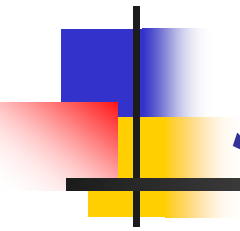
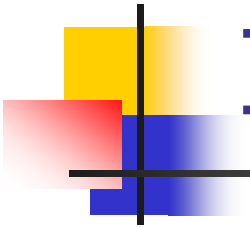


# *COMP 354: Introduction to Software Engineering*

## Software Metrics and Analytics

Based on Chapter 23 of the textbook

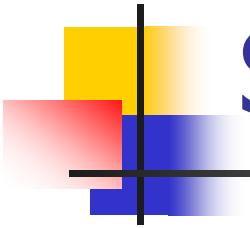
# Measures, Metrics, and Indicators



- A measure provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process.
- A metric is a quantitative measure of the degree to which a system, component, or process possesses a given attribute.
- An indicator is a metric or combination of metrics that provide insight into the software process, a software project, or the product itself.

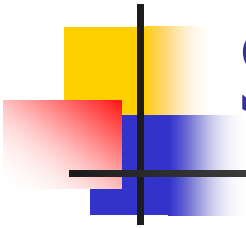
# Attributes of Effective Metrics

- Simple and computable. It should be relatively easy to learn how to derive the metric.
- Empirically and intuitively persuasive. Satisfies the engineer's intuitive notions about the product attribute.
- Consistent and objective. The metric should yield results that are unambiguous.
- Consistent in its use of units and dimensions. Computation of the metric should not lead to bizarre combinations of units.
- Programming language independent. Metrics should be based on the analysis model, the design model, or the structure of the program itself.
- Effective mechanism for quality feedback. Should provide a software engineer with information that can lead to a higher quality end-product.



# Software Analytics

- Key performance indicators (KPI s) are metrics that are used to track performance and trigger remedial actions when their values fall in a predetermined range.
- How do you know that metrics are meaningful in the first place?
- Software analytics is the systematic computational analysis of software engineering data or statistics to provide managers and software engineers with meaningful insights and empower their teams to make better decisions.



# Software Analytics

Software analytics can help developers make decisions regarding:

- Targeted testing.
- Targeted refactoring.
- Release planning.
- Understanding customers.
- Judging stability.
- Targeting inspection.

# Requirements Model Metrics

- Requirement specificity (lack of ambiguity):

  $$Q_1 = n_{ui} / n_r$$

  where $n_{ui}$ is the number of requirements for which all reviewers had identical interpretations.
- $Q_1$ close to 1 is good.
- Assume there are $n_r$ requirements in a specification:

  $$n_r = n_f + n_{nf}$$

  $n_f$ is the number of functional requirements

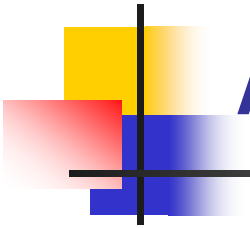
  $n_{nf}$ is the number of nonfunctional requirements

# Mobile Software Requirements Model Metrics

- Number of static screen displays. ($N_{sp}$)
- Number of dynamic screen displays. ($N_{dp}$)
- Number of persistent data objects.
- Number of external systems interfaced.
- Number of static content objects.
- Number of dynamic content objects.
- Number of executable functions.
- Customization index $C = N_{dp} / (N_{dp} + N_{sp})$

  $C$ ranges from 0 to 1, larger $C$ is better

# Architectural Design Metrics



Architectural design metrics

- Structural complexity = *g*(fan-out).
- Data complexity = *f*(input & output variables, fan-out).
- System complexity = *h*(structural & data complexity).

Morphology metrics: a function of the number of modules and the number of interfaces between modules

