Product metrics

Software quality,
metrics for analysis model,
metrics for design model,
metrics for source code,
metrics for testing,
metrics for maintenance.

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 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

- 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 Domain Value	Count		Weighting Factor: 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?

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 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

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

- 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

- 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 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 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