Software Quality Evalaution

COMP 3700.002
Software Modeling and Design

Shehenaz Shaik

Software Quality: IEEE Definitions

- "Totality of features of a software product that bears on its ability to satisfy given needs." [Source: IEEESTD-729]

- "Composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer." [Source: IEEE-STD-729]

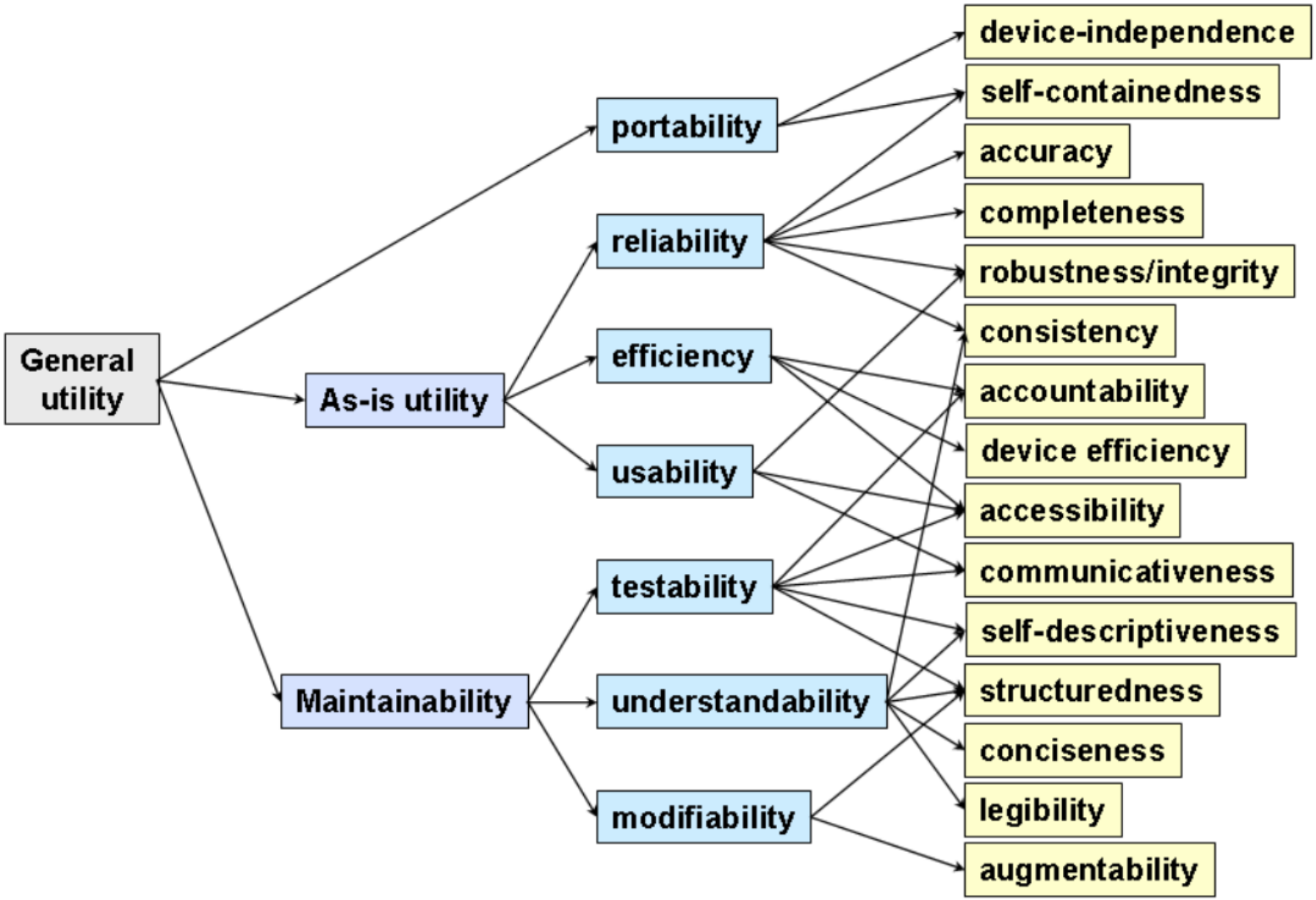
Design Quality

- Fitness to purpose
  - Does it do what is needed?
  - Does it do it in the way that its users need it to?
  - Does it do it reliably enough? fast enough? safely enough? securely enough?
  - Will it be affordable? will it be ready when its users need it?
  - Can it be changed as the needs change?

