Measuring Internal Product Attributes

(Size)

Software Size

- Size measures only indicate how much of an entity we have.
- Size alone cannot directly indicate external attributes such as effort, productivity, and cost
- It doesn’t measure the external attributes like “coding difficulties”, however, it is a good measure to predict software development time and resources. e.g.,
  - Productivity=size/effort
  - Defect density = Defect count/size

Properties of Valid Software Size Measurement

- Three properties for any valid measure of software size:
  - Nonnegativity: All systems have nonnegative size.
  - Null value: The size of a system with no elements is zero.
  - Additivity: The size of the union of two modules is the sum of the sizes of the two modules after subtracting the size of the intersection

Code Size

- The most commonly used measure of source code size is the number of lines of code (LOCs).
- We must explain how each of the following is handled:
  - Blank lines
  - Comment lines
  - Data declarations
  - Lines that contain several separate instructions
- One count can be as much as five times larger than another, simply because of the difference in counting technique

LOC Definitions

- NCLOC: Non-Commented Lines of Code (Comments and blank lines removed), sometimes called **effective lines of a code**.
- CLOC: Number of comment lines of program text (CLOC)
- Total size (LOC) = NCLOC + CLOC
- Density of comments = CLOC / LOC
- Is NCLOC a valid measure?
- The number of executable statements (ES): counts separate statements on the same physical line as distinct. It ignores comment lines, data declarations, and headings

Halstead's Approach

- 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. The basic metrics for these tokens are the following:
    - μ_1 = Number of unique operators
    - μ_2 = Number of unique operands
    - N_1 = Total occurrences of operators
    - N_2 = Total occurrences of operands
- In addition to the above, Halstead defines the following :
- μ_1^* = Number of potential operators.
  μ_2^* = Number of potential operands.

Halstead's Approach

- **Halstead Program Length:** The total number of operator occurrences and the total number of operand occurrences. N = $N_1 + N_2$

  Estimated program length is, $N^\wedge = \mu_1 \log_2 \mu_1 + \mu_2 \log_2 \mu_2$
- **Halstead Vocabulary:** $\mu = \mu_1 + \mu_2$
- **Program Volume:** V = Size * ($\log_2$ vocabulary) = N * $\log_2(\mu)$
- **Program Difficulty:** D = $(\mu_1 / 2) * (N_2 / \mu_2)$
- **Programming Effort:** E = D * V = Difficulty * Volume

Halstead's Approach

For example, the FORTRAN statement

$$A(I) = A(J)$$

has one operator (=) and two operands ($A(I)$ and $A(J)$).

The *length* of P is defined to be $N = N_1 + N_2$, while the *vocabulary* of P is $\mu_1 = \mu_2 + \mu_3$. The *volume* of a program, akin to the number of mental comparisons needed to write a program of length N or the minimum number of bits to represent a program, is

$$V = N \times \log_2 \mu$$

Halstead derived measures for a number of other attributes including *program level*, *difficulty*, and *effort*. According to Halstead, one can use these metrics and an estimate of the number of mental discriminations per second to compute the time required to write a program.

Alternative Size Measures

- We can measure the size in terms of
  - Number of bytes of computer storage
  - Number of characters in the program text

If α is the average number of characters per line of program text, then we have the rescaling

$$\text{CHAR} = \alpha\ \text{LOC}$$

which expresses a stochastic relationship between LOC and CHAR. Similarly, we can use any constant multiple of the proposed measures as an alternative valid size measure. Thus, we can use KLOC (thousands of LOCs) or KDSI (thousands of delivered source instructions) to measure program size.

Design Size

- Count design elements rather than LOCs
- 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
- 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 of functions and procedures in the package.

