Software Engineering

Prof. Dr. Bertrand Meyer

March 2007 – June 2007

Lecture 28: Software metrics
Measurement

“To measure is to know”

“When you can measure what you are speaking about and 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 meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the state of Science, whatever the matter may be.”

"If you cannot measure it, you cannot improve it."

Lord Kelvin

“You can't control what you can't measure”

Tom de Marco

“Not everything that counts can be counted, and not everything that can be counted counts.”

Albert Einstein
Why measure software?

Understand issues of software development

Make decisions on basis of facts rather than opinions

Predict conditions of future developments
Software metrics: methodological guidelines

Measure only for a clearly stated purpose

Specifically: software measures should be connected with quality and cost

Assess the validity of measures through controlled, credible experiments

Apply software measures to software, not people

GQM (see below)
Example: software quality

External quality factors:
- Correctness
- Robustness
- Ease of use
- Security
- ...

Compare:
- “This program is much more correct than the previous development”
- “There are 67 outstanding bugs, of which 3 are `blocking` and 12 `serious`. The new bug rate for the past three months has been two per week.”
Absolute and relative measurements

Without Historical Data
Variance between +20% to -145%
(Mostly Level 1 & 2)

With Historical Data
Variance between -20% to +20%
(Level 3)

(Based on 120 projects in Boeing Information Systems)

What to measure in software

Effort measures
- Development time
- Team size
- Cost

Quality measures
- Number of failures
- Number of faults
- Mean Time Between Failures
Cost models

Purpose: estimate in advance the effort attributes (development time, team size, cost) of a project

Problems involved:

- Find the appropriate parameters defining the project (making sure they are measurable in advance)
- Measure these parameters
- Deduce effort attributes through appropriate mathematical formula

Best known model: COCOMO (B. W. Boehm)
Difficulty of cost control

Most industry projects late and over budget, although situation is improving
Cost estimation still considered black magic by many; does it have to be?

Source: Standish report

Source: van Genuchten (1991)
Average overrun: 22%
Difficulty of effort measurement: an example

(after Ghezzi/Jazayeri/Mandrioli)

Productivity:
- Software professional: a few tens of lines of code per day
- Student doing project: much more!

Discrepancy due to: other activities (meetings, administration, ...); higher quality requirements; application complexity; need to understand existing software elements; communication time in multi-person development
Effort measurement

Standard measure: person-month (or “man-month”)

Even this simple notion is not without raising difficulties:

- Programmers don’t just program
- \( m \text{ persons } \times n \text{ months} \) is not interchangeable with \( n \text{ persons } \times m \text{ months} \)

Brooks: “The Mythical Man-Month”
Project parameters

Elements that can be measured in advance, to be fed into cost model

Candidates:

- Lines of code (LOC, KLOC, SLOC..)
- Function points
- Application points
Lines of code

Definition: count number of lines in program

Conventions needed for: comments; multi-line instructions; control structures; reused code.

Pros as a cost estimate parameter:

- Appeals to programmers
- Fairly easy to measure on final product
- Correlates well with other effort measures

Cons:

- Ambiguous (several instructions per line, count comments or not, …)
- Does not distinguish between programming languages of various abstraction levels
- Low-level, implementation-oriented
- Difficult to estimate in advance.
Some more O-O measures

Weighted Methods Per Class (WMC)

Depth of Inheritance Tree of a Class (DIT)

Number of Children (NOC)

Coupling Between Objects (CBO)

Response for a Class (RFC)
Function points

Definition: one end-user business function

Five categories (and associated weights):
- Inputs (4)
- Outputs (5)
- Inquiries (4)
- Files (10)
- Interfaces to other systems (7)

Pros as a cost estimate parameter:
- Relates to functionality, not just implementation
- Experience of many years, ISO standard
- Can be estimated from design
- Correlates well with other effort measures

Cons:
- Oriented towards business data processing
- Fixed weights
Application points

Definition: high-level effort generators
Examples: screen, reports, high-level modules

Pro as a cost estimate parameter:
- Relates to high-level functionality
- Can be estimated very early on

Con:
- Remote from actual program
Cost models: COCOMO

Basic formula:

\[ \text{Effort} = A \times \text{Size}^B \times M \]

For \text{Size}, use:

- Action points at stage 1 (requirements)
- Function points at stage 2 (early design)
- Function points and SLOC at stage 3 (post-architecture)

0.91 to 1.23 (depending on novelty, risk, process...)