Design Quality

- Not a measure of software in isolation
  - It is a measure of the relationship between software and its application domain
  - Might not be able to measure this until you place the software into its environment
  - Quality will be different in different environments!
- During design
  - Predict how well the software will fit its purpose
  - Understand the purpose (requirements analysis)
  - Look for quality predictors

Boehm's Taxonomy



OPA Framework

- A set of OBJECTIVES are defined that correspond to project level goals and objectives (maintainability, testability etc.)
- Achieving those objectives require adherence to certain PRINCIPLES that characterize the process by which software is developed.
- Adherence to a process governed by these principles should result in a product that possesses ATTRIBUTES considered to be desirable and beneficial.

Objectives

- Adaptability
  - Ease with which software can accommodate to changing requirements.
- Correctness
  - Strict adherence to specifications.
- Maintainability
  - Ease with which corrections can be made to respond to recognized inadequacies.
- Portability
  - Ease in transferring software to another host environment.
- Reliability
  - Error-free behavior of software over time.
- Reusability
  - Use of software developed in other applications.
- Testability
  - Ability to evaluate conformance with specifications.

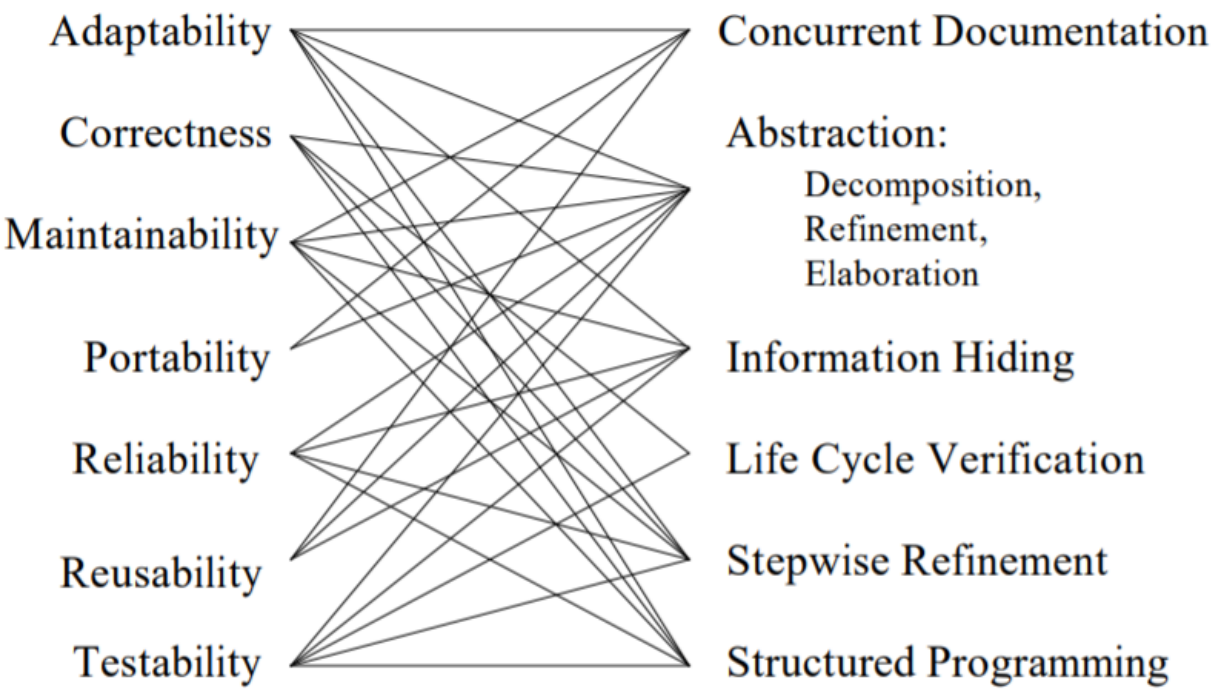
Principles

- Abstraction
  - Hierarchical Decomposition
  - Functional Decomposition
- Concurrent Documentation
  - Management of supporting documents (system specifications, user manuals, etc.) throughout the life cycle.
- Information Hiding
  - Insulating the internal details of component behavior.
- Life Cycle Verification
  - Verification of requirements throughout the design, development, and maintenance phases of the life cycle.
- Stepwise Refinement
  - Utilizing convergent design.
- Structured Programming
  - Using a restricted set of program control constructs

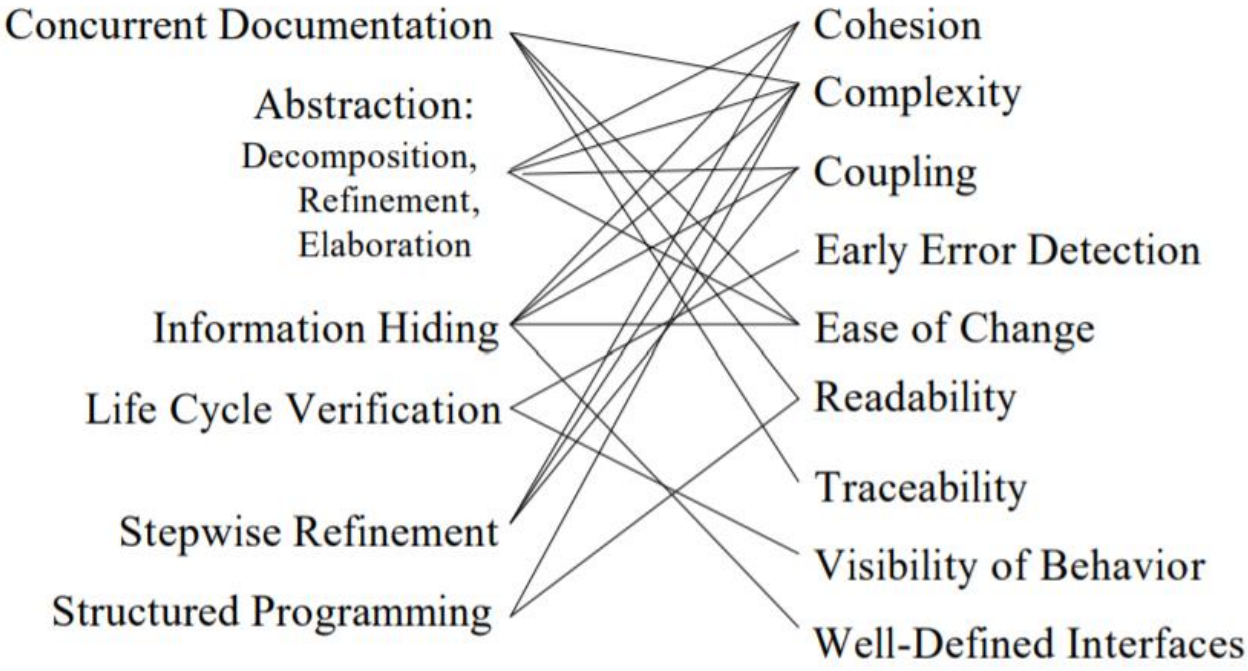
Attributes

- Cohesion
  - Binding of statements within a software component
- Complexity
  - Abstract measure of work associated with a software component
- Coupling
  - Interdependence among software components
- Early Error Detection
  - Indication of faults in requirements, specification and design prior to implementation
- Ease of Change
  - Software that accommodates enhancements or extensions
- Readability
  - Difficulty in understanding a software component
- Traceability
  - Ease in retracing complete history of a software component from its current status to its design
- Visibility of Behavior
  - Provision of a review process for error checking
- Well-Defined Interfaces
  - Definitional clarity and completeness of a shared boundary between software and/or hardware