Size = $n + a$

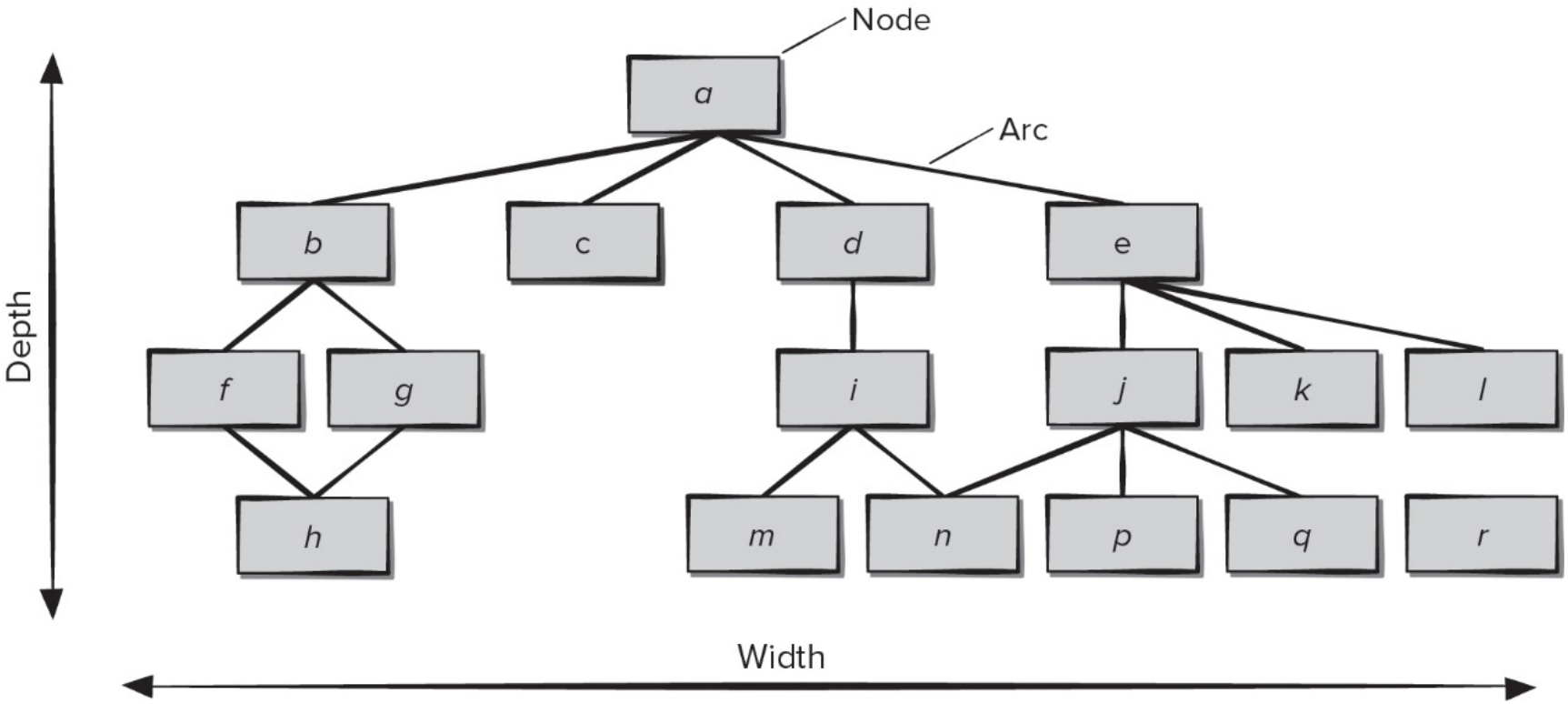
$n$ = number of nodes, $a$ = number of arcs

Depth = longest path root to leaf node

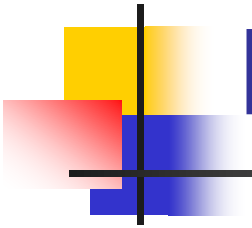
Width = maximum number if nodes at each level

# Morphology Metrics



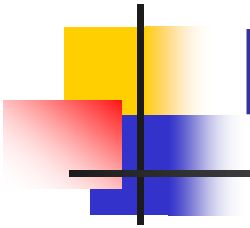


# Object-Oriented Design Metrics



- Weighted methods per class (WMC). The number of methods and their complexity are reasonable indicators of the amount of effort required to implement and test a class.
- Depth of the inheritance tree (DIT). A deep class hierarchy (DIT is large) leads to greater design complexity.
- Number of children (NOC). As NOC increases, the amount of testing (required to exercise each child in its operational context) will also increase.

