Software Measurement:
Product, Process and Project Metrics

- **Quality and Project Management: Chapters 30 & 32**

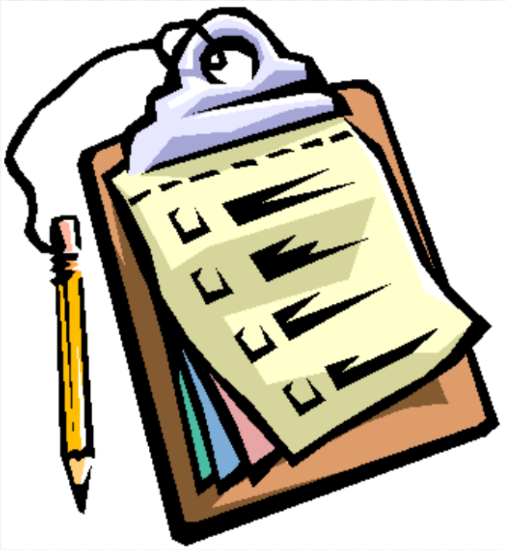
Slide Set to accompany

Software Engineering: A Practitioner's Approach, 8/e

by Roger S. Pressman and Bruce R. Maxim

Agenda

- Product Metrics
- Process and Project Metrics
- Software Quality Metrics



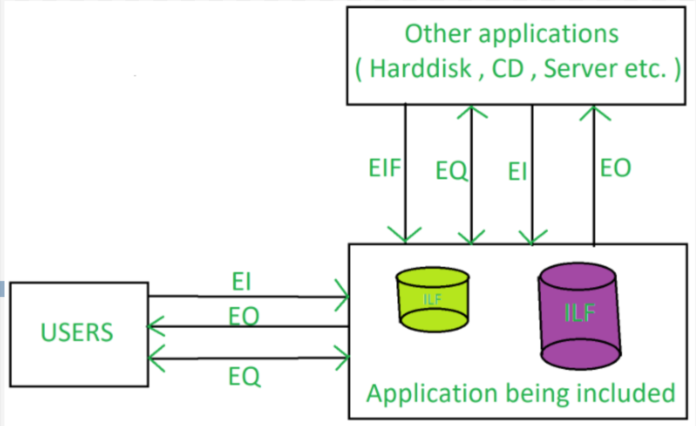
Product Metrics

Metrics for the Requirements Model

- Technical work in software engineering begins with the creation of the requirements model
- Product metrics that provide insight into the quality of the analysis model are desirable

1. Function-based metrics
   - The function point metric (FP) is used as a means for **measuring the functionality** delivered by a system
   - FP measures software size (functional sizing)
2. Specification metrics

Function Point (FP)



- Types of FP
  - Transaction Functions:
  - made up of the processes that are **exchanged** between the user or external applications and the application being measured
    - External Inputs (EI) → Input screen and tables
    - External Outputs (EO) → Output screen and reports
    - External Inquiries (EQ) → Queries and interrupts
  - Data Functions:
  - made up of internal and external **resources** that affect the system
    - Internal Logical Files (ILF) → Databases and directories
    - External Interface Files (EIF) → Shared databases and routines

Transactional FP

- External Input (EI):
  - A transaction function in which data goes “into” the application from outside the boundary to inside
    - Data may come from a data input screen or another application
    - Data can be either control or business information
- External Outputs (EO):
  - A transaction function in which data comes “out” of the system
    - Reports or output files sent to other applications
- External Inquiries (EQ):
  - A transaction function with both input and output components that result in data retrieval

Data FP

- Internal Logical Files (ILF)
  - A group of logically related data or control information that resides entirely within the application boundary
  - The primary intent of an ILF is to hold data required by one or more processes of the application
- External Interface Files (EIF)
  - A group of logically related data or control information that is used by the application
  - The data resides entirely outside the application boundary and is maintained in an ILF by another application
  - An interface has to be developed to get the data from the file