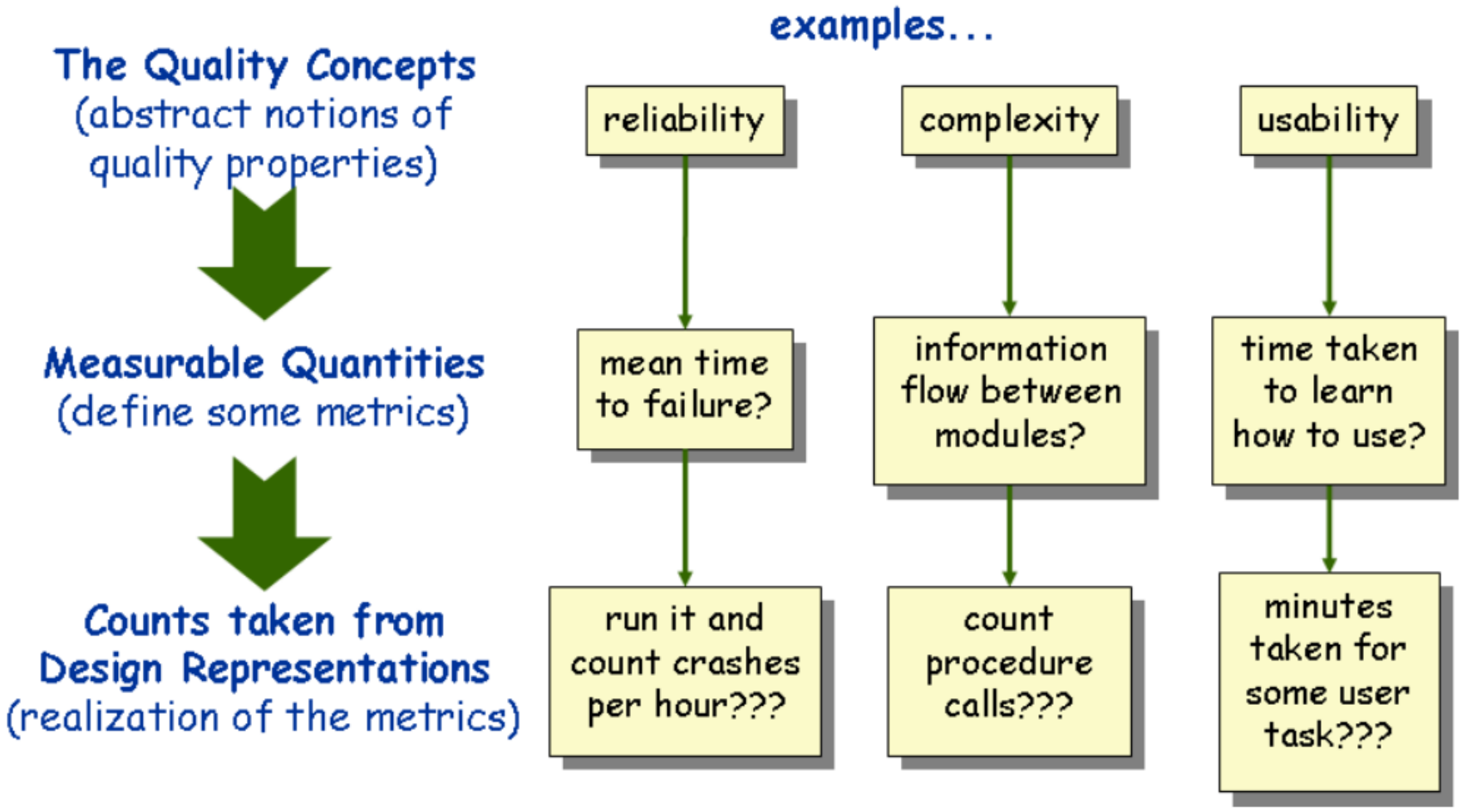
Objective/Principles Linkages



Principles/Attributes Linkages



Measuring Quality



Measuring OO Design Quality

- What is Metric:
  - The continuous application of measurement-based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products.

