Measuring Internal Product Attributes: Size

Basel Dudin

dudinbk@mcmaster.ca

Outline

- Aspects of Software Size
- Length (LOC)
  - Lines of Code
  - Halstead's Theory
- Functionality
  - Unadjusted Function Point Count (UFC)

Aspects of Software Size

- Size: One of the most useful attributes of a software product that can be measured without having to execute the system.
- Can be described by ***length***, ***functionality***, and ***complexity***:
  - *Length* is the physical product size (traditionally code length).
  - *Functionality* is a measure of the functions supplied by the product to the user.
  - *Complexity* is a multi-faceted attribute which can be interpreted in multiple ways.
- ***Reuse*** is also an issue in size, specifically the amount or size of reuse within a program.

Length

- Length is the "physical size" of the product.
- In a software development effort, there are three major development products: **specification**, **design**, and **code**.
- The length of the specification can indicate how long the design is likely to be, which in turn is a predictor of code length.
- Traditionally, code length refers to text-based code length.

Length: Lines of Code (LOC)

- The most commonly used measure of source code program length is the number of lines of code (LOC).
  - *NCLOC:* non-commented source line of code or effective lines of code (ELOC).
  - *CLOC:* commented source line of code.
- By measuring NCLOC and CLOC separately we can define:

total length (LOC) = NCLOC + CLOC

- The ratio: *CLOC/LOC* measures the density of comments in a program.

Length: Lines of Code (2)

Variations of LOC:

- Count 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 only executable statements, not including exception conditions.

Example 1

for (i = 0; i < 100; i++) printf("hello"); ***/* How many lines of code is this? */***

- 1 Physical Line of Code (LOC)
- 2 Logical Lines of Code (LLOC) (for and printf statements)
- 1 comment line

Example 2

- */* Now how many lines of code is this? */*
- for (i = 0; i < 100; i++)
- {
- printf("hello");
- }
- 5 Physical Lines of Code (LOC): is placing braces work to be estimated?
- 2 Line of Code (LLOC): what about all the work writing non-statement lines?
- 1 comment line: tools must account for all code and comments regardless of comment placement.

LOC: Pros and Cons

- Advantages of LOC
  - Simple and automatically measurable
  - Correlates with programming effort (& cost)
- Disadvantage of LOC
  - Vague definition
  - Language dependability
  - Not available for early planning
  - Developers' skill dependability
  - Encouraging "sumo" development!

Halstead's Theory

- A program *P* is a collection of tokens, composed
- of two basic elements: ***operands*** and ***operators***
- ***Operands*** are variables, constants, addresses (e.g.
- ***Operators*** are defined operations in a programming language (language constructs such as conditional, iterative and procedural statements)

Parameters in Halstead's Theory

- Number of distinct operators in the program ($\mu 1$)
- Number of distinct operands in the program ($\mu 2$)
- Total number of occurrences of operators in the program ($N1$)
- Total number of occurrences of operands in the program ($N2$)
- Program vocabulary (μ)

  $\mu = \mu 1 + \mu 2$

Parameters in Halstead's Theory

- Program length is the total number of occurrences of operators and operands:
- $N=N1+N2$
- Program volume is the number of mental comparisons needed to write a program of length N
- $V = N \log2 \mu = (N1 + N2) \log2 (\mu1 + \mu2)$
- Program level (L):
- $L=V^*/V$ or $L= 1/D = (2/\mu1)^* (\mu2/N2)$
- Where V^* is the minimum size potential volume (i.e., minimal size of implementation) and D is program difficulty

Parameters in Halstead's Theory

- Program length is the total number of occurrences of operators and operands:

$$N = N1 + N2$$

- Program estimated length (Ń)

$$\acute{N} = \mu 1 \log 2\, \mu 1 + \mu 2 \log 2\, \mu 2$$

- Effort required to generate program *P*: number of elementary discriminations

$$E = V \times D = V/L = (\mu 1 * N2)/(2 * \mu 2) * (N * \log 2\, \mu)$$

Parameters in Halstead's Theory

- Time required for developing program *P* is the total effort divided by the number of elementary discriminations per second

$$T=E/\beta$$

- In cognitive psychology β is usually a number between 5 and 20
- Halstead claims that $\beta=18$

