CTIS359
Principles of Software Engineering

Software Quality

"Quality is never an accident.
it is always the result of intelligent effort."
John Ruskin

Today

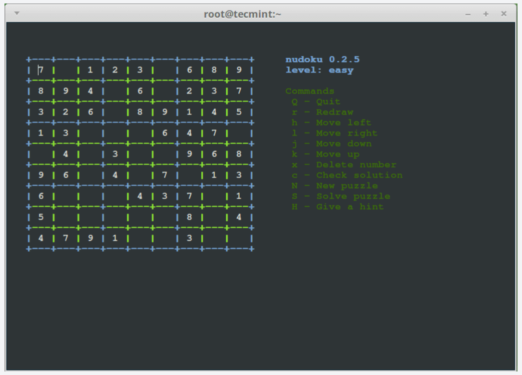
- Define the term "quality"
- (General) Viewpoints on quality
- Define the term "Software Quality (SQ)"
- Look at the **general models** of software quality that have gained acceptance within the SWE community
- Software metrics and an how they relate to software quality
    - Measurement Scales and Scale Types

What is "quality"?

Windows 10

macOS 10.15: Catalina

Cockpit Software



```
+---+---+---+---+---+---+---+---+---+      nudoku 0.2.5
| 7 |   | 1 | 2 | 3 |   | 6 | 8 | 9 |      level: easy
+---+---+---+---+---+---+---+---+---+
| 8 | 9 | 4 |   | 6 |   | 2 | 3 | 7 |      Commands
+---+---+---+---+---+---+---+---+---+       Q - Quit
| 3 | 2 | 6 |   | 8 | 9 | 1 | 4 | 5 |       r - Redraw
+---+---+---+---+---+---+---+---+---+       h - Move left
| 1 | 3 |   |   |   | 6 | 4 | 7 |   |       l - Move right
+---+---+---+---+---+---+---+---+---+       j - Move down
|   | 4 |   | 3 |   |   | 9 | 6 | 8 |       k - Move up
+---+---+---+---+---+---+---+---+---+       x - Delete number
| 9 | 6 |   | 4 |   | 7 |   | 1 | 3 |       c - Check solution
+---+---+---+---+---+---+---+---+---+       N - New puzzle
| 6 |   |   |   | 4 | 3 | 7 |   | 1 |       S - Solve puzzle
+---+---+---+---+---+---+---+---+---+       H - Give a hint
| 5 |   |   |   |   |   | 8 |   | 4 |
+---+---+---+---+---+---+---+---+---+
| 4 | 7 | 9 | 1 |   |   | 3 |   |   |
+---+---+---+---+---+---+---+---+---+
```

Nudoku CLI-based Sudoku game

The term "quality" has a relative meaning

What is "quality"?

- Question:
  If **Nudoku CLI-based Sudoku game** is not free, would you purchase it or use it again?

What is "quality"?

- Question:
  Would you still complain about **MS Windows** operating system if it was **free**?

  Would you be so demanding?

What is "quality"?

- Question:
  If **Cockpit Software in Boeing** satisfies more user wants/needs than **Cockpit Software in an Airbus**, then a pilot finds that Boeing product has higher quality than product Airbus Product?
  - Ask 5000 pilots?
    - Junior
    - Senior
    - Jet/Jumbo/Fight

What is "quality"?

- What about this case?
  Your satisfaction on **Windows XP** might be high enough to compensate for the dissatisfaction attributable to use **Windows 8** and/or **10**.

Defining (Software) Quality

- Quality has always been a difficult topic to define, and software quality has been exceptionally difficult.
- The reason is that perceptions of quality vary from person to person and from object to object.
- For **software** quality for a specific application, the **perceptions** of quality differ among clients, developers, users, managers, software quality personnel, testers, senior executives, and other stakeholders.
  - The perceptions of quality also differ among quality consultants, academics, and litigation attorneys.
- Many definitions have been suggested over the years, **but none have been totally satisfactory or totally adopted by the software industry, including those embodied in international standards.**

Source: The Economics of Software Quality -Capers Jones (2011)

Defining (Software) Quality

- The reason that quality in general and software quality in particular have been elusive and hard to pin down is because the word "quality" has many nuances and overtones.
- For example, among the attributes of quality that can be found these some:
    1. *Elegance or beauty* in the eye of the beholder
    2. *Fitness of use* for various purposes
    3. *Satisfaction of user requirements*, both explicit and implicit
    4. *Freedom from defects*, perhaps to Six Sigma levels
    5. *High efficiency of defect removal* activities
    6. *High reliability* when operating
    7. *Ease of learning* and *ease of use*
    8. *Clarity of user guides* and HELP materials
    9. *Ease of access* to customer support
    10. *Rapid repairs* of reported defects

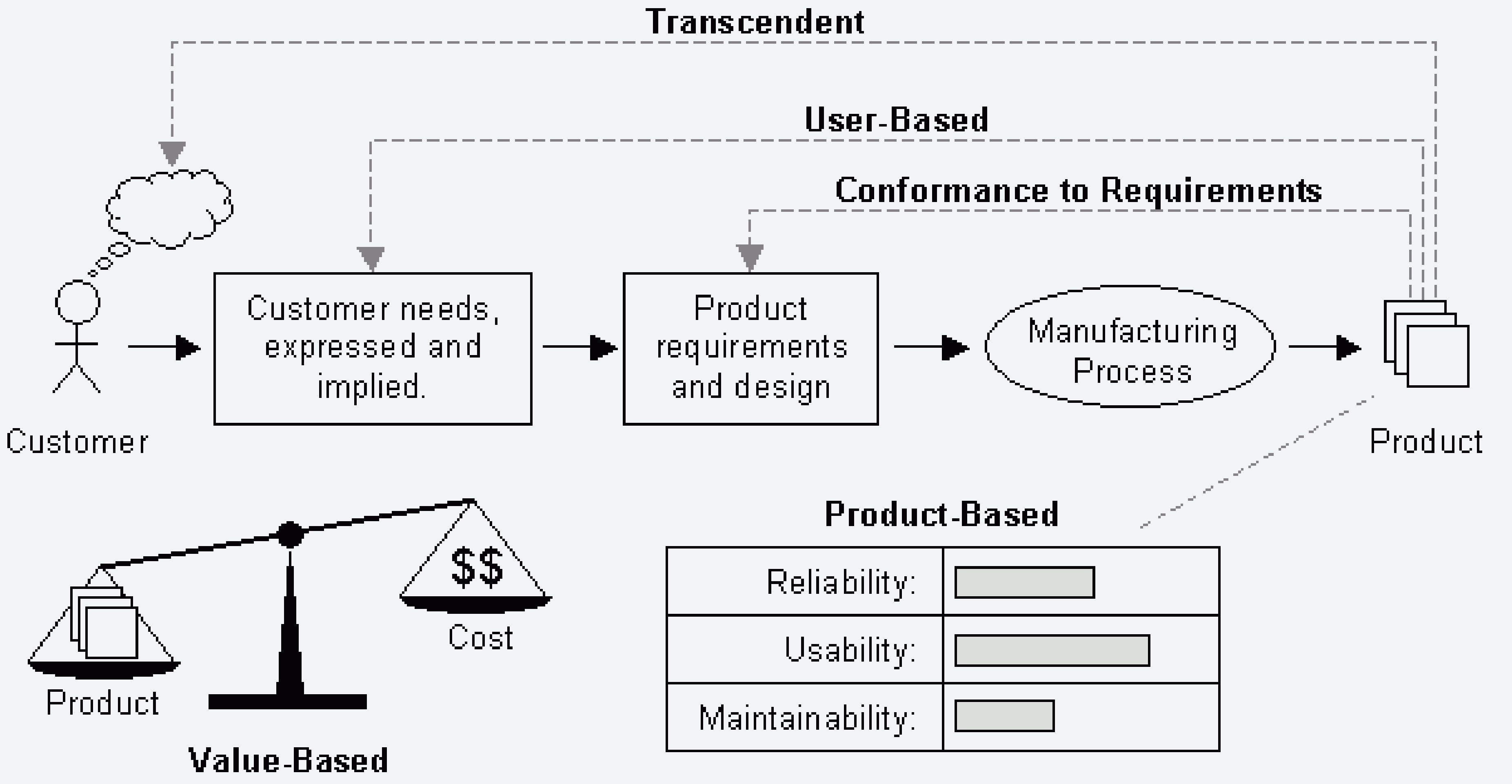
Source: The Economics of Software Quality -Capers Jones (2011)

Different perspectives on quality

- The following are **the most common perspectives** on **quality** (in general) [Garvin, Crosby]:
  - Transcendent
  - User-Based
  - Conformance to Requirements
  - Product-Based
  - Value-Based

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Different perspectives on quality



Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Different perspectives on quality

- The following are **the most common perspectives** on **quality** (in general) [Garvin, Crosby]:
  - Transcendent
  - User-Based
  - Conformance to Requirements
  - Product-Based
  - Value-Based

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

The transcendent view of quality

- This view of quality equates quality with "innate (inherent) excellence".
- According to this perspective, **quality can't be described with words**, it can **only be understood through experience**.
- This view of quality relies on perception and experience but isn't completely subjective. For example wine tasting experts generally agree on the relative merits of different vintages of wine which suggests that the activity isn't completely subjective.
    - *Robert Pirsig takes on the topic of quality in Zen and the Art of Motorcycle Maintenance and concludes, "Quality is not objective... It doesn't reside in the material world....Quality is not subjective...It doesn't reside merely in the mind....Quality is neither a part of mind, nor is it a part of matter. It is a third entity which is independent of the two." [Pirsig]*

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

The transcendent view of quality

- This view of quality is a poor basis from which to develop an operational definition of quality.
  - The most obvious problem is measurability. There is NO precise way of quantifying innate excellence. The transcendent view of quality is **better suited** to **products** that **appeal** to the **senses**.
  - Software is practical and utilitarian and therefore NOT well suited to the transcendent view of quality.

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Different perspectives on quality

- The following are **the most common perspectives** on **quality** (in general) [Garvin, Crosby]:
  - Transcendent
  - User-Based
  - Conformance to Requirements
  - Product-Based
  - Value-Based

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

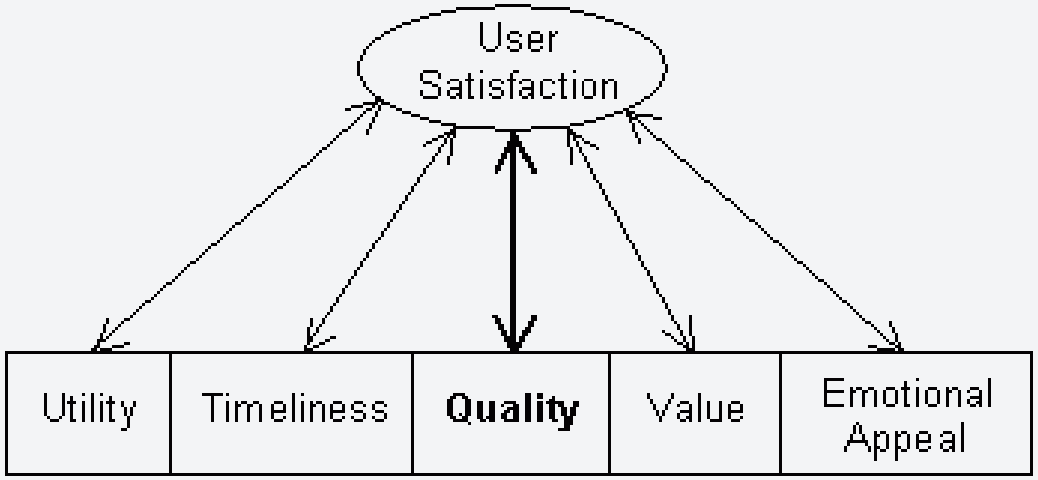
The user-based view of quality

- This view is certainly **more down-to-earth** than the transcendent view.
- According to this view, **quality** is the satisfaction of user wants or needs.
- ***If product A satisfies more user wants and needs than product B, then product A has higher quality than product B.*** (**Ask the user**.)
- There are two potential problems with this definition of quality though.
  - The first problem is that of designing a product that simultaneously meets the needs of a diverse group of users.
    - For example, the popular image editing program Adobe Photoshop is used by average consumers, professional photographers, and high-end animation studios. It would be difficult to design a product that simultaneously meets the needs of this diverse group of customers. What might be satisfying to one customer might be completely unacceptable to another.

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

The user-based view of quality

- The second problem with this perspective is more fundamental.
  - User satisfaction is based on an **aggregate of product factors**--only one of which is quality.



Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

The user-based view of quality

- A user might be satisfied with a low quality product if the other factors that influence satisfaction are high enough to compensate for the dissatisfaction attributable to low quality.
- To say that user satisfaction is a good measure of product quality is to ignore the other factors.
  - **For example**, there is a robust market for classic Jaguar automobiles from the early 90's. This suggests that these cars are providing user satisfaction. However, Jaguars of this vintage are notorious for electrical and mechanical problems. These cars might be meeting the overall needs of their buyers, but even their most enthusiastic admirers would freely admit that they are not high quality.
  - More likely, it is **a combination of product features** including their instinctive beauty and style that make them appealing in spite of their reputation for poor quality.

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

The user-based view of quality

- There is certainly value in having a definition of quality that defines **quality relative to user wants and needs**.
- However, before this perspective can serve as a basis for quality management, there has to be some way of **measuring the extent to which product quality is contributing to user satisfaction independent of other product features that might also be influencing overall user satisfaction**.

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Different perspectives on quality

- The following are **the most common perspectives** on **quality** (in general) [Garvin, Crosby]:
  - Transcendent
  - User-Based
  - Conformance to Requirements
  - Product-Based
  - Value-Based

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Quality as conformance to requirements

- Quality **as conformance to requirements** is **a popular operational definition of quality**. Philip Crosby is probably the best-known advocate of this interpretation [Crosby, Quality is Free, p. 17].
- It is also the definition used in the well-known ISO 9000 quality standard.
- With this perspective, there is no ambiguity about what quality is.
- A quality product is one that conforms to specified requirements and design.
- According to this definition of quality, a **meal at McDonalds** is **high quality** as long as it conforms to the specifications of the franchise. By the same token a meal at the **Four Seasons** could be considered of lesser quality if, for example, the salad fork isn't chilled.

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Quality as conformance to requirements

- If this definition of quality bothers you it's probably because you are equating quality with luxury or goodness [Crosby].
  - The source of the confusion is in the use of the same term for two different concepts.
  - A manufacturer may plan a product with a certain level of quality (luxury or goodness). In order to deliver on this plan, the manufacturer will need to maintain high standards of quality (conformance to requirements) during production.

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Quality as conformance to requirements

- Quality as conformance to requirements is a good operational definition of quality because manufacturers of all types of products can
  - **specify quality goals** and
  - **control progress** towards the accomplishment of these goals

Quality as conformance to requirements

- Another potentially confusing aspect of defining quality as "conformance to requirements" is that it seems to disregard the **needs** of the **user**.
- Quality as conformance to requirements doesn't discount the importance of <u>meeting the needs of the user</u>, it just assumes that <u>these needs will be completely expressed in the requirements</u>.
    - This puts the burden on the requirements process to capture the true requirements with <u>accuracy</u> and <u>precision</u>.

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Quality as conformance to requirements

- The principle advantage of defining quality as conformance to requirements is that it simplifies the production or implementation phase of the product life cycle.
  - It is easier to manage and control the production or implementation phase of the product life cycle.
- When quality is defined as satisfaction of user needs the whole life cycle is working toward subjective hard-to-measure goals which makes the whole process hard to control.
- When quality is defined as conformance to requirements, the requirements phase isn't any easier to control, but the design and production phase are working toward an objective easier-to-measure goal--conformance to written requirements.
- The advantage of defining quality as conformance to requirements is that it makes it possible to define quality goals in specific terms and then measure and control progress towards these goals.
  - This perspective is the definition of quality preferred by manufactures because it provides an objective **easy-to-measure goal for quality** during the production phase of the product life cycle.

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Different perspectives on quality

- The following are **the most common perspectives** on **quality** (in general) [Garvin, Crosby]:
  - Transcendent
  - User-Based
  - Conformance to Requirements
  - Product-Based
  - Value-Based

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

The product-based approach to quality

- With the product-based approach to quality, differences in quality are measured by differences in the quantity of some ingredient or attribute of the product.
  - For example, one way to measure the quality of fabric is by **thread count**. Fabrics with a higher thread count are usually considered more desirable.
- In practice, the quality of a product is often a function of multiple attributes, some more important than others.
  - For example, **3 attributes** that can be used to measure the quality of a digital camera are resolution, light sensitivity and frame rate. These aren't the only attributes by which the quality of a digital camera can be measured and how much each attribute contributes to overall product quality will be different for different buyers.
  - Camera Quality = $W_{\text{Resolution}}$ * Resolution + $W_{\text{Sensitivity}}$ * Light Sensitivity + $W_{\text{Storage Capacity}}$ * Storage Capacity

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

The product-based approach to quality

- The challenge when defining product quality using this definition is determining the **attributes** and their **weights**.
    - If the choice of attributes and weights are **subjective**, there may be no choice of attributes and weights that is optimal for all customers.

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Different perspectives on quality

- The following are **the most common perspectives** on **quality** (in general) [Garvin, Crosby]:
  - Transcendent
  - User-Based
  - Conformance to Requirements
  - Product-Based
  - Value-Based

The value-based approach to quality

- The value-based approach to quality considers **value** to be **an aspect of quality**.
- Adding features to a product and controlling its defects, beyond a certain point, will likely increase its cost.
- Higher cost means less value. Tying value to quality means that **as the cost of a product goes up, its perceived quality goes down**.
- According to this approach, a quality product is one that has the features the customer desires, limited defects (if any), and is offered at a price that is acceptable to the customer.

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

The value-based approach to quality

- This approach to quality explains why, when shopping for flooring, linoleum starts to look better than tile after seeing the price of each.
  - In practice, this view of quality confuses product quality with desire for a product.
  - **Ex:** A customer may admire the quality of a pair of shoes, but upon seeing the price may loose interest in purchasing them. The quality of the shoes haven't changed but the customer's desire for owning a pair diminishes because they don't represent good value for the customer.

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

What is Software Quality?

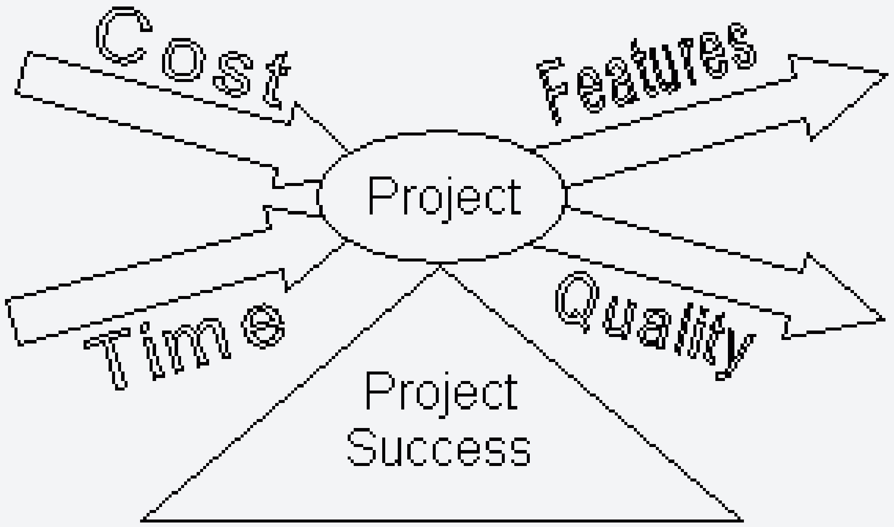
- The perspectives on quality presented are purposefully general-they apply to any product in any environment.
- The motivation for defining software `quality` is that it is one of the <u>4 constraints</u> or <u>variables</u> of a project (along with **`time`**, **`cost`** and **`features`**) that must be planned and managed.
    - You can't plan and manage the quality goals of a project unless there is **<u>universal agreement among all stakeholders about what quality is</u>**.