Design Size

- Object-oriented designs add new abstraction mechanisms: objects, classes, interfaces, operations, methods, associations, inheritance, etc.
- Thus, we will measure the size in terms of packages, design patterns, classes, interfaces, abstract classes, operations, and methods.
  - ***Packages*****:** Number of sub packages, 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

Design Size

- **Weighted Methods per Class (WMC) measure:** measured by summing the weights of the methods in a class, where weights are unspecified complexity factors for each method
- Both the number of methods and the number of attributes can serve as class size measures
- One set of studies found that the number of methods is a better predictor of class change-proneness than the number of attributes

Requirement Analysis and Specification Size

- Requirement and specification documents generally combine text, graphs, and special mathematical diagrams and symbols.
- It may be difficult to generate a single-size measure because a requirement analysis often consists of a mix of document types.
- 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

Requirement Analysis and Specification Size

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

i. *Use case diagrams*: Number of use cases, actors, and relationships of various types

ii. *Use case*: Number of scenarios, size of scenarios in terms of steps, or activity diagram model elements

iii. *Domain model (expressed as a UML class diagram)*: Number of classes, abstract classes, interfaces, roles, operatons, and attributes

iv. *UML OCL specifications*: Number of OCL expressions, OCL clauses

v. *Alloy models*: Number of alloy statements—signatures, facts, predicates, functions, and assertions (Jackson 2002)

vi. *Data-flow diagrams used in structured analysis and design*: Processes (bubbles nodes), external entities (box nodes), data-stores (line nodes) and data-flows (arcs)

Cost of a Project

- The cost of a project is due to:
  - The requirements for software, hardware, and human resources
  - The cost of software development is due to the human resources needed
  - Most cost estimates are measured in ***person-months (PM)***

Software Cost Estimation

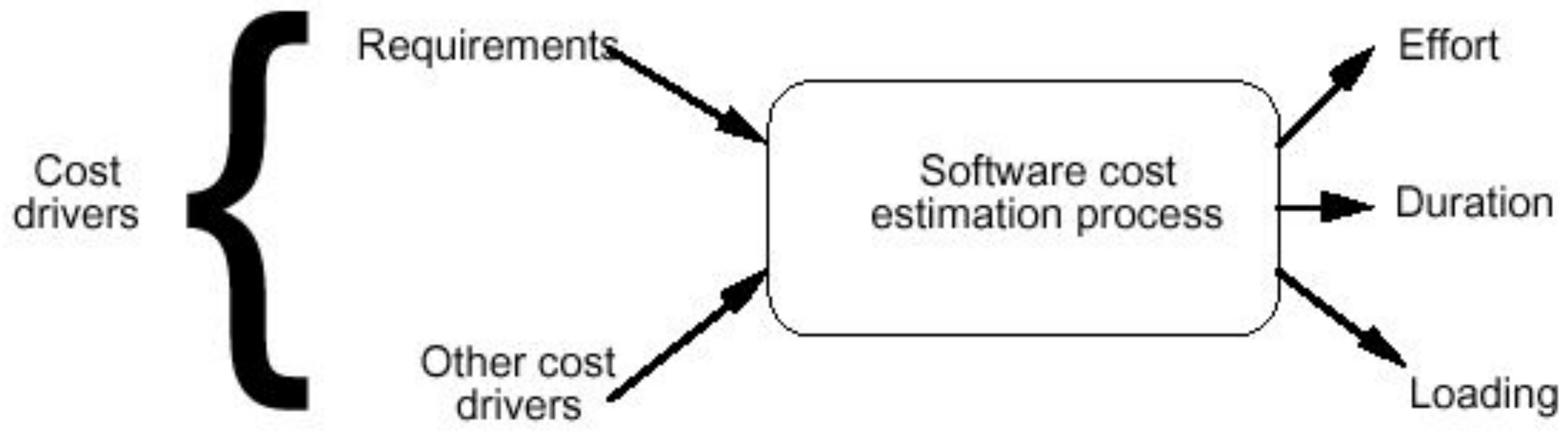
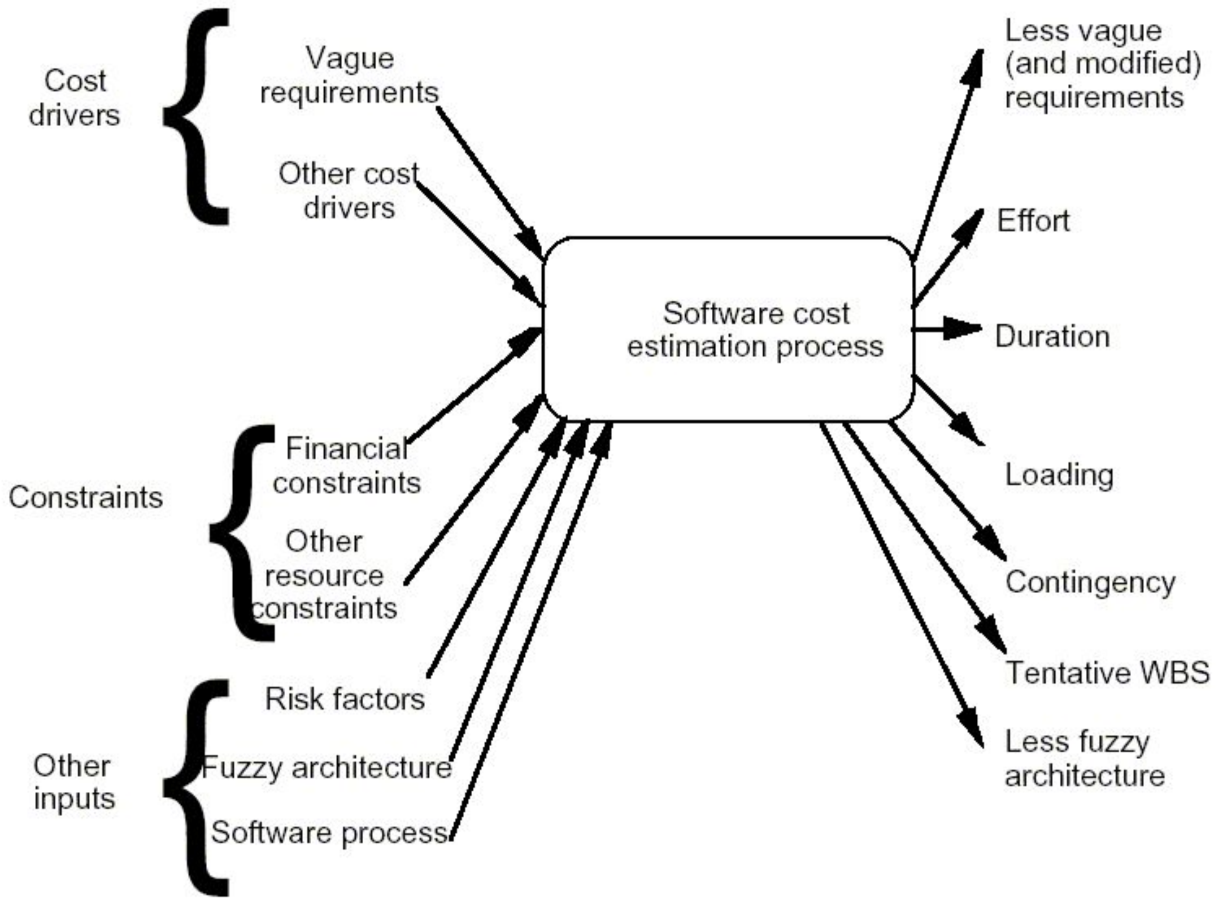