Computing Function Points-I

Information Domain Value	Count	Weighting factor Simple	Average	Complex		
External Inputs (EIs)		3	4	6	=	
External Outputs (EOs)		4	5	7	=	
External Inquiries (EQs)		3	4	6	=	
Internal Logical Files (ILFs)		7	10	15	=	
External Interface Files (EIFs)		5	7	10	=	
Count total						

- Organizations that use FP methods develop criteria for determining whether a particular entry is simple, average, or complex
- The determination of complexity is somewhat subjective

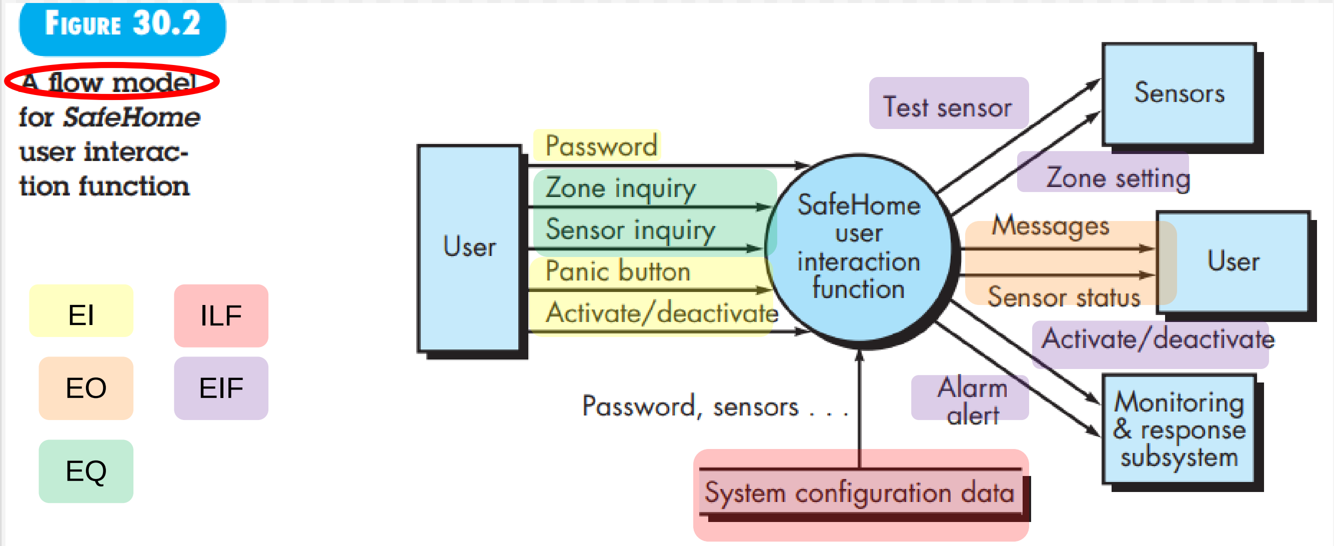
Computing Function Points-II

- To compute function points (FP):

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

- count total is the sum of all FP entries
- The Fi (i = 1 to 14) are ***value adjustment factors*** (VAF) based on responses to 14 questions [Lon02] (see page 660 of 8th edition)
  - For example: Does the system require reliable backup and recovery?
  - Each question is answered using an ordinal scale that ranges from 0 (not important or applicable) to 5 (absolutely essential)
- Based on the projected FP value derived from the requirements model, the project team can estimate the overall implementation size of the system
  - Assume that past projects have found that one FP translates into 60 lines of code (with an object-oriented language)
  - These historical data provide the project manager with important planning information that is **based on the requirements model rather than preliminary estimates**

An Example of Computing FP-I



- Consider the flow diagram for a user interaction function within SafeHome software
- The function manages user interaction, accepting a user password to activate or deactivate the system, and allows inquiries on the status of security zones and various security sensors.
- The function displays a series of prompting messages and sends appropriate control signals to various components of the security system

An Example of Computing FP-II

Information Domain Value	Count		Weighting factor Simple	Average	Complex		
External Inputs (EIs)	3		3	4	6	=	9
External Outputs (EOs)	2		4	5	7	=	8
External Inquiries (EQs)	2		3	4	6	=	6
Internal Logical Files (ILFs)	1		7	10	15	=	7
External Interface Files (EIFs)	4		5	7	10	=	20
Count total						→	50