What is Software Quality?

- Define **time**? → No need: a day, week or month
- Define **cost**? → No need: money
- Define a **feature**? → No need: While stakeholders might argue over the utility of a feature, no one would argue whether or not something is a feature.
- However, as the multiple perspectives on quality presented above demonstrate, there is **no universal agreement** on the meaning of the term quality.
  - If the definition of software quality isn't well-defined and understood by all stakeholders, disagreements and misunderstandings are likely to follow.

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

What is Software Quality?



Project success is finding the right balance among cost, time, features and quality

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

What is Software Quality?

- **Project success is delivering the features and quality the customer desires at a cost and schedule that is acceptable.**
  - Cost and time must be balanced with features and quality.
  - Once a project is started, if there isn't enough time and money to deliver the features and quality promised, something has to give. It should be the least important constraint, but in practice it's hard to resist compromising the most convenient constraint.
  - **Cost**, **time**, **features** are well-defined and easily recognizable. If one of these is compromised, the effects will be immediately obvious.
  - On the other hand, if **quality** isn't well-defined it can be compromised in favor of other more visible project constraints without any immediate consequences. Compromise schedule, budget or features and you have to publicly acknowledge a failure to meet the original project goals.
  - Compromise on quality and it's someone else's problem for another day.
- The bottom line is: **when quality is well-defined, tradeoffs among project constraints are more likely to be based on true long-term priorities rather than short-term conveniences**.

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

What is Software Quality?

- Quality is important but it's not everything.
- Quality is only one component of project success. ***Project success is delivering desired features at an acceptable level of quality according to the agreed upon schedule and budget.***
- Users may be willing to accept a little less quality if it means faster delivery, less cost, or more features.
- **Engineers** have an especially hard time compromising on quality because:
    1. They are closely involved with product creation and so understandably have pride of authorship.
    2. Engineers are directly affected by poor quality because they have to maintain the software into the future.
    3. Conversely, engineers are less affected by compromises to the other project constraints.
- When it comes to setting project goals, quality is just one component of project success that must be balanced with other considerations such as cost and schedule.

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

The definition: Quality

- **Merriam-Webster Dictionary definition:** "*degree of excellence*".

Source: https://www.merriam-webster.com/dictionary/quality

The definition: Quality

- **ISO definition:** "*The totality* (sum, whole) *of features and characteristics of a product or service that bear on its ability to satisfy* ***stated*** *or* ***implied needs***".
  - In simpler words, one can say that a product has good quality when it "complies with the requirements specified by the client".
- When projected on analytical work, quality can be defined as "*delivery of reliable information within an agreed span of time under agreed conditions, at agreed costs, and with necessary aftercare*".

Source: http://www.fao.org/3/W7295E/w7295e03.htm

Quality Definition - IEEE

- The IEEE provides two definitions relevant to the topic: one for quality in general and another for software quality in particular. The IEEE standard glossary of SWE terminology defines quality as:
- **Quality** –
- (1) The degree to which a system, component, or **process** meets specified requirements.
- (2) The degree to which a system, component, or **process** meets customer or user needs or expectations. [IEEE Std. 610-12-1990]

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Quality Definition - IEEE

- This definition offers what are probably the **two most common interpretations** of the word quality,
- (1) conformance to requirements, and (2) measure of user satisfaction.
- As long as the requirements completely and accurately reflect user needs and expectations, these two definitions are equivalent.
- The definition for quality above is also broader than most definitions because it explicitly includes software processes as well as the products of these processes.
  - Process quality is important because the only practical way of producing a quality product is with a quality process.

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Software Quality Definition - IEEE

- The second definition from the IEEE is from the IEEE standard for quality metrics. It defines software quality as:
- **Software Quality** –
- *the degree to which software possesses a desired combination of attributes*. [IEEE Std 1061-1998]
- Despite its brevity, this definition incorporates both the user-based and product-based views of quality.
- Using the desired in the definition suggests a user-based view of quality where quality is relative to the desires and priorities of the users and project stakeholders.
- Using the word *attributes* in the definition implies that software quality is best measured by the degree to which certain product attributes are present.
  - The quality attributes of software are the familiar "ilities" of software including usability, reliability, maintainability, etc.

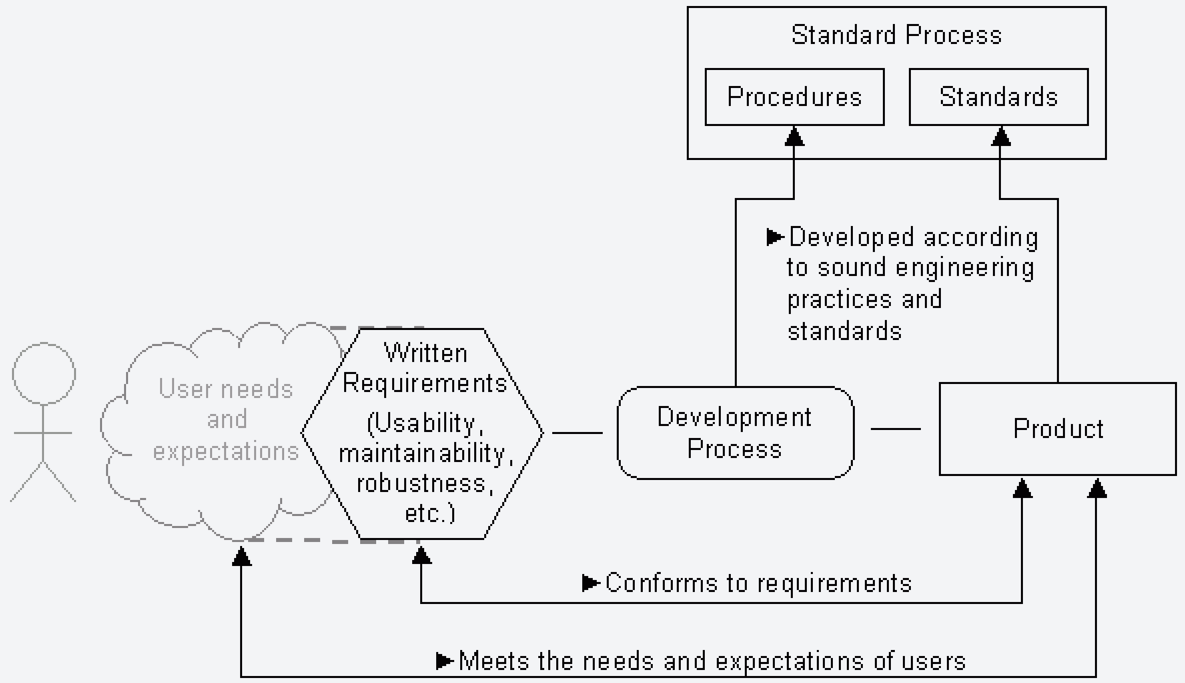
Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Software Quality (Refined-Merged) Definition

- **Software Quality –**
- degree to which the software, (1) conforms to specified requirements, (2) meets the needs and expectations of customers, users and stakeholders in general (3) is designed and developed according to sound engineering practices and standards.
  - The three parts to this definition are NOT three alternative definitions, but rather the three aspects of software quality.
  - Each contributes to the overall quality of the product. Software quality is the degree to which ALL three of these aspects are met.

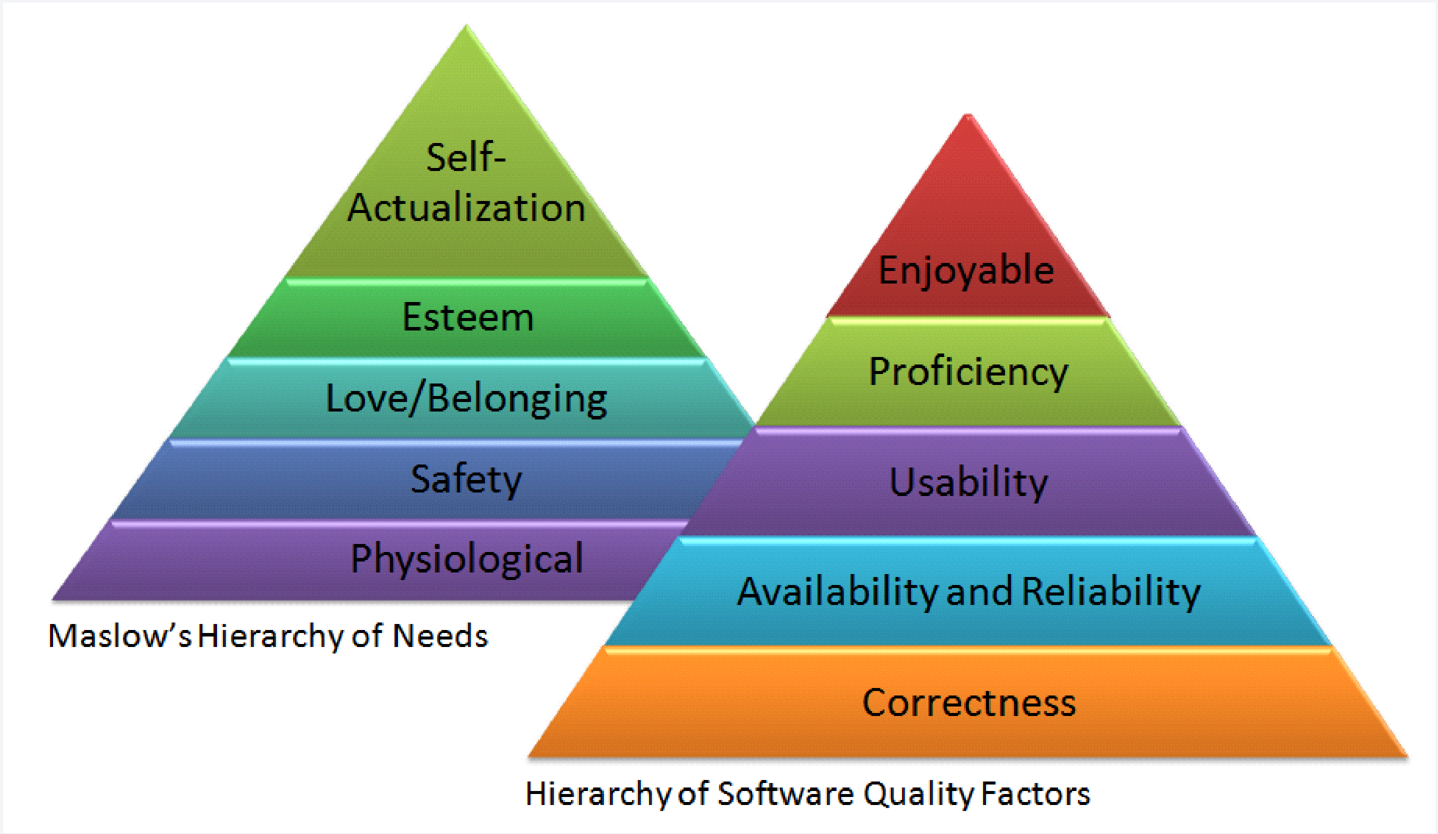
Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Software Quality (Refined-Merged) Definition


Standard Process
Procedures
Standards
Developed according to sound engineering practices and standards
User needs and expectations
Written Requirements (Usability, maintainability, robustness, etc.)
Development Process
Product
Conforms to requirements
Meets the needs and expectations of users

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Hierarchical Nature of Software Quality



Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

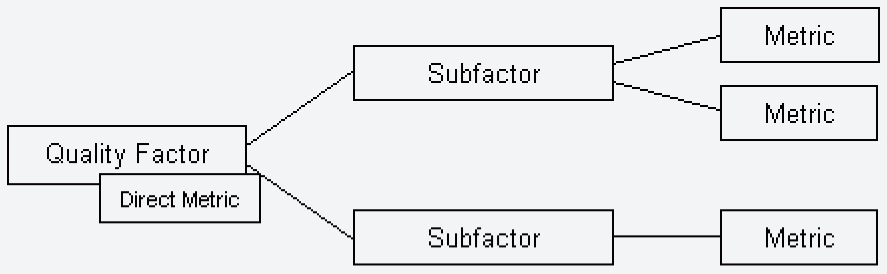
Software Quality Models

- Models are common tools in software development.
  - Software life-cycle models (spiral, waterfall, etc.),
  - Process improvement models (CMMI, ISO 9000, etc.)
  - Cost estimation models (COCOMO, SLIM, etc.)
- Models are used to manage complexity.
- Models provide an abstraction of a system or phenomenon from a particular perspective for a particular purpose.
- Software quality models provide a framework for **expressing** and **measuring** software quality as **a collection of desirable attributes** (the product-based perspective on quality mentioned above).

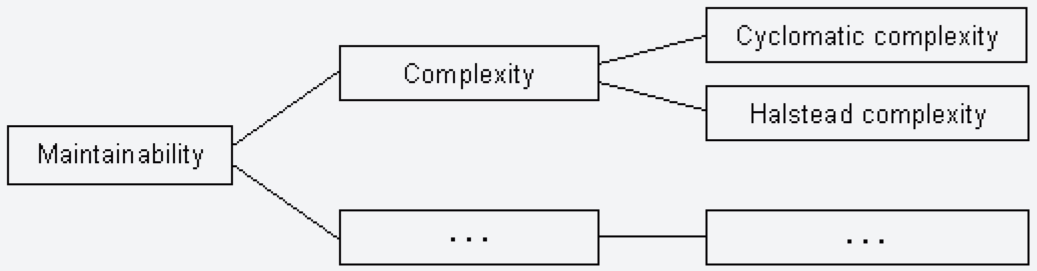
Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Software Quality Models

Framework...



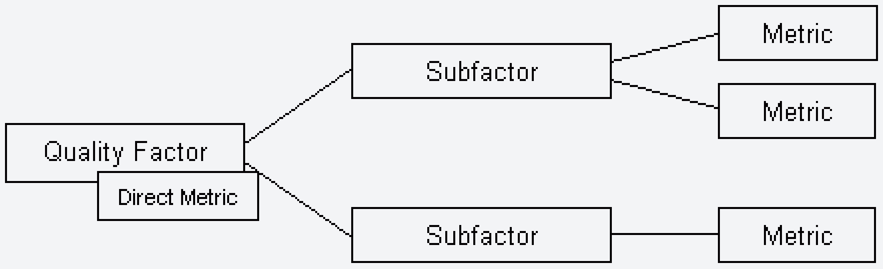
Example...



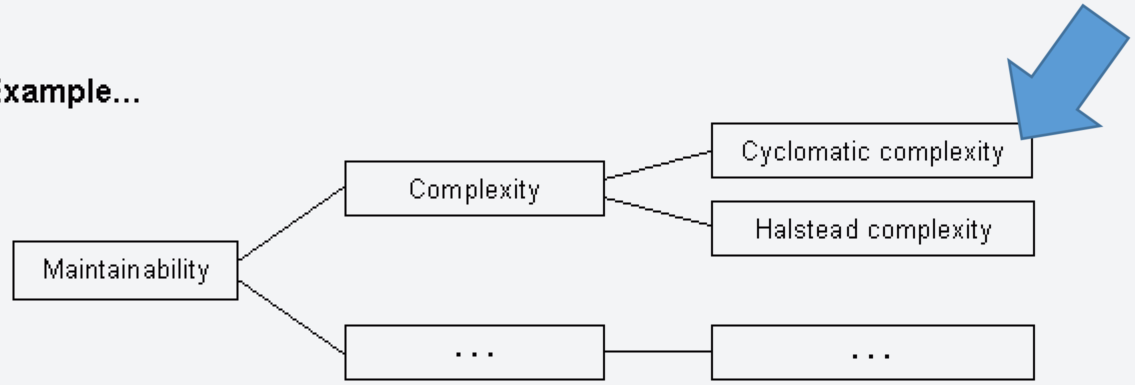
Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Software Quality Models

Framework...

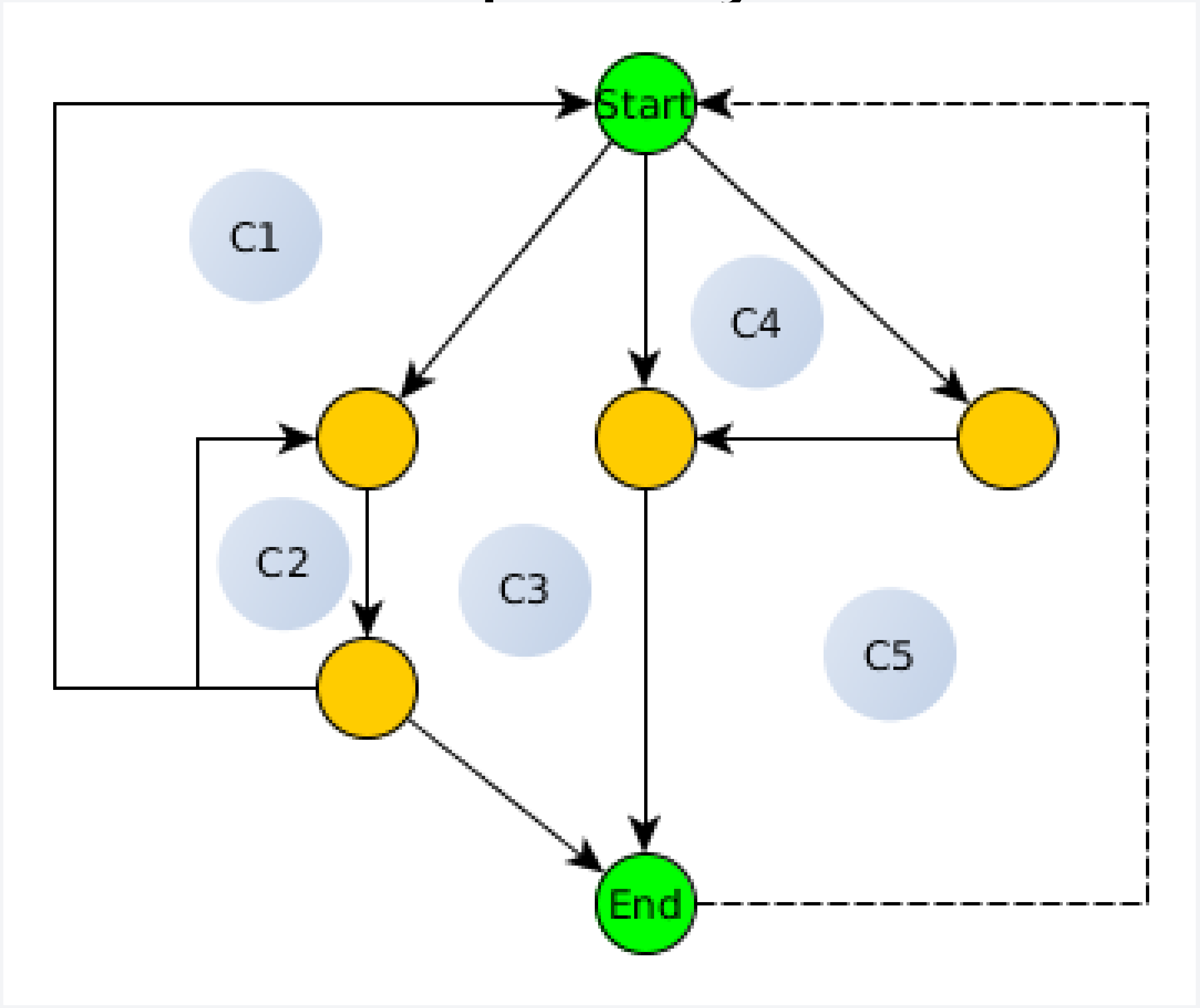


Example...