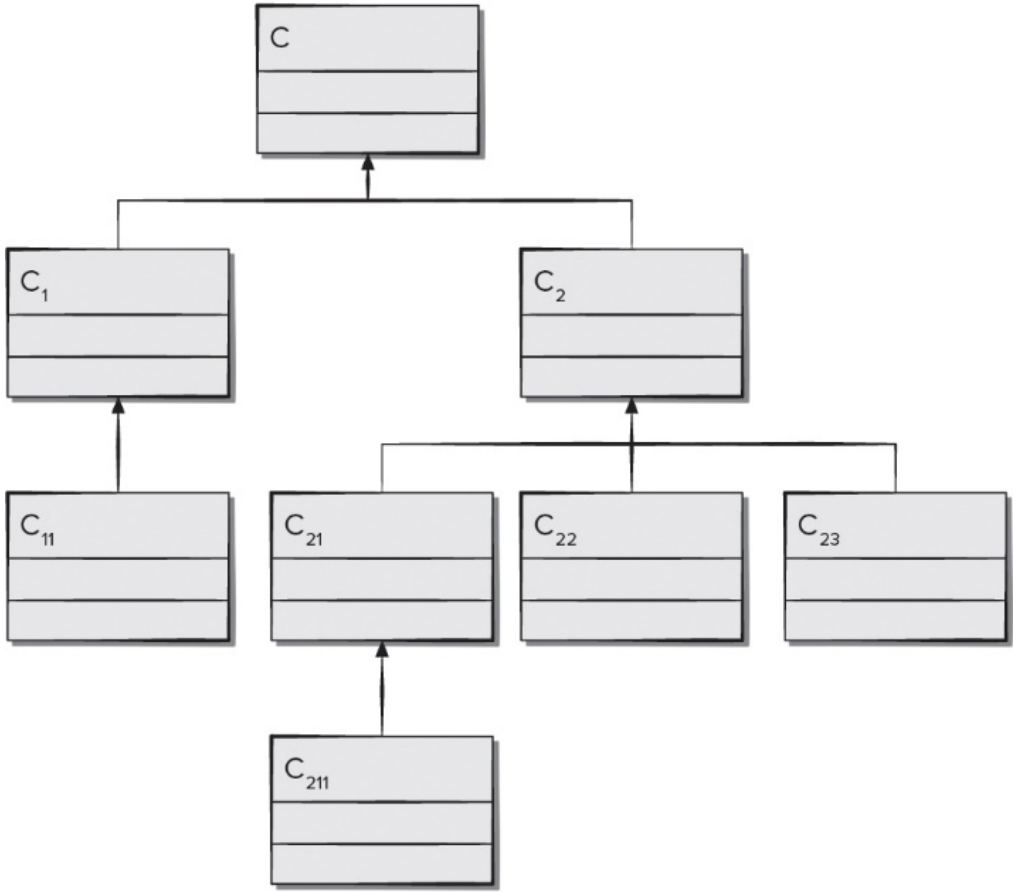
# Object-Oriented Design Metrics



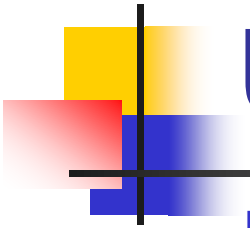
- Coupling between object classes (CBO). High values of CBO indicate poor reusability and make testing of modifications more complicated.
- Response for a class (RFC). The number of methods that can potentially be executed in response to a message received by an object of the class.
- Lack of cohesion in methods (LCOM). LCOM is the number of methods that access one or more of the same attributes

# Class Hierarchy



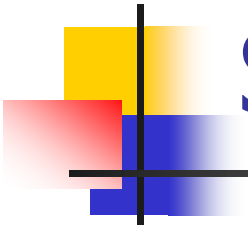


# User Interface Design Metrics



- Interface metrics. Ergonomics measures (for example, memory load, typing effort, recognition time, layout complexity)
- Aesthetic (graphic design) metrics. Aesthetic design relies on qualitative judgment but some measures are possible (for example, word count, graphic percentage, page size)
- Content metrics. Focus on content complexity and on clusters of content objects that are organized into pages
- Navigation metrics. Address the complexity of the navigational flow and they are applicable only for static Web applications.

# Source Code Metrics



Halstead's Software Science: a comprehensive collection of metrics all predicated on the number (count and occurrence) of operators and operands within a component or program.

$n_1$ = number of distinct operators that appear in a program

$n_2$ = number of distinct operands that appear in a program

$N_1$ = total number of operator occurrences

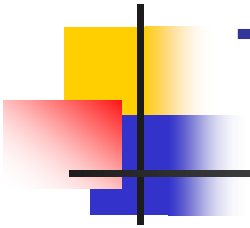
$N_2$ = total number of operand occurrences

Program length $N = n_1 \log_2 n_1 + n_2 \log_2 n_2$

Program volume $V = N \log_2 (n_1 + n_2)$

Volume Ratio $L = ( 2 / n_1 ) \times ( n_2 / N_2 )$

$L$ is ratio of most compact form of the program to actual program size



# Testing Metrics

- Testing effort estimated using metrics derived from Halstead measures
  - Program level $PL = 1 / L$
  - Effort $e = V / PL$
- Some OO design metrics have influence on "testability"
  - 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).

# Maintenance Metrics

- IEEE Std. 982.1-2005 software maturity index (SMI) that provides an indication of the software product stability (based on changes made).

