Course Notes Set 13:

Software Metrics

Computer Science and Software Engineering
Auburn University

Software Metrics

- Software metrics is a broad area of research.
- Essentially refers to the measurement of certain attributes of a software: [Pressman]
  - **Process**
    - Give insight into what works and what doesn't in the process (e.g., the model, tasks, milestones, etc.).
    - The goal is long-term process improvement.
  - **Project**
    - Give insight into the status of an ongoing project, track potential risks, identify problems earlier, adjust workflow and tasks, evaluate the project team's ability to control quality.
    - The goal is to keep a project on schedule and within quality boundaries.
  - **Product**
    - Give insight into internal characteristics of the product such as appropriateness of analysis, design, and code models, the effectiveness of test cases, and the overall product quality.

Software Metrics

- **Measure** - a datum that is a quantification of a software attribute
- **Measurement** - the collection of one or more measures
- **Metric** - a relation of the individual measures in a meaningful way
- **Indicator** - a metric or combination of metrics that provide insight which enables process, project, or product improvement.

- **Example**:
  - Measures = # tokens in a statement, # of conditions in an IF, level of nesting
  - Metric = complexity

[Pressman 4th Ed]

Software Metrics

- A measurement process
  - Derive and formulate appropriate metrics.
  - Collect the necessary data.
  - Compute the metrics.
  - Interpret the metrics.
  - Evaluate the product in light of the metrics.

Software Quality Metrics

- In any assessment of software quality, some form of measurement must occur.
- The measurement may be
  - Direct (errors per KLOC)
  - Indirect (usability)
- Various taxonomies of "quality factors" have been proposed:
  - McCall, et al.
  - FURPS (Functionality, Usability, Reliability, Performance, Supportability)
- No matter the taxonomy or method of measurement, no real measurement of quality ever occurs; only ***surrogates*** can ever be measured.
- A fundamental problem is identifying appropriate surrogates to serve as indicators of software quality.

A Few Measures and Metrics

- Lines of code (LOC)
- Function Points (FP)
- Reliability Metrics
- Complexity Metrics
  - Halstead Metrics
  - McCabe Metrics
  - Complexity Profile Graph

Lines of Code (LOC)

- Direct measurement
- Can be used as a productivity indicator (e.g. KLOC per person)
- Can be used as the basis of quality indicators (e.g. errors per KLOC)
- Positive
  - Easily measured and computed.
  - Guaranteed measurable for all programs.
- Negative
  - What to count? Is this count language-independent?
  - Better suited to procedural languages than non-procedural ones.
  - Can it devalue shorter, but better-designed programs?

A Partial List of Size Metrics

- number of lines in the source file
- number of language statements in the source file
- number of semicolons in the source file
- Halstead's length, vocabulary, and volume
- number of bytes in the source file
- number of bytes in the object file
- number of machine code instructions
- number of comments
- number of nodes in the parse tree
- length of longest branch in the parse tree

Function Points (FP)

- Subjective, indirect measure
- To be measured early in the life cycle (e.g. during requirements analysis), but can be measured at various points.
- Measures the functionality of software, with the intent of estimating a project's size (e.g., Total FP) and monitoring a project's productivity (e.g., Cost per FP, FP per person-month)
- Developed at IBM and rooted in classic information systems applications
- Software Productivity Research, Inc. (SPR) developed a FP superset known as "Feature Points" to incorporate software that is high in algorithmic complexity but low in input/output.
- A program's FP metric is computed based on the program's information domain and functionality complexity, with empirically-derived weighting factors.

Function Points

- The FP metric is computed by considering five factors which directly impact the visible, external aspects of software:
  - Inputs to the application
  - Outputs generated by the application
  - User inquiries
  - Data files to be accessed by the application
  - Interfaces to other applications
- Initial trial and error produced empirical weights for each of the five items along with a complexity adjustment for the overall application.
  - The weights reflect the approximate difficulty associated with implementing each of the five factors.
  - The complexity adjustment reflects the approximate overall level of complexity of the application (e.g.: Is distributed processing involved? Is data communications involved? etc.)