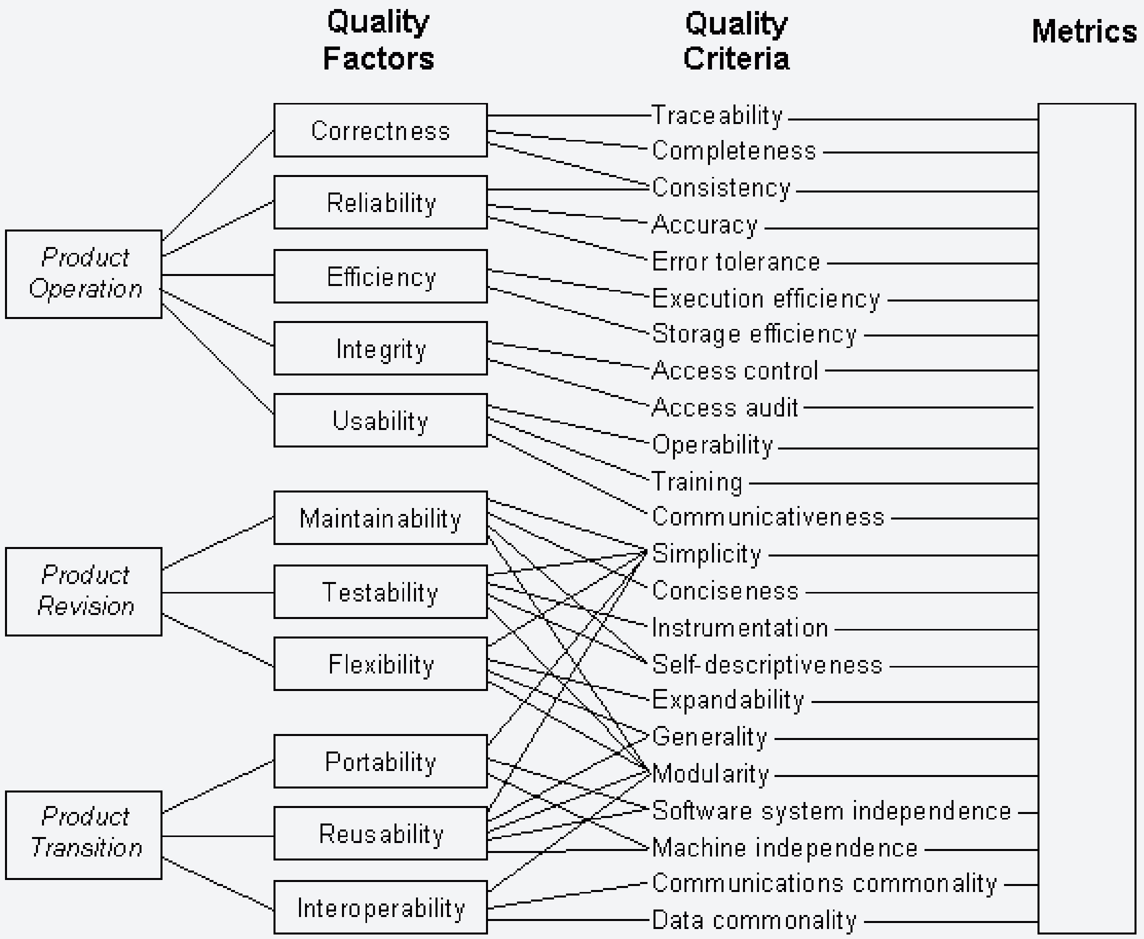


Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Cyclomatic Complexity



McCall's Quality Model



Source:
http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

McCall's Quality Model

- Like most SQ models the high level quality factors in McCall's model generally represent external emergent system properties.
  - Emergent system properties are system characteristics not attributable to specific elements of the system but rather those that emerge from the structure and interaction of individual system elements.
- Low level quality factors in the model refer to more tangible internal properties of the system.

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

McCall's Quality Model

- McCall's model links high-level quality characteristics of interest to end users to low-level system properties that developers understand.
    - It provides a vocabulary of quality factors that users and managers can use when setting the quality goals of a project.
    - Ex: End users may decide that usability is an important quality requirement. When quality requirements are stated in terms of **high-level quality factors**, it's not always clear to technical staff what specific product properties are needed in order to accomplish the desired system qualities.
- McCall's quality model **breaks down** high-level quality factors into their constituent components or quality criteria.
    - These low level quality criteria are more familiar to technical staff and represent product properties that can be built directly into work products.

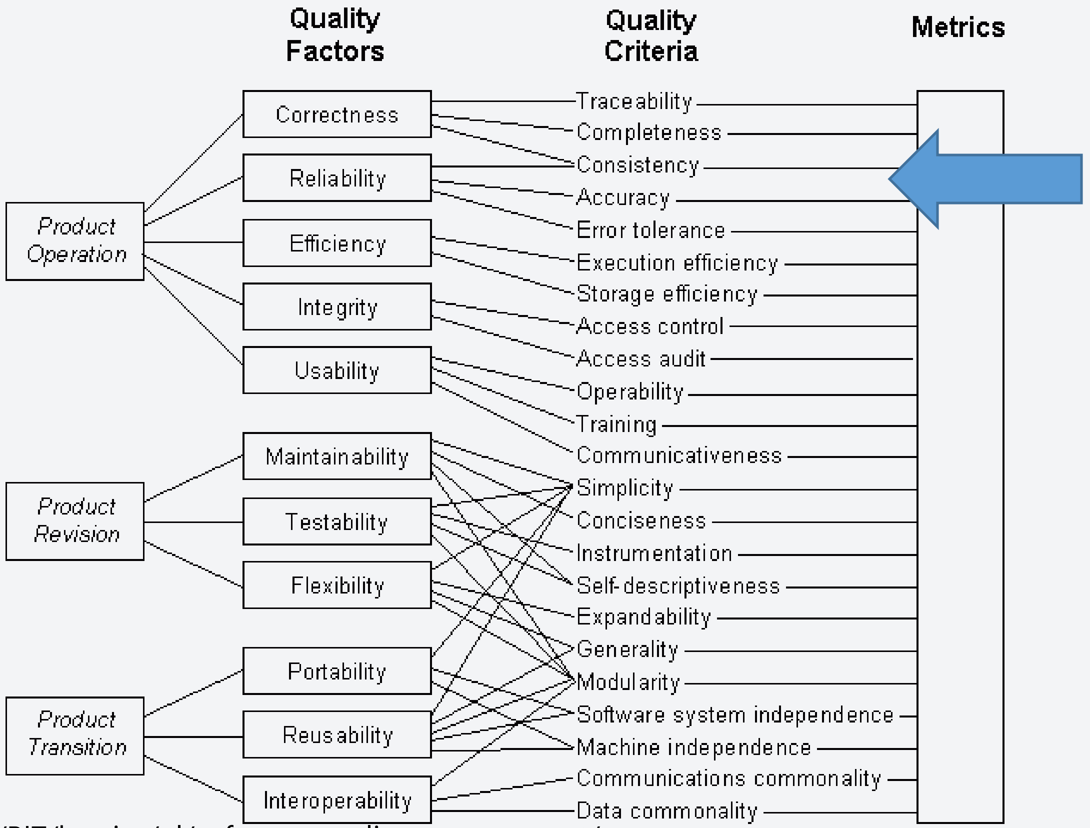
Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

McCall's Quality Model

- Besides suggesting quality characteristics and their relationships, McCall's model--and the others that will be examined--also **suggest metrics (one or more metrics) for measuring the degree to which quality characteristics** are present.
- McCall's model breaks down high-level quality factors that are difficult to measure into low-level detailed quality criteria which are more tangible and measurable.
- Ex: **Storage efficiency** may have the associated metrics:
    1. total disk space required
    2. average internal memory required while the program is running
- *Every quality criteria has at least one metric, but metrics may also be associated with some high level quality factors.*

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

McCall's Quality Model



Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

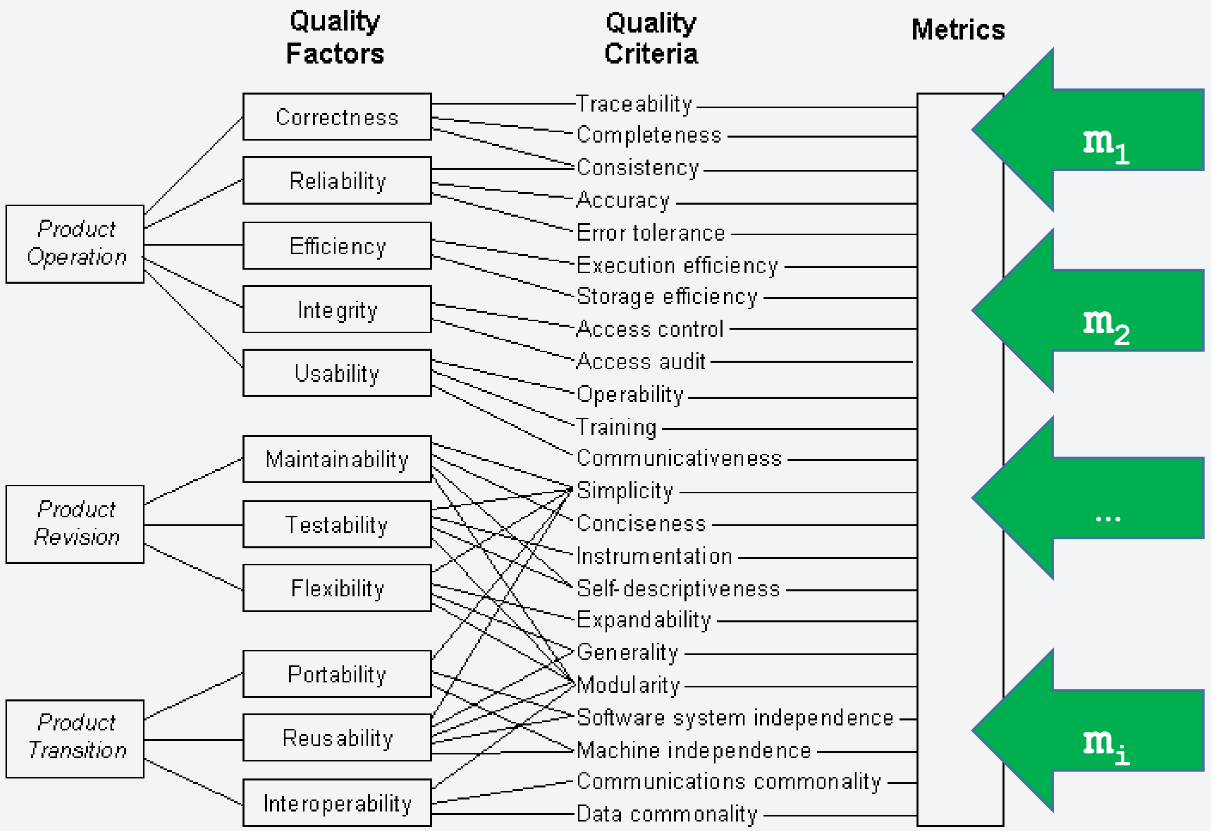
McCall's Quality Model

- McCall's model provides a quantitative measure of quality.
- Metric values are expressed in quantitative terms and the relationship between quality criteria and quality factors is formally expressed with a **linear equation** of the form:
- $QF_a = c_1 * m_1 + c_2 * m_2 + \ldots + c_n * m_n$ where:
  - QF_a is the degree to which quality factor a is present.
  - c_i is a constant which denotes the relative importance of the measurement m_i to quality factor a.
  - This constant may be determined by using regression analysis on historical data from similar projects.
  - m_i is a metric value associated with quality factor QF_a .

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

QF_a

McCall's Quality Model

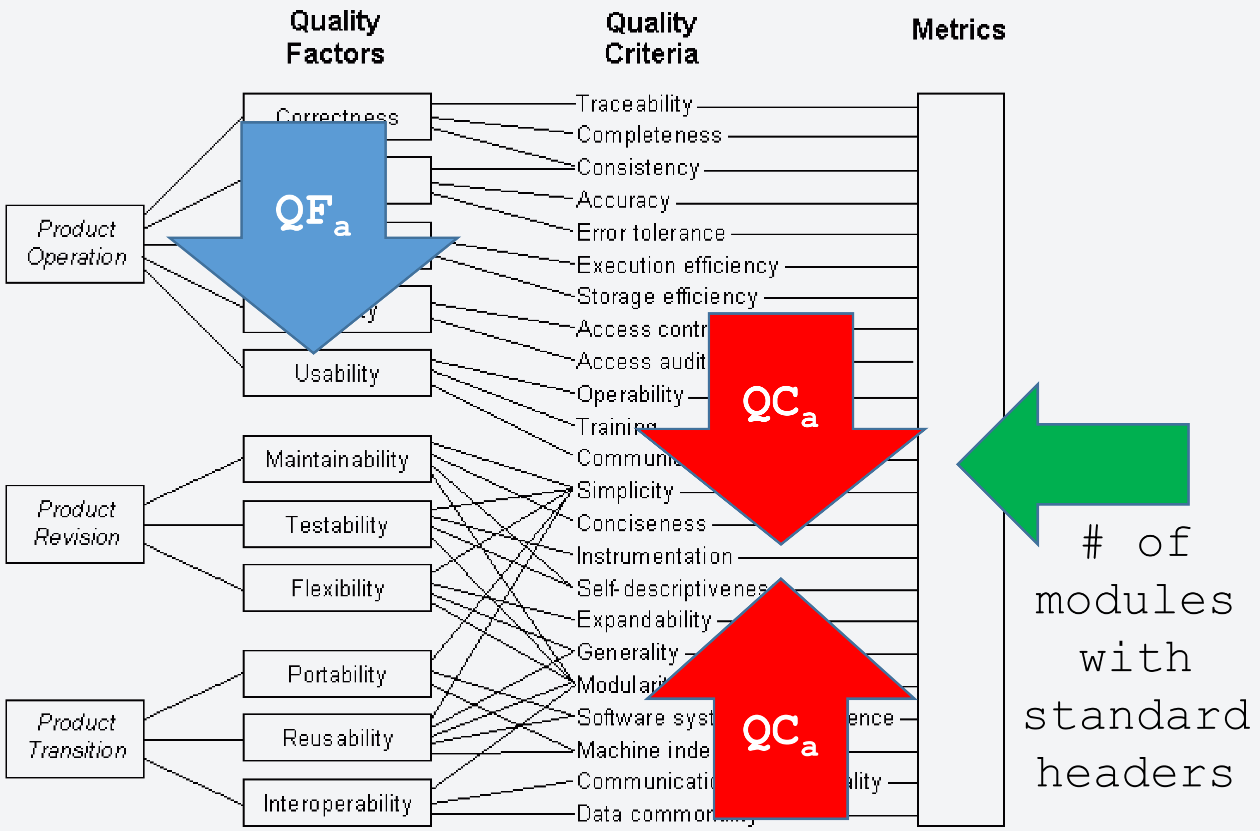


Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

McCall's Quality Model - Example

- **Ex:** The quality criteria "self-descriptiveness" which is associated with quality factor *maintainability* might have the metric
    - "*number of modules with standard headers*".
        - If 9 out of 10 modules have a standard header, and historical data suggests that this metric has a 30% influence on the quality factor maintainability,
    - the formula becomes:
- $QF_{\texttt{maintainability}} = \ldots + 30\% * 9/10$

McCall's Quality Model - Example



Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

McCall's Quality Model

- The formal relationship between metrics and QF_i is captured in the equation's coefficients (c_i).
- McCall showed that for a limited domain significant correlation could be established between predicted **quality** and actual **quality**.
- However, its not clear that a quality model calibrated for one domain is generalizable to other domains.
- Like other software models calibrated from empirical data, the **accuracy** of **McCall's model** depends heavily on tuning and using the model on projects that are similar in type.

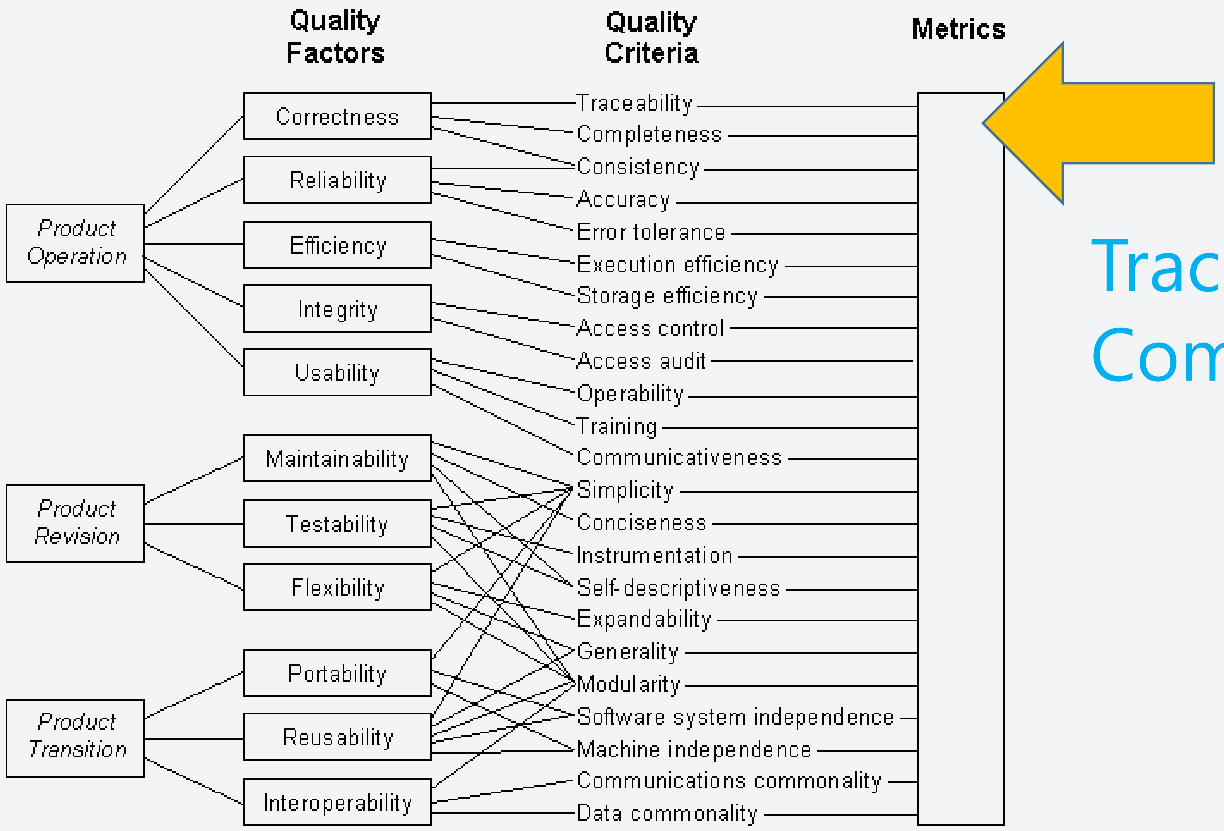
Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

McCall's Quality Model

- One of the ways McCall's SQ model is distinguished from the others that are presented here is that *it defines software broadly to include NOT ONLY code but also requirements and design*.
    - For example, traceability and completeness are two desirable characteristics of requirements and design documents.