  $MT$ = number of modules in current release

  $F_c$ = number of modules in current release that have been changed

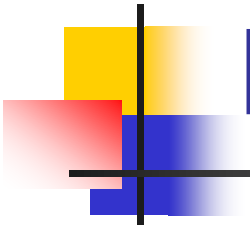
  $F_a$ = number of modules in current release that have been added

  $F_d$ = number of preceding release modules deleted

  $$SMI = [MT - (F_a + F_c + F_d)]/MT$$

- As SMI approaches 1.0, the product begins to stabilize.

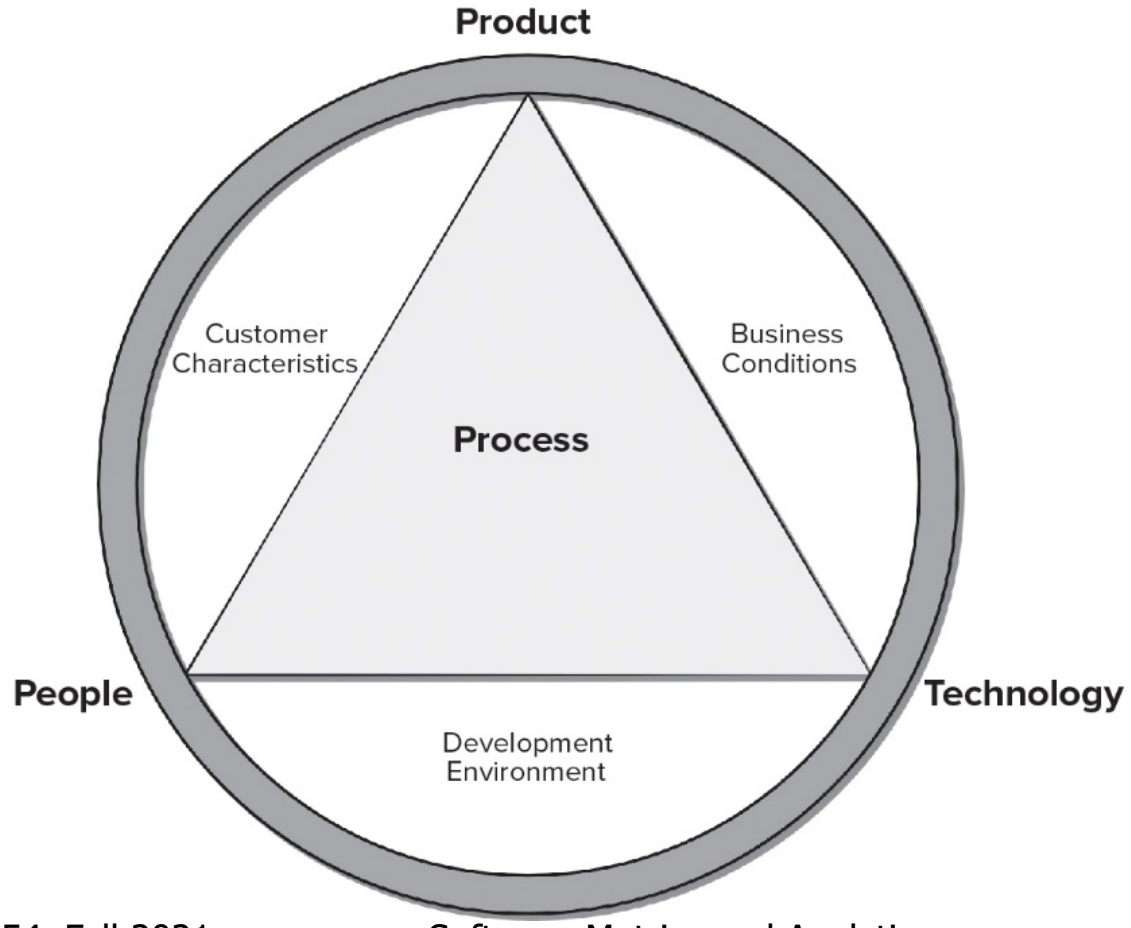
# Process and Project Metrics



- Process metrics collected across all projects, over long periods of time. Their intent is to provide a set of 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 project team's ability to control quality of software work products.

