# Object-Oriented Software Engineering

**Extra Chapter:**
**Software Metrics**

Lecture 14

## Why Measure Software?

**Projects often**
- Go over budget
- Miss deadlines
- Exhibit poor quality

**Measurements can be used to improve the process and quality**
- Both short term (current) and long term (future)

**Systematic improvement is not possible without measurement**

# Software Metrics

**Definition**

- The measurement of the software development *process* and its resulting *product*

**The software product includes the source code, executable code, and related documentation**

**Used to**

- Estimate cost
- Estimate schedule
- Measure productivity
- Measure quality



# Characteristics of Good Metrics

- Can be used to predict, not just describe
- Simple; precisely definable
- Objective
- Easily obtained; low cost
- Valid; relevant to intended measurement
- Robust; relatively insensitive to insignificant changes

# Metrics Classification

**Process**

- Measurements of the development process
- Examples
  - —Development time
  - —Methodology (Waterfall, Spiral, etc.)
  - —Average years of experience for development staff

**Product**

- Measurements of software at any point in development
- Examples
  - —Design complexity
  - —Program size (source or executable) in lines of code (LOC)
  - —Pages of documentation



# More Classifications

**Objective versus subjective**

- Do different observers measure the same values?
- Examples
  - —Objective: LOC, development time
  - —Subjective: developer experience

**Primitive versus computed**

- Examples
  - —Primitive: LOC, development time
  - —Computed: LOC/Person-month, Defects/KLOC

# Data Examples

**Nominal**
- Program type: database, network, application

**Ordinal**
- Programmer experience: low, medium, high

**Interval**
- Program complexity: 5, 4, 3, 2, 1

**Ratio**
- LOC: 2000 is twice as large as 1000



# Process Metrics

**For individuals**
- Work distribution
- Estimated versus actual task duration and effort
- Code covered by unit testing
- Number of defects found by unit testing

# Process Metrics

**For project teams**

- Work effort distribution
- Requirements status (number approved, implemented, and verified)
- Percentage of test cases passed
- Estimated versus actual duration between milestones
- Staffing levels
- Defects found during testing and inspection
- Defect status
- Requirements stability
- Number of tasks planned and completed



# Process Metrics

**For development organizations**

- Defect levels in released products
- Product development cycle time
- Schedule and effort estimating accuracy
- Reuse effectiveness
- Planned and actual cost

# Product Metrics

**Size**
- Lines of code
- File, Flow, Process (FFP)
- Function points

**Complexity**
- More about these later

**Quality**
- Defect counts
- Reliability: mean time to failure
- Maintainability: complexity, %comments



# Size: Lines of Code

**Most common**

**Units**
- Lines of code (LOC)
- Thousands delivered source instructions (KDSI)

**Can be unreliable given the development stage**

# Problems with LOC Metrics

**Only a portion of the development effort**

- What about requirements? Models?

**Different languages produce programs with different LOC**

**How should they be counted? Comments?**

**Not all code written is delivered to the client**

**The most accurate LOC measurement is obtained when the project is complete**



# Size: File, Flow, Process

**Files, flows and processes (FFP)**

- File: collection of logically or physically related records that are permanently resident
- Flow: a data interface between the product and the environment
- Process: functionally defined logical or arithmetic manipulation of data

**Can be determined using the architectural design**

**Sum the number of files, flows, and processes**

- Size = Fi + Fl + Pr
- C = b * Size
- Where b is an organization-dependent constant

**The metric was never updated to include databases**

# Size: Function Points

**Based on the number of**

- inputs (Inp)
- outputs (Out)
- inquiries (Inq)
- master files (Maf)
- and interfaces (Inf)

**Rough approximation of size is calculated by FP =**

- (4 * Inp) + (5 * Out) + (4 * Inq) + (10 * Maf) + (7 * Inf)

