Software Product Metrics

- Software quality
- A framework for product metrics
- A product metrics taxonomy
- Metrics for the analysis model
- Metrics for the design model
- Metrics for maintenance

(Source: Pressman, R. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2005)

Examples of Metrics from Everyday Life

- Working and living
  - Cost of utilities for the month
  - Cost of groceries for the month
  - Amount of monthly rent per month
  - Time spent at work each Saturday for the past month
  - Time spent mowing the lawn for the past two times
- College experience
  - Grades received in class last semester
  - Number of classes taken each semester
  - Amount of time spent in class this week
  - Amount of time spent on studying and homework this week
  - Number of hours of sleep last night
- Travel
  - Time to drive from home to the airport
  - Amount of miles traveled today
  - Cost of meals and lodging for yesterday

Why have Software Product Metrics?

- Help software engineers to better understand the attributes of models and assess the quality of the software
- Help software engineers to gain insight into the design and construction of the software
- Focus on specific attributes of software engineering work products resulting from analysis, design, coding, and testing
- Provide a systematic way to assess quality based on a set of clearly defined rules
- Provide an “on-the-spot” rather than “after-the-fact” insight into the software development

Software Quality

Software Quality Defined

- Definition:

  Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software

- Three important points in this definition
  - Explicit software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality
  - Specific standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will most surely result
  - There is a set of implicit requirements that often goes unmentioned (e.g., ease of use). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect

Properties of Software Quality Factors

- Some factors can be directly measured (e.g. defects uncovered during testing)
- Other factors can be measured only indirectly (e.g., usability or maintainability)
- Software quality factors can focus on three important aspects
  - Product operation: Its operational characteristics
  - Product revision: Its ability to undergo change
  - Product transition: Its adaptability to new environments

ISO 9126 Software Quality Factors

- Functionality
  - The degree to which the software satisfies stated needs
- Reliability
  - The amount of time that the software is available for use
- Usability
  - The degree to which the software is easy to use
- Efficiency
  - The degree to which the software makes optimal use of system resources
- Maintainability
  - The ease with which repair and enhancement may be made to the software
- Portability
  - The ease with which the software can be transposed from one environment to another

A Framework for Product Metrics

Measures, Metrics, and Indicators

- These three terms are often used interchangeably, but they can have subtle differences
- Measure
  - Provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process
- Measurement
  - The act of determining a measure
- Metric
  - (IEEE) A quantitative measure of the degree to which a system, component, or process possesses a given attribute
- Indicator
  - A metric or combination of metrics that provides insight into the software process, a software project, or the product itself

Purpose of Product Metrics

- Aid in the evaluation of analysis and design models
- Provide an indication of the complexity of procedural designs and source code
- Facilitate the design of more effective testing techniques
- Assess the stability of a fielded software product

Activities of a Measurement Process

- Formulation
  - The derivation (i.e., identification) of software measures and metrics appropriate for the representation of the software that is being considered
- Collection
  - The mechanism used to accumulate data required to derive the formulated metrics
- Analysis
  - The computation of metrics and the application of mathematical tools
- Interpretation
  - The evaluation of metrics in an effort to gain insight into the quality of the representation
- Feedback
  - Recommendations derived from the interpretation of product metrics and passed on to the software development team

Characterizing and Validating Metrics

- A metric should have desirable mathematical properties
  - It should have a meaningful range (e.g., zero to ten)
  - It should not be set on a rational scale if it is composed of components measured on an ordinal scale
- If a metric represents a software characteristic that increases when positive traits occur or decreases when undesirable traits are encountered, the value of the metric should increase or decrease in the same manner
- Each metric should be validated empirically in a wide variety of contexts before being published or used to make decisions
  - It should measure the factor of interest independently of other factors
  - It should scale up to large systems
  - It should work in a variety of programming languages and system domains

Collection and Analysis Guidelines

- Whenever possible, data collection and analysis should be automated
- Valid statistical techniques should be applied to establish relationships between internal product attributes and external quality characteristics
- Interpretative guidelines and recommendations should be established for each metric

Goal-oriented Software Measurement

- Goal/Question/Metric (GQM) paradigm
- GQM technique identifies meaningful metrics for any part of the software process
- GQM emphasizes the need to
  - Establish an explicit measurement goal that is specific to the process activity or product characteristic that is to be assessed
  - Define a set of questions that must be answered in order to achieve the goal
  - Identify well-formulated metrics that help to answer these questions
- GQM utilizes a goal definition template to define each measurement goal

(More on next slide)

Goal-oriented Software Measurement (continued)

- Example use of goal definition template

  Analyze the SafeHome software architecture for the purpose of evaluating architecture components. Do this with respect to the ability to make SafeHome more extensible from the viewpoint of the software engineers, who are performing the work in the context of product enhancement over the next three years.

- Example questions for this goal definition

  1) Are architectural components characterized in a manner that compartmentalizes function and related data?
  2) Is the complexity of each component within bounds that will facilitate modification and extension?