FP Counting Method (Original)

		Complexity Multipliers			
	Count	Low	Average	High	FP
Inputs		3	4	6	
Outputs		4	5	7	
Inquiries		3	4	6	
Files		7	10	15	
Interfaces		5	7	10	
				TOTAL A	

0..5 where 0 = no effect, 5 = essential

Complexity Adjustment	
Backup and recovery	
Data communications	
Distributed processing	
Critical performance	
Heavily used operational environment	
Online data entry	
Transaction complexity	
Online master file updates	
Complex external processing	
Complex internal processing	
Reusability	
Conversion and installation	
Multiple sites	
Change facilitation	
TOTAL B	

Adjusted Total Function Points (AFP)
AFP = TOTAL A * (0.65 + 0.01*TOTAL B)

FP Counting Example

		Complexity Multipliers			
	Count	Low	Average	High	FP
Inputs	7		4		28
Outputs	10	4			40
Inquiries	6	3			18
Files	17	7			119
Interfaces	4		7		28
				TOTAL A	233

0..5 where 0 = no effect, 5 = essential

	Complexity Adjustment
Backup and recovery	4
Data communications	5
Distributed processing	1
Critical performance	1
Heavily used operational environment	4
Online data entry	5
Transaction complexity	1
Online master file updates	5
Complex external processing	1
Complex internal processing	1
Reusability	1
Conversion and installation	1
Multiple sites	1
Change facilitation	1
TOTAL B	32

Adjusted Total Function Points (AFP) **226**

AFP = TOTAL A * (0.65 + 0.01*TOTAL B)

Complexity Metrics

- Not a measure of computational complexity
- Measures psychological complexity, specifically structural complexity; that is, the complexity that arises from the structure of the software itself, independent of any cognitive issues.
- Many complexity metrics exist: H. Zuse lists over 200 in his 1990 taxonomy.
- Complexity metrics can be broadly categorized according to the fundamental software attribute measures on which they are based:
  - software science parameters
  - control-flow
  - data-flow
  - information-flow
  - hybrid

Halstead Metrics

- Software Science is generally agreed to be the beginning of systematic research on metrics as predictors for qualitative attributes of software.
- Proposed by Maurice Halstead in 1972 as a mixture of information theory, psychology, and common sense.
- These are ***linguistic metrics***.
- Based on four measures of two fundamental software attributes, operators and operands:
  - n_1 - number of unique operators
  - n_2 - number of unique operands
  - N_1 - total number of operators
  - N_2 - total number of operands

Halstead Metrics

- Halstead conjectures relationships between these fundamental measures and a variety of qualitative attributes:
  - Length: $N = N_1 + N_2$
  - Vocabulary: $n = n_1 + n_2$
  - Volume: $V = N*\log_2(n)$
  - Level: $L = (2*n_2)/(n_1 N_2)$
  - Difficulty: $D = 1/L$
  - Effort: $E = V * D$
  - Bugs: $B = (E ** (2/3)) / 3000$
- Halstead also defines a number of other attributes:
  - potential volume, intelligence content, program purity, language level, predicted number of bugs, predicted number of seconds required for implementation

Halstead Metrics

- Extensive experiments involving Halstead metrics have been done and the metrics generally hold up well.
  - Even the bug prediction metric has been supported: A study of various programs ranging in size from 300 to 12,000 executable statements suggested that the bug prediction metric was accurate to within 8%. [Lipow, M. IEEE TSE, 8:437-439(1982)]
- Generally used as maintenance metrics.
- A few caveats:
  - Operator/Operand ambiguity
    - Is code always code and data always data?
  - Operator types
    - Some control structures are inherently more complex than others.
  - Level of nesting
    - Nesting adds complexity to code.

