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
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

\[ n_1 = \text{no. of distinct operators in program} \]
\[ n_2 = \text{no. of distinct operands in program} \]
\[ N_1 = \text{total number of operator occurrences} \]
\[ N_2 = \text{total number of operand occurrences} \]

Program Length: \[ N = N_1 + N_2 \]

Program volume: \[ V = N \log_2 (n_1 + n_2) \]

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

Specification abstraction level: \[ L = \frac{(2 * n_2)}{(n_1 * N_2)} \]

Program Effort: \[ E = \frac{(n_1 + N_2 * (N_1 + N_2) * \log_2 (n_1 + n_2))}{(2 * n_2)} \]

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

\[ I_F \subseteq I \]

\[ O_F \subseteq O \]

\( O_F \) is the image set of \( I_F \) under the mapping \( p \)
### Sample Interfailure Times Data

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

Execution time in seconds between successive failures.

(Read left to right in rows).
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.”
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 randomly 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.