Software Process Metrics

• Outline
  – Today's topic is process metrics, the measurements we can make related to the software development and maintenance process itself
    • Predictive process metrics
      – Effort and cost
    • Specification metrics
Software Cost Estimation

• COCOMO
  – Stands for COnstructive COst MOdel
  – A method for modelling software development, to yield estimates of effort and cost before undertaking the project
  – Based on a mathematical model of effort, plus empirical constants to parameterize the model
Estimating Effort Using COCOMO

- **Simple COCOMO effort prediction**
  - The simplest COCOMO model uses the estimate
    \[ \text{Effort} = a \ (\text{Size})^b \]
  - where
    - \text{Effort} is measured in person-months, and
    - \text{Size} is the predicted size of the software in KDSI
      ("thousands of delivered source instructions")
Estimating Effort Using COCOMO

• Simple COCOMO effort prediction
  – The simplest COCOMO model uses the estimate
    \[ \text{Effort} = a \, (\text{Size})^b \]
  – \(a\) and \(b\) are empirically derived constants depending on the kind of software:
    • “organic” – stand-alone in-house data processing systems
      \(a = 2.4, \quad b = 1.05\)
    • “embedded” – real-time or hardware linked systems
      \(a = 3.6, \quad b = 1.2\)
    • “semi-detached” – in between the two above
      \(a = 3.0, \quad b = 1.12\)
Problems Estimating Size

• The downside of COCOMO
  – The simple COCOMO model is claimed to give good order of magnitude estimates of required effort
  – But depends on a size estimate – which some say is just as hard to estimate as effort!
  – Example:
    • In one experiment managers were asked to estimate software size given the complete specifications
    • The average deviation from the actual size was 64%
    • Only 25% of the estimates were within 25% of the actual size
Estimating Time Using COCOMO

• Simple COCOMO development time prediction
  – COCOMO uses a similar model for time given effort
    \[ \text{Time} = a \ (\text{Effort})^b \]
  – where
    • Time is measured in months, and
    • Effort is measured in person-months
Estimating Time Using COCOMO

• Simple COCOMO development time prediction
  – COCOMO uses a similar model for time given effort
  \[ \text{Time} = a \ (\text{Effort})^b \]
  – Again, \( a \) and \( b \) are (different) empirically derived constants depending on the kind of software:
    • “organic” – stand-alone in-house data processing systems
      \( a = 2.5, \ b = 0.38 \)
    • “embedded” – real-time or hardware linked systems
      \( a = 2.5, \ b = 0.32 \)
    • “semi-detached” – in between the two above
      \( a = 2.5, \ b = 0.35 \)
Regression-Based Cost Estimation

• Where does the COCOMO model come from?
  – COCOMO is based on empirical measurements of the actual effort and cost of past software projects as a function of software size
  – And the derivation of a regression equation to explain them

(a different sense of “regression”)
Regression-Based Cost Estimation

• Where does the COCOMO model come from?
Regression-Based Cost Estimation

• Where does the COCOMO model come from?
  – Analysis of the historical data indicates that the logarithm of the effort required to produce a software system has a linear relationship with the logarithm of the size of the software, that is:
    \[ \log \text{Effort} = \log a + b \log \text{Size} \]
    – where \( \log a \) is the y-intercept of the line and \( b \) is the slope of the line
  – Solving for Effort yields the COCOMO effort model
    \[ \text{Effort} = a \ (\text{Size})^b \]
  – A similar empirical observation of the historical relationship between Time and Effort yields the COCOMO model for estimating time required
Specification-Based Size Metrics

• How can we predict size independently of code?
  – Predictions of effort, cost and time depending on code size have two inherent difficulties:
    • Prediction based on KDSI or KLOC just involves just replacing one difficult prediction problem (effort, cost or time) with another one (code size)
    • KDSI and KLOC are actually measures of length, not size (which must take into account functionality)
Specification-Based Size Metrics

• How can we predict size independently of code?
  – Code complexity size measures, which would be better, can’t be predicted any more easily than code length
  – If our size measures are based on the specification of functionality of the software rather than its eventual code, perhaps we can more accurately estimate size once the specification is known
Function Point Analysis

• Analyzing the functional specification
  – The number of function points [Albrecht 1979] is a popular and widely used size metric
    • Designed to reflect the size of the functionality of a piece of software from the end user’s point of view, independently of the code that implements it
  – Computed from detailed system specification (available early in the development cycle) using the equation:

\[
FP = UFC \cdot TCF
\]

– where
  • **UFC** is the unadjusted function count, a count of the number of different user visible functions required by the spec, and
  • **TCF** is the technical complexity factor, a constant between 0.65 and 1.35, determined by 14 questions about the system
Counting Functions

