Software Metrics

1. *When you can measure what you are speaking about, and can express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind: It may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science.*

   Lord Kelvin, a physicist

2. *In truth, a good case could be made that if your knowledge is meagre and unsatisfactory, the last thing in the world you should do is make measurements; the chance is negligible that you will measure the right things accidentally.*

   George Miller, a psychologist

Software Metrics

- Product vs. process

- Most metrics are indirect:
  - No way to measure property directly or
  - Final product does not yet exist

- For predicting, need a model of relationship of predicted variable with other measurable variables.

- Three assumptions (Kitchenham)

  1. We can accurately measure some property of software or process.

  2. A relationship exists between what we can measure and what we want to know.

  3. This relationship is understood, has been validated, and can be expressed in terms of a formula or model.

- Few metrics have been demonstrated to be predictable or related to product or process attributes.

Software Metrics (2)

- Code
  - Static
  - Dynamic

- Programmer productivity

- Design

- Testing

- Maintainability

- Management
  - Cost
  - Duration, time
  - Staffing

3

Code Metrics

- Estimate number of bugs left in code.

    - From static analysis of code

    - From dynamic execution

- Estimate future failure times: operational reliability

Static Analysis of Code

- Halstead's Software Physics or Software Science

  n1 = no. of distinct operators in program
  n2 = no. of distinct operands in program
  N1 = total number of operator occurrences
  N2 = total number of operand occurrences

  Program Length: $N = N1 + N2$

  Program volume: $V = N \log_2 (n1 + n2)$

  (represents the volume of information (in bits) necessary
   to specify a program.)

  Specification abstraction level: $L = (2 * n2) / (n1 * N2)$

  Program Effort: $E = (n1 + N2 * (N1 + N2) * \log_2 (n1 + n2)) / (2 * n2)$

  (interpreted as number of mental discrimination required
   to implement the program.)

McCabe's Cyclomatic Complexity

Hypothesis: Difficulty of understanding a program is largely determined by complexity of control flow graph.

- Cyclomatic number V of a connected graph G is the number of linearly independent paths in the graph or number of regions in a planar graph.
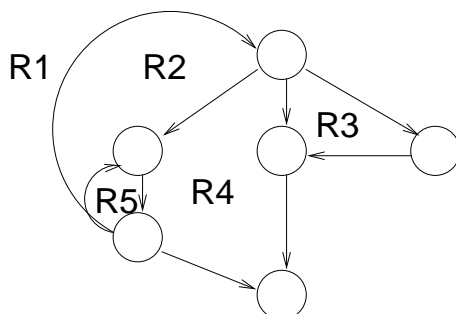
- Claimed to be a measure of testing diffiiculty and reliability of modules.

- McCabe recommends maximum V(G) of 10.

Static Analysis of Code (Problems)

- Doesn't change as program changes.

- High correlation with program size.

- No real intuitive reason for many of metrics.

- Ignores many factors: e.g., computing environment, application area, particular algorithms implemented, characteristics of users, ability of programmers,.

- Very easy to get around. Programmers may introduce more obscure complexity in order to minimize properties measured by particular complexity metric.

Static Analysis of Code (Problems con't)

- Size is best predictor of inherent faults remaining at start of program test.

- One study has shown that besides size, 3 significant additional factors:

    1. Specification change activity, measured in pages of specification changes per k lines of code.

    2. Average programmer skill, measured in years.

    3. Thoroughness of design documentation, measured in pages of developed (new plus modified) design documents per k lines of code.

Bug Counting using Dynamic Measurement

- Estimate number remaining from number found.

  - Failure count models

  - Error seeding models

- Assumptions:

  - Seeded faults equivalent to inherent faults in difficulty of detection.

  - A direct relationship between characteristics and number of exposed and undiscovered faults.

  - Unreliability of system will be directly proportional to number of faults that remain.

  - A constant rate of fault detection.

Bug Counting using Dynamic Measurement (2)

- What does an estimate of remaining errors mean?

  Interested in performance of program, not in how many bugs it contains.

- Most requirements written in terms of operational reliability, not number of bugs.

- Alternative is to estimate failure rates or future interfailure times.

Estimating Failure Rates

- Input-Domain Models: Estimate program reliability using test cases sampled from input domain.

    - Partition input domain into equivalence classes, each of which usually associated with a program path.

    - Estimate conditional probability that program correct for all possible inputs given it is correct for a specified set of inputs.

    - Assumes outcome of test case given information about behavior for other points close to test point.

- Reliability Growth Models: Try to determine future time between failures.

Reliability Growth Models

Software Reliability: The probability that a program
will perform its specified function for a stated time
under specified conditions.

- Execute program until "failure" occurs, the underlying
  error found and removed (in zero time), and resume
  execution.

- Use a probability distribution function for the interfailure
  time (assumed to be a random variable) to predict future
  times to failure.

- Examining the nature of the sequence of elapsed times
  from one failure to the next.

- Assumes occurrence of software failures is a stochastic
  process.

Software Uncertainty

Assumption: The mechanism that selects successive inputs during execution is unpredictable (random).