Parameters in Halstead's Theory

- Remaining bugs: the number of bugs left in the software at the delivery time
- $B = E^{2/3} /3000$
- Conclusion: the bigger program needs more time to be developed and more bugs remained

Example

- For the following C program:

```
#include<stdio.h>

main()
{
int a ;

scanf (“%d”, &a);
if ( a >= 10 )
if ( a < 20 ) printf ("10 < a< 20 %d\n" , a);

else          printf ("a >= 20      %d\n" , a);

else          printf ("a <= 10       %d\n" , a);

}
```

Example

- Determine number of operators ($\mu 1$).
- Determine number of operands ($\mu 2$).
- Determine the program length in terms of the total number of occurrences of operators (*N1*) and operands (*N2*): $N = N1 + N2$
- Estimate program length

Example

Operators	Number of occurrences	Operators	Number of occurrences
#	1	<=	1
include	1	\n	3
stdio.h	1	printf	3
< … >	1	<	3
main	1	>=	2
(…)	7	if … else	2
{ … }	1	&	1
int	1	,	4
;	5	%d	4
scanf	1	" … "	4
$\mu_1 = 20$		$N_1 = 47$	

Example

Operands	Number of occurrences
a	10
10	3
20	3
$\mu_2 = 3$	$N_2 = 16$
$\mu_1 = 20$	$N_1 = 47$
Program length: $N = N_1 + N_2 = 63$	
Program Estimated length: $\hat{N} = \mu_1 \log_2 \mu_1 + \mu_2 \log_2 \mu_2 = 20 \log_2 20 + 3 \log_2 3 = 91.1934$	

Example 2

- Given the following code:

```
1: read x,y,z;
2: type = “scalene”;
3: if (x == y or x == z or y == z) type =“isosceles”;

4: if (x == y and x == z) type =“equilateral”;
5: if (x >= y+z or y >= x+z or z >= x+y) type =“not a triangle”;
6: if (x <= 0 or y <= 0 or z <= 0) type =“bad inputs”;

7: print type;
```

Example 2

- Calculate Halstead's
  - (c1) Number of operators;
  - (c2) Number of operands;
  - (c3) Program vocabulary;
  - (c4) Occurrences of operators in the program;
  - (c5) Occurrences of operands in the program;
  - (c6) Program length;
  - (c7) Program volume;
  - (c8) Program estimated length.

Example 2

Operators				Operands	
read	1	==	5	strings	5
,	2	or	6	x	9
;	7	and	1	y	8
“ ... ”	6	>=	3	z	8
=	5	<=	3	0	3
if	4	+	3	type	6
( )	4	print	1		

Example 2

- (c1) Number of distinct operators in the program: $\mu 1 = 14$
- (c2) Number of distinct operands in the program: $\mu 2 = 6$
- (c3) Program vocabulary: $\mu = \mu 1 + \mu 2 = 20$
- (c4) Total number of occurrences of operators in the program: N1 = 51
- (c5) Total number of occurrences of operands in the program: N2 = 39
- (c6) Program length: N = N1 + N2 = 90
  (c7) Program volume: V = N log2 μ = 90 log2 (20) = 388.9735
- (c8) Program estimated length: $\acute{N}=\mu 1 \log 2 \mu 1 + \mu 2 \log 2 \mu 2 = 14 \log 2 14 + 6 \log 2 6 = 68.81274$

Challenges with Halstead's Theory

- Developed in the context of assembly languages and too fine grained for modern programming languages.
- The treatment of basic and derived measures is somehow confusing.
- The notions of time to develop and remaining bugs are arguable.
- Unable to be extended to include the size for specification and design.

Functionality

- Depending on the programmer and/or coding standards, the "line of code" could be, and usually is, written on many separate lines

- Example:

```
for (i=0; i<100; ++i)
 {
printf("hello");

} /* Now how many lines of code is this? */
```

- 4 Physical Lines of Code (is placing braces worth to be estimated?)