Figure 1. Classical view of software estimation process.

Actual View


Cost drivers
Vague requirements
Other cost drivers
Constraints
Financial constraints
Other resource constraints
Other inputs
Risk factors
Fuzzy architecture
Software process
Software cost estimation process
Less vague (and modified) requirements
Effort
Duration
Loading
Contingency
Tentative WBS
Less fuzzy architecture

Effort (using LOC)

- Effort Equation
  - **PM = C * (KDSI)^n** (person-months)
    - where **PM** = number of person-month (=152 working hours),
    - **C** = a constant,
    - **KDSI** = thousands of "delivered source instructions" (DSI) and
    - **n** = a constant.

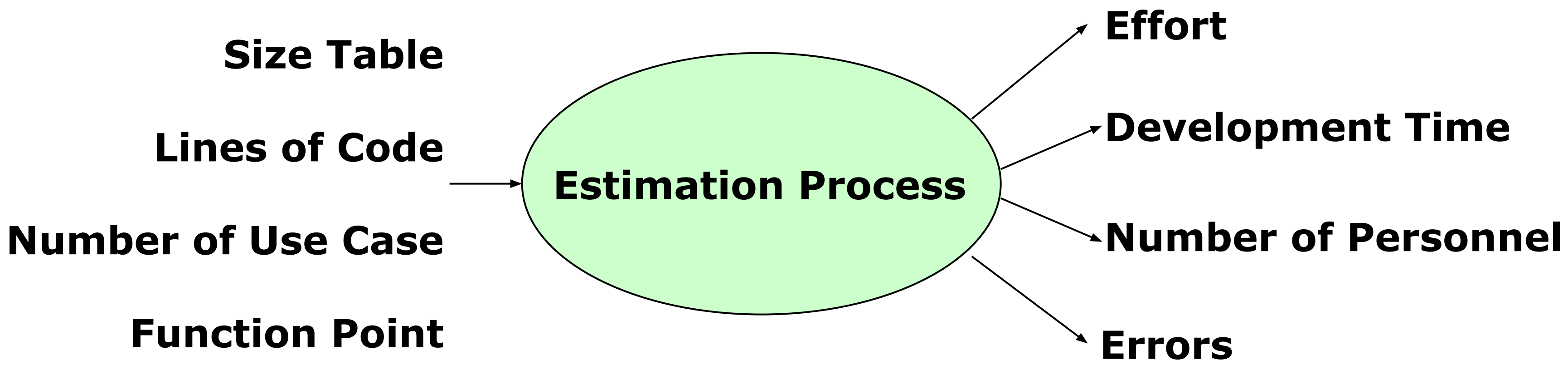
Productivity

- Productivity equation
  - **(DSI) / (PM)**
    - where **PM** = number of person-month (=152 working hours),
    - **DSI** = "delivered source instructions"

Schedule and Average Staffing

- Schedule equation
  - **TDEV = C * $(PM)^n$ (**months)
    - where TDEV = number of months estimated for software development.
- Average Staffing Equation
  - **(PM) / (TDEV)** (FSP)
    - where FSP means Full-time-equivalent Software Personnel.

Cost Estimation Process



Project Size - Metrics

1. **Number of functional requirements**
2. **Cumulative number of functional and non-functional requirements**
3. **Number of Customer Test Cases**
4. **Number of ‘typical sized’ use cases**
5. **Number of inquiries**
6. **Number of files accessed (external, internal, master)**
7. **Total number of components (subsystems, modules, procedures, routines, classes, methods)**
8. **Total number of interfaces**
9. **Number of System Integration Test Cases**
10. **Number of input and output parameters (summed over each interface)**
11. **Number of Designer Unit Test Cases**
12. **Number of decisions (if, case statements) summed over each routine or method**
13. **Lines of Code, summed over each routine or method**

Project Size - Metrics

<u>Availability of Size Estimation Metrics:</u>

	Development Phase	Available Metrics
a	Requirements Gathering	1, 2, 3
b	Requirements Analysis	4, 5
d	High Level Design	6, 7, 8, 9
e	Detailed Design	10, 11, 12
f	Implementation	12, 13

Function Points

- A function point calculates software size with the help of logical design and performance of functions as per user requirements.
- Function points are a unit of measure used to define the value that the end user derives, or the functional business requirements the software is designed to accomplish
- Function Point (FP) is **an element of software development that helps to approximate the cost of development early in the process**

Function Points Calculation

STEP 1: Measure size in terms of the amount of functionality in a system. Function points are computed by first calculating an ***unadjusted function point count*** **(UFC).** Counts are made for the following categories

- *External inputs* – those items provided by the user that describe distinct application-oriented data (such as file names and menu selections)
- *External outputs* – those items provided to the user that generate distinct application-oriented data (such as reports and messages, rather than the individual components of these)
- *External inquiries* – interactive inputs requiring a response
- *External files* – machine-readable interfaces to other systems
- *Internal files* – logical master files in the system

Function Points

- **<u>STEP 2</u>:** Multiply each number by a weight factor, according to complexity (**simple**, **average** or **complex**) of the parameter, associated with that number. The value is given by a table:

Parameter	simple	average	complex
users inputs	3	4	6
users outputs	4	5	7
users requests	3	4	6
files	7	10	15
external interfaces	5	7	10

Function Points

- **STEP 3:** Calculate the total **UFP** (Unadjusted Function Points)
- **STEP 4:** Calculate the total **TCF** (Technical Complexity Factor) by giving a value between 0 and 5 according to the importance of the following points (next slide):