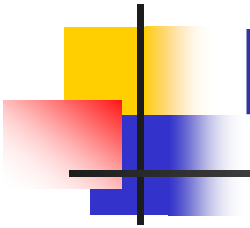
# Determinants of Software Quality and Organizational Effectiveness





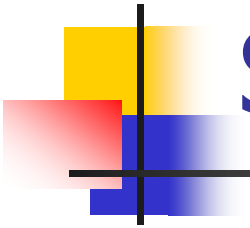
# Process Measurement

- We measure the efficacy of a software process indirectly – by deriving metrics based on the outcomes that can be derived from the process:
  - 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



# Process Metrics Guidelines

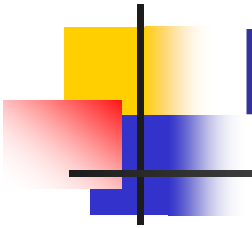
- Use common sense and organizational sensitivity when interpreting metrics data.
- Provide regular feedback to the individuals and teams who collect measures and metrics.
- Don’t use metrics to appraise individuals.
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them.
- Never use metrics to threaten individuals or teams.
- Metrics data that indicate a problem area should not be considered “negative.” These data are merely an indicator for process improvement.
- Don’t obsess on a single metric to the exclusion of other important metrics.



# Software Measurement

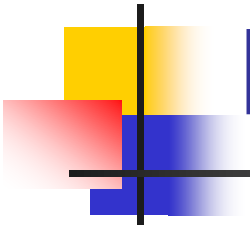
- Direct measures of the software process include cost and effort applied.
- Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time.
- Indirect measures of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many others.
- Direct measures are relatively easy to collect, the quality and functionality of software are more difficult to assess and can be measured only indirectly.

# Normalized Size-Oriented Metrics



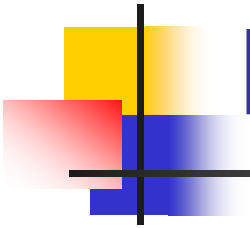
- errors per KLOC (thousand lines of code)
- defects per KLOC
- $ per LOC
- pages of documentation per KLOC
- errors per person-month
- errors per review hour
- LOC per person-month
- $ per page of documentation

# Normalized Function-Oriented Metrics



- errors per FP (thousand lines of code)
- defects per FP
- $ per FP
- pages of documentation per FP
- FP per person-month

# Why Opt For Function-Oriented Metrics



- Programming language independent.
- Used readily countable characteristics that are determined early in the software process.
- Does not "penalize" inventive (short) implementations that use fewer L O C that other more clumsy versions.
- Makes it easier to measure the impact of reusable components.

# Software Quality Metrics

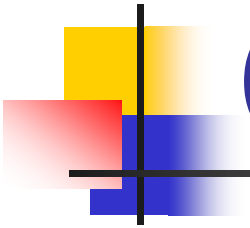
- Correctness. degree to which the software performs its required function (for example, defects per K L O C).
- Maintainability. degree to which a program is amenable to change (for example, M T T C - mean time to change).
- Integrity. degree to which a program is impervious to outside attack.

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

threat = probability specific attack occurs
security = probability specific attack is repelled

- Usability. quantifies ease of use (for example, error rate).

# Defect Removal Efficiency (DRE)



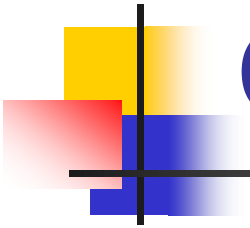
- DRE is a measure of the filtering ability of quality assurance and control actions as they are applied throughout all process framework activities.

  $DRE = E / (E + D)$

  $E$ = number of errors found before delivery

  $D$ = number or errors found after delivery

- The ideal value for DRE is 1. No defects ($D = 0$) are found be the consumers of a work product after delivery.
- The value of DRE begins to approach as $E$ increases many the team is catching its own errors.



# Goal Driven Metrics Program

1. Identify your business goals.
2. Identify what you want to know or learn.
3. Identify your subgoals.
4. Identify the entities and attributes related to your subgoals.
5. Formalize your measurement goals.
6. Identify quantifiable questions and the related indicators that you will use to help you achieve your measurement goals.
7. Identify the data elements that you will collect to construct the indicators.
8. Identify the measures to be used, and make these definitions operational.
9. Identify the actions that you will take to implement the measures.
10. Prepare a plan for implementing the measures.

# Metrics for Small Organizations

- time (hours or days) elapsed from the time a request is made until evaluation is complete, $t_{queue}$ .
- effort (person-hours) to perform the evaluation, $W_{eval}$ .
- time (hours or days) elapsed from completion of evaluation to assignment of change order to personnel, $t_{eval}$ .
- effort (person-hours) required to make the change, $W_{change}$ .
- time required (hours or days) to make the change, tchange.
- errors uncovered during work to make change, $E_{change}$ .
- defects uncovered after change is released to the customer base, $D_{change}$ .