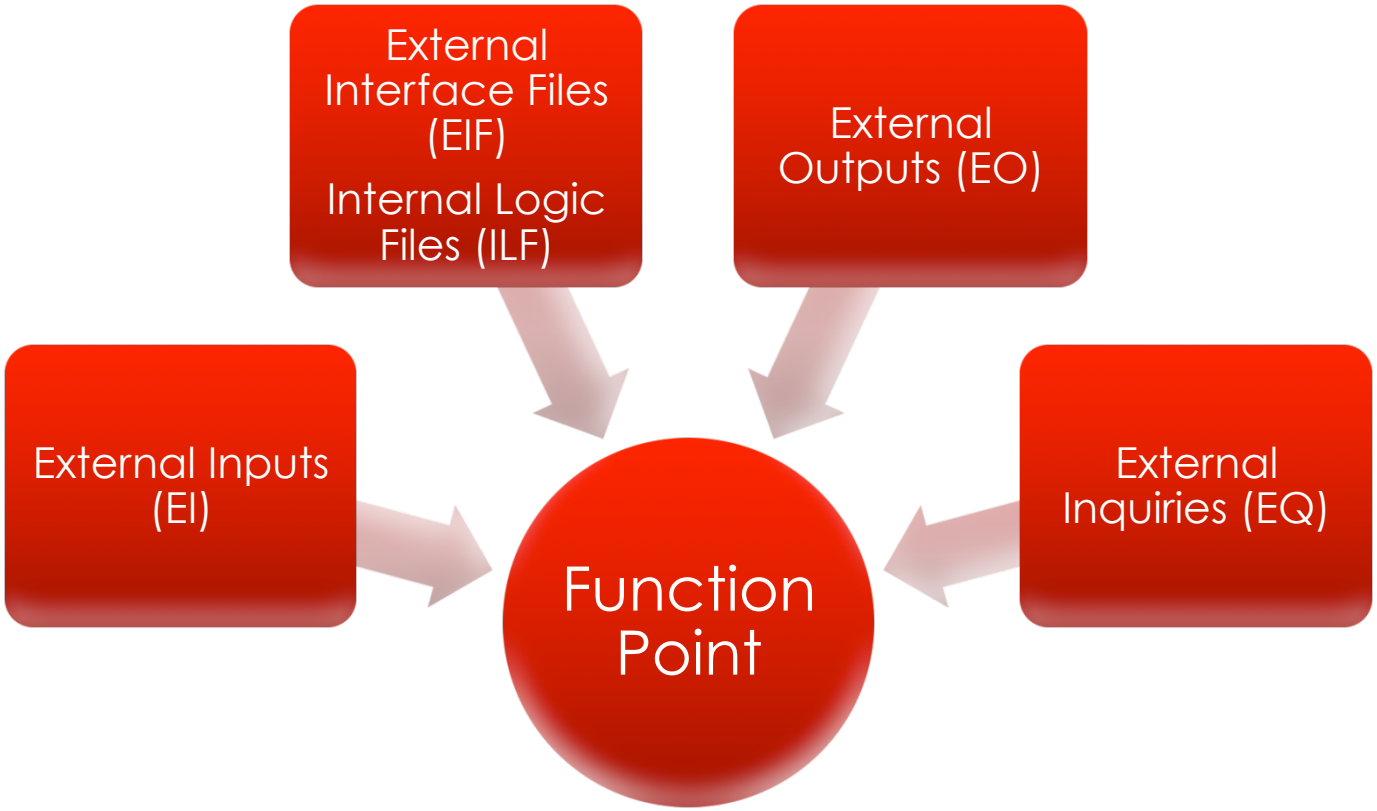
- 2 Logical Lines of Code (What about all the work writing non-statement lines?)

- 1 Comment Line (?)

Functionality

- **Function Point (FP) is a weighted measure of software functionality.**
- The idea is that a product with more functionality will be larger in size.
- Function-oriented metrics are indirect measures of software which focus on functionality and utility.
- The first function-oriented metrics was proposed by Albrecht (1979~1983) who suggested a productivity measurement approach called the **Function Point (FP)** method.
- Function points (FPs) measure the amount of functionality in a system based upon the *system specification*.

Function Points



Functions

- Function Point (FP) is a weighted measure of software functionality.
- FP is computed in two steps:

1) Calculating ***Unadjusted Function point Count (UFC)***.