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

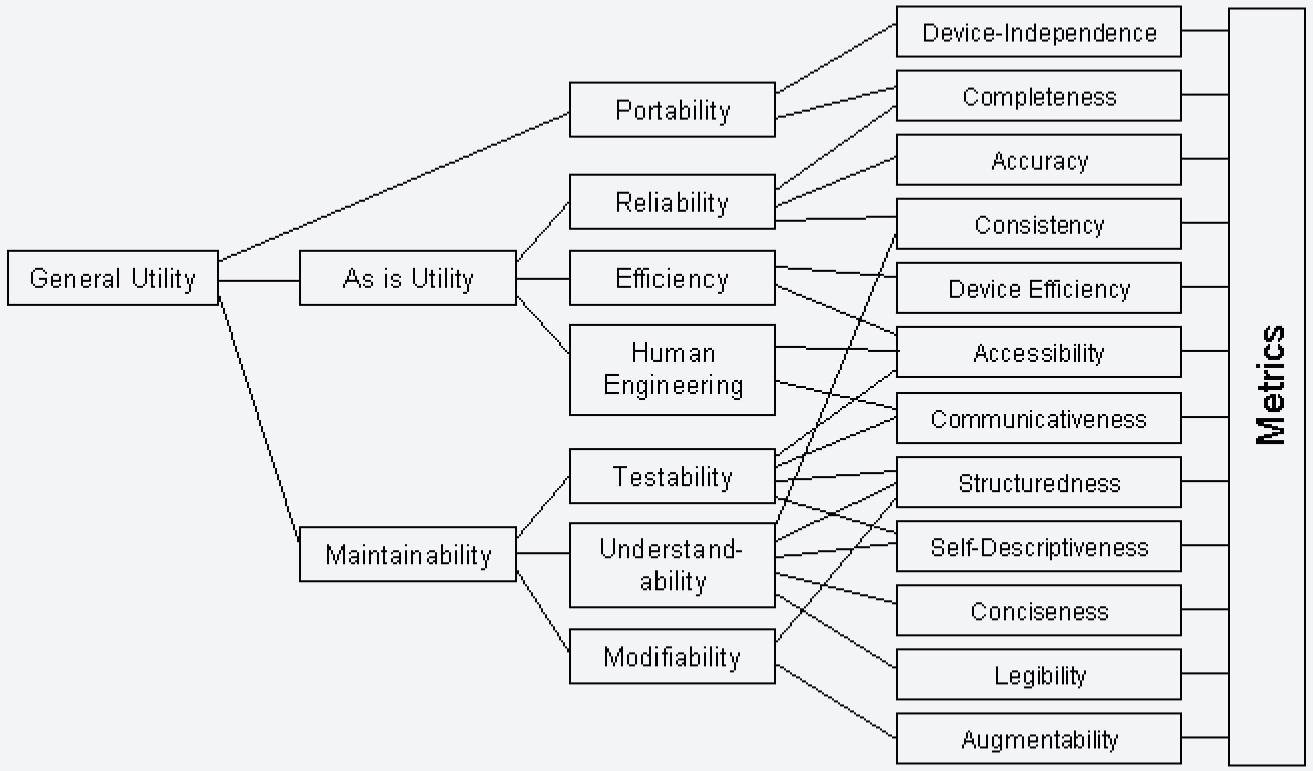
McCall's Quality Model - Example



Traceability
Completeness

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Boehm's Quality Model



Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Boehm's Quality Model

- Using the similar approach in McCall's model, Boehm's model defines the overall system quality (a.k.a general utility) is the aggregation of
  - portability
  - usability (a.k.a "utility")
  - maintainability
- Like McCall's SQ model, Boehm's quality model is hierarchical with metrics defined for lower-level quality attributes.

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Boehm's Quality Model vs. McCall's

- One obvious difference is its factoring of quality characteristics.
  - As an example, in McCall's model maintainability and testability are at the same level which means in McCall's model *a product can be maintainable without being testable and vice versa*.
  - In Boehm's model, testability is a subfactor of maintainability, which means *an increase in testability implies an increase in maintainability*.
    - By itself this isn't necessary an important distinction but it does point out the difficulty of having a definitive quality model.

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Boehm's Quality Model

- Boehm's model applies only to code.
- Boehm's quality model includes 151 metrics which provide measurements of 12 primitive quality characteristics (a.k.a. primitive constructs).
- Each metric is expressed in the form of a question.
    - Stated in this way the metrics form a checklist of good programming practice.

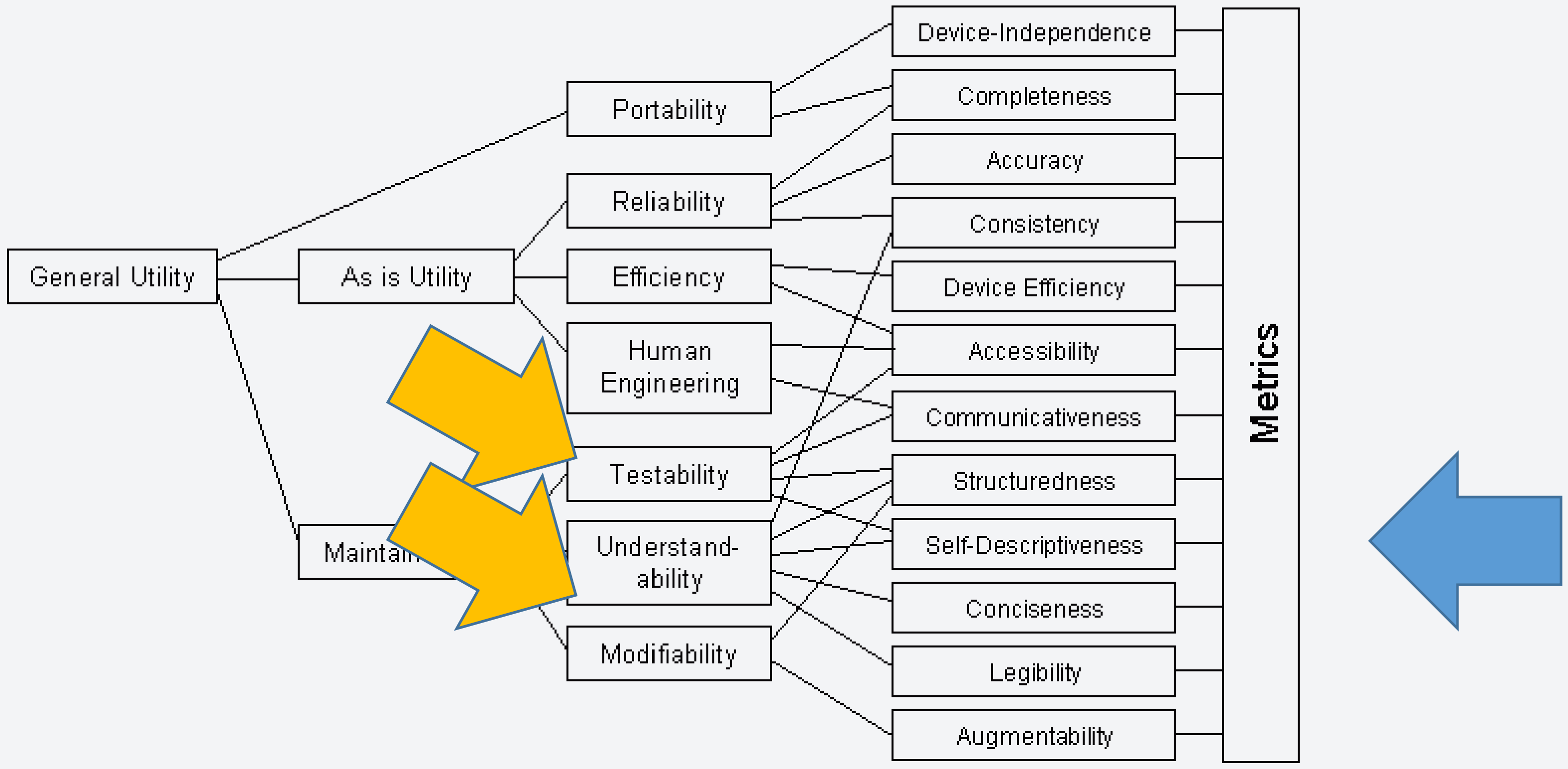
Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Boehm's Quality Model - Example

- Next slides show a partial checklist for the quality characteristic **`self-descriptiveness`**.
  - **Definition of self-descriptiveness**: is the extent to which a program or module contains enough information for the reader to understand the function of the module and its dependencies.
  - **`Self-descriptiveness`** is a **measure** of a product's **`testability`** and **`understandability`**.

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Boehm's Quality Model - Example



Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

A partial checklist Example

1. Does each program module contain a header block of commentary which describes program name, purpose, modification history, and assumptions?
2. Are the functions of the modules as well as inputs/outputs adequately defined to allow module testing?
3. Where there is module dependence, is it clearly specified by commentary, program documentation, or inherent program structure.
4. Are variable names descriptive of the physical or functional property represented?
5. Do functions contain adequate descriptive information (i.e., comments) so that the purpose of each is clear?

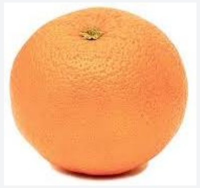
Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

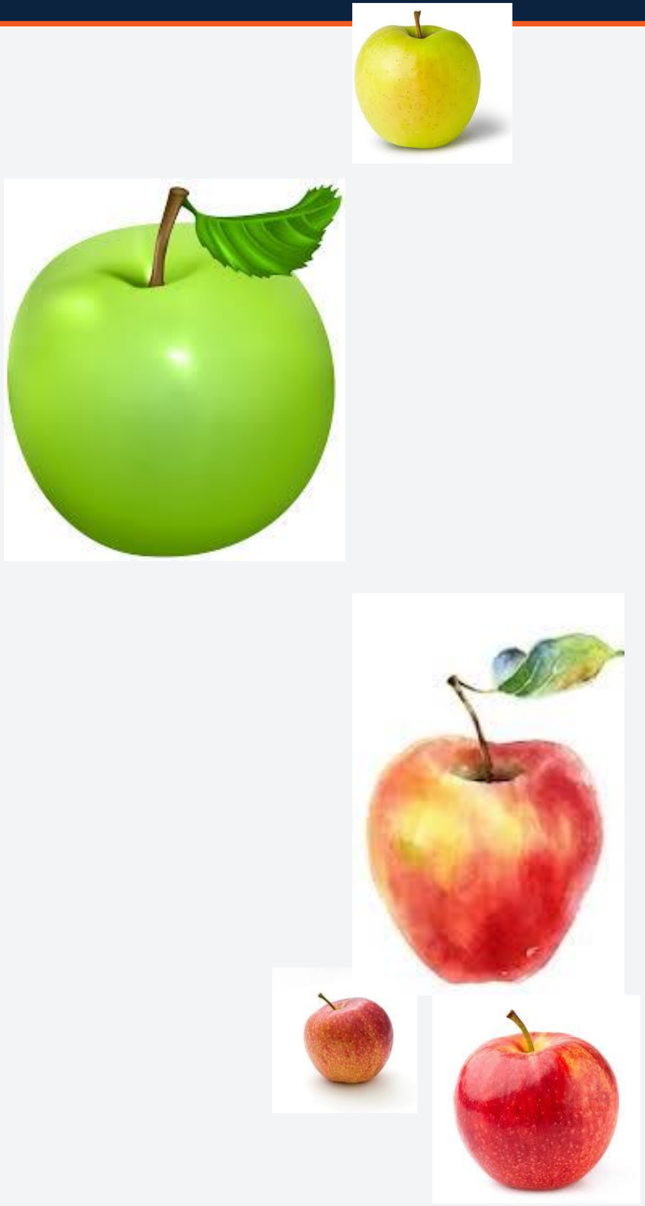
Boehm's Quality Model

- By expressing quality metrics in the form of checklists, the metrics used for quality measurement can also be used for code reviews and general process documentation and improvement.
  - The metrics developed in Boehm's original work are for **Fortran programs**, making them mostly outdated by today's standards. ☹
- However, the quality characteristics and their factorings in Boehm's Quality Model are just as applicable today as when they were proposed in 1978, demonstrating that the principles of quality are timeless. ☺

Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

Measurement Scales and Scale Types



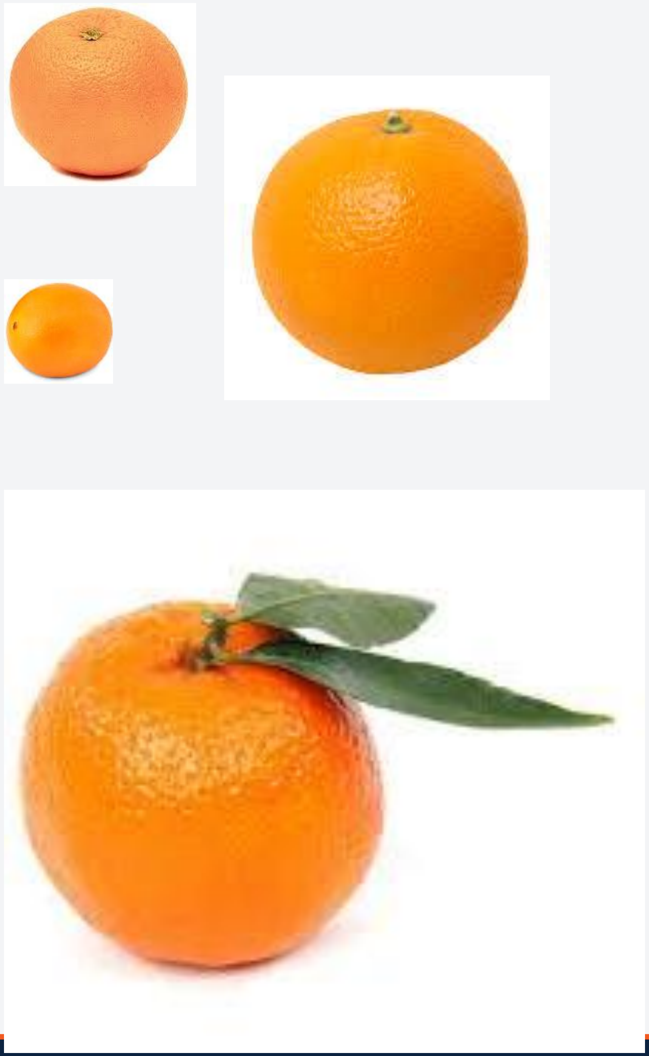


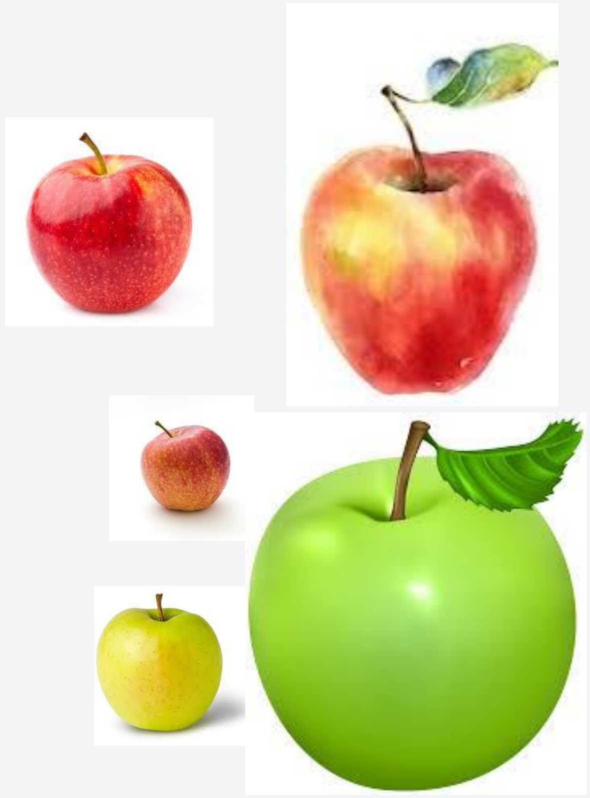
Nominal

Category → Color

Nominal

Category → Type



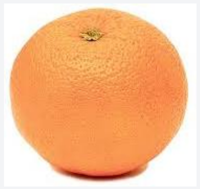


Ordinal

→Order

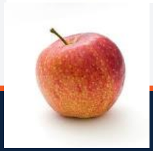
1st

2nd



3rd

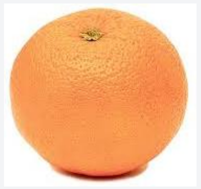
4th



Ordinal

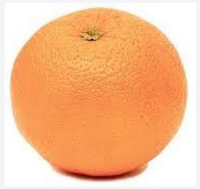
1st

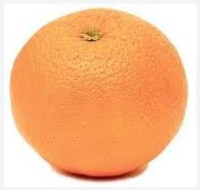
2nd

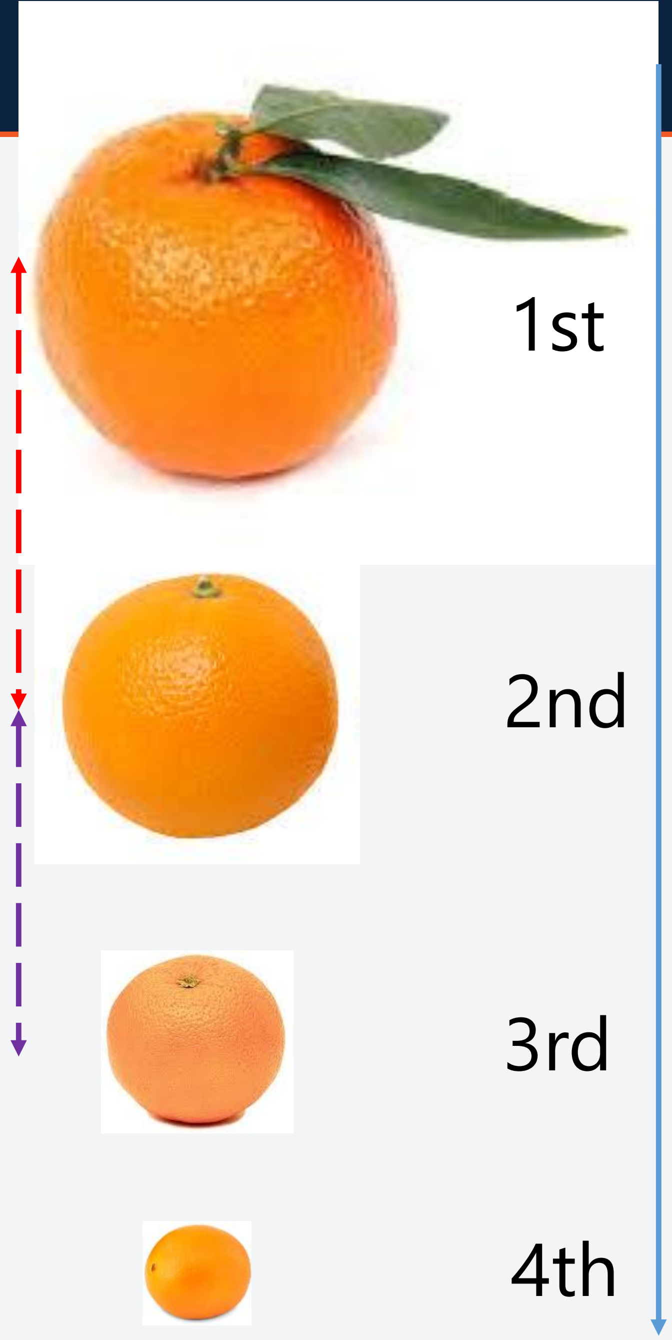


3rd

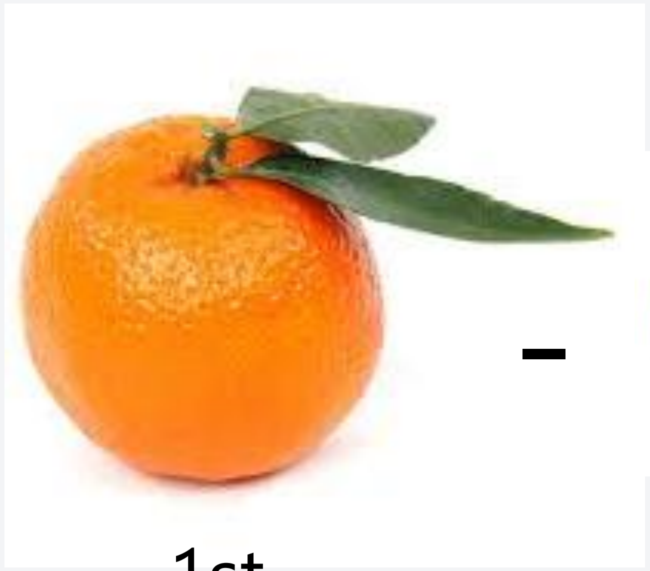
4th

> 

 >



How much bigger?