McCabe Metrics

- Tom McCabe was the first to propose that complexity depends only on the decision structure of a program and is therefore derivable from a control flow graph.
- In this context, complexity is a synonym for testability and structuredness. McCabe's premise is that the complexity of a program is related to the difficulty of performing path testing.
- These are ***structural metrics***.
- McCabe metrics are a family of related metrics including:
  - Cyclomatic Complexity
  - Essential Complexity
  - Module Design Complexity
  - Design Complexity
  - Pathological Complexity

Cyclomatic Complexity

- Cyclomatic Complexity, v(G), is a measure of the amount of control structure or decision logic in a program.
- Studies have shown a high correlation between v(G) and the occurrence of errors and it has become a widely accepted indicator of software quality.
- Based on the flowgraph representation of code:
  - Nodes - representing one or more procedural statements
  - Edges - the arrows represent flow of control
  - Regions - areas bounded by edges and nodes; includes the area outside the graph
- Cyclomatic Complexity is generally computed as:
  - v(G) = number of regions in the flowgraph
  - v(G) = number of conditions in the flowgraph + 1
  - v(G) = number of edges - number of nodes + 2

Cyclomatic Complexity

- Cyclomatic complexity can be used to
  - Determine the maximum number of test cases to ensure that all <u>independent paths</u> through the code have been tested.
  - Ensure the code covers all the decisions and control points in the design.
  - Determine when modularization can decrease overall complexity.
  - Determine when modules are likely to be too buggy.

Cyclomatic Complexity Thresholds

- 1-10
  - A simple module, without much risk
- 11-20
  - More complex, moderate risk
- 21-50
  - Complex, high risk
- Greater than 50
  - Untestable, very high risk (however, there are exceptions)

[From SEI reports]