- For the purposes of this example, we assume that $\sum(F_i)$ is 46 (a moderately complex product)
- Therefore:

$$FP = 50 \times [0.65 + (0.01 \times 46)] = 56$$

Specification-Based Metrics

- Davis proposes a list of characteristics to assess the quality of the requirements specification [Dav93]:
  - Specificity (lack of ambiguity)
  - Completeness
  - Correctness
  - Understandability
  - Verifiability
  - Internal and external consistency
  - Achievability
  - Concision
  - Traceability
  - Modifiability
  - Precision
  - Reusability

Specifity

- Assume that there are n_r requirements in a specification
- To determine the specificity of requirements, Davis suggests a metric that is based on the consistency of the reviewers' interpretation of each requirement:

$$Q_1 = \frac{n_{ui}}{n_r}$$

- where n_{ui} is the number of requirements for which all reviewers had identical interpretations
- The closer the value of Q to 1, the lower is the ambiguity of the specification

Architectural Design Metrics

- Architectural design complexity metrics
  - Structural complexity = g(fan-out)
  - Data complexity = f(input & output variables, fan-out)
  - System complexity = h(structural & data complexity)
- Morphology (i.e., shape) metrics: a function of the number of modules and the number of interfaces between modules

Structural Complexity

- For hierarchical architectures (e.g., call-and-return architectures), structural complexity of a module *i* is defined as:

$$S(i) = f^2_{\mathrm{out}}(i)$$

- where $f_{out}(i)$ is the fan-out of module *i*

Data Complexity

- Provides an indication of the complexity in the internal interface for a module ***i*** and is defined as:

$$D(i) = \frac{v(i)}{[f_{out}(i) + 1]}$$

- where *v*(*i*) is the number of input and output variables that are passed to and from module *i*

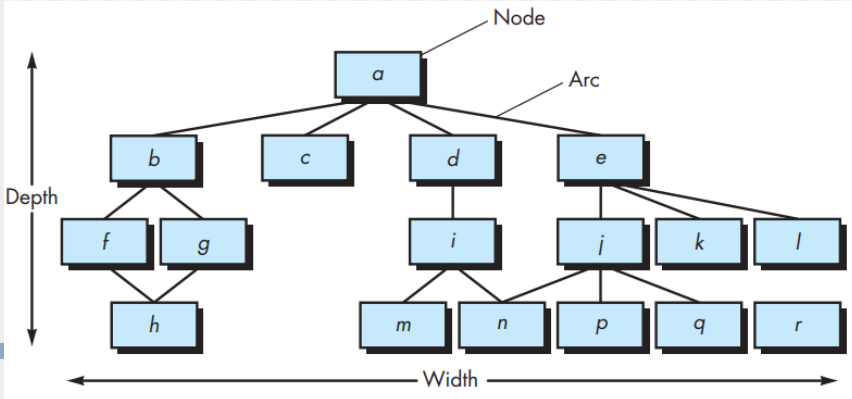
System Complexity

- The sum of structural and data complexity, specified as:

$$C(i) = S(i) + D(i)$$

- As each of these complexity values increases, the overall architectural complexity of the system also increases
- This leads to a greater likelihood that integration and testing effort will also increase

Morphology Metrics



- Referring to the call-and-return architecture:

$$\text{Size} = n + a$$

- n is the number of nodes and a is the number of arcs
- In this example:
  - Size = 17 + 18 = 35
- Depth = longest path from the root (top) node to a leaf
  - For the above architecture, depth = 4
- Width = maximum number of nodes at any one level of the architecture
  - For the above architecture, width = 6
- The arc-to-node ratio, r = a/n, measures the connectivity density of the architecture and may provide a simple indication of the coupling of the architecture.