1st

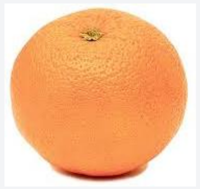
-

2nd

=?

2nd

-

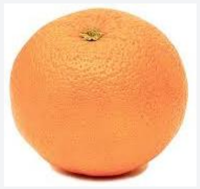


3rd

=?

1st

2nd

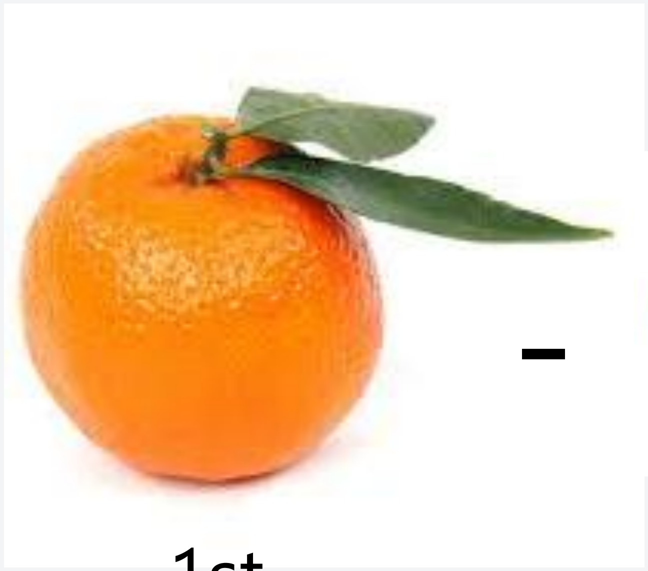

3rd

4th

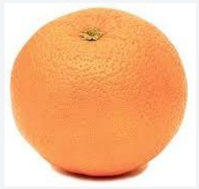
How much bigger?

length

weight

 - =?

1st 2nd

-  =?

2nd 3rd

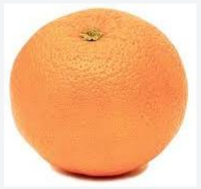
How much bigger?

length

weight

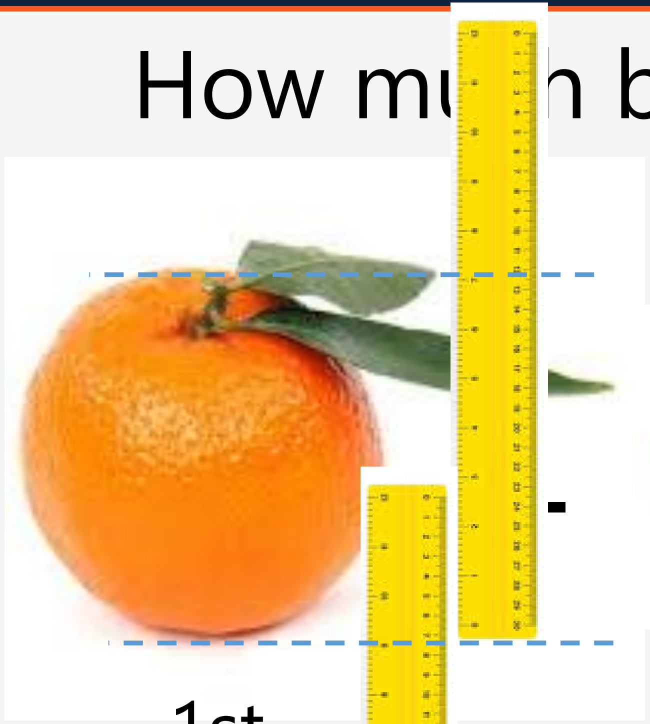
1st

2nd



3rd

4th



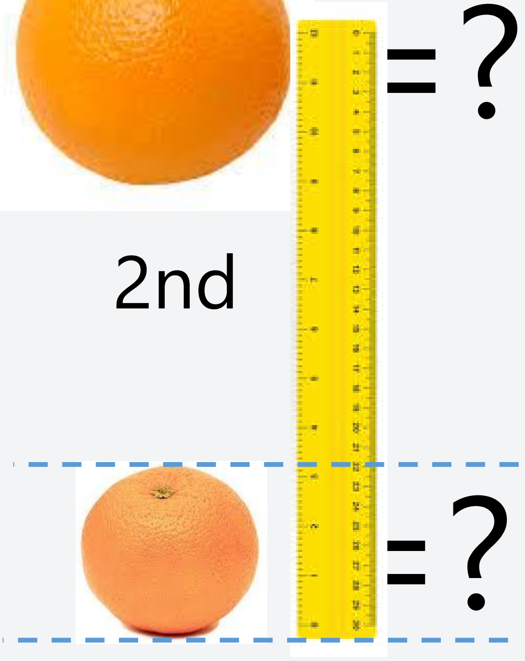
1st

2nd

=?

2nd

–



=?

3rd

Measurement Scales and Scale Types

- We classify measurement scales as one of **five major types**:
    1. Nominal
    2. Ordinal
    3. Interval
    4. Ratio
    5. Absolute
- There are other scales (ex: logarithmic scale) that can be defined.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Nominal Scale

- Suppose we define classes or categories, and then place each entity in a particular class or category, based on the value of the attribute.
- This categorization is the basis for the most primitive form of measurement, the *nominal scale*.
- Thus, the nominal scale has two major characteristics:
  - The empirical relation system **consists only of different classes**; there is **no notion of ordering** among the classes.
  - Any distinct numbering or symbolic representation of the classes is an acceptable measure, but there is **no notion of magnitude** associated with the numbers or symbols.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Nominal Scale Example

- Suppose that we are investigating the set of all known **software faults** in our code, and we are trying to capture the *location* of the faults. Then we seek a measurement scale with faults as entities and location as the attribute.
- We can use a common but primitive mapping to identify the fault location: we denote a **fault** as `specification`, `design`, or `code`, according to where the fault was first introduced.
  - Notice that this classification imposes no judgment about which class of faults is more severe or important than another.
  - However, we have a clear distinction among the classes, and every fault belongs to exactly one class.
- Any mapping, *M*, that assigns the 3 different classes to 3 different numbers satisfies the representation condition and is therefore an acceptable measure.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Nominal Scale Example

- The mappings M_1 and M_2 defined by are acceptable. In fact, any two mappings, *M* and *M′*, will always be related in a special way: *M′* can be obtained from *M* by a one-to-one mapping. The mappings need not involve numbers; distinct symbols will suffice. Thus, the class of admissible transformations for a nominal scale measure is the set of all **one-to-one mappings**.

$$M_1(x) = \begin{cases} 1, & \text{if } x \text{ is specification fault} \\ 2, & \text{if } x \text{ is design fault} \\ 3, & \text{if } x \text{ is code fault} \end{cases}$$

$$M_2(x) = \begin{cases} 101, & \text{if } x \text{ is specification fault} \\ 2.73, & \text{if } x \text{ is design fault} \\ 69, & \text{if } x \text{ is code fault} \end{cases}$$

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Ordinal Scale

- We can often **augment** the nominal scale with information about an **ordering** of the classes or categories creating an *ordinal scale*.
- The ordering leads to analysis not possible with nominal measures.
- The ordinal scale has the following characteristics:
    - The empirical relation system consists of classes that are **ordered with respect to the attribute**.
    - Any mapping that preserves the ordering (i.e., any **monotonic** function) is acceptable.
    - The numbers represent ranking only, so addition, subtraction, and other arithmetic operations have NO MEANING.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Ordinal Scale Example

- Suppose our set of entities is a set of software modules, and the attribute we wish to capture quantitatively is **`complexity`**.
- Initially, we may define 5 distinct classes of module complexity: *trivial*, *simple*, *moderate*, *complex*, and *incomprehensible*.
- There is an implicit order relation of *less complex than* on these classes; that is, all trivial modules are less complex than simple modules, which are less complex than moderate modules, etc.
- In this case, since the measurement mapping must preserve this ordering, we cannot be as free in our choice of mapping as we could with a nominal measure.
- Any mapping, *M*, must map each distinct class to a different number, as with nominal measures. But we must also ensure that the more complex classes are mapped to bigger numbers.
- Therefore, *M* must be a monotonically increasing function.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Ordinal Scale Example

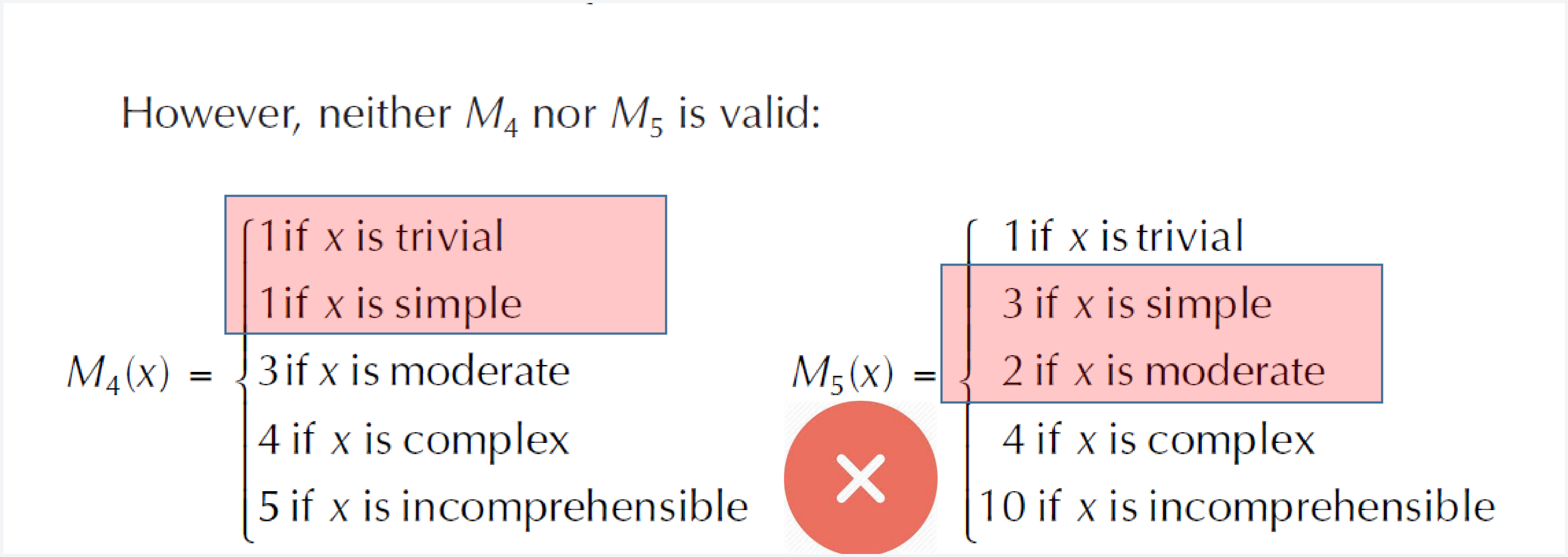
- Each of the mappings M_1, M_2, and M_3 is a valid measure, since each satisfies the representation condition.

$$M_1(x) = \begin{cases} 1 \text{ if } x \text{ is trivial} \\ 2 \text{ if } x \text{ is simple} \\ 3 \text{ if } x \text{ is moderate} \\ 4 \text{ if } x \text{ is complex} \\ 5 \text{ if } x \text{ is incomprehensible} \end{cases} \qquad M_2(x) = \begin{cases} 1 \text{ if } x \text{ is trivial} \\ 2 \text{ if } x \text{ is simple} \\ 3 \text{ if } x \text{ is moderate} \\ 4 \text{ if } x \text{ is complex} \\ 10 \text{ if } x \text{ is incomprehensible} \end{cases}$$

$$M_3(x) = \begin{cases} 0.1 \text{ if } x \text{ is trivial} \\ 1001 \text{ if } x \text{ is simple} \\ 1002 \text{ if } x \text{ is moderate} \\ 4570 \text{ if } x \text{ is complex} \\ 4573 \text{ if } x \text{ is incomprehensible} \end{cases}$$

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Ordinal Scale Example



However, neither M_4 nor M_5 is valid:

$$M_4(x) = \begin{cases} 1 \text{ if } x \text{ is trivial} \\ 1 \text{ if } x \text{ is simple} \\ 3 \text{ if } x \text{ is moderate} \\ 4 \text{ if } x \text{ is complex} \\ 5 \text{ if } x \text{ is incomprehensible} \end{cases} \qquad M_5(x) = \begin{cases} 1 \text{ if } x \text{ is trivial} \\ 3 \text{ if } x \text{ is simple} \\ 2 \text{ if } x \text{ is moderate} \\ 4 \text{ if } x \text{ is complex} \\ 10 \text{ if } x \text{ is incomprehensible} \end{cases}$$

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Interval Scale Type

- The interval scale carries more information and is more powerful than **nominal** or **ordinal** scale.
- This scale captures info about **the size of the intervals** that separate the classes, so that we can in some sense understand the size of the jump from one class to another.
- Thus, an interval scale can be characterized in the following way:
  - An interval scale preserves order, as with an ordinal scale.
  - An interval scale preserves differences but not ratios. That is, **we know the difference between any two of the ordered classes** in the range of the mapping, but computing the ratio of two classes in the range does not make sense.
  - Addition and subtraction are acceptable on the interval scale, but NOT multiplication and division.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Interval Scale Type Example 1

- We can measure air temperature on a Fahrenheit or Celsius scale.
- Thus, we may say that
  - it is usually 20° Celsius on a summer's day in London,
  - while it may be 30° Celsius on the same day in Washington, DC.
- The interval from one degree to another is the same, and we consider each degree to be a class related to heat.
- That is, moving from 20° to 21° in London increases the heat in the same way that moving from 30° to 31° does in Washington.
- However, we CANNOT say that it is 2/3 as hot in London as Washington; neither can we say that it is 50% hotter in Washington than in London. Similarly, we CANNOT say that a 90° Fahrenheit day in Washington is twice as hot as a 45° Fahrenheit day in London.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Interval Scale Type Example 2

- The timing of an event's occurrence is a classic use of interval scale measurement.
  - We can measure the timing in units of years, days, hours, or some other **standard measure**, where each time is noted relative to a given fixed event.
- Software development projects can be measured, by referring to the project's start day.
  - We say that we are on day 87 of the project, when we mean that we are measuring 87 days from the first day of the project.
  - Thus, using these conventions, it is meaningless to say "*Project X started twice as early as project Y*" but meaningful to say "*the time between project X's beginning and now is twice the time between project Y's beginning and now*."

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Interval Scale Type Example 2

- On a given project, suppose the PM is measuring time in months from the day work started: April 1, 2020.
- But the contract manager is measuring time in years from the day that the funds were received from the customer: January 1, 2021.
- If M is the project manager's scale and M' the contract manager's scale, we can transform the contract manager's time into the project manager's by using the following admissible transformation:

$$M = 12M' + 9$$

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Interval Scale Type Example 3

- Suppose that the difference in complexity between a <u>trivial</u> and <u>simple</u> system is the same as that between a <u>simple</u> and <u>moderate</u> system.
- Then any interval measure of complexity must preserve these differences.
  - Where this equal step applies to each class, we have an attribute measurable on an interval scale.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Interval Scale Type Example 3

$$M_1(x) = \begin{cases} 1 \text{ if } x \text{ is trivial} \\ 2 \text{ if } x \text{ is simple} \\ 3 \text{ if } x \text{ is moderate} \\ 4 \text{ if } x \text{ is complex} \\ 5 \text{ if } x \text{ is incomprehensible} \end{cases}$$

$M_2 = 2M_1 - 2$

$$M_2(x) = \begin{cases} 0 \text{ if } x \text{ is trivial} \\ 2 \text{ if } x \text{ is simple} \\ 4 \text{ if } x \text{ is moderate} \\ 6 \text{ if } x \text{ is complex} \\ 8 \text{ if } x \text{ is incomprehensible} \end{cases}$$

$M_3 = 2M_1 + 1.1$

$$M_3(x) = \begin{cases} 3.1 \text{ if } x \text{ is trivial} \\ 5.1 \text{ if } x \text{ is simple} \\ 7.1 \text{ if } x \text{ is moderate} \\ 9.1 \text{ if } x \text{ is complex} \\ 11.1 \text{ if } x \text{ is incomprehensible} \end{cases}$$

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Ratio Scale Type

- Sometimes, we would like to be able to say that *one liquid is twice as hot as another*, or *that one project took twice as long as another*.
- This need for ratios gives rise to the ratio scale, the most useful scale of measurement, and one that is **common in the physical sciences**.
- A *ratio scale* has the following characteristics:
  - It is a measurement mapping that preserves ordering, the size of intervals between entities, and ratios between entities.
  - There is **a zero element**, representing total lack of the **attribute**.
  - The measurement mapping must start at zero and increase at equal intervals, known as units.
  - All arithmetic can be meaningfully applied to the classes in the range of the mapping.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Ratio Scale Type Example

- The length of software code is also measurable on a ratio scale. As with other physical objects, we have empirical relations like *twice as long*.
- The Notion of a zero-length object exists—**an empty piece of code**.
- We can measure program length in a variety of ways, including **LOC**, **KLOC**, the **# of characters** contained in the program, the **# of executable statements**, and more.
- Suppose *M* is the measure of program length in LOC, while *M′* captures length as # of characters. Then we can transform one to the other by computing $M' = aM$, where *a* is the average # of characters per LOC.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Absolute Scale Type

- As the scales of measurement carry more information, the defining classes of admissible transformations have become increasingly restrictive.
- The absolute scale is the most restrictive of all.
- For any two measures, *M* and *M*′, there is **`only one admissible transformation`**: the identity transformation.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Absolute Scale Type

- That is, there is only one way in which the measurement can be made, so *M* and *M'* must be equal. The *absolute scale* has the following properties:
  - The measurement for an absolute scale is made simply by counting the number of elements in the entity set.
  - The attribute always takes the form "# of occurrences of *x* in the entity."
  - There is only one possible measurement mapping, namely the actual count, and there is only one way to count elements.
  - All arithmetic analysis of the resulting count is meaningful.

