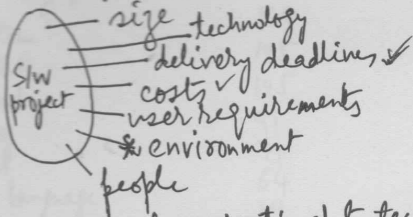


SW Project



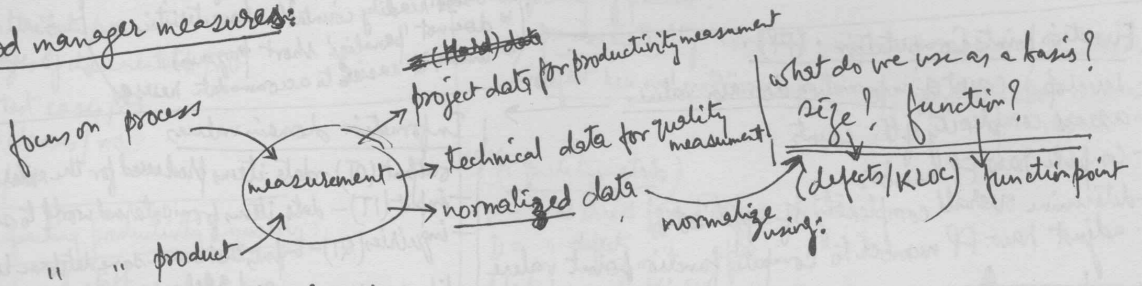
\* Risk is inversely proportional to technology maturity.

\* use metrics to improve SW quality.

Measurement & Metrics  $\Rightarrow$  Total Quality Management (TQM) connection

- TQM  $\rightarrow$  continuous process improvement
- "Kaisen"  $\rightarrow$  the use of measurement isolate common defects and remove them
- SW metrics  $\rightarrow$  a basis for Kaisen

A good manager measures:



- \* Staff experience has a major impact.
- \* Irrational schedules force mistakes and lower quality
- \* unstable requirements

Metrics for the project:

normalize:

- + use LOC or function points (FP) as a basis for ~~error~~ normalization
- examine effort applied (in work units)
- examine deliverables produced
- examine after-the-fact quality

collecting project (Hard) data:

- number of people assigned to a project
- effort (person-months) expended on project tasks
- duration (calendar units) for each project task
- volume of documentation, source code, test cases
- number of defects found and reported

Technical metric example: - cyclomatic complexity or McCabe complexity metric

(measures algo. complexity of a program component)  
(correlates to error-proneness of a module)

\* dedicate testing resources to those modules with high cyclomatic complexity.

collecting support data:  
(SWP)

## collecting support (soft) data: (why?)

- business constraints on the project
- skill of the project team
- experience of the project manager
- stability of requirements
- difficulty of the problem to be solved
- adequacy of the software engineering environment (methods and tools)
- user satisfaction with end-product
- value of the software to the business

## Normalization: Size Vs Function:

- \* LOC "reward" verbose programs
- \* LOC does not accommodate 4GLs particularly well.
- \* LOC is difficult to use in mixed language applications.
- \* LOC can present problems when reuse is applied.

(Size)  
 \* the LOC measure ← most widely used  
 \* a physical artifact of the software engineering process

(the function point measure)  
 \* independent of programming language.  
 \* uses readily countable characteristics.  
 \* does not 'penalize' short programs.  
 \* makes it easier to accommodate reuse.

## Function points Computation: (FP):

- develop a count of information domain values
- assess complexity of the counts
- compute raw FP number
- determine overall complexity of application
- adjust raw FP number to compute function point value

$$FP = \frac{\text{raw FP count}}{\text{value}} \times [0.65 + (0.01 \times \sum C_i)]$$

for FP  
 \* A variation: "feature point"

- it add new count "algorithm"
- it is computed the same way as FP

## Taking algorithms into account: Feature points:

number of user input	—	<sup>weight</sup> × 4 = —
no of " outputs	—	× 5 = —
" " inquiries	—	× 4 = —
" " files	—	× 7 = —
" " interface	—	× 7 = —
algorithms	—	× 3 = —

— count-total  
 — complexity multiplier

FP (feature points)

\* Algorithm: any specific method for solving a certain kind of problem

- For the purposes of feature points, algorithm
- is a set of logical and computational steps that solves a specific, bounded problem or sub-problem.
- has input (or a start value); produces output.
- often corresponds to a cohesive module in a well designed program.