2) Multiplying the UFC by a ***Value Adjustment Factor (VAF)***

- The final (adjusted) Function Point is:

$$FP = UFC \times VAF$$

External Inputs (EI)

- ***External Inputs – IFPUG Definition*:**
  - An external input (EI) is an elementary process that processes data or control information that comes from outside the application boundary
  - The primary intent of an EI is to maintain one or more ILFs and/or to alter the behavior of the system
  - **Example:**
    Data entry by users
    Data or file feeds by external applications

External Outputs (EO)

- ***External Outputs – IFPUG Definition:***
  - An external output(EO) is an elementary process that sends data or control information outside the application boundary
  - The primary intent of an external output is to present information to a user through processing logic other than, or in addition to, the retrieval of data or control information
  - The processing logic must contain at least one mathematical formula or calculation, create derived data, maintain one or more ILFs, or alter the behavior of the system

- **Example:**
- Reports created by the application being counted, where the reports include derived information

External Inquiries (EQ)

- ***External Inquiries – IFPUG Definition:***
- An external inquiry (EQ) is an elementary process that sends data or control information outside the application boundary
- The primary intent of an external inquiry is to present information to a user through the retrieval of data or control information from an ILF or EIF
- The processing logic contains no mathematical formulas or calculations, and creates no derived data
- No ILF is maintained during the processing, nor is the behavior of the system altered
- **Example:**
  Reports created by the application being counted, where the report does not include any derived data

Internal Logic Files (ILF)

- ***Internal Logical Files – IFPUG Definition*:**
  - An ILF is a user-identifiable group of logically related data or control information maintained within the boundary of the application
  - The primary intent of an ILF is to hold data maintained through one or more elementary processes of the application being counted
- Examples:
  - Tables in a relational database
  - Files

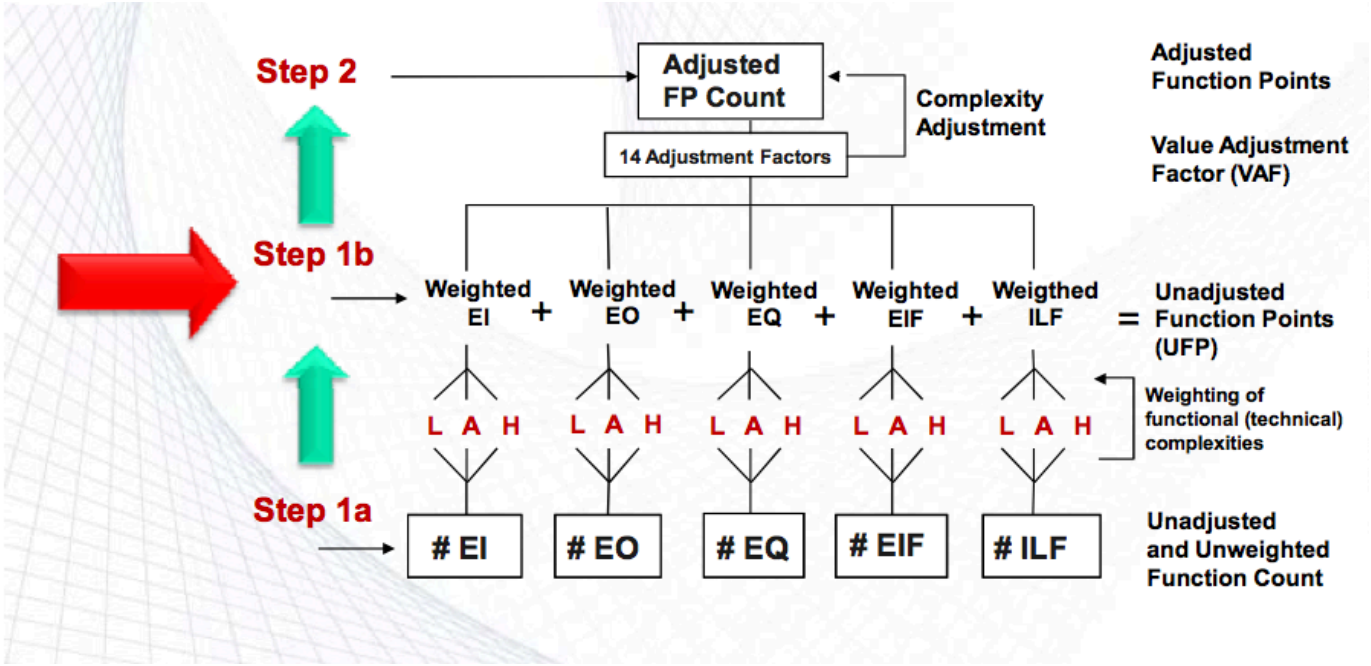
External Interface Files (EIF)