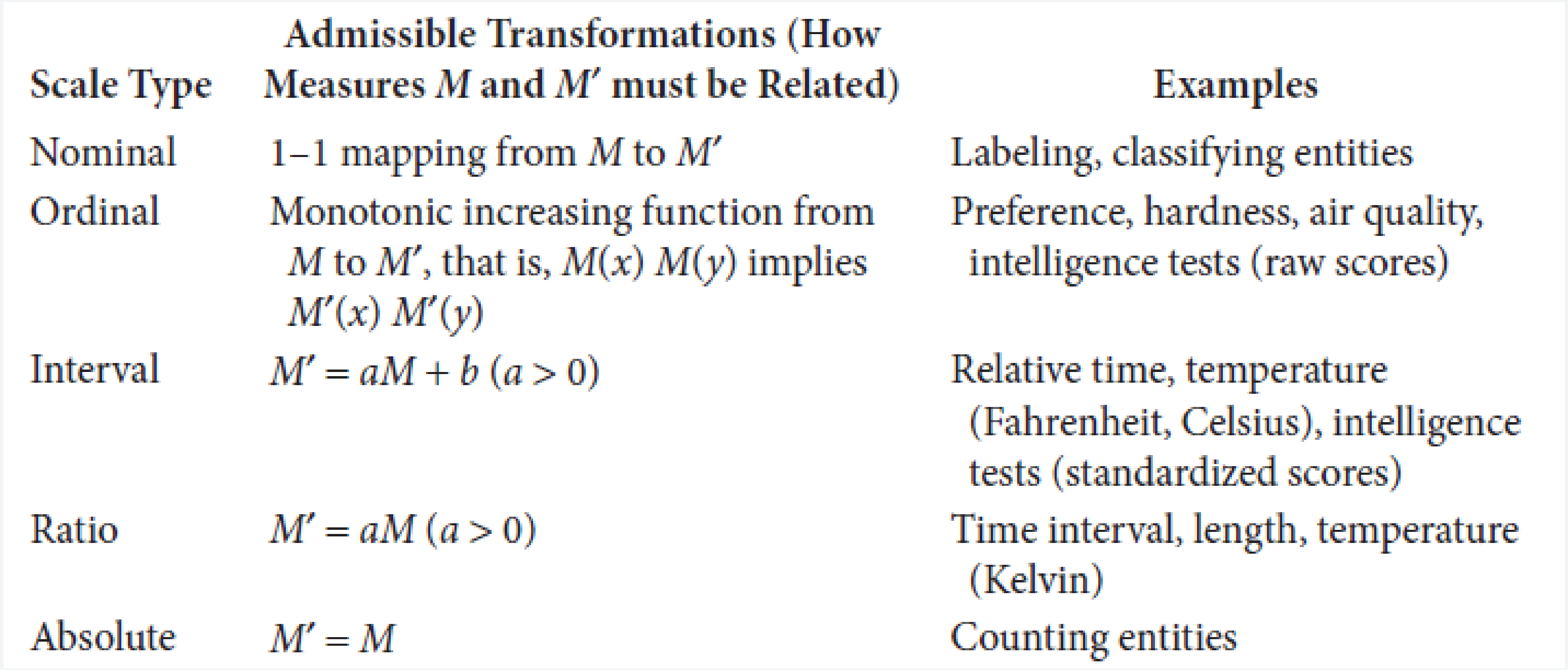
Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Absolute Scale Type Example

- A common mistake is to assume that LOC is an *absolute* scale measure of length, because it is obtained by counting.
- However, it is the *attribute* that determines the scale type.
- As we have seen, the length of programs cannot be absolute, because there are many different ways to measure it
    - LOC,
    - KLOC,
    - # of characters
    - # of bytes
- It is **incorrect** to say that LOC is an absolute scale measure of program length.
- However, LOC is an absolute scale measure of the **attribute** "**`number of lines of code`**" of a program. For the same reason, "number of years" is a ratio scale measure of a person's age; it cannot be an absolute scale measure of age, because we can also measure age in months, hours, minutes, or seconds.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

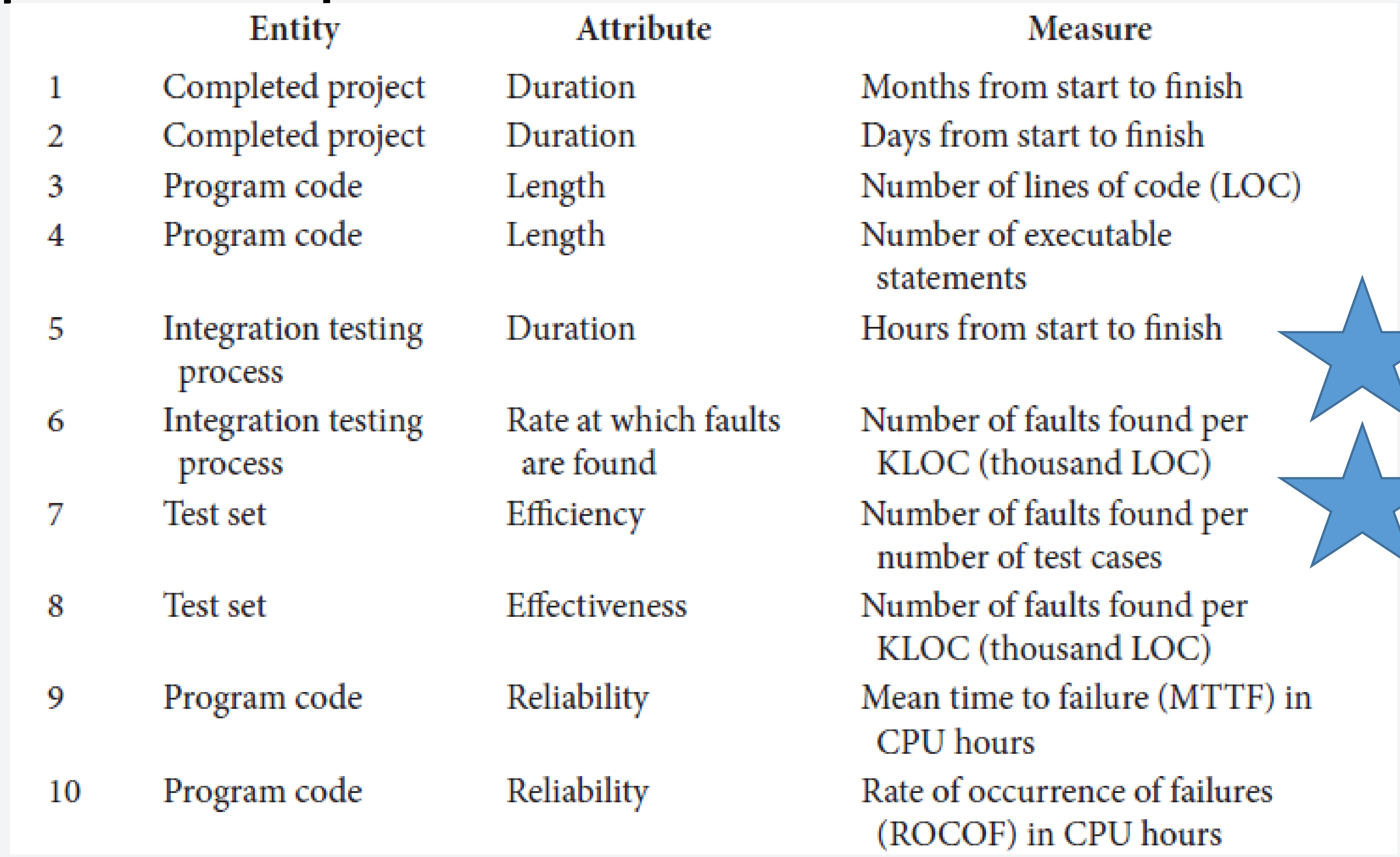
Scales of Measurement



Scale Type	Admissible Transformations (How Measures M and M' must be Related)	Examples
Nominal	1–1 mapping from M to M'	Labeling, classifying entities
Ordinal	Monotonic increasing function from M to M', that is, $M(x)$ $M(y)$ implies $M'(x)$ $M'(y)$	Preference, hardness, air quality, intelligence tests (raw scores)
Interval	$M' = aM + b$ $(a > 0)$	Relative time, temperature (Fahrenheit, Celsius), intelligence tests (standardized scores)
Ratio	$M' = aM$ $(a > 0)$	Time interval, length, temperature (Kelvin)
Absolute	$M' = M$	Counting entities

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Examples of Specific Measures Used in SWE



	Entity	Attribute	Measure
1	Completed project	Duration	Months from start to finish
2	Completed project	Duration	Days from start to finish
3	Program code	Length	Number of lines of code (LOC)
4	Program code	Length	Number of executable statements
5	Integration testing process	Duration	Hours from start to finish
6	Integration testing process	Rate at which faults are found	Number of faults found per KLOC (thousand LOC)
7	Test set	Efficiency	Number of faults found per number of test cases
8	Test set	Effectiveness	Number of faults found per KLOC (thousand LOC)
9	Program code	Reliability	Mean time to failure (MTTF) in CPU hours
10	Program code	Reliability	Rate of occurrence of failures (ROCOF) in CPU hours

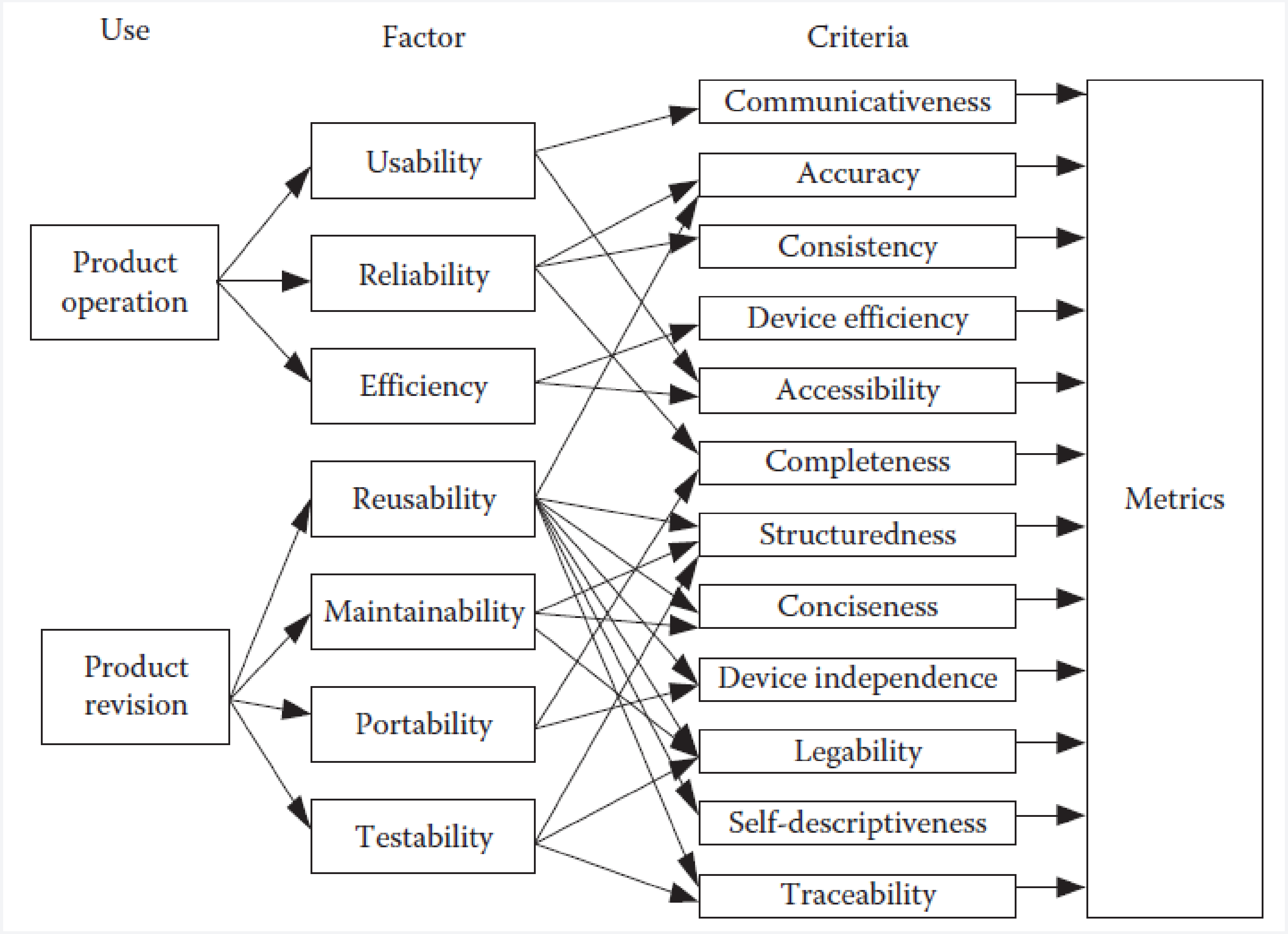
Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Software Quality (Models & Measures)

- Since software quality involves **many diverse factors**, SWEs have developed (usually tree-like) **models** of the **interaction between multiple quality factors**.
- The upper branches hold important **high-level quality factors** of software products that we would like to quantify
  - **Ex:** reliability, usability, etc.
- Each quality **factor** is composed of **lower-level criteria**
  - **Ex:** modularity, data commonality, etc.
- The **criteria** are **easier** to **understand** and **measure** than the **factors**; thus, **actual measures (metrics) are proposed for the criteria**.
- The tree describes the pertinent relationships between factors and their dependent criteria, so we can measure the factors in terms of the dependent criteria measures.
  - This notion of divide and conquer has been implemented as a standard approach to measuring software quality (IEEE 1061-2009).

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Software Quality (Models & Measures)



Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Software Quality

- A principal objective of SWE is to improve the quality of software products. But quality, like beauty, is very much in the eyes of the beholder.
- In the philosophical debate about the meaning of software quality, proposed definitions include:
  - Fitness for purpose
  - Conformance to specification
  - Degree of excellence
  - Timeliness

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Software Quality

- However, from a measurement perspective, we must be able to define quality in terms of specific software product attributes of interest to the user.
- That is, we want to know how to measure the extent to which these attributes are present in our software products.
  - external product attributes
  - internal product attributes

Software Quality - External Attribute

- We defined external product attributes as those that can be measured only with respect to **how** the **product** relates to its **environment**.
  - For example, if the product is software code, then its reliability (defined in terms of the probability of failure-free operation) is an external attribute; it is dependent on **both** the machine environment and the user.
  - Whenever we think of software code as our product and we investigate an external attribute that is dependent on the user, we inevitably are dealing with an attribute synonymous with a particular view of quality (i.e., a *quality attribute*).

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Software Quality - Internal Attribute

- We considered a range of internal attributes believed to affect quality in some way.
- Many practitioners and researchers measure and analyze internal attributes because they **may be predictors of external attributes**. There are two major advantages to doing so.
  - First, the internal attributes are often available for measurement early in the life cycle, whereas external attributes are measurable only when the product is complete (or nearly so).
  - Second, internal attributes are often easier to measure than external ones.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Modeling Software Quality

- Because quality is really a composite of many characteristics, the notion of quality is **usually captured in a model** that depicts the composite characteristics and their relationships.
- Many of the models blur the distinction between internal and external attributes, making it difficult for us to understand exactly what quality is.
  - Still, the models are useful in articulating what people think is important, and in identifying the commonalities of view.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

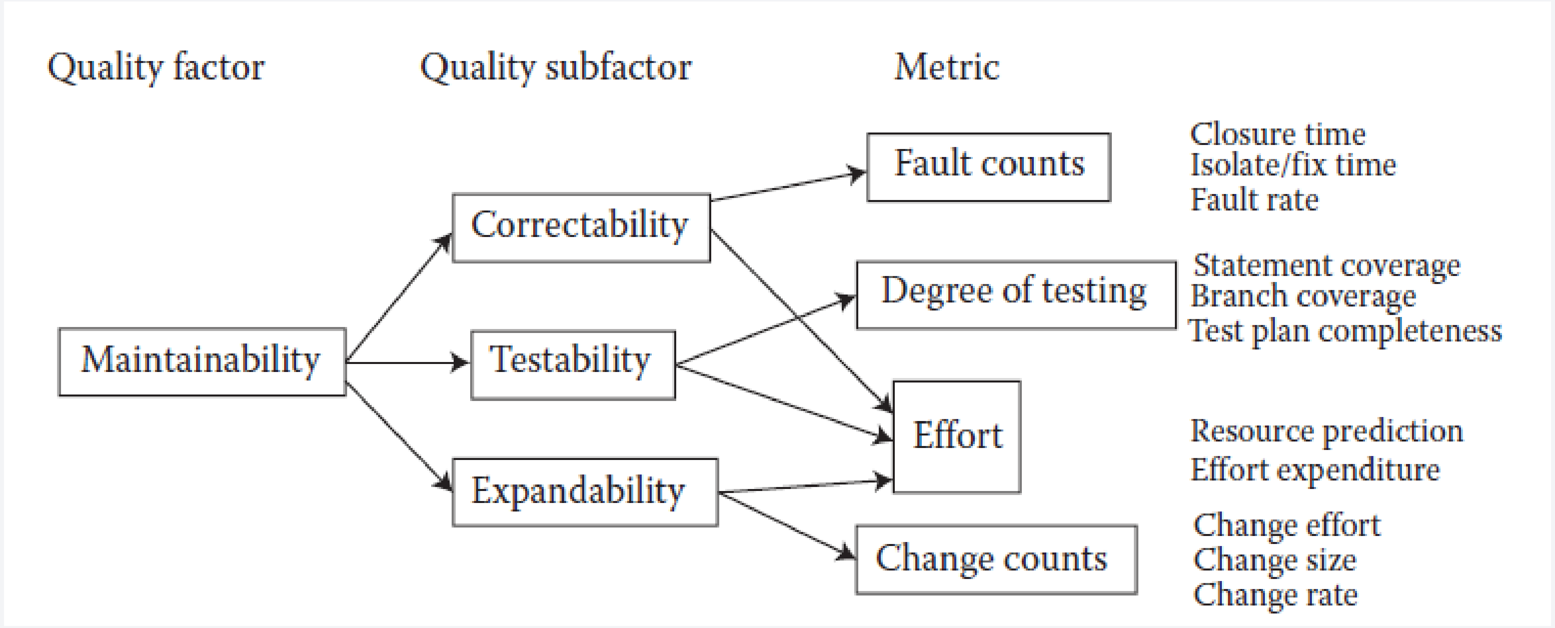
Software Quality - Early Models

- Two early models described quality using a decomposition approach
  - McCall et al. 1977
  - Boehm et al. 1978
- In models such as these, the model-builders focus on the final product (usually the executable code), and identify key attributes of quality from **the user's perspective**.
  - These key attributes, called *quality factors*, are normally high-level external attributes like *reliability*, *usability*, and *maintainability*.
  - But they may also include several attributes that arguably are internal, such as *testability* and *efficiency*.
- Each of the models assumes that the quality factors are still at too high a level to be meaningful or to be measurable directly.
  - Hence, they are further decomposed into lower-level attributes called *quality criteria* or *quality subfactors*.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

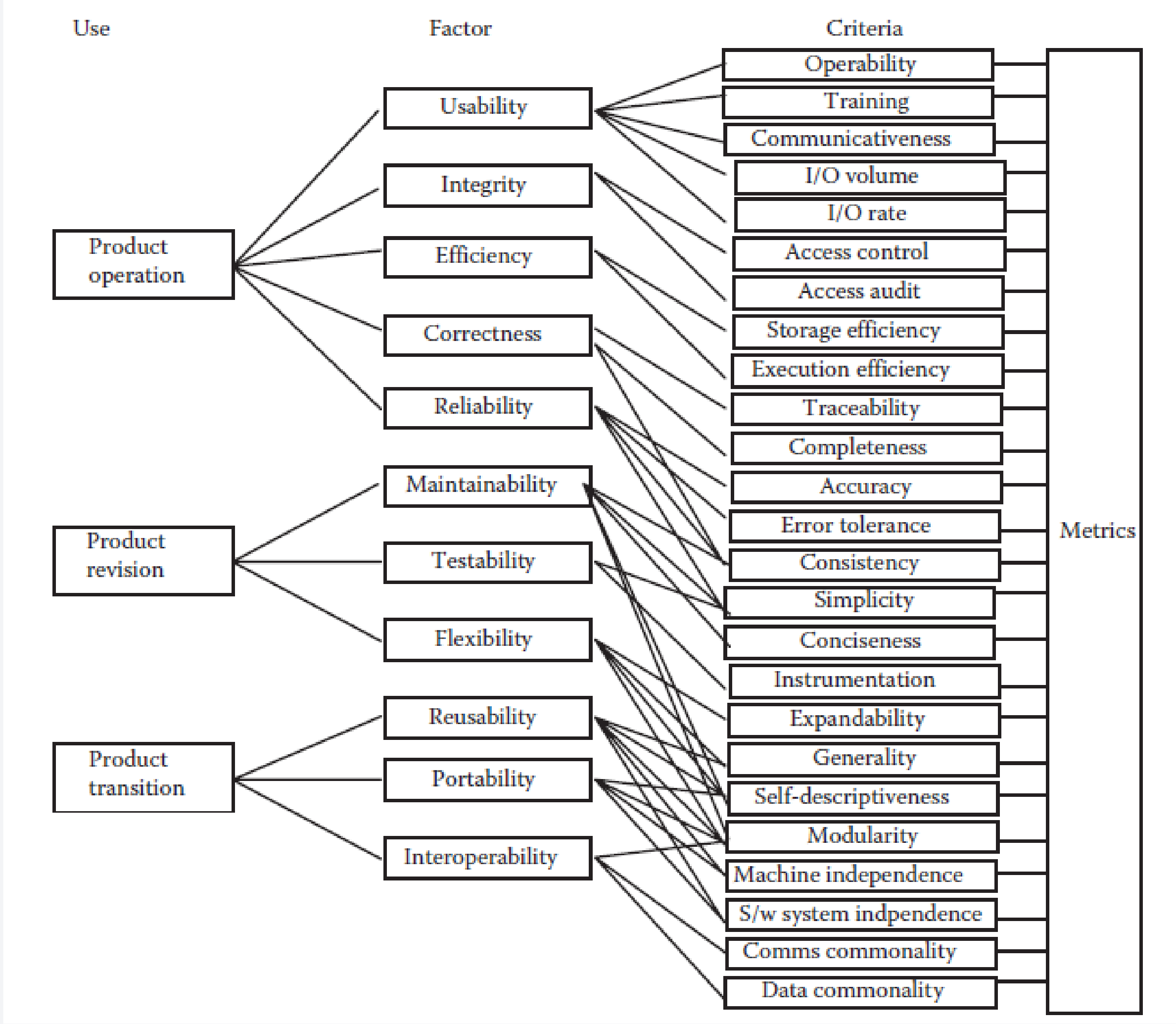
Software Quality

- A further level of decomposition is required, in which the quality criteria are associated with a set of low-level, directly measurable attributes (both product and process) called *quality metrics*.