## Information Domain values

- output (OT) - data items produced for the external world.
- input (IT) - data items from external world to software
- inquiries (QT) - inputs that cause some database lookups and response
- files (FT) - externally observable data stores
- interfaces (EI) - connections to other systems or databases

## Domain value complexity

	Simple	Average	Complex
output	4		
input	3	5	7
inquiry	4	4	6
files	7	5	7
interface	5	10	15
		7	10

## Computing raw values:

raw FP values =

$$(OT_{\text{simple}} \times 4) + (OT_{\text{avg}} \times 5) + (OT_{\text{complex}} \times 7) +$$

$$(IT_{\text{simple}} \times 3) + (IT_{\text{avg}} \times 4) + (IT_{\text{complex}} \times 6) +$$

$$(QT_{\text{simple}} \times 4) + (QT_{\text{avg}} \times 5) + (QT_{\text{complex}} \times 7) +$$

$$(FT_{\text{simple}} \times 7) + (FT_{\text{avg}} \times 10) + (FT_{\text{complex}} \times 15) +$$

$$(EI_{\text{simple}} \times 5) + (EI_{\text{avg}} \times 7) + (EI_{\text{complex}} \times 10)$$

## Taking complexity into account:

(Factors are rated on a scale of 0 (not important) to 5 (very important)):

data communications  
 distributed functions  
 heavily used  
 transaction rate  
 on-line data entry  
 and user efficiency

on-line update  
 complex processing  
 installation ease  
 operational ease  
 multiple sites  
 multiple change

Complexity factors,  $CF_i$

## Computing Function Points by "Balkfiring" :

Assembly language	statements per FP
C	320
Fortran	128
COBOL	105
Basic	105
Alg	91
LISP	71
O-O language	64
4GL	29
	20

2. 34 000 lines of PORTMAN  
3650 lines " assembly  
8400 " " C

$$\text{total function points} = \frac{34000}{105} + \frac{3650}{320} + \frac{8400}{128}$$

adjust <sup>total by</sup> complexity factor 0.70 (very low) and 1.30 (very high)

## Productivity metrics:

- effort (person-months) / nv\*  $\rightarrow$  nv = KLOC or FP
- cost / nv
- calendar months / nv
- product attribute / nv
  - \* pages of documentation / nv
  - \* test cases / nv
  - \* classes / nv
- people / nv

## Factors impacting productivity & quality:

1. inexperienced staff
  2. irrational schedules
  3. inexperienced managers
  4. unstable requirements
  5. poor SW engg. methods
  6. no inspections
  7. perfunctory testing
  8. no measurement
  9. low design reuse
  10. low code reuse
- } Process factors

## Quality Metrics:

\* the quality of product:

after-the-fact quality metrics focus on reported defects.  
technical metrics focus on SW design attributes.

\* the quality of process:

process maturity - computed using a process assessment method.

defect removal metrics - computed using information about errors and defects.

≡ (A quality metrics)

E = total errors found before delivery

D = " defects " after "

defect removal efficiency,  $DRE = \frac{E}{E+D}$

## Balancing Agility and discipline

hybrid agile and plan-driven methods provide different but highly effective balances of agility and discipline to fit their unique situations.

A way to plan your program and incorporate both agility and discipline in proportion to your project's needs.

The criteria we develop are based on your project's particular risks with respect to the use of agile or plan-driven methods.

### Risk-based method (5-steps method)

1. Risk analysis: (Rate project's environmental, agile, and plan-driven risks. If uncertain about ratings, buy information via prototyping, data collection, and analysis.)
2. Risk comparison: (Compare risks (agility risks and plan-driven risks) and go with lower risks method.)
3. Architecture analysis: (if confusion, architect application to encapsulate agile parts, ~~and go~~ Go risk-based agile in agile parts and risk-based plan-driven elsewhere.)
4. Tailor life cycle: (establish an overall project ~~strategy~~ strategy by integrating individual risk mitigation plans.)
5. execute and monitor: (monitor progress and risks/opportunities, readjust balance and process as appropriate.)

### Top six conclusions:

1. neither agile nor plan-driven methods ~~provide~~ provide a silver bullet.
2. Agile and plan-driven methods have home grounds where one clearly dominates the other.
3. Future trends are towards application developments that need both agility and discipline.
4. Some balance methods are emerging.
5. It is better to build your method up ~~that~~ then to tailor it down. (starting with minimum sets of practices and only adding extras where it can be clearly justified by cost-benefit.)
6. methods are important, but potential silver bullets are more likely to be found in areas dealing with people, values, communications, expectations management.  
value-proposition about <sup>proposed</sup> S/W system.  
(value-based S/W eng.)