

$$\text{est N} = 8_* 3 + 5_* 2.32 = 24 + 11.6 = 35.6$$

while the actual N is 26. This would be considered borderline. It is probably not a bad approximation for such a small program.

- From experience, I have found that if N and est N are not within about 30 percent of each other, it may not be reasonable to apply any of the other software science measures.

5.3 Product Metrics

Volume, V

- Halstead thought of volume as a 3D measure, when it is really related to the number of bits it would take to encode the program being measured.
  In other words:

$$V = N * \log_2(\eta_1 + \eta_2)$$

EXAMPLE 5.13

Calculate V for the code of Example 5.10.

$$V = 26 * \log_2 13 = 26 * 3.7 = 96.2$$

- The volume gives the number of bits necessary to encode that many different values. This number is hard to interpret.

5.3 Product Metrics

Potential Volume, V*

- The potential volume is the minimal size of a solution to the problem, solved in any language. Halstead assumes that in the minimal implementation, there would only be two operators: the name of the function and a grouping operator. The minimal number of operands is η^*_2 :

$$V^* = (2 + \eta^*_2)\log_2(2 + \eta^*_2)$$

Implementation Level, L

- Since we have the actual volume and the minimal volume, it is natural to take a ratio. Halstead divides the potential volume by the actual. This relates to how close the current implementation is to the minimal implementation as measured by the potential volume. The implementation level is unitless.

$$L = V^* / V$$

- The basic measures described so far are reasonable. Many of the ideas of operands and operators have been used in many other metric efforts. The remaining measures are given for historical interest and are not recommended as being useful or valid.

5.3 Product Metrics

Effort, E

- Halstead wanted to estimate how much time (effort) was needed to implement this algorithm. He used a notion of elementary mental discriminations (emd).

$$E = V / L$$

- The units are elementary mental discriminations (emd). Halstead's effort is not monotonic—in other words, there are programs such that if you add statements, the calculated effort decreases.

Time, T

- Next, Halstead wanted to estimate the time necessary to implement the algorithm. He used some work developed by a psychologist in the 1950s, John Stroud. Stroud had measured how fast a subject could view items passed rapidly in front of his face. S is the Stroud number (emd/sec) taken from those experiments. Halstead used 18 emd/sec as the value of S.

$$T = E / S$$

5.3 Product Metrics

5.3.3 HENRY–KAFURA INFORMATION FLOW

- Sallie Henry and Dennis Kafura developed a metric to measure the intermodule complexity of source code. The complexity is based on the flow of information into and out of a module. For each module, a count is made of all the information flows into the module, in_i, and all the information flows out of the module, out_i. These information flows include parameter passing, global variables, and inputs and outputs. They also use a measure of the size of each module as a multiplicative factor. LOC and complexity measures have been used as this weight.

$$HK_i = weight_i * (out_i * in_i)^2$$

The total measure is the sum of the HK_i from each module.

5.3 Product Metrics

EXAMPLE 5.14

Calculate the HK information flow metrics from the following information. Assume the weight of each module is 1.

mod #	a	b	c	d	e	f	g	h
in_i	4	3	1	5	2	5	6	1
out_i	3	3	4	3	4	4	2	6

mod	a	b	c	d	e	f	g	h
HK_i	144	81	16	225	64	400	144	36

HK for the whole program will be 1110.

5.4 Process Metrics

Productivity

- Productivity is one of the basic process metrics. It is calculated by dividing the total delivered source lines by the programmer-days attributed to the project. The units are normally LOC/programmer-day. In many projects in the 1960s the productivity was 1 LOC/programmer-day. In large projects, the typical productivity will range from 2 to 20 LOC/programmer-day. In small, individual projects, the productivity can be much higher.

EXAMPLE 5.15

The project totaled 100 KLOC. Twenty programmers worked on the project for a year. This year included the whole effort for the requirements, design, implementation, testing, and delivery phases. Assume that there are about 240 workdays in a year (20 days a month for 12 months, no vacations). The productivity is 100,000 LOC / (20 * 240 days) = 20.8 LOC/programmer-day.