- ***External Interface files – IFPUG Definition*:**
  - An external interface file (EIF) is a user identifiable group of logically related data or control information referenced by the application, but maintained within the boundary of another application
  - The primary intent of an EIF is to hold data referenced through one or more elementary processes within the boundary of the application counted
  - This means an EIF counted for an application must be in an ILF in another application

Example:

- As for ILF, but maintained in a different system

Function Point Counting



Source: B.H. Far – University of Calgary

Unadjusted FP Count (UFC)

Element	Low (Simple)	Average	High (Complex)
External Inputs (N_{EI})	3	4	6
External Outputs (N_{EO})	4	5	7
External Inquiries (N_{EQ})	3	4	6
External Interface Files (N_{EIF})	5	7	10
Internal Logic Files (N_{ILF})	7	10	15

Unadjusted Function Count (UFC)

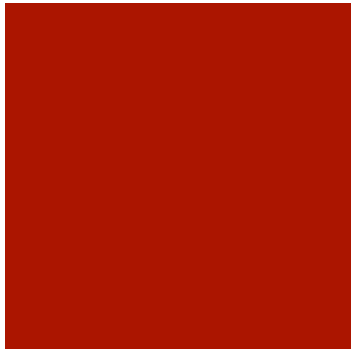
- Defined as a complexity rating that is associated with each of the defined counts according to function point complexity weights illustrated on previous table

$$\mathbf{UFC = 4N_{EI}+5N_{EO}+4N_{EQ}+7N_{EIF}+10N_{ILF}}$$

Weighted Technical Complexity

Element	Low	Average	High	Sum
External Inputs (N_{EI})	____X3	____X4	____X6	
External Outputs (N_{EO})	____X4	____X5	____X7	
External Inquiries (N_{EQ})	____X3	____X4	____X6	
External Interface Files (N_{EIF})	____X5	____X7	____X10	
Internal Logic Files (N_{ILF})	____X7	____X10	____X15	
	Unadjusted Function Points (UFP)			