Metrics for OO Design-I

- Whitmire [Whi97] describes nine distinct and measurable characteristics of an OO design:
  - Size
    - A static count of OO entities such as classes or operations, coupled with the depth of an inheritance tree
  - Complexity
    - How classes of an OO design are interrelated to one another
  - Coupling
    - Counting the physical connections between elements of the OO design
    - e.g., the number of messages passed between objects
  - Sufficiency
    - “the degree to which an abstraction [class] possesses the features required of it…”
  - Completeness
    - Whether a class delivers the set of properties that fully reflect the needs of the problem domain

Metrics for OO Design-II

- Cohesion
  - The degree to which all operations working together to achieve a single, well-defined purpose
- Primitiveness
  - The degree to which an operation is atomic
- Similarity
  - The degree to which two or more classes are similar in terms of their structure, function, behavior, or purpose
- Volatility
  - Measures the likelihood that a change will occur

Class-Oriented Metrics

- The **CK** metrics suite
  - The most widely referenced sets of OO metrics
  - proposed by **C**hidamber and **K**emerer [Chi94]:

1. weighted methods per class
2. depth of the inheritance tree
3. number of children
4. coupling between object classes
5. response for a class
6. lack of cohesion in methods

Weighted Methods per Class (WMC)

- Assume that n methods of complexity c1, c2, …, cn are defined for a class C
- The specific complexity metric that is chosen (e.g., cyclomatic complexity) should be normalized so that nominal complexity for a method takes on a value of 1.0

$$\mathrm{WMC} = \Sigma\, c_i$$

- The number of methods and their complexity indicates the amount of effort required to implement and test a class
- The larger the number of methods, the more complex is the inheritance tree
- As the number of methods grows for a given class, it is likely to become more and more application specific, thereby limiting potential reuse
- For all of these reasons, WMC should be kept as low as possible

Depth of the Inheritance Tree (DIT)

- The maximum length from the node to the root of the tree
- As DIT grows, it is likely that lower-level classes will inherit many methods
  - Leading to potential difficulties when attempting to predict the behavior of a class
- A deep class hierarchy (DIT is large) also leads to greater design complexity
- On the positive side, large DIT values imply that many methods may be reused

Number Of Children (NOC)

- The subclasses that are immediately subordinate to a class in the class hierarchy
- As NOC increases,
  - the abstraction represented by the parent class can be diluted if some of the children are not appropriate members of the parent class
  - the amount of testing also increases (required to exercise each child in its operational context)

Coupling Between Object classes (CBO)

- The number of collaborations listed for a class on its CRC index card
- As CBO increases,
  - it is likely that the reusability of a class will decrease
  - modifications and testing are also complicated
- CBO for each class should be kept as low as is reasonable

Response For a Class (RFC)

- The response set of a class:
  - “a set of methods that are executed in response to a message received by an object of that class”
- RFC is the number of methods in the response set
- As RFC increases, the effort required for testing also increases
  - because the test sequence grows
- As RFC increases, the overall design complexity of the class increases

Lack of Cohesion in Methods (LCOM)

- Each method within a class accesses one or more attributes
- LCOM: The number of methods that access one or more of the same attributes
- If no methods access the same attributes → LCOM = 0
- Consider a class with six methods. Four of the methods have one or more attributes in common (i.e., they access common attributes) → LCOM = 4
- If LCOM is high, methods may be coupled to one another via attributes
  - This increases the complexity of the class design
- Although there are cases in which a high LCOM is justifiable, it is desirable to keep cohesion high
  - Keep LCOM low

User Interface Design Metrics

- Significant literature on the design of UI
- But little information on quality metrics
- In the following, we present some design metrics that may have application for:
  - websites, browser-based applications, and mobile applications
  - Many of these metrics are applicable to all user interfaces
- UI design metrics:
  1. Interface Metrics
  2. Aesthetic (Graphic Design) Metrics
  3. Content Metrics
  4. Navigation Metrics

Interface Metrics

Suggested Metric	Description
Layout appropriateness	The relative position of entities within the interface
Layout complexity	Number of distinct regions[9] defined for an interface
Layout region complexity	Average number of distinct links per region
Recognition complexity	Average number of distinct items the user must look at before making a navigation or data input decision
Recognition time	Average time (in seconds) that it takes a user to select the appropriate action for a given task
Typing effort	Average number of keystrokes required for a specific function
Mouse pick effort	Average number of mouse picks per function
Selection complexity	Average number of links that can be selected per page
Content acquisition time	Average number of words of text per Web page
Memory load	Average number of distinct data items that the user must remember to achieve a specific objective