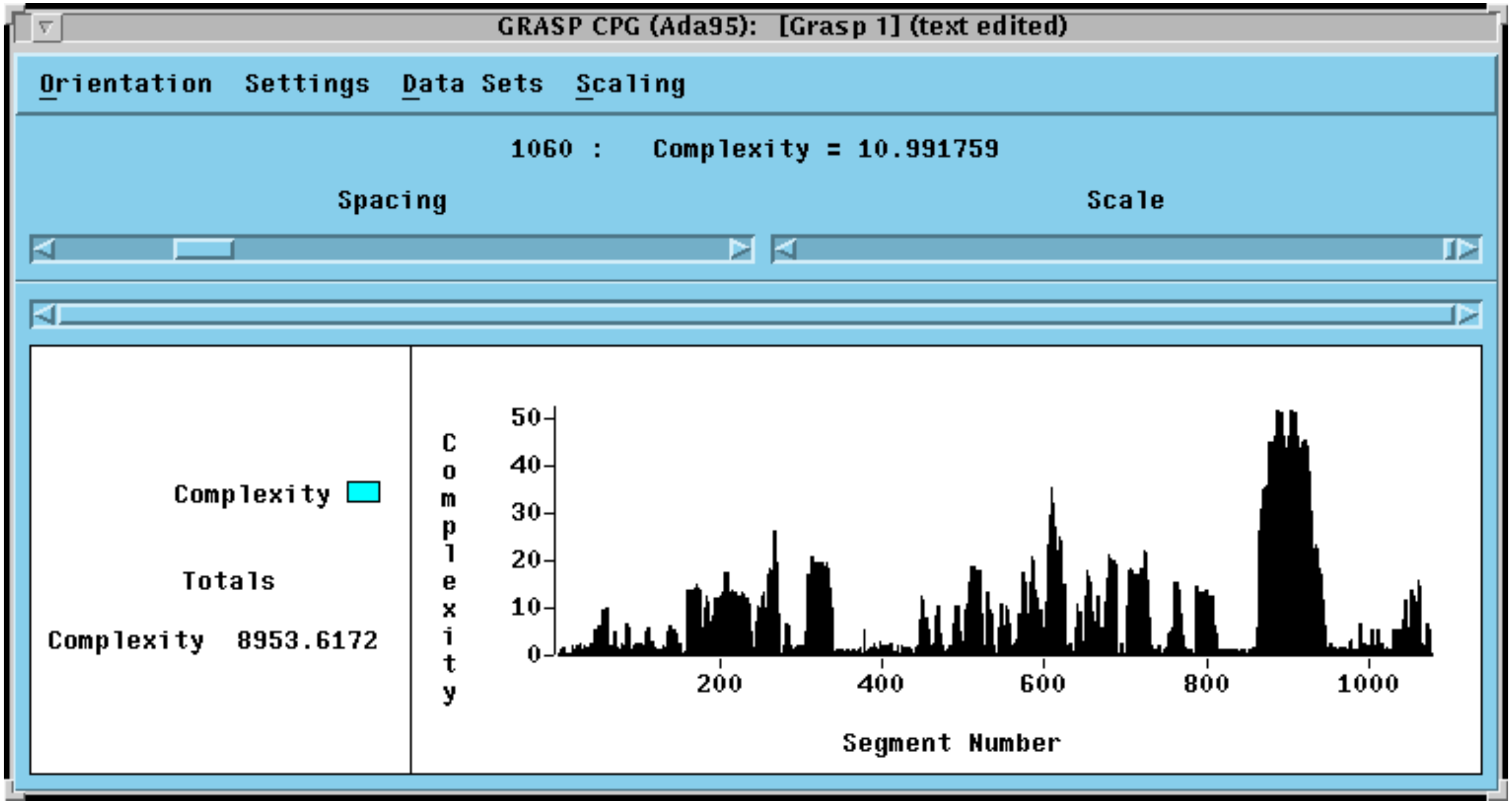
Complexity Profile Graph

- Algorithmic level graph of complexity profile
- Fine-grained metric
  - for each production in the grammar
- Profile of program unit rather than single-value metric
- Complexity values from each ***measurable unit*** in a program unit are displayed as a set to form the complexity profile graph.
- Adds the advantages of visualization to complexity measurement.

Complexity Profile Graph

- A program unit is parsed and divided into ***segments***
  - e.g., each simple declaration or statement is a single segment, composite statements are divided into multiple segments
- Each segment is a measurable unit.
- Segments are non-overlapping and all code is covered
  - i.e., all tokens are included in exactly one segment
- The complexity for each segment is a bar in the CPG.

Complexity Profile Graph



Computing the CPG

- **Content**
  - $C = \ln(\text{reserved words} + \text{operators} + \text{operands})$
- **Breadth**
  - B = number of statements within a construct
- **Reachability**
  - R = 1 + number of operators in predicate path
- **Inherent**
  - I = assigned value based on type of control structure
- **Total**
  - $T = s_1C + s_2B + s_3R + s_4I$
  - where s_1, s_2, s_3, s_4 are scaling factors

Maintainability Index

- Quantitative measurement of an operational system's maintainability, developed by industry (Hewlett-Packard, and others) and research groups (Software Engineering Test Laboratory at University of Idaho, and others).
- A combination of Halstead metrics, McCabe metrics, LOC, and comment measures.
- MI formula calibrated and validated with actual industrial systems.
- Used as both an instantaneous metric as well as a predictor of maintainability over time.
- MI measurement applied during software development can help reduce lifecycle costs.

[From SEI reports]

MI Formula

171 – 5.2*ln(aveV) – 0.23*aveV(g') - 16.2*ln(aveLOC) – 50*sin(sqrt(2.4*perCM))

Where:

aveV = average Halstead volume V per module
aveV(g') = average cyclomatic complexity per module
aveLOC = average LOC per module
perCM = average percent of comment lines per module

[From SEI reports]

Using the MI

- Systems can be checked periodically for maintainability.
- MI can be integrated into development to evaluate code quality as it is being built.
- MI can be used to assess modules for risk as candiates for modification.
- MI can be used to compare systems with each other.

[From SEI reports]