Attributes of Effective Software Metrics

- Simple and computable
  - It should be relatively easy to learn how to derive the metric, and its computation should not demand inordinate effort or time
- Empirically(by means of observation or experience rather than theory or pure logic) and intuitively persuasive
  - The metric should satisfy the engineer's intuitive notions about the product attribute under consideration
- Consistent and objective
  - The metric should always yield results that are unambiguous

(More on next slide)

Attributes of Effective Software Metrics (continued)

- Consistent in the use of units and dimensions
  - The mathematical computation of the metric should use measures that do not lead to bizarre(very strange or unusual) combinations of units
- Programming language independent
  - Metrics should be based on the analysis model, the design model, or the structure of the program itself
- An effective mechanism for high-quality feedback
  - The metric should lead to a higher-quality end product

A Product Metrics Taxonomy

Metrics for the Analysis Model

- Functionality delivered
  - Provides an indirect measure of the functionality that is packaged within the software
- System size
  - Measures the overall size of the system defined in terms of information available as part of the analysis model
- Specification quality
  - Provides an indication of the specificity and completeness of a requirements specification

Metrics for the Design Model

- Architectural metrics
  - Provide an indication of the quality of the architectural design
- Component-level metrics
  - Measure the complexity of software components and other characteristics that have a bearing on quality
- Interface design metrics
  - Focus primarily on usability
- Specialized object-oriented design metrics
  - Measure characteristics of classes and their communication and collaboration characteristics

Metrics for Source Code

- Complexity metrics
  - Measure the logical complexity of source code (can also be applied to component-level design)
- Length metrics
  - Provide an indication of the size of the software

"These metrics can be used to assess source code complexity, maintainability, and testability, among other characteristics"

Metrics for Testing

- Statement and branch coverage metrics
  - Lead to the design of test cases that provide program coverage
- Defect-related metrics
  - Focus on defects (i.e., bugs) found, rather than on the tests themselves
- Testing effectiveness metrics
  - Provide a real-time indication of the effectiveness of tests that have been conducted
- In-process metrics
  - Process related metrics that can be determined as testing is conducted

Metrics for the Analysis Model

Function Points

Introduction to Function Points

- First proposed by Albrecht in 1979; hundreds of books and papers have been written on functions points since then
- Can be used effectively as a means for measuring the functionality delivered by a system
- Using historical data, function points can be used to
  - Estimate the cost or effort required to design, code, and test the software
  - Predict the number of errors that will be encountered during testing
  - Forecast the number of components and/or the number of projected source code lines in the implemented system
- Derived using an empirical relationship based on
  1) Countable (direct) measures of the software's information domain
  2) Assessments of the software's complexity

Information Domain Values

- Number of external inputs
  - Each external input originates from a user or is transmitted from another application
  - They provide distinct application-oriented data or control information
  - They are often used to update internal logical files
  - They are not inquiries (those are counted under another category)
- Number of external outputs
  - Each external output is derived within the application and provides information to the user
  - This refers to reports, screens, error messages, etc.
  - Individual data items within a report or screen are not counted separately

(More on next slide)

Information Domain Values (continued)

- Number of external inquiries
  - An external inquiry is defined as an online input that results in the generation of some immediate software response
  - The response is in the form of an on-line output
- Number of internal logical files
  - Each internal logical file is a logical grouping of data that resides within the application's boundary and is maintained via external inputs
- Number of external interface files
  - Each external interface file is a logical grouping of data that resides external to the application but provides data that may be of use to the application

Function Point Computation

1) Identify/collect the information domain values
2) Complete the table shown below to get the count total
   - Associate a weighting factor (i.e., complexity value) with each count based on criteria established by the software development organization
3) Evaluate and sum up the adjustment factors (see the next two slides)
   - "F_i" refers to 14 value adjustment factors, with each ranging in value from 0 (not important) to 5 (absolutely essential)
4) Compute the number of function points (FP)

$$FP = \text{count total} * [0.65 + 0.01 * \text{sum}(F_i)]$$

Information		**Weighting Factor**			
Domain Value	**Count**	**Simple**	**Average**	**Complex**	
External Inputs	_____ x	3	4	6	= _____
External Outputs	_____ x	4	5	7	= _____
External Inquiries	_____ x	3	4	6	= _____
Internal Logical Files	_____ x	7	10	15	= _____
External Interface Files	_____ x	5	7	10	= _____
Count total					_____

Value Adjustment Factors

1) Does the system require reliable backup and recovery?
2) Are specialized data communications required to transfer information to or from the application?
3) Are there distributed processing functions?
4) Is performance critical?
5) Will the system run in an existing, heavily utilized operational environment?
6) Does the system require on-line data entry?
7) Does the on-line data entry require the input transaction to be built over multiple screens or operations?