Aesthetic (Graphic Design) Metrics

Suggested Metric	Description
Word count	Total number of words that appear on a page
Body text percentage	Percentage of words that are body versus display text (e.g., headers)
Emphasized body text percentage	Portion of body text that is emphasized (e.g., bold, capitalized)
Text positioning count	Changes in text position from flush left
Text cluster count	Text areas highlighted with color, bordered regions, rules, or lists
Link count	Total links on a page
Page size	Total bytes for the page as well as elements, graphics, and style sheets
Graphic percentage	Percentage of page bytes that are for graphics
Graphics count	Total graphics on a page (not including graphics specified in scripts, applets, and objects)
Color count	Total colors employed
Font count	Total fonts employed (i.e., face + size + bold + italic)

Content Metrics

Suggested Metric	Description
Page wait	Average time required for a page to download at different connection speeds
Page complexity	Average number of different types of media used on page, not including text
Graphic complexity	Average number of graphics media per page
Audio complexity	Average number of audio media per page
Video complexity	Average number of video media per page
Animation complexity	Average number of animations per page
Scanned image complexity	Average number of scanned images per page

Navigation Metrics

Suggested Metric	Description
Page-linking complexity	Number of links per page
Connectivity	Total number of internal links, not including dynamically generated links
Connectivity density	Connectivity divided by page count

Metrics for Source Code

- Halstead's Software Science: a comprehensive collection of metrics all predicated on the number (count and occurrence) of operators and operands within a program
- n1 = number of distinct operators in a program
- n2 = number of distinct operands in a program
- N1 = total number of operator occurrences
- N2 = total number of operand occurrences
- The overall program length:

$$N = n1 \log_2 n1 + n2 \log_2 n2$$

- The program volume:

$$V = N \log_2 (n1 + n2)$$

  - V will vary with programming language and represents the volume of information (in bits) required to specify a program
  - Lower volume is more desirable

Metrics for Testing-I

- Testing metrics
  1. Metrics that attempt to predict the likely number of tests required at various testing levels
  2. Metrics that focus on test coverage for a given component
- Architectural design metrics provide information on
  - the ease or difficulty of testing
  - and the need for specialized testing software (e.g., stubs and drivers)
- Cyclomatic complexity (a component-level design metric) lies at the core of basis path testing
  - Modules with high cyclomatic are more likely to be error prone than modules whose cyclomatic complexity is lower
  - Cyclomatic complexity can be used to target modules as candidates for extensive unit testing