Methods Per Class (MPC)

- Average number of methods per object class = Total number of methods / Total number of object classes
  - Larger number
    - More complicated testing
    - Difficult to extend
    - Increase in amount of code reuse, from inheritance through subclasses
      - But, ad hoc and random code reuse may be dangerous

Inheritance Dependencies

- Inheritance tree depth = max (inheritance tree path length)
  - Depth, preferable to width, of an inheritance tree
    - Greater amount of method sharing
  - Harder to test than a broad tree
  - Harder to understand with a larger number of layers

Degree of Coupling Between Objects

- Average number of uses dependencies per object = total number of arcs / total number of objects
- arcs = max (number of uses arcs) - in an object uses network
- arcs - attached to any single object in a uses network

- Higher degree of coupling between objects
  - Application maintenance more complicated, as object interactions and interconnections are more complex
  - Reduced suitability for reuse internally and externally to an application
  - Increased amounts of errors can be caused by the complexity of the interactions
  - Harder to test

Degree of Cohesion of Objects

- Degree of dependencies of parts within a single component
- Low cohesion
  - Higher degree of errors →
    - Raises complexity of application
    - Reduces application's reliability
- Objects that have fewer dependencies on other objects for data → more likely to be reusable.

Average Method Complexity (AMC)

- = Sum of cyclomatic complexity of all Methods / Total number of application methods
- Higher the complexity
  - More difficult to maintain
  - Harder to understand the application
  - Harder to test
  - Reduce application reliability

Depth of Inheritance Tree (DIT)

- For a certain class, the length of the path from that class to the root of the class hierarchy.
- Deeper a class in hierarchy
  - Likely to inherit more methods
    - More complex to predict its behavior
    - Greater possible reuse of inherited methods
- Deeper trees
  - Greater design complexity
    - More classes and methods

Number of Children (NOC)

- Number of immediate subclasses
- Greater number of children
  - Greater amount of reuse
  - Improper abstraction of parent class is more likely
  - Reflects class's potential influence on the design
    - More testing is needed

Total Number of Methods per Class (TOM)

- Includes all inherited methods
- Larger number of methods
  - Larger amount of testing needed

Percentage Public and Protected (PAP)

- Percentage of public member variables in the related class.
- Higher percentage
  - Related classes need to be tested more

Size Metrics

- Method-Level Size Metrics
  - Number of parameters required by a method
  - Number of operators and operands used in a method
  - Number of Instance Variables used by the method to perform the functionality
- Class-Level Size Metrics
  - Number of Methods in a class
  - Number of Member Variables in a class
  - Size of Class Interface
    - Number of public member variables available to other classes
  - Total number of executable statements in all methods
- System-Level Size Metrics
  - Number of Classes in the system
  - Total Number of Methods and Global Functions in the system
  - Total Number of Member Variables in all classes
  - Total Number of Global Variables in all classes

Reading assignment

- References
  - Quantitative evaluation of software quality, Barry William Boehm, John R Brown, Myron Lipow, ICSE '76: Proceedings of the 2nd international conference on Software engineering, October 1976, Pages 592–605.
  - Managing Software Quality: A Measurement Framework for Assessment and Prediction, Richard E. Nance, James D. Arthur, 2012. – Book chapter #2.
  - Object Oriented Metrics, http://yunus.hun.edu.tr/~sencer/oom.html

Next sessions…

- GoF Patterns (Contd.)