**Can be more accurate than LOC (why?)**

**As with previous size metrics, can be inaccurate depending on design phase**



# ISO/IEC 15939

**An international standard defining the software measurement process**

**Published in 2002**

**Covers these topics**

- Measurement activities
- Required information
- Application of measurement analysis results
- Determining validity of analysis results

**Can be used by both developers and clients**

# Tenets of Documentation

**“If you didn’t write it down, it didn’t happen”**

**“If it wasn’t recorded, it can’t be measured”**

**“If it wasn’t measured, it can’t be controlled”**

**Successful software engineers discipline themselves to document their work**

- Record your research
- Record your thoughts when weighing design options
- Record your rationale
- Record your effort



# Process Metrics for Your Team

**Per team member**

- Total hours
- Hours per week
- LOC per week
- Defects identified

**Per week**

- LOC per subsystem

**Per subsystem**

- Changes in complexity
- Defects found/corrected

# References

"Software Metrics", EvelandE. Mills,
http://www.sei.cmu.edu/publications/documents/cms/cm.012.html

"Software Engineering", Stephen R. Schach

"A Software Metrics Primer", Karl E. Wiegers,
http://www.processimpact.com/articles/metrics_primer.html



# Complexity Metrics

**There are many!**

**As with previous metrics, some are better suited for**

- A particular design phase
- A particular programming language/paradigm

**Examples**

- Software Science [Halstead]
- Cyclomatic [McCabe]
- Fan-in / Fan-out [Henry and Kafura]
- Object-oriented [Morris]

# Halstead's Software Science

**Attempts to estimate programming effort**

**Classifies program tokens as *operators* and *operands***

**Counts these attributes**

- n1= the number of distinct operators
- n2= the number of distinct operands
- N1= the total number of operators
- N2= the total number of operands



# Halstead Complexity Metrics

**Program length**

- N= N1+ N2

**Program vocabulary**

- n= n1+ n2

**Volume**

- V= N * (log2n)

**Difficulty**

- D= (n1/2) * (N2/n2)

**Effort**

- E= D * V

# Example: Halstead

```
void foo (int k, int x) {
   if (k < 3) {
      if (k < 1) {
         x = x * x;
      } else {
         x = x * k;
      }
   }
}
```

N1 = 29; N2 = 13
n1 = 13; n2 = 5
N = 42; n = 18
V = 42 log218 = 176.4
D = (13/2)(13/5) = 16.9
E = (16.9)(176.4) = 2981.16



# Halstead Metrics

**Some debate over their validity**

**Advantages**

- Does not require in-depth analysis of program structure
- Predicts maintenance effort
- Simple to calculate/automate

**Disadvantages**

- Depends of completed code

# McCabe's Cyclomatic Complexity

**Measures complexity based on program structure**

**Partitions program into nodes and edges**

**Cyclomatic complexity**

- CC(G) = number(edges) -number(nodes) + 1

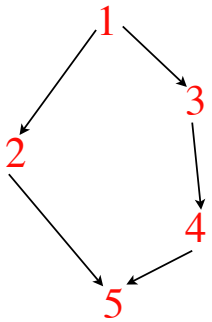


# Examples: Cyclomatic

```
1 statement_a
2 statement_b
```

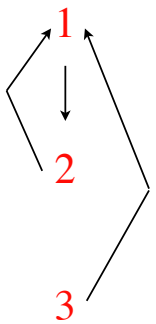

CC = 1 – 2 + 1 = 0

```
1 if (condition)
2     block_1
3 else
4     block_2
5
```




CC = 5 – 5 + 1 = 1

```
1 while (condition)
2     block_1
3
```




CC = 3 –3 + 1 = 1

# Cyclomatic Metrics

**Advantages**

- Good maintenance predictor
- Can be used in design phase to compare options
- Applicable earlier than Halstead

**Disadvantages**

- Measures control complexity, not data complexity
- Same weight placed on nested and non-nested loops
- Many simple comparison and decision structures may produce a misleading figure