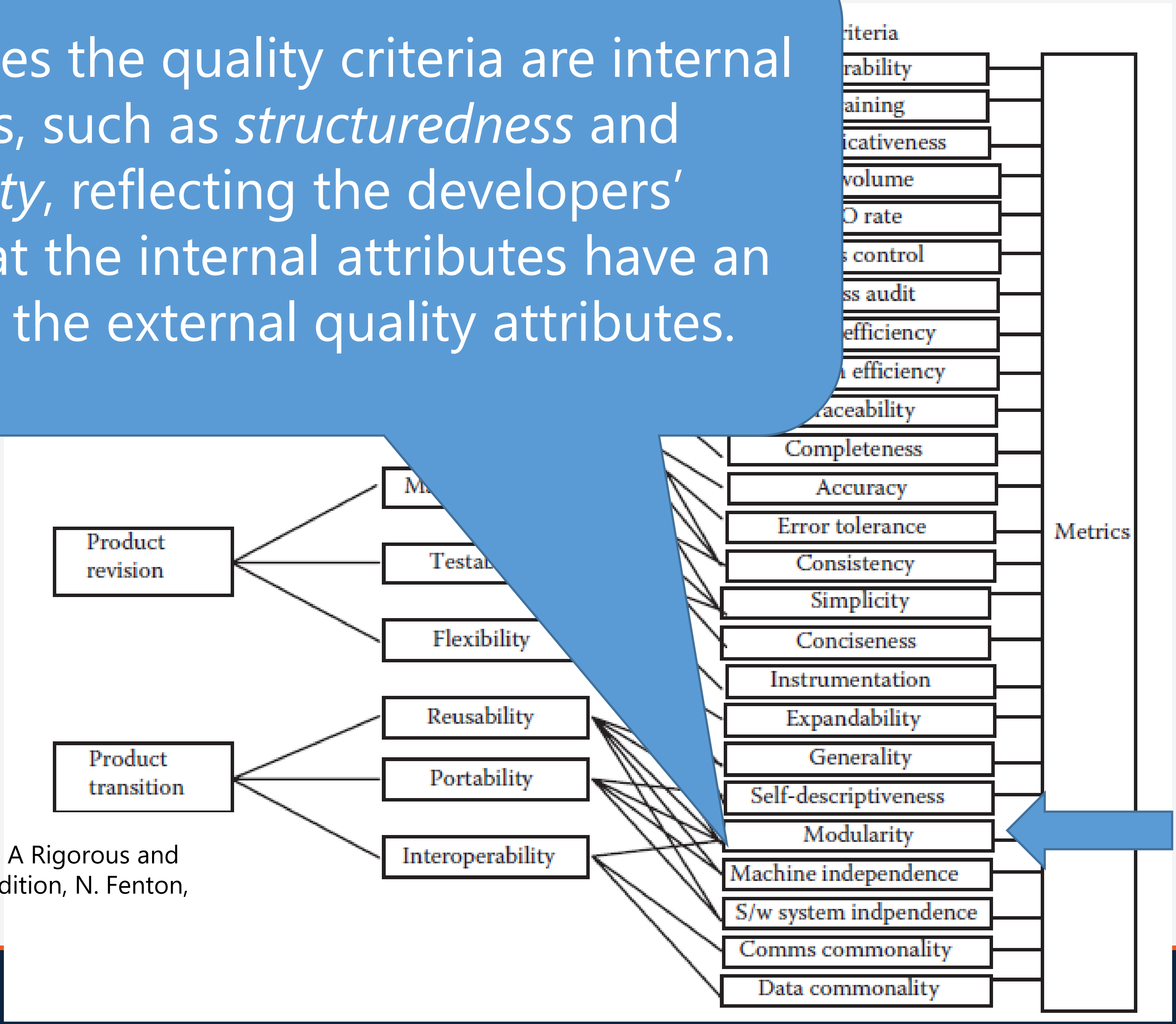
(This structure has been adapted from an IEEE standard for software quality metrics methodology, which uses the term *subfactor* rather than *criteria* (IEEE Standard 1061 2009).)

Software Quality - Early Models (McCall et al.)



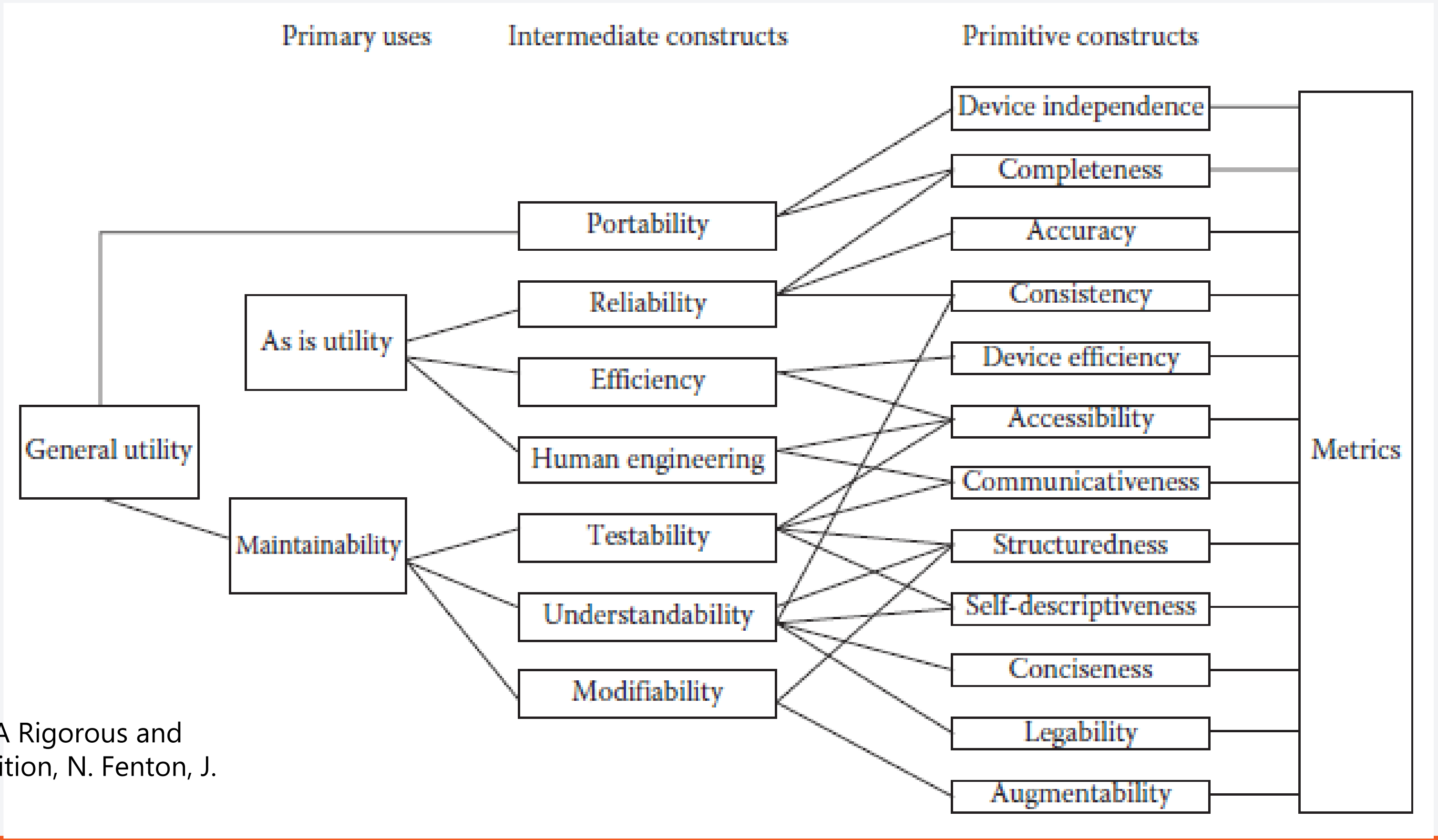
Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Sometimes the quality criteria are internal attributes, such as *structuredness* and *modularity*, reflecting the developers' belief that the internal attributes have an affect on the external quality attributes.



Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Software Quality - Early Models (Boehm et al.)



Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Software Quality Models

1. *The fixed model approach*
   - Boehm et. al.'s Model
   - McCall et. al.'s Model
   - etc.
2. *The "define your own quality model" approach*

Software Quality Models: Fixed Model

- *The fixed model approach*:
- We assume that ALL IMPORTANT quality factors needed to monitor a project are a subset of those in a **published** model.
- To control and measure each attribute, we accept the model's associated criteria and metrics and, most importantly, the proposed relationships among factors, criteria, and metrics.
    - Then, we use the data collected to determine the quality of the product.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Software Quality Models: Define your own

- *The "define your own quality model" approach*:
- We accept the general philosophy that quality is composed of many attributes, but we do not adopt a given model's characterization of quality.
- Instead, we meet with prospective users to reach a consensus on which quality attributes are important for a given product.
- Together, we decide on a decomposition (possibly guided by an existing model) in which we agree on specific measures for the lowest-level attributes (criteria) and specific relationships between them.
  - Then, we measure the quality attributes objectively to see if they meet specified, quantified targets.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Software Quality Models: Fixed Model

- The McCall model includes 41 metrics to measure the 23 quality criteria generated from the quality factors.
- Measuring any factor requires us first to consider a checklist of conditions that may apply to
  - the requirements (R)
  - the design (D)
  - the implementation (I)
- The condition is designated "yes" or "no," depending on whether or not it is met.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Software Quality Models: Fixed Model

Example

- To see how the metrics and checklists are used, consider measuring the **criterion** *completeness* for the **factor** *correctness*.
- The checklist for *completeness* is:
  1. Unambiguous references (input, function, output) [R,D,I].
  2. All data references defined, computed, or obtained from external source [R,D,I].
  3. All defined functions used [R,D,I].
  4. All referenced functions defined [R,D,I].
  5. All conditions and processing defined for each decision point R,D,I].
  6. All defined and referenced calling sequence parameters agree [D,I].
  7. All problem reports resolved [R,D,I].
  8. Design agrees with requirements [D].
  9. Code agrees with design [I].

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Software Quality Models: Fixed Model

Example

- Notice that there are
  - 6 conditions that apply to requirements[R]
  - 8 to design [D]
  - 8 to implementation[I]
- We can assign a 1 to a yes answer and 0 to a no, and we can compute the completeness metric in the following way to yield a measure that is a number between 0 and 1:

$$\frac{1}{3}\left(\frac{\text{Number of yes for R}}{6}+\frac{\text{Number of yes for D}}{8}+\frac{\text{Number of yes for I}}{8}\right)$$

Software Quality Models: Fixed Model

Example

- Since the model tells us that *correctness* depends on *completeness*, *traceability*, and *consistency*, we can calculate analogous measures for the latter two.
- Then, the measure for `correctness is the mean of their measures:`
- That is, *x*, *y*, and *z* are the metrics for *completeness*, *traceability*, and *consistency*, respectively.
  - $Correctness=(x+y+z)/3$

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Software Quality Models: Fixed Model

- The McCall model was originally developed for the **U.S. Air Force**, and its use was promoted within the **U.S. Department of Defense (DoD)** for evaluating software quality.
- But, many other standard (and competing) measures have been employed in the DoD; no single set of measures has been adopted as a department-wide standard.

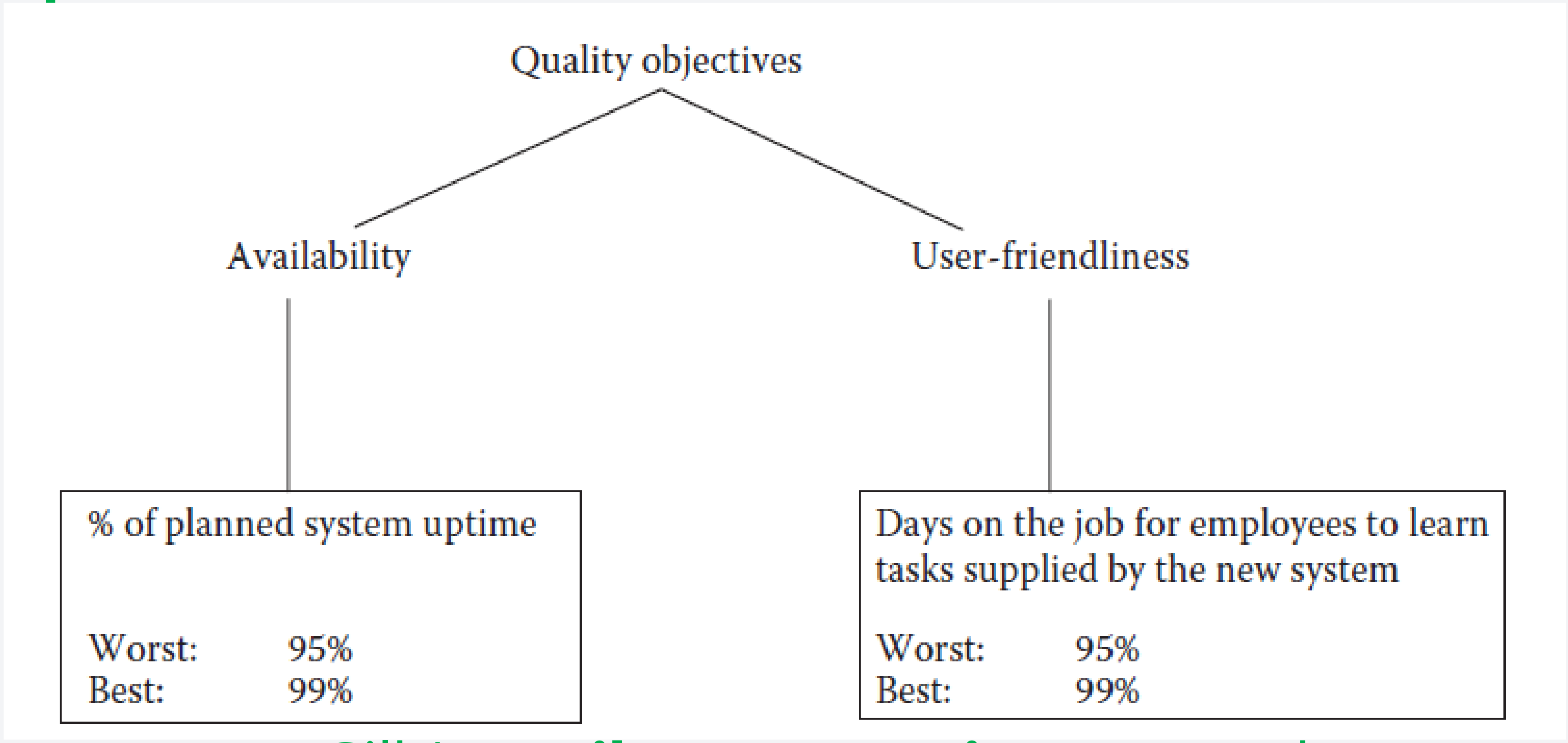
Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Define-Your-Own Models

- Gilb's method can be thought of as "design by measurable objectives"; it complements his philosophy of *evolutionary development*.
  - The software engineer delivers the product incrementally to the user, based on the importance of the different kinds of functionality being provided.
- To assign priorities to the functions, the user identifies key software attributes in the specification.
- These attributes are described in measurable terms, so the user can determine whether measurable objectives (in addition to the functional objectives) have been met.
  - This simple but powerful technique can be used to good effect on projects of all sizes, and can be applied within agile processes.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Define-Your-Own Models - Example


Quality objectives
Availability
User-friendliness
% of planned system uptime
Worst: 95%
Best: 99%
Days on the job for employees to learn tasks supplied by the new system
Worst: 95%
Best: 99%

Gilb's **attribute expansion** approach.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

ISO/IEC 9126-1 and ISO/IEC 25010 Standard Quality Models

- For many years, the user community sought a single model for depicting and expressing quality.
- The advantage of a universal model is clear: it makes it easier to compare one product with another.
  - In 1992, a derivation of the McCall model was proposed as the basis for an international standard for software quality measurement and adopted.
  - It evolved into *ISO/IEC Standard 9126-1* (ISO/IEC 9126-1 2003).
  - In 2011, ISO/IEC 25010 "Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)" replaced ISO/IEC 9126-1 (ISO/IEC 25010 2011).
- In the ISO/IEC 25010 standard, software quality is defined to be the following: "*The degree to which a software product satisfies stated and implied needs when used under specified conditions.*"

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

ISO/IEC 9126-1 and ISO/IEC 25010 Standard Quality Models

- The quality is decomposed into 8 characteristics:
  1. **Functional suitability**
  2. **Performance efficiency**
  3. **Compatibility**
  4. **Usability**
  5. **Reliability**
  6. **Security**
  7. **Maintainability**
  8. **Portability**

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

ISO/IEC 9126-1 and ISO/IEC 25010 Standard Quality Models

- The standard claims that these 8 are comprehensive; that is, any component of software quality can be described in terms of some aspect of one or more of the 8 characteristics.
- In turn, each of the 8 is defined in terms of other attributes on a relevant aspect of software, and each can be refined through multiple levels of subcharacteristics.
  - ISO/IEC 25010:2011 contains definitions of numerous subcharacteristics and metrics; many of these were derived from those in 3 technical reports issued as part of ISO/IEC 9126 (ISO/IEC 9126-2 2003, ISO/IEC 9126-3 2003, ISO/IEC 9126-4 2004).