2.94 (early design)

Cost driver estimation
COCOMO cost drivers (examples)

Early design:
- Product reliability & complexity
- Required reuse
- Platform difficulty
- Personnel capability
- Personnel experience
- Schedule
- Support facilities

Postarchitecture:
- Product reliability & complexity
- Database size
- Documentation needs
- Required reuse
- Execution time & storage constraints
- Platform volatility
- Personnel experience & capability
- Use of software tools
- Schedule
- Multisite development
About cost models

Easy to criticize, but seem to correlate well with measured effort in well-controlled environments

Useful only in connection with long-running measurement and project tracking policy; cf CMMI, PSP/TSP

Worth a try if you are concerned with predictability and cost control
Complexity models

Aim: estimate complexity of a software system

Examples:
- Lines of code
- Function points
- Halstead’s volume measure: $N \log \eta$, where $N$ is program length and $\eta$ the program vocabulary (operators + operands)
- McCabe’s cyclomatic number: $C = e - n + 2p$, where $n$ is number of vertices in control graph, $e$ the number of edges, and $p$ the number of connected components
Reliability models

Goal: to estimate the reliability - essentially, the likelihood of faults - in a system.

Basis: observed failures

Source: hardware reliability studies; application to software has been repeatedly questioned, but the ideas seem to hold
Reliability models: basic parameters

Interfailure times
Average: Mean Time To Failure: MTTF

Mean Time To Repair: MTTR
- Do we stop execution to repair?
- Can repair introduce new faults?

Reliability: $R$

$$R = \frac{MTTF}{1 + MTTF}$$
MTTF: the AutoTest experience

Apparent shape:

\[ b = \frac{1}{a + b \cdot t} \]
Reliability models

Attempt to predict the number of remaining faults and failures.

Example: Motorola’s zero-failure testing

\[ \text{Failures} (t) = a e^{-b(t)} \]

Zero-failure test hours:

\[
\frac{\ln \left( \frac{f}{(0.5 + f)} \right) \times h}{\ln \left( \frac{(0.5 + f)}{t + f} \right)}
\]

- Desired number of failures
- Hours to last failure
- Test failures so far
- Test failures so far
Software Metrics
using EiffelStudio

With material by
Yi Wei & Marco Piccioni

June 2007
What to measure

Product properties
- Lines of Code
- Number of classes
- Cohesion & Coupling
- Conformance of code to OO principles

Process properties
- Man-month spent on software
- Number of bugs introduced per hour
- Ratio of debugging/developing time
- CMM, PSP
Traditional Metrics

McCabe Cyclomatic Complexity (CC)

Source Lines of Code (SLOC)

Comment Percentage (CP)
McCabe Cyclomatic Complexity

A measure based on a connected graph of the module (shows the topology of control flow within the program)

Definition

\[ M = E - N + P \] where

- \( M \) = cyclomatic complexity
- \( E \) = the number of edges of the graph
- \( N \) = the number of nodes of the graph
- \( P \) = the number of connected components.
Example of Cyclomatic Complexity

\[
\text{if } \text{condition} \text{ then} \\
\quad \text{code 1} \\
\text{else} \\
\quad \text{code 2} \\
\text{end}
\]

\[
E = 4, \quad N = 4, \quad P = 2, \\
M = 4 - 4 + 2 = 2
\]
Source Lines of Code

A measure of the number of physical lines of code

Different counting strategies:
- Blank lines
- Comment lines
- Automatically generated lines

EiffelBase has 63,474 lines, Vision2 has 153,933 lines, EiffelStudio (Windows GUI) has 1,881,480 lines in all compiled classes.

Code used in examples given here and below are got from revision 68868 in Origo subversion server.
Comment Percentage

Ratio of the number of commented lines of code divided by the number of non-blank lines of code.

Critique:
If you need to comment your code, you better refactor it.
OO metrics

**Weighted Methods Per Class (WMC)**

**Depth of Inheritance Tree of a Class (DIT)**

**Number of Children (NOC)**

**Coupling Between Objects (CBO)**

**Response for a Class (RFC)**
Weighted Methods Per Class

Sum of the complexity of each feature contained in the class.
Feature complexity: (e.g. cyclomatic complexity)
When feature complexity assumed to be 1,
WMC = number of features in class

In Eiffel base, there are 5,341 features,
In Vision2 (Windows), there are 10,315 features,
In EiffelStudio (Windows GUI), there are 89,630 features.
Depth of Inheritance Tree of a Class

Length of the longest path of inheritance ending at the current module

for CHAIN, DIT=7
Number of children

Number of immediate subclasses of a class.

In Eiffel base, there are 3 classes which have more than 10 immediate subclasses:

ANY
COMPARABLE
HASHABLE

And of course, ANY has most children.
Coupling between objects

Number of other classes to which a class is coupled, i.e., suppliers of a class.