Metrics for Testing-II

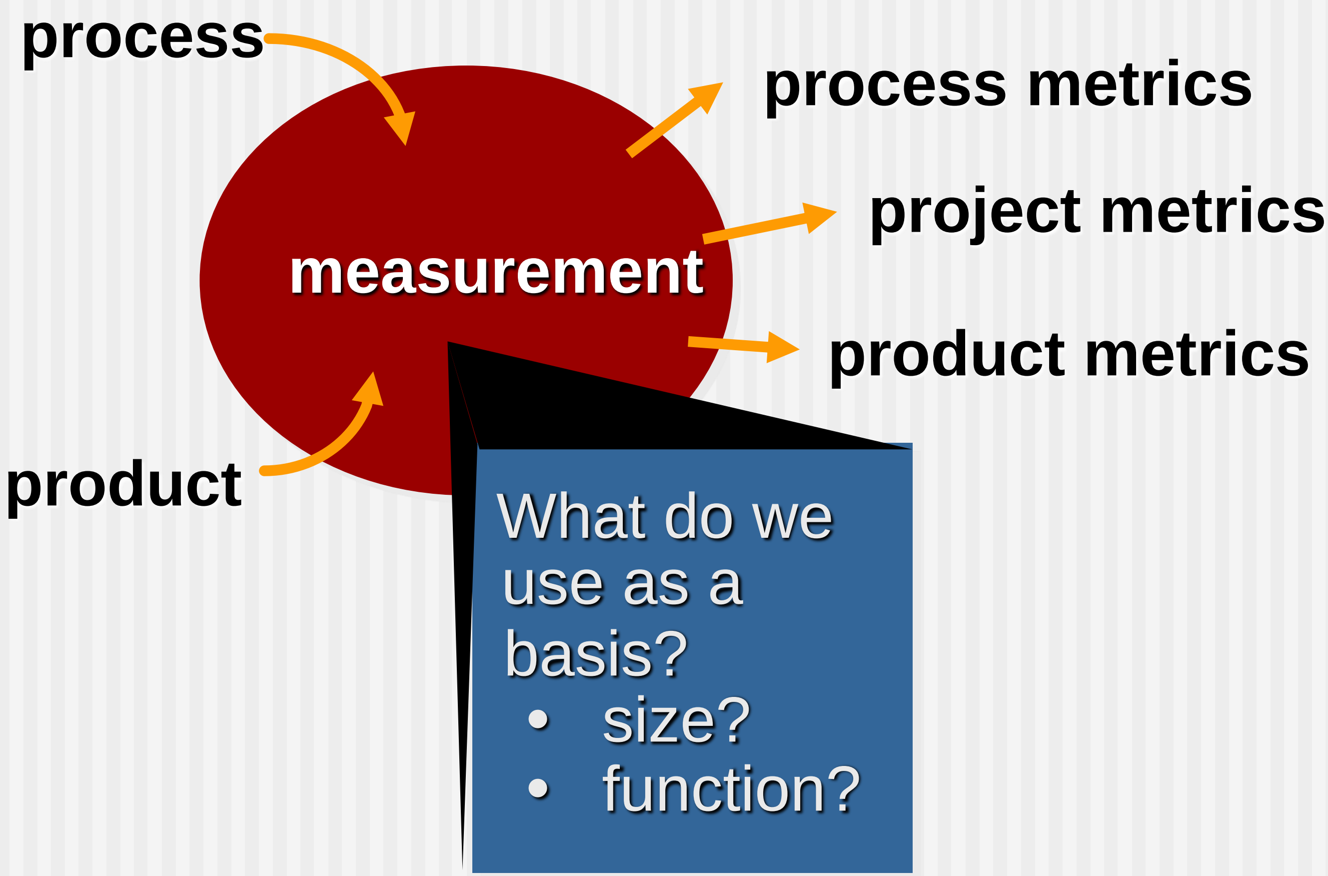
- Testing effort can also be estimated using metrics derived from Halstead measures
  - See page 676 of 8^{th} edition
- Binder [Bin94] suggests a broad array of design metrics that have a direct influence on the “testability” of an OO system.
  - Lack of cohesion in methods (LCOM).
  - Percent public and protected (PAP).
  - Public access to data members (PAD).
  - Number of root classes (NOR).
  - Fan-in (FIN).
  - Number of children (NOC) and depth of the inheritance tree (DIT).

Metrics for Maintenance

- IEEE Std. 982.1-1988 [IEE94] suggests a *software maturity index* (SMI) that provides an indication of the stability of a software product (based on changes that occur for each release of the product). The following information is determined:
  - M_T = the number of modules in the current release
  - F_c = the number of modules in the current release that have been changed
  - F_a = the number of modules in the current release that have been added
  - F_d = the number of modules from the preceding release that were deleted in the current release
- The software maturity index is computed in the following manner:
  - $SMI = [M_T - (F_a + F_c + F_d)]/M_T$
- As SMI approaches 1.0, the product begins to stabilize.

Process and Project Metrics

A Good Manager Measures



Why Do We Measure?

