Software Requirements Metrics

- Fairly primitive and predictive power limited.
- Function Points
  - Count number of inputs and output, user interactions, external interfaces, files used.
  - Assess each for complexity and multiply by a weighting factor.
  - Used to predict size or cost and to assess project productivity.
- Number of requirements errors found (to assess quality)
- Change request frequency
  - To assess stability of requirements.
  - Frequency should decrease over time. If not, requirements analysis may not have been done properly.
Programmer Productivity Metrics

- Because software intangible, not possible to measure directly.

- If poor quality software produced quickly, may appear to be more productive than if produce reliable and easy to maintain software (measure only over software development phase).
  - More does not always mean better.
  - May ultimately involve increased system maintenance costs.

- Common measures:
  - Lines of source code written per programmer month.
  - Object instructions produced per programmer month.
  - Pages of documentation written per programmer month.
  - Test cases written and executed per programmer month.
Programmer Productivity Metrics (2)

- Take total number of source code lines delivered and divide by total time required to complete project.
  - What is a source line of code? (declarations? comments? macros?)
  - How treat source lines containing more than a single statement?
  - More productive when use assembly language? (the more expressive the language, the lower the apparent productivity)
  - All tasks subsumed under coding task although coding time represents small part of time needed to complete a project.

- Use number of object instructions generated.
  - More objective.
  - Difficult to estimate until code actually produced.
  - Amount of object code generated dependent on high-level language programming style.
Using pages of documentation penalizes writers who take time to express themselves clearly and concisely. So difficult to give average figure.

- For large, embedded system may be as low as 30 lines/programmer-month.
- Simple business systems may be 600 lines.
- Studies show great variability in individual productivity. Best are twenty times more productive than worst.
Software Design Metrics

- Number of parameters
  - Tries to capture coupling between modules.
  - Understanding modules with large number of parameters will require more time and effort (assumption).
  - Modifying modules with large number of parameters likely to have side effects on other modules.

- Number of modules.

- Number of modules called (estimating complexity of maintenance).
  Fan-in: number of modules that call a particular module.
  Fan-out: how many other modules it calls.
  - High fan-in means many modules depend on this module.
  - High fan-out means module depends on many other modules.
    Makes understanding harder and maintenance more time-consuming.
Software Design Metrics (2)

- Data Bindings

  Triplet \((p,x,q)\) where \(p\) and \(q\) are modules and \(X\) is variable within scope of both \(p\) and \(q\)

- Potential data binding:
  - \(X\) declared in both, but does not check to see if accessed.
  - Reflects possibility that \(p\) and \(q\) might communicate through the shared variable.

- Used data binding:
  - A potential data binding where \(p\) and \(q\) use \(X\).
  - Harder to compute than potential data binding and requires more information about internal logic of module.

- Actual data binding:
  - Used data binding where \(p\) assigns value to \(x\) and \(q\) references it.
  - Hardest to compute but indicates information flow from \(p\) to \(q\).
Cohesion metric

- Construct flow graph for module.
  - Each vertex is an executable statement.
  - For each node, record variables referenced in statement.

- Determine how many independent paths of the module go through the different statements.
  - If a module has high cohesion, most of variables will be used by statements in most paths.

  - Highest cohesion is when all the independent paths use all the variables in the module.
Management Metrics

- Techniques for software cost estimation
  1. Algorithmic cost modeling:
     - Model developed using historical cost information that relates some software metric (usually lines of code) to project cost.
     - Estimate made of metric and then model predicts effort required.
     - The most scientific approach but not necessarily the most accurate.
  2. Expert judgement
  3. Estimation by analogy: useful when other projects in same domain have been completed.
Management Metrics (2)

4. Parkinson’s Law: Work expands to fill the time available.
   - Cost is determined by available resources
   - If software has to be developed in 12 months and you have 5 people available, then effort required is estimated to be 60 person months.

5. Pricing to win: estimated effort based on customer’s budget.

6. Top-down estimation: cost estimate made by considering overall function and how functionality provided by interacting sub-functions. Made on basis of logical function rather than the components implementing that function.

7. Bottom-up function: cost of each component estimated and then added to produce final cost estimate.
Algorithmic Cost Modeling

- Build model by analyzing the costs and attributes of completed projects.
- Dozens of these around -- most well-known is COCOMO.
  - Assumes software requirements relatively stable and project will be well managed.
  - Basic COCOMO uses estimated size of project (primarily in terms of estimated lines of code) and type of project (organic, semi-detached, or embedded).
    \[
    \text{Effort} = A \times \text{KDSI}^b
    \]
    where A and b are constants that vary with type of project.
  - More advanced versions add a series of multipliers for other factors:
    product attributes (reliability, database size, complexity)
    computer attributes (timing and storage constraints, volatility)
    personnel attributes (experience, capability)
    project attributes (use of tools, development schedule)

and allow considering system as made up of non-homogeneous subsystems.
Evaluation of Management Metrics

- Parameters associated with algorithmic cost models are highly organization-dependent.
  - Mohanty: took same data and tried on several models. Estimates ranged from $362,000 to $2,766,667.
  - Another person found estimates from 230 person-months to 3857 person-months.

- Relies on the quantification of some attribute of the finished software product but cost estimation most critical early in project when do not know this number.
  - Lines of code: very difficult to predict or even define.
  - Function points:
    - Heavily biased toward a data processing environment
    - Assessment of complexity factors leads to wide variations in estimates.
Evaluation of Management Metrics (2)

- Value of parameters must be determined by analysis of historical project data for organization. May not have that data or may no longer be applicable (technology changes quickly).

- Need to be skeptical of reports of successful usage of these models.
  - Project cost estimates are self-fulfilling: estimated often used to define project budget and product adjusted so that budget figure is realized.

- No controlled experiments.

- Some interesting hypotheses (that seem to be well accepted):
  - Time required to complete a project is a function of total effort required and not a function of number of software engineers involved.
    - A rapid buildup of staff correlates with project schedule slippages
    - Throwing people at a late project will only make it later.
Evaluation of Management Metrics (3)

- Programmer ability swamps all other factors in factor analyses.
- Accurate schedule and cost estimates are primarily influenced by the experience level of those making them.

Warning about using any software metrics:

Be careful not to ignore the most important factors simply because they are not easily quantified.