University of Zurich[UZH]

Department of Informatics

Martin Glinz

Software Quality

Chapter 6

Software Product Quality

6.1 External vs. Internal Product Quality

6.2 Internal Software Product Quality

6.3 External Software Product Quality

6.4 Dependability

External vs. internal software product quality

- External quality is the quality of a (software) product as perceived by its stakeholders
- Internal quality is the quality of the software, particularly of the source code that eventually delivers external quality
- Note that the standard ISO/IEC 25010:2011 uses a different notion of external and internal quality (see below)

6.1 External vs. Internal Product Quality

6.2 Internal Software Product Quality

6.3 External Software Product Quality

6.4 Dependability

About internal software product quality

- Measuring
  - Measuring internal quality characteristics
  - Predicting external quality from internal quality data
- Mining
  - Mining internal quality characteristics
  - Predicting quality-relevant phenomena from mined data

Measuring internal software product quality

- Classic measurment of static source code properties
  - Size
  - Complexity
  - Cohesion and couping
  - Depth of inheritance trees
  - Method fan-in/fan-out
  - ...
- In combination with process measurements:
  - Error and defect rates
  - Defect density per module
  - ...

Measurement-based analysis

- Simple measurement
  - For example, measure the size of of methods (in terms of LoC) and identify outliers (very short and too long methods)
- Static/Dynamic program analysis

  Can, for example, identify
  - non-initialized variables
  - dead code
  - data flow anomalies
- Architectural analysis
  - For example, identify cycles in in the method call hierarchy

Predicting external quality

- Using internal quality measurements for predicting external quality characteristics, for example
  - Predicting system reliability by measuring error occurrence rates during statistical (random) testing or by measuring defect density
  - Predicting portability by measuring source code characteristics such as percentage of platform-dependent code
- Proving internal quality properties, in particular safety and liveness properties for predicting safety and security characteristics of a system
- Inspecting internal quality properties for predicting external quality characteristics such as maintainability

Mining internal product quality

Basic idea:

From big repositories of data about software, ...

using suitable procedures, ...

elicit information, which...

- tells us about the current internal quality of the software
- allows predictions about quality relevant phenomena

Data repositories

- Version history of software artifacts (particularly source code)
- Change history
- Problem report database
- Test suites and test summaries
- Review reports
- Process measurement databases (effort, duration, productivity, error cost,...)
- Developers' e-mail and chat threads
- ...

What and how to mine

- Identify certain patterns and anomalies
  - For example, an analysis of test summaries reveals a pattern of erroneous usage of some library
- Learning certain patterns (using machine learning algorithms)
  - For example, we might be able to learn from the change history of a system that in most cases, changes in module X imply changes in modules X1, A, and F

Predicting quality-relevant phenomena

- Example: With machine learning technology, we might find a statistically significant correlation between some measurable properties of a module in the system's version archive and the error-proneness of a module
  → From such data, we can derive a predictor for error-proneness

→ Another example: if we have learned change correlations between modules (see previous slide) we can derive a predictor for modules that also need to be changed if some given module is modified.

- Significant correlation under stable conditions is sufficient for constructing predictors – no causality analysis needed

Reading assignment

Read the following papers about mining quality-relevant data from software repositories:

- Zimmermann et al. (2005): Mining Version Histories to Guide Software Changes
- Nagappan, Ball, Zeller (2006): Mining Metrics to Predict Component Failures
- Bird et al. (2009): Does Distributed Development Affect Software Quality?: An Empirical Case Study of Windows Vista

6.1 External vs. Internal Product Quality

6.2 Internal Software Product Quality

6.3 External Software Product Quality

6.4 Dependability

Classifying external product quality