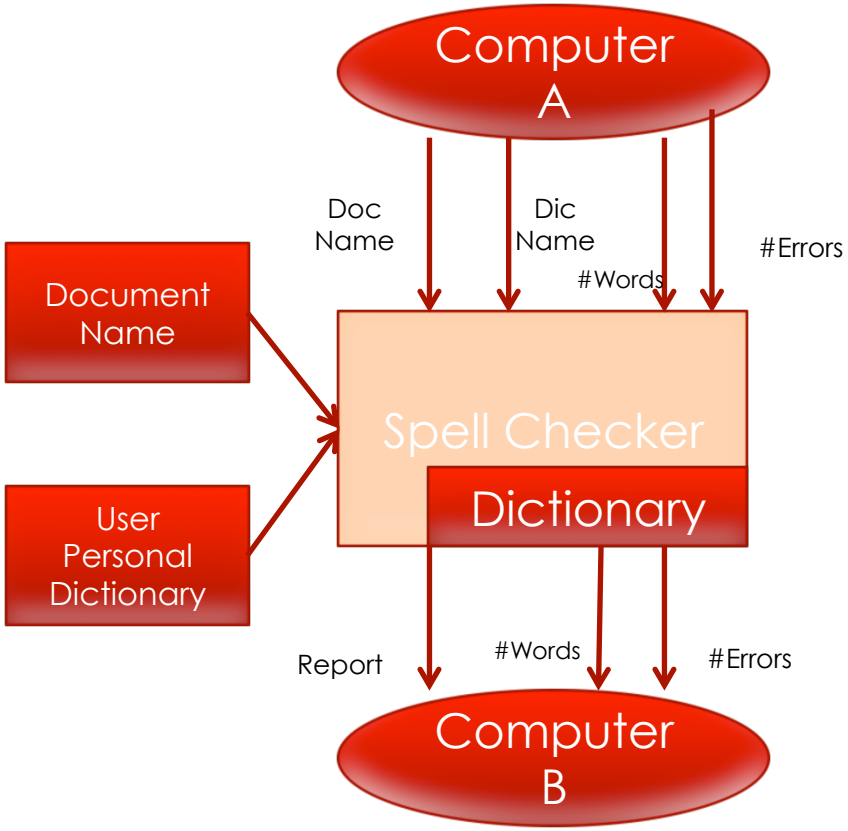
Example – Spell Checker



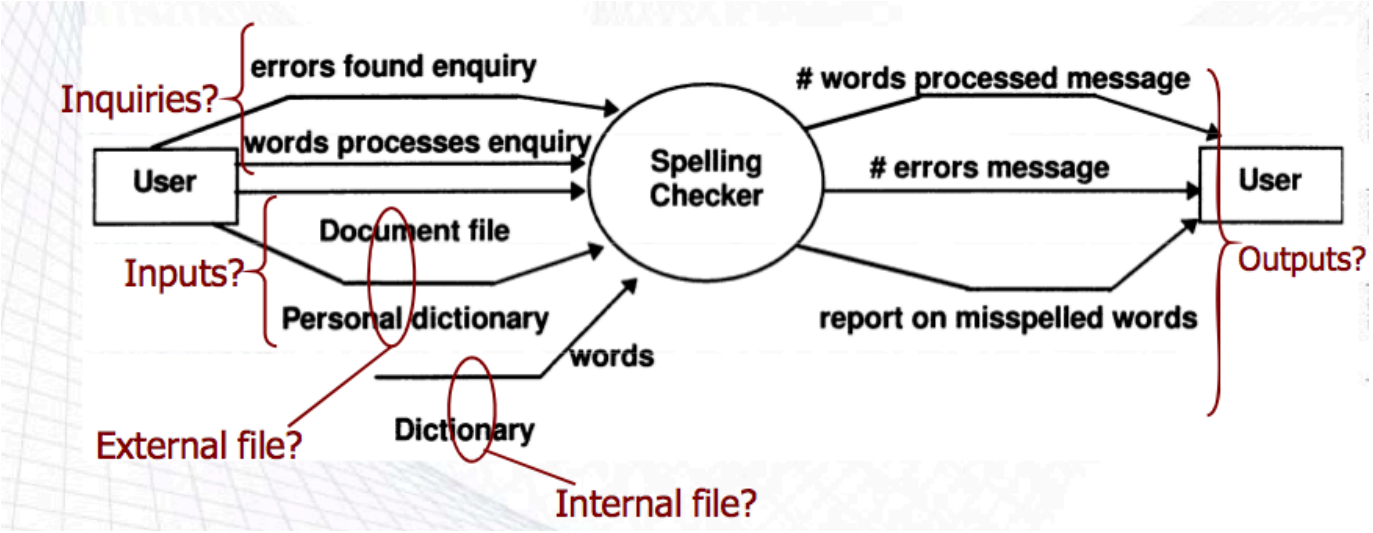
Specification – Spell Checker:

- Checks all words in a document by comparing them to a list of words in the internal dictionary and an optional user-defined dictionary
- After processing the document sends a report on all misspelled words to standard output
- On request from user shows number of words processed on standard output
- On request from user shows number of spelling errors detected on standard output
- Requests can be issued at any point in time while processing the document file

Spell Checker Model



Context Diagram



Solution

- EI: Doc. name + User Dic. name → 2
- EO: Report + #Words + #Errors → 3
- EQ: --
- EIF: Document + User Dictionary → 2
- ILF: Dictionary → 1

Solution

Element	Low (Simple)	Average	High (Complex)
External Inputs (N_{EI})	3	4	6
External Outputs (N_{EO})	4	5	7
External Inquiries (N_{EQ})	3	4	6
External Interface Files (N_{EIF})	5	7	10
Internal Logic Files (N_{ILF})	7	10	15

$N_{EI} = 2$

$N_{EO} = 3$

$N_{EQ} = 0$

$N_{EIF} = 3$

$N_{ILF} = 1$

QUESTIONS?