In Eiffel base, there are 3 classes which directly depend on more than 20 other classes, they are:

- STRING_8
- STRING_32
- TUPLE

Class SED_STORABLE_FACILITIES indirectly depends on 91 other classes.
Response for a Class

Number of features that can potentially be executed in a feature, i.e., transitive closure of feature calls.

```
foo is do
    bar
end

bar is
    f1
    f2
end
```

RFC=3
Metrics tool in EiffelStudio

A code quality checking tool with seamlessly working style:

Coding - Metricing - Problem solving - Coding

Highly customizable:

Define your own metrics to match particular requires

Metric archive comparison:

Compare measurement of your software to others

Automatic metric quality checking:

Get warned when some quality criterion are not met
Metrics tool – Evaluate metric
Metrics tool – Investigate result

- Metric name: Classes
- Type: Basic
- Unit: Class
- Value: 377

Input domain:
root_cluster base net time web

Results:

<table>
<thead>
<tr>
<th>Class</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML_TABLE</td>
<td>web.table</td>
</tr>
<tr>
<td>HTML_TABLE_CONSTANTS</td>
<td>web.table</td>
</tr>
<tr>
<td>STDIN</td>
<td>web.stdio</td>
</tr>
<tr>
<td>STDOUT</td>
<td>web.stdio</td>
</tr>
<tr>
<td>SHARED_STDOUT</td>
<td>web.stdio</td>
</tr>
<tr>
<td>SHARED_STDIN</td>
<td>web.stdio</td>
</tr>
<tr>
<td>HTML_PAGE</td>
<td>web.html</td>
</tr>
<tr>
<td>HTML_TEXT</td>
<td>web.html</td>
</tr>
<tr>
<td>HTML_GENERATOR</td>
<td>web.html</td>
</tr>
<tr>
<td>HTML_CONSTANTS</td>
<td>web.html</td>
</tr>
<tr>
<td>HTML</td>
<td>web.html</td>
</tr>
</tbody>
</table>
Metrics tool – Define new metric
<table>
<thead>
<tr>
<th>Metric name</th>
<th>Current value</th>
<th>Previous value</th>
<th>Difference</th>
<th>Filter</th>
<th>Result</th>
<th>Calculated time</th>
<th>Input domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncommented features</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td>05/26/2007 9:24:52.311 PM</td>
<td>sample</td>
</tr>
<tr>
<td>Features</td>
<td>62</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>05/26/2007 7:10:30.899 AM</td>
<td>sample</td>
</tr>
<tr>
<td>Classes</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>05/26/2007 7:19:40.375 AM</td>
<td>sample</td>
</tr>
<tr>
<td>Classes</td>
<td>242</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>05/26/2007 7:40:30.734 AM</td>
<td>base</td>
</tr>
</tbody>
</table>
Metrics tool - Archive

Archive Management
Location: c:\my_archive.xml

Setup input domain:

Select metric:

Root_cluster

Archive Comparison
Compare

Select reference archive (URL acceptable):

Select current archive (URL acceptable):
Software metrics: methodological guidelines

Measure only for a clearly stated purpose

Specifically: software measures should be connected with quality and cost

Assess the validity of measures through controlled, credible experiments

Apply software measures to software, not people

GQM (see below)
GQM (Goal/Question/Metric)

Process for a measurement campaign:

1. Define goal of measurement
   Analyze... with the purpose of ... the ... from the point of view of ... in the context of ...

   Example: Analyze testing phase with the purpose of estimating the costs from the point of view of the manager in the context of Siemens Train Division’s embedded systems group

2. Devise suitable set of questions
   Example: do faults remain that can have major safety impact?

3. Associate metric with every question
GQM (Goal/Question/Metric) (Basili et al.)

Process for a measurement campaign:

1. Define goal of measurement
   Analyze... with the purpose of ... the ... from the point of view of ... in the context of ...

   Example: Analyze testing phase with the purpose of estimating the costs from the point of view of the manager in the context of Siemens Train Division’s embedded systems group

2. Devise suitable set of questions
   Example: do faults remain that can have major safety impact?

3. Associate metric with every question
Final observations about this course

(Starting with quotations from the first lecture)
So, what is software engineering?

The production of operational software satisfying defined standards of quality.
Software engineering

... includes programming, but is more than programming.

As von Clausewitz did not write: “the continuation of programming through other means”.
The five components of software engineering

- **Describe**
  - Requirements, design specification, documentation...

- **Implement**
  - Design, programming

- **Assess**
  - Testing and other V&V* techniques

- **Manage**
  - Plans, schedules, communication, reviews...

- **Operate**
  - Deployment, installation, *Validation & Verification*
Some principles: about the course

Do not focus on design and implementation (covered in other courses)

In the course (less in the project) de-emphasize programming

Emphasize software activities that are not usually covered in programming courses and are hard to convey in a standard university setting

Emphasize group work

Emphasize management and communication aspects

Stay away from fuzzy science (e.g. Halstead’s “software science”)

Try to keep course in sync with project; project is main thread