Input space I

I_F

Program p

O_F

Output space O
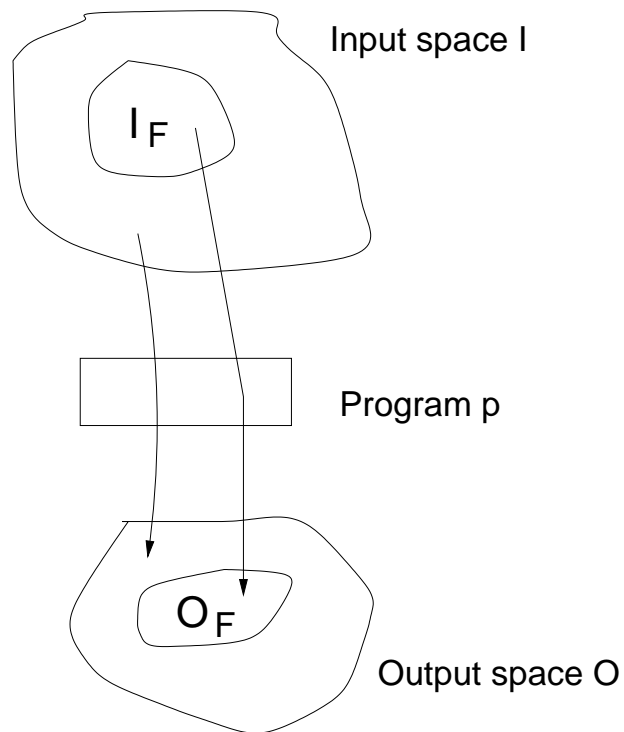
O_F is the image set of I_F under the mapping p

Sample Interfailure Times Data

3	30	113	81	115	9	2	91	112	15
138	50	77	24	108	88	670	120	26	114
325	55	242	68	422	180	10	1146	600	15
36	4	0	8	227	65	176	58	457	300
97	263	452	255	197	193	6	79	816	1351
148	21	233	134	357	193	236	31	369	748
0	232	330	365	1222	543	10	16	529	379
44	129	810	290	300	529	281	160	828	1011
445	296	1755	1064	1783	860	983	707	33	868
724	2323	2930	1461	843	12	261	1800	865	1435
30	143	109	0	3110	1247	943	700	875	245
729	1897	447	386	446	122	990	948	1082	22
75	482	5509	100	10	1071	371	790	6150	3321
1045	648	5485	1160	1864	4116				

Execution time in seconds between successive failures.

(Read left to right in rows).

Using the Models
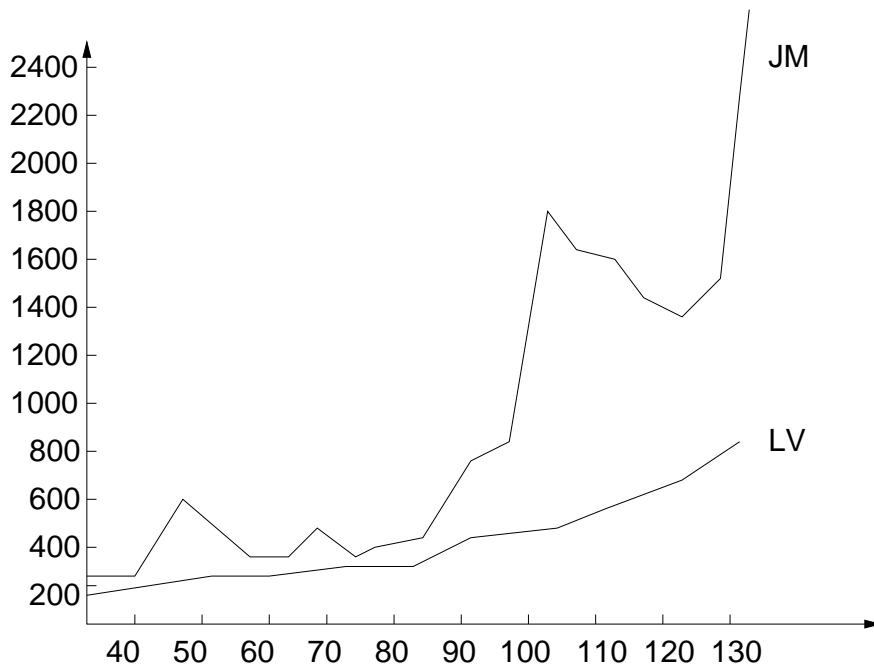
- Different models can give varying results for the same data; there is no way to know a priori which model will provide the best results in a given situation.

- "The nature of the software engineering process is too poorly understood to provide a basis for selecting a particular model."

Problems with Software Reliability Modeling

- There is no physical reality on which to base our assumptions.

- Assumptions are not always valid for all, or any, programs:

  - Software fault (and failures they cause) are independent.

  - Inputs for software selected randonly from an input space.

  - Test space is representative of the operational input space.

  - Each software failure is observed.

  - Faults are corrected without introducing new ones.

  - Each fault contributes equally to the failure rate.

- Data collection requirements may be impractical.

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.

Programmer Productivity Metrics (3)

- 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.

Software Design Metrics (3)

- 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).

    $$Effort = A * KDSI^b$$

    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.