TCF=0.65+0.01*DI

Function Points

- **Technical Complexity Factors:**

1. Data Communication
2. Distributed Functions
3. Performance
4. Heavily Utilized Hardware
5. High Transaction Rates
6. Online Data Entry
7. Online Updating
8. End-user Efficiency
9. Complex Computations
10. Reusability
11. Ease of Installation
12. Ease of Operation
13. Portability
14. Maintainability/Facility change

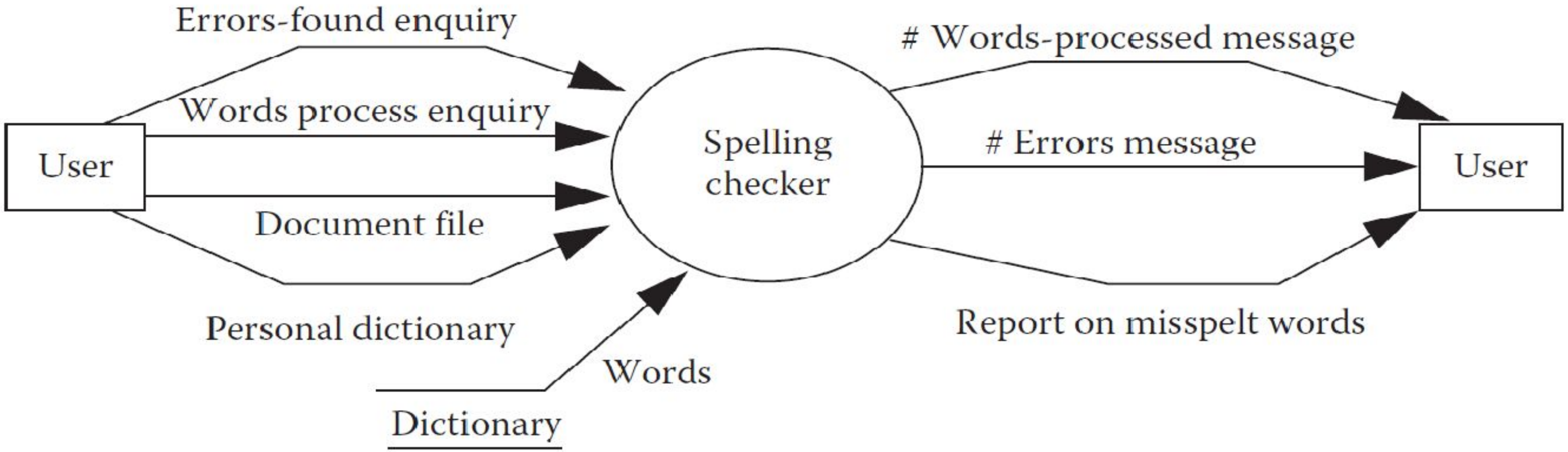
Each component or subfactor is rated from 0 to 5, where 0 means the subfactor is irrelevant, 3 means it is average, and 5 means it is essential to the system being built

Function Points

- **STEP 5:** Sum the resulting numbers to obtain **DI** (degree of influence)
- **STEP 6: TCF** (Technical Complexity Factor) by given by the formula
  - **TCF=0.65+0.01*DI**
- **STEP 6:** Function Points are given by the formula
  - **FP=UFP*TCF**

Function Points Example

Spell-checker spec: The checker accepts as input a document file and an optional personal dictionary file. The checker lists all words not contained in either of these files. The user can query the number of words processed and the number of spelling errors found at any stage during processing.



A = # external inputs = 2, B = # external outputs = 3, C = # inquiries = 2,
D = # external files = 2, and E = # internal files = 1

Function Points Example

- The two external inputs are: document file-name, personal dictionary-name.
- The three external outputs are: misspelled word report, number-of-words-processed message, number-of-errors-so-far message.
- The two external inquiries are: words processed, errors so far.
- The two external files are: document file, personal dictionary.
- The one internal file is: dictionary.