- The **UFC** is obtained by summing weighted counts of the number of **inputs**, **outputs**, **logical master files**, **interface files** and **queries** visible to the system user, where:
  - an **input** is a user or control data element entering an application;
  - an **output** is a user or control data element leaving an application;
  - a **logical master file** is a data store acted on by the application user (an internal file or database);
  - an **interface file** is a file or input/output data that is used by another application (an external file or database);
  - a **query** is an input-output combination (i.e. an input that results in an immediate output).
Using Function Points

• A better size metric
  – FP’s are used extensively as a size metric in preference to KLOC, for example in equations for productivity, defect density and cost / effort prediction
  – **Advantages:**
    • language-independent
    • can be computed early in a project
    • does not have to be predicted; derived directly from the spec
  – **Disadvantages:**
    • unnecessarily complex: evidence is that TCF adds little; effort prediction after adding the TCF is often no better than UFC alone
    • difficult to compute, uses a large degree of subjectivity
    • some doubt they actually measure functionality
Using Function Points

• **Bottom line:**
  – FPs are common, popular, better than KLOC, and apparently work
  – The International Function Point Users Group formalizes rules for Function Point counting to ensure that counts are comparable across different systems and organizations.
Function Points: An Example

• Spell Checker Specification
  – Accepts as input a document file, a dictionary file and an optional user dictionary file
  – The checker lists all words in the document file not contained in either of the dictionary files
  – User can query the number of words processed and the number of spelling errors found at any stage in the process
Function Points: An Example

- Spell Checker Specification

\[ \begin{align*}
A &= \# \text{inputs} = 2 \\
B &= \# \text{outputs} = 3 \\
C &= \# \text{internal files ("logical master files")} = 1 \\
D &= \# \text{external files ("interface files")} = 2 \\
E &= \# \text{queries} = 2
\end{align*} \]

\[ UFC = 4A + 5B + 7C + 10D + 4E = 58 \]
# Function Points vs. Program Length

## SLOC per function point

<table>
<thead>
<tr>
<th>Language</th>
<th>median</th>
<th>high</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly</td>
<td>98</td>
<td>320</td>
</tr>
<tr>
<td>C</td>
<td>99</td>
<td>333</td>
</tr>
<tr>
<td>C++</td>
<td>53</td>
<td>80</td>
</tr>
<tr>
<td>C#</td>
<td>59</td>
<td>70</td>
</tr>
<tr>
<td>Excel</td>
<td>191</td>
<td>315</td>
</tr>
<tr>
<td>HTML</td>
<td>40</td>
<td>48</td>
</tr>
<tr>
<td>Java</td>
<td>53</td>
<td>134</td>
</tr>
<tr>
<td>JavaScript</td>
<td>53</td>
<td>63</td>
</tr>
</tbody>
</table>

[www.qsm.com/resources/function-point-languages-table](http://www.qsm.com/resources/function-point-languages-table)
Dusty old version of previous slide

<table>
<thead>
<tr>
<th>Language</th>
<th>Source Statements per FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembler</td>
<td>320</td>
</tr>
<tr>
<td>C</td>
<td>150</td>
</tr>
<tr>
<td>Algol</td>
<td>106</td>
</tr>
<tr>
<td>COBOL</td>
<td>106</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>106</td>
</tr>
<tr>
<td>Pascal</td>
<td>91</td>
</tr>
<tr>
<td>RPG</td>
<td>80</td>
</tr>
<tr>
<td>PL/1</td>
<td>80</td>
</tr>
<tr>
<td>MODULA-2</td>
<td>71</td>
</tr>
<tr>
<td>PROLOG</td>
<td>64</td>
</tr>
<tr>
<td>LISP</td>
<td>64</td>
</tr>
<tr>
<td>BASIC</td>
<td>64</td>
</tr>
<tr>
<td>4 GL Database</td>
<td>40</td>
</tr>
<tr>
<td>APL</td>
<td>32</td>
</tr>
<tr>
<td>SMALLTALK</td>
<td>21</td>
</tr>
<tr>
<td>Query languages</td>
<td>16</td>
</tr>
<tr>
<td>Spreadsheet languages</td>
<td>6</td>
</tr>
</tbody>
</table>
Summary

• **Software Process Metrics**
  – Process metrics attempt to predict properties of the software process, such as effort, time and cost
  – Process predictions need good estimates of size
  – Function points provide a good code-independent way to estimate the size of a software problem

• **Reference**
  – Somerville Ch. 23, Project Planning

• **Next time**
  – Introduction to Software (In)security

• **Remember**
  – Assignment #5 due Thursday