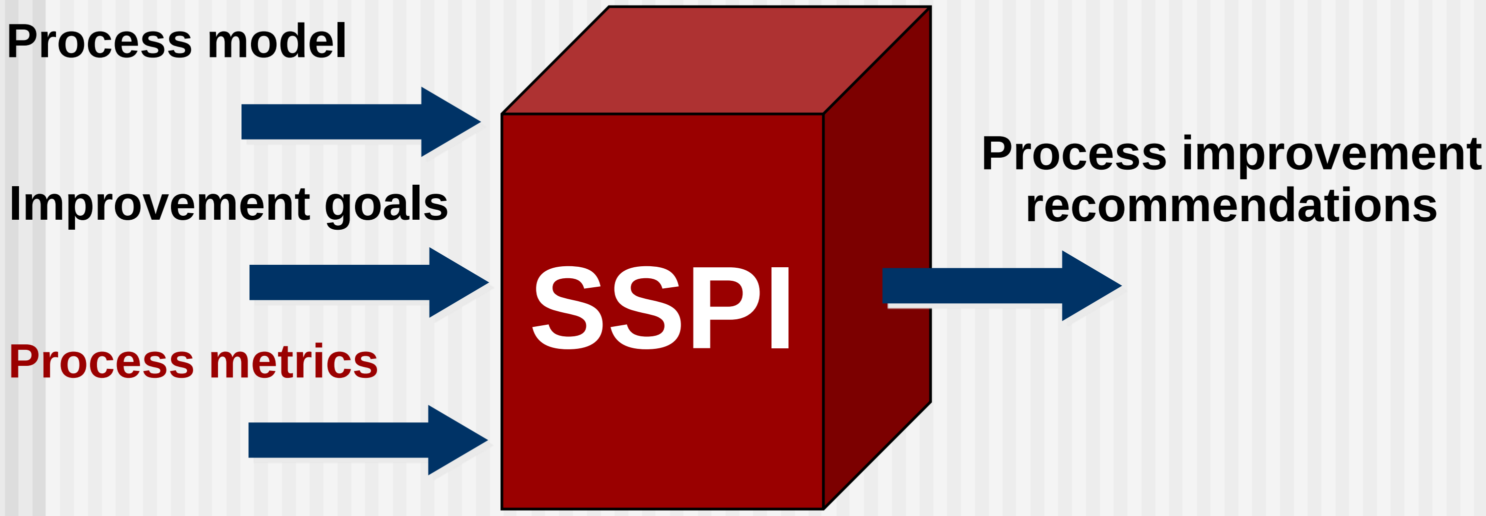
- **Process metrics** are collected across all projects and over long periods of time
  - Their intent is to provide a set of process indicators that lead to long-term software process improvement
- **Project metrics** enable a software project manager to:
  - assess the status of an ongoing project
  - track potential risks
  - uncover problem areas before they go “critical”
  - adjust work flow or tasks
  - evaluate the project team’s ability to control quality of software work products

Process Measurement

- We measure the efficacy of a software process indirectly.
  - That is, we derive a set of metrics based on the outcomes that can be derived from the process.
  - Outcomes include
    - measures of errors uncovered before release of the software
    - defects delivered to and reported by end-users
    - work products delivered (productivity)
    - human effort expended
    - calendar time expended
    - schedule conformance
    - other measures.
- We also derive process metrics by measuring the characteristics of specific software engineering tasks.
  - For example, the effort and time spent performing the umbrella activities and framework activities

Statistical Software Process Improvement

- A more rigorous approach called statistical software process improvement (SSPI)
  - As an organization becomes more comfortable with the **collection and use** of process metrics



Project Metrics

- Intent
  - To minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks
  - To assess product quality on an ongoing basis and, when necessary, modify the technical approach to improve quality
- Typical Project Metrics
  - Effort/time per software engineering task
  - Errors uncovered per review hour
  - Scheduled vs. actual milestone dates
  - Changes (number) and their characteristics
  - Distribution of effort on software engineering tasks

Comparing Projects

- But how does an organization compare metrics that come from different individuals or projects?
- Consider a simple example:
  - Team A found 342 errors during the software process prior to release
  - Team B found 184 errors
  - Which team is more effective in uncovering errors throughout the process?
    - Because you do not know the size or complexity of the projects, you cannot answer this question
- If the measures are normalized, it is possible to create software metrics that enable comparison to broader organizational averages

Size-Oriented Metrics

- Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the size of the software
  - E.g., lines of code (LOC), person-months effort, cost, number of pages of documentation, …

Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
⋮	⋮	⋮	⋮	⋮	⋮		

Typical Size-Oriented Metrics

- Choosing lines of code (LOC) as a normalization value
  - errors per KLOC (thousand lines of code)
  - defects per KLOC
  - $ per LOC
  - pages of documentation per KLOC
- Choosing effort as a normalization value
  - errors per person-month
  - LOC per person-month
- Other interesting metrics
  - errors per review hour
  - $ per page of documentation

Function-Oriented Metrics

- Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value
- The most widely used function-oriented metric is the function point (FP)
- Proponents
  - programming language–independent
  - Based on data that are more likely to be known early in the evolution of a project
    - Making FP more attractive as an estimation approach
- Opponents
  - Requires some “sleight of hand” in that computation is based on subjective rather than objective data
  - No direct physical meaning—it’s just a number

Typical Function-Oriented Metrics

- errors per FP
- defects per FP
- $ per FP
- pages of documentation per FP
- FP per person-month

Software Quality Metrics

Measuring Quality

- Correctness — the degree to which a program operates according to specification
- Maintainability—the degree to which a program is amenable to change
- Integrity—the degree to which a program is impervious to outside attack
- Usability—the degree to which a program is easy to use

Correctness

- Defects (lack of correctness) are those problems reported by a user of the program after the program has been released
- Defects are counted over a standard period of time, typically one year
- The most common measure for correctness:
  - **Defects per KLOC**

Maintainability

- The ease with which a program can be
  - corrected if an error is encountered,
  - adapted if its environment changes, or
  - enhanced if the customer desires a change in requirements
- There is no way to measure maintainability directly
  - We must use indirect measures
- A simple time-oriented metric is **mean time to change (MTTC)**
  - The time it takes to analyze the change request, design an appropriate modification, implement the change, test it, and distribute the change to all user

Integrity

- A system's ability to withstand attacks to its security
  - Both accidental and intentional attacks
- To measure integrity, two additional attributes must be defined:
  - **Threat**: The probability that an attack of a specific type will occur within a given time
    - Can be estimated or derived from empirical evidence
  - **Security**: The probability that the attack of a specific type will be repelled
    - Can be estimated or derived from empirical evidence
- The integrity of a system:

$$\text{Integrity} = \Sigma[1 - (\text{threat} \times (1 - \text{security}))]$$

Integrity: An Example

- If
  - threat (the probability that an attack will occur) is 0.25
  - security (the likelihood of repelling an attack) is 0.95
- Then
  - the integrity of the system is 0.99 **(very high)**
- If, on the other hand,
  - the threat probability is 0.50
  - the likelihood of repelling an attack is only 0.25,
  - the integrity of the system is 0.63 **(unacceptably low)**

Usability

- Usability is an attempt to quantify ease of use
- It can be measured in terms of the usability characteristics:
  - Is the system usable without continual help?
  - Does the user know where she is at all times?
  - Are interaction mechanisms, icons, and procedures consistent across the interface?
  - Does the interaction anticipate errors and help the user correct them?
  - Is the interaction simple?
  - ...
- (see Chapter 15 for details)

Defect Removal Efficiency

- DRE: A quality metric that provides benefit at both the project and process level

$$\textbf{DRE} = E\,/(E + D)$$

where:

E is the number of errors found before delivery of the software to the end-user

D is the number of defects found after delivery.

DRE Advice

- The ideal value for DRE is 1
  - That is, no defects are found in the software
- As E increases (for a given value of D), the overall value of DRE begins to approach 1
- If DRE is low as you move through analysis and design, spend some time improving the way you conduct formal technical reviews
  - For finding as many errors as possible before delivery

Task-Specific DRE

- DRE can be used within the project to assess a team's ability to find errors before they are passed to the next framework activity or software engineering task
- For example, requirements analysis produces a requirements model
  - It can be reviewed to find and correct errors
  - Those errors that are not found during the review of the requirements model are passed on to design (where they may or may not be found)

$$\mathrm{DRE}_i = \frac{E_i}{E_i + E_{i+1}}$$

E_i: number of errors found during action i
E_{i+1}: number of errors found during action i+1 (not discovered in action i)

Further Reading

The End