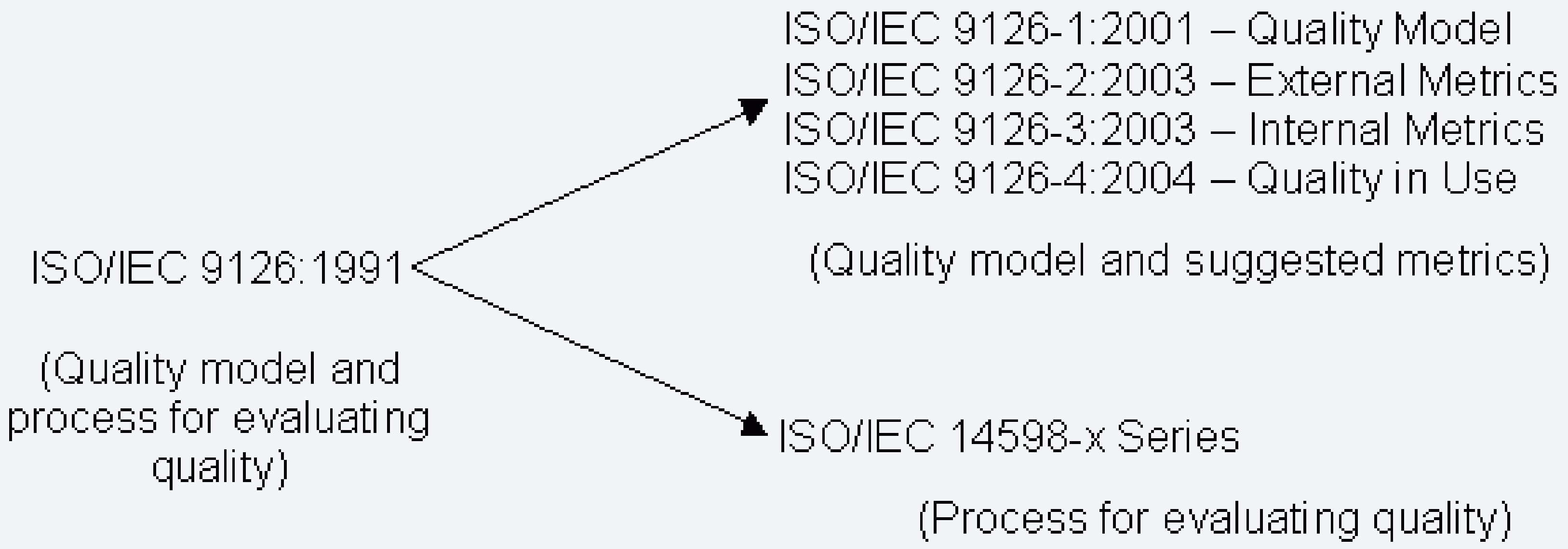
Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

ISO/IEC 9126-1 and ISO/IEC 25010 Standard Quality Models

- Another standards document describes an evaluation process for **evaluating software product quality** (ISO/IEC 25040:2011 2011). The process is described in terms of **5 major stages**:
  1. Establish evaluation requirements by determining the evaluation objectives, quality requirements, and the extent of the evaluation.
  2. Specify the evaluation by selecting measures, criteria for measurement, and evaluation.
  3. Design the evaluation activities.
  4. Conduct the evaluation.
  5. Conclude the evaluation by analyzing results, preparing reports, providing feedback, and storing results appropriately. The standard also describes the roles of various stakeholders involved in software quality evaluations.
- The process can be used for the evaluation of the quality of pre-developed software, COTS software or custom software and can be used during or after the development process.

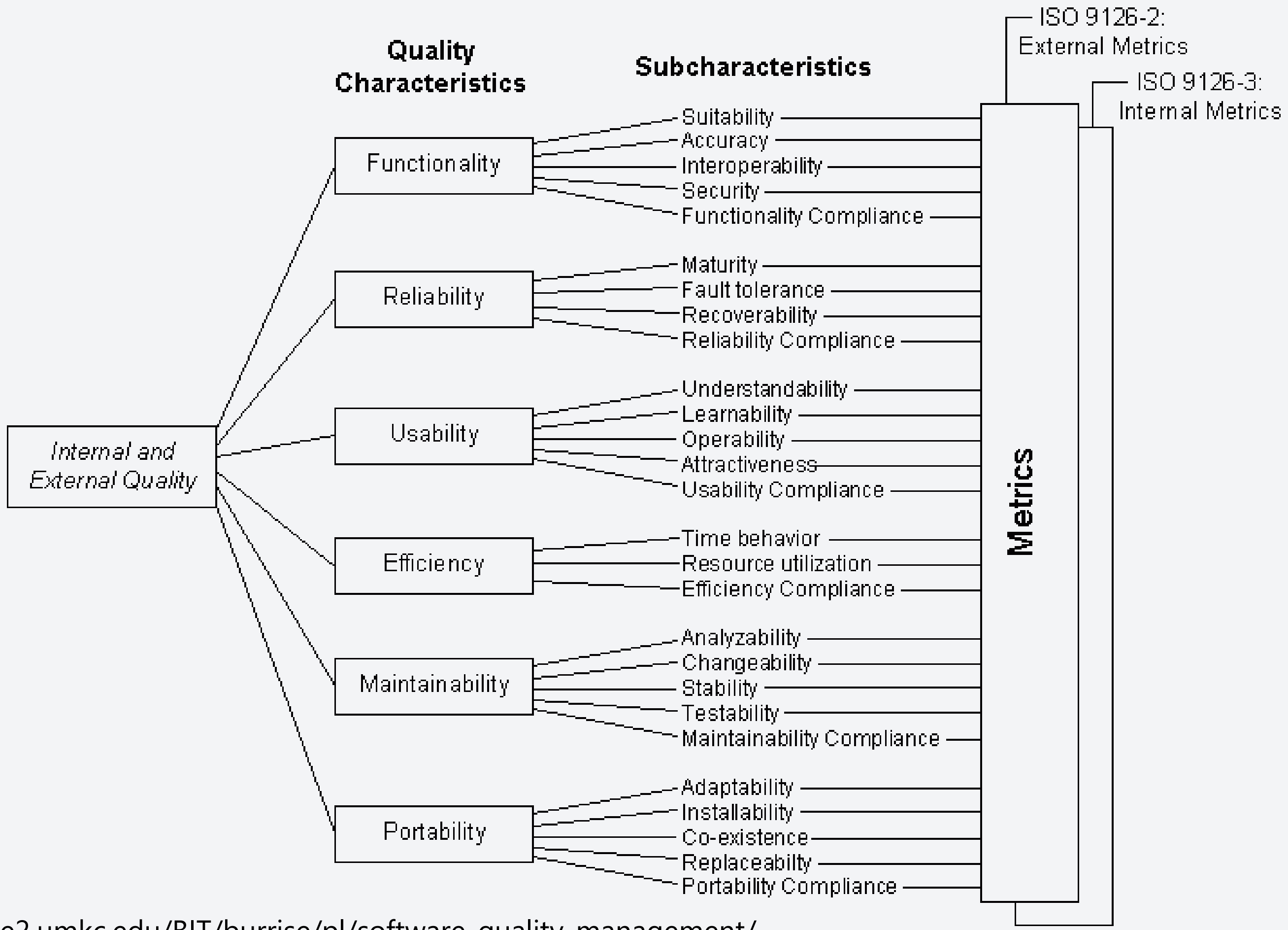
Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

The ISO/IEC 9126 Product Quality Standard


ISO/IEC 9126:1991
(Quality model and process for evaluating quality)
ISO/IEC 9126-1:2001 – Quality Model
ISO/IEC 9126-2:2003 – External Metrics
ISO/IEC 9126-3:2003 – Internal Metrics
ISO/IEC 9126-4:2004 – Quality in Use
(Quality model and suggested metrics)
ISO/IEC 14598-x Series
(Process for evaluating quality)

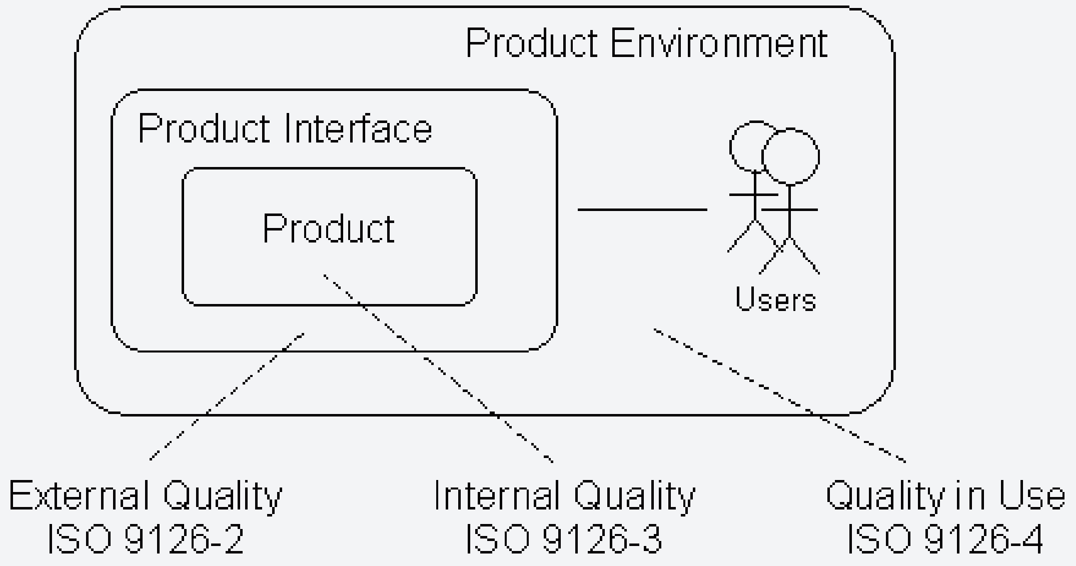
Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

ISO 9126 Quality Model



Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

ISO 9126 Quality Model



Source: http://sce2.umkc.edu/BIT/burrise/pl/software_quality_management/

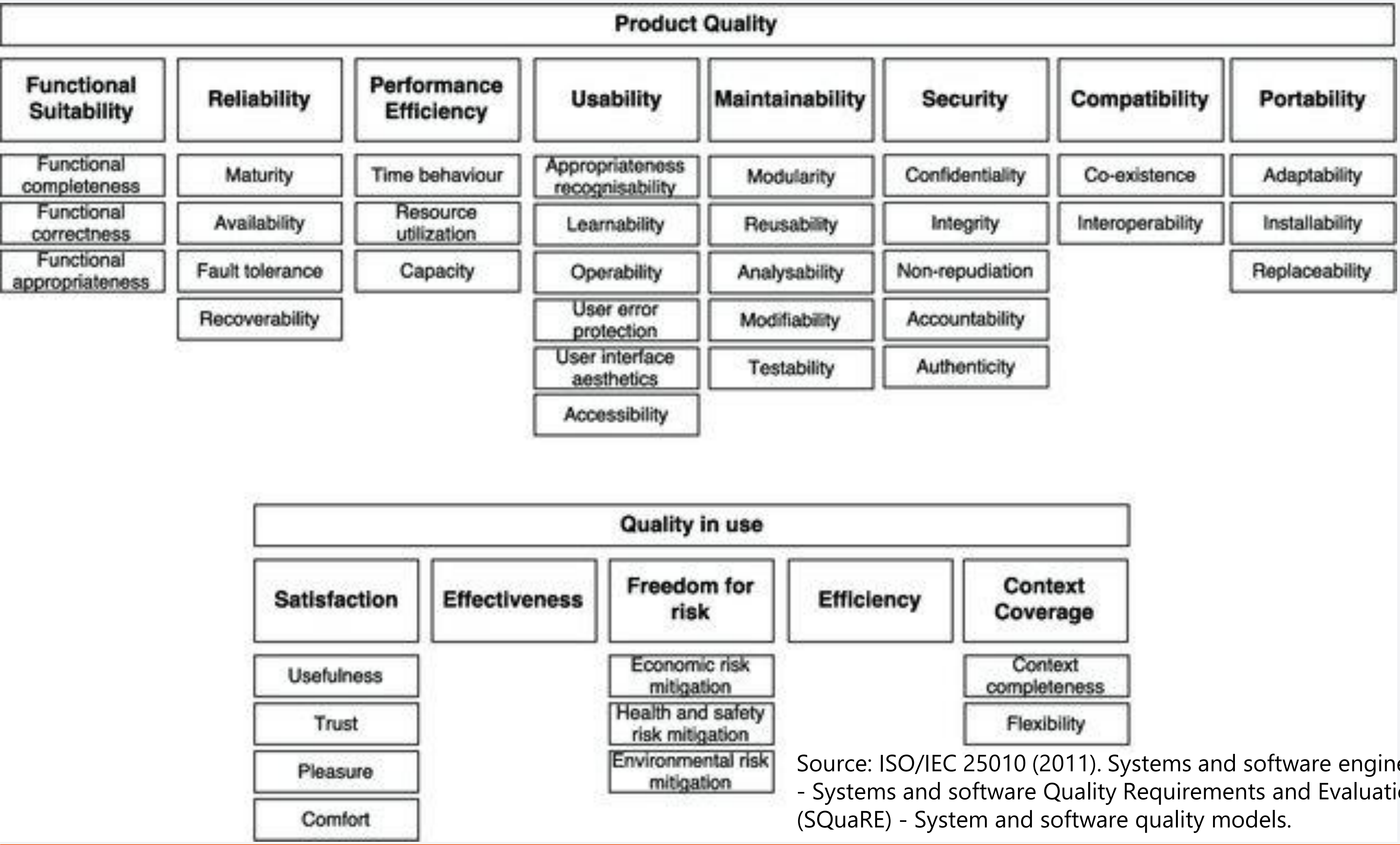
ISO 25010 Quality Model

- ISO 25010, which was published in 2011, superseded ISO 9126 published in 2001.
- **The main difference between the two lies in how they categorize and define non-functional software quality requirements.**
- ISO 25010 added two additional product quality characteristics to the six specified in ISO 9126 — adding security and compatibility.

ISO 25010 Quality Model

- ISO25010 describes two quality models:
  - **The quality in use** model composed of five characteristics (some of which are further sub-divided into sub-characteristics) that relate to the outcome of interaction when a product is used in a particular context of use.
  - **A product quality** model composed of eight characteristics (which are further sub-divided into sub-characteristics) that relate to static properties of software and dynamic properties of the computer system.
- The characteristics and sub-characteristics provide consistent terminology for specifying, measuring and evaluating system and software product quality. They also provide a set of quality characteristics against which stated quality requirements can be compared for completeness.

ISO 25010 Quality Model



Source: ISO/IEC 25010 (2011). Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models.

ISO 25010 Quality Model

- **Functional Suitability**
- Functional Suitability refers to how well a product or system is able to provide functions that meet the stated and implied needs.
- **Functional Completeness:** Refers to the set of functions that covers all of the specified tasks and user objectives.
- **Functional Correctness:** Refers to how well a product or system provides the correct results with the needed degree of precision.
- **Functional Appropriateness:** Refers to how well functions are able to accomplish specified tasks and objectives.

ISO 25010 Quality Model

- **Reliability**
- Reliability refers to how well a system, product, or component performs specified functions under specified conditions.
- **Maturity:** Refers to how well a system, product, or component is able to meet your needs for reliability.
- **Availability:** Refers to whether a system, product, or component is operational and accessible.
- **Fault Tolerance:** Refers to how well a system, product, or component operates despite hardware and/or software faults.
- **Recoverability:** Refers to how well a product or system can recover data in the event of an interruption or failure.

ISO 25010 Quality Model

- **Performance Efficiency**
- Performance Efficiency refers to the performance related to the amount of resources used.
- **Time Behavior:** Refers to the response and processing times, and throughput rates of a product or system while it's performing its functions.
- **Resource Utilization:** Refers to the amounts and types of resources used by a product or system while performing its functions.
- **Capacity:** Refers to the maximum limits of a product or system parameter.

ISO 25010 Quality Model

- **Usability**
- Usability refers to how well a product or system can be used to achieve specified goals effectively, efficiently, and satisfactorily.
- **Appropriateness Recognizability:** Refers to how well you can recognize whether a product or system is appropriate for your needs.
- **Learnability:** Refers to how easy it is to learn how to use a product or system.
- **Operability:** Refers to whether a product or system has attributes that make it easy to operate and control.
- **User Error Protection:** Refers to how well a system protects users against making errors.
- **User Interface Aesthetics:** Refers to whether a user interface is pleasing.
- **Accessibility:** Refers to how well a product or system can be used with the widest range of characteristics and capabilities.

ISO 25010 Quality Model

- **Security**
- Security refers to how well a product or system protects information and data from security vulnerabilities.
- **Confidentiality:** Refers to how well a product or system is able to ensure that data is only accessible to those who have authorized access.
- **Integrity:** Refers to how well a system, product, or component is able to prevent unauthorized access and modification to computer programs and/or data.
- **Non-repudiation:** Refers to how well actions or events can be proven to have taken place.
- **Accountability:** Refers to the actions of an unauthorized user can be traced back to them.
- **Authenticity:** Refers to how well the identity of a subject or resource can be proved.

ISO 25010 Quality Model

- **Compatibility**
- Compatibility refers to how well a product, system, or component can exchange information as well as perform its required functions while sharing the same hardware or software environment.
- **Co-existence:** Refers to how well a product can perform its required functions efficiently while sharing a common environment and resources with products, without negatively impacting any other product.
- **Interoperability:** Refers to how well two or more systems, products, or components are able to exchange information and use that information.

ISO 25010 Quality Model

- **Maintainability**
- Maintainability refers to how well a product or system can be modified to improve, correct, or adapt to changes in the environment as well as requirements.
- **Modularity:** Refers to whether the components of a system or program can be changed with minimal impact on the other components.
- **Reusability:** Refers to how well an asset can be used in more than one system.
- **Analysability:** Refers to the effectiveness of an impact assessment on intended changes. In addition, it also refers to the diagnosis of deficiencies or causes of failures, or to identify parts to be modified.
- **Modifiability:** Refers to how well a product or system can be modified without introducing defects or degrading existing product quality.
- **Testability:** Refers to how effective the test criteria is for a system, product, or component. In addition, it also refers to the tests that can be performed to determine whether the test criteria has been met.

ISO 25010 Quality Model

- **Portability**
- Portability refers to how well a system, product, or component can be transferred from one environment to another.
- **Adaptability:** Refers to how well a product or system can be adapted for different or evolving hardware, software, or other usage environments.
- **Installability:** Refers to how successfully a product or system can be installed and/or uninstalled.
- **Replaceability:** Refers to how well a product can replace another comparable product.

Direct and Derived Measurement

- Many of the examples we have used employ direct mappings from attribute to number, and we use the number to answer questions or assess situations.
- But when there are complex relationships among attributes, or when an attribute must be measured by combining several of its aspects, then we need a model of how to combine the related measures.
- It is for this reason that we distinguish direct measurement from derived measurement.

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Direct and Derived Measurement

- Direct measurement of an attribute of an entity involves no other attribute or entity.
    - For example, length of a physical object can be measured without reference to any other object or attribute.
- On the other hand, measures of the density of a physical object can be derived in terms of *mass* and *volume*; we then use a model to show us that the relationship between the three is
    - *Density* = *Mass/Volume*

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Direct and Derived Measurement

- The direct measures are **commonly used in SWE**:
  - Size of source code (measured by LOC)
  - Schedule of the testing process (measured by elapsed time in hours)
  - # of defects discovered (measured by counting defects)
  - Time a programmer spends on a project (measured by months worked)

Examples of Common Derived Measures Used in SWE

Programmer productivity	LOC produced/person-months
Module defect density	# of defects/module size
Defect detection efficiency	# of defects detected/total # of defects
Requirements stability	# of initial requirements/total # of requirements
Test coverage	# of test requirements covered/total # of test requirements
System spoilage	Effort spent fixing faults/total project effort

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Rescaling - Example

- A derived measure of testing efficiency T=D/E, where
  - D is the # of defects discovered
  - E is effort in person-months
- Here D is an absolute scale measure, while E is on the ratio scale.
- Since absolute is stronger than ratio scale, it follows that **T is a ratio scale measure**.
- Consequently, the acceptable rescalings of T arise from rescalings of E into other measures of effort (person-days, person-years, etc.)

Source: Software Metrics, A Rigorous and Practical Approach, 3rd Edition, N. Fenton, J. Bieman, 2015.

Example – Effort

- Consider another effort measure $E=2.7v+121w+26x+12y+22z-497$ cited by DeMillo and Lipton (1981).
  - *E* is supposed to represent person-months
  - *v* is the # of program instructions
  - *w* is a subjective complexity rating
  - *x* is the # of internal documents generated on the project
  - *y* is the # of external documents
  - *z* is the size of the program in words
- DeMillo and Lipton correctly point out that E should be on a ratio scale, but it cannot be ratio in this equation because *w*, an ordinal measure, restricts *E* to being ordinal.
- Thus, the equation is meaningless. However, *E* could still be an ordinal scale measure of effort if we drop the pre-condition that *E* expresses effort in person-months.