(More on next slide)

Value Adjustment Factors (continued)

8) Are the internal logical files updated on-line?
9) Are the inputs, outputs, files, or inquiries complex?
10) Is the internal processing complex?
11) Is the code designed to be reusable?
12) Are conversion and installation included in the design?
13) Is the system designed for multiple installations in different organizations?
14) Is the application designed to facilitate change and for ease of use by the user?

Function Point Example

Information Domain Value	**Count**		**Weighting Factor**			
			Simple	**Average**	**Complex**	
External Inputs	3	x	3	4	6	= 9
External Outputs	2	x	4	5	7	= 8
External Inquiries	2	x	3	4	6	= 6
Internal Logical Files	1	x	7	10	15	= 7
External Interface Files	4	x	5	7	10	= 20
Count total						**50**

- $FP = count\ total * [0.65 + 0.01 * sum(F_i)]$
- $FP = 50 * [0.65 + (0.01 * 46)]$
- $FP = 55.5$ (rounded up to 56)

Interpretation of the FP Number

- Assume that past project data for a software development group indicates that
  - One FP translates into 60 lines of object-oriented source code
  - 12 FPs are produced for each person-month of effort
  - An average of three errors per function point are found during analysis and design reviews
  - An average of four errors per function point are found during unit and integration testing
- This data can help project managers revise their earlier estimates
- This data can also help software engineers estimate the overall implementation size of their code and assess the completeness of their review and testing activities

Metrics for the Design Model

Architectural Design Metrics

- These metrics place emphasis on the architectural structure and effectiveness of modules or components within the architecture
- They are “black box” in that they do not require any knowledge of the inner workings of a particular software component

Hierarchical Architecture Metrics

- Fan out: the number of modules immediately subordinate to the module *i*, that is, the number of modules directly invoked by module *i*
- Structural complexity
  - $S(i) = f^2_{out}(i)$, where $f_{out}(i)$ is the "fan out" of module *i*
- Data complexity
  - $D(i) = v(i)/[f_{out}(i) + 1]$, where v(*i*) is the number of input and output variables that are passed to and from module *i*
- System complexity
  - $C(i) = S(i) + D(i)$
- As each of these complexity values increases, the overall architectural complexity of the system also increases
- This leads to greater likelihood that the integration and testing effort will also increase

Hierarchical Architecture Metrics (continued)

- Shape complexity
  - `size = n + a`, where `n` is the number of nodes and `a` is the number of arcs
  - Allows different program software architectures to be compared in a straightforward manner
- Connectivity density (i.e., the arc-to-node ratio)
  - `r = a/n`
  - May provide a simple indication of the coupling in the software architecture

Metrics for Object-Oriented Design

- Size
  - Population: a static count of all classes and methods
  - Volume: a dynamic count of all instantiated objects at a given time
  - Length: the depth of an inheritance tree
- Coupling
  - The number of collaborations between classes or the number of methods called between objects
- Cohesion
  - The cohesion of a class is the degree to which its set of properties is part of the problem or design domain
- Primitiveness
  - The degree to which a method in a class is atomic (i.e., the method cannot be constructed out of a sequence of other methods provided by the class)

Specific Class-oriented Metrics

- Weighted methods per class
  - The normalized complexity of the methods in a class
  - Indicates the amount of effort to implement and test a class
- Depth of the inheritance tree
  - The maximum length from the derived class (the node) to the base class (the root)
  - Indicates the potential difficulties when attempting to predict the behavior of a class because of the number of inherited methods
- Number of children (i.e., subclasses)
  - As the number of children of a class grows
    - Reuse increases
    - The abstraction represented by the parent class can be diluted by inappropriate children
    - The amount of testing required will increase

(More on next slide)

Specific Class-oriented Metrics (continued)

- Coupling between object classes
  - Measures the number of collaborations a class has with any other classes
  - Higher coupling decreases the reusability of a class
  - Higher coupling complicates modifications and testing
  - Coupling should be kept as low as possible
- Response for a class
  - This is the set of methods that can potentially be executed in a class in response to a public method call from outside the class
  - As the response value increases, the effort required for testing also increases as does the overall design complexity of the class
- Lack of cohesion in methods
  - This measures the number of methods that access one or more of the same instance variables (i.e., attributes) of a class
  - If no methods access the same attribute, then the measure is zero
  - As the measure increases, methods become more coupled to one another via attributes, thereby increasing the complexity of the class design

Metrics for Maintenance

Metrics for Maintenance

- Software maturity index (SMI)
  - Provides an indication of the stability of a software product based on changes that occur for each release
- $SMI = [M_T - (F_a + F_c + F_d)]/M_T$
  where
  M_T = #modules in the current release
  F_a = #modules in the current release that have been added
  F_c = #modules in the current release that have been changed
  F_d = #modules from the preceding release that were deleted in the current release
- As the SMI (i.e., the fraction) approaches 1.0, the software product begins to stabilize
- The average time to produce a release of a software product can be correlated with the SMI