Function Points Example

TABLE 8.2 Function Point Complexity Weights

Item	Simple	Weighting Factor Average	Complex
External inputs	3	4	6
External outputs	4	5	7
External inquiries	3	4	6
External files	7	10	15
Internal files	5	7	10

Function Points Example

- 2 users inputs: document file name, personal dictionary name (average)
- 3 users outputs: fault report, word count, misspelled error count (average)
- 2 users requests: #treated words?, #found errors? (average)
- 1 internal file: dictionary (average)
- 2 external files: document file, personal dictionary (av).

$$UFP = 4 \times 2 + 5 \times 3 + 4 \times 2 + 10 \times 1 + 7 \times 2 = 55$$

Function Points Example

Technical Complexity Factors:

- *F*1. Data Communication
- *F*2. Distributed Data Processing
- *F*3. Performance Criteria
- *F*4. Heavily Utilized Hardware
- *F*5. High Transaction Rates
- *F*6. Online Data Entry
- F7. Online Updating
- *F*8. End-user Efficiency
- *F*9. Complex Computations
- *F*10. Reusability
- *F*11. Ease of Installation
- *F*12. Ease of Operation
- *F*13. Portability
- F14. Maintainability

- Each component or subfactor is rated from 0 to 5, where 0 means the subfactor is irrelevant, 3 means it is average, and 5 means it is essential to the system being built.
- Sum the resulting numbers to obtain **DI (degree of influence)**

Function Points Example

- The following formula combines the 14 ratings into a final technical complexity factor

$$\mathrm{TCF} = 0.65 + 0.01 \sum_{i=1}^{14} F_i$$

i.e.,

$$\mathrm{TCF} = 0.65 + 0.01 * \mathrm{DI}$$

- The final calculation of FPs multiplies the UFC by the technical complexity factor TCF

$$\mathrm{FP} = \mathrm{UFC} \times \mathrm{TCF}$$

Function Points Example

- Function Points
  - FP=UFP*(0.65+0.01*DI)= 55*(0.65+0.01*30)=52.25
  - That means **FP=52.25**
- ** *considering* DI *or* TCF *= 30*

Relation between LOC and FP

- Relationship:
  - ***LOC = Language Factor * FP***
  - where
    - **LOC** (Lines of Code)
    - **FP** (Function Points)

Relation between LOC and FPs

Language	LOC/FP
assembly	320
C	128
Cobol	105
Fortan	105
Pascal	90
Ada	70
OO languages	30
4GL languages	20

Relation between LOC and FP

Assuming LOC's per FP for:

Java = 53,

C++ = 64

aKLOC = FP * LOC_per_FP / 1000

It means for the SpellChekcer Example: (Java)

LOC=52.25*53=2769.25 LOC or 2.76 KLOC

IFPUG

- International Function Point User Group meets regularly to discuss FPs and their applications, and they publish guidelines with counting rules.
- Please visit https://ifpug.org/

COCOMO I

- COCOMO has three different models (each one increasing with detail and accuracy):
  - **Basic**, applied early in a project
  - **Intermediate**, applied after requirements are specified.
  - **Advanced**, applied after design is complete
- COCOMO has three different modes:
  - **Organic** – "relatively small software teams develop software in a highly familiar, in-house environment" [Bohem]
  - **Embedded** – operate within tight constraints, product is strongly tied to "complex of hardware, software, regulations, and operational procedures" [Bohem]
  - **Semi-detached** – intermediate stage somewhere between organic and embedded. Usually up to 300 KDSI

COCOMO I

- COCOMO uses two equations to calculate effort in man months (MM) and the number on months estimated for project (TDEV)
- MM is based on the number of thousand lines of delivered instructions/source (KDSI)

$$MM = a(KDSI)^b * EAF$$

$$TDEV = c(MM)^d$$

- EAF is the Effort Adjustment Factor derived from the Cost Drivers, EAF for the basic model is 1
- The values for a, b, c, and d differ depending on which mode you are using

Mode	a	b	c	d
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

COCOMO I

- A simple example:

Project is a flight control system (mission critical) with 310,000 DSI in embedded mode

- Reliability must be very high (RELY=1.40). So we can calculate:
- **Effort** = $1.40*3.6*(310)^{1.20}$ = 5093 MM
- **Schedule** = $2.5*(5093)^{0.32}$ = 38.4 months
- **Average Staffing** = 5093 MM/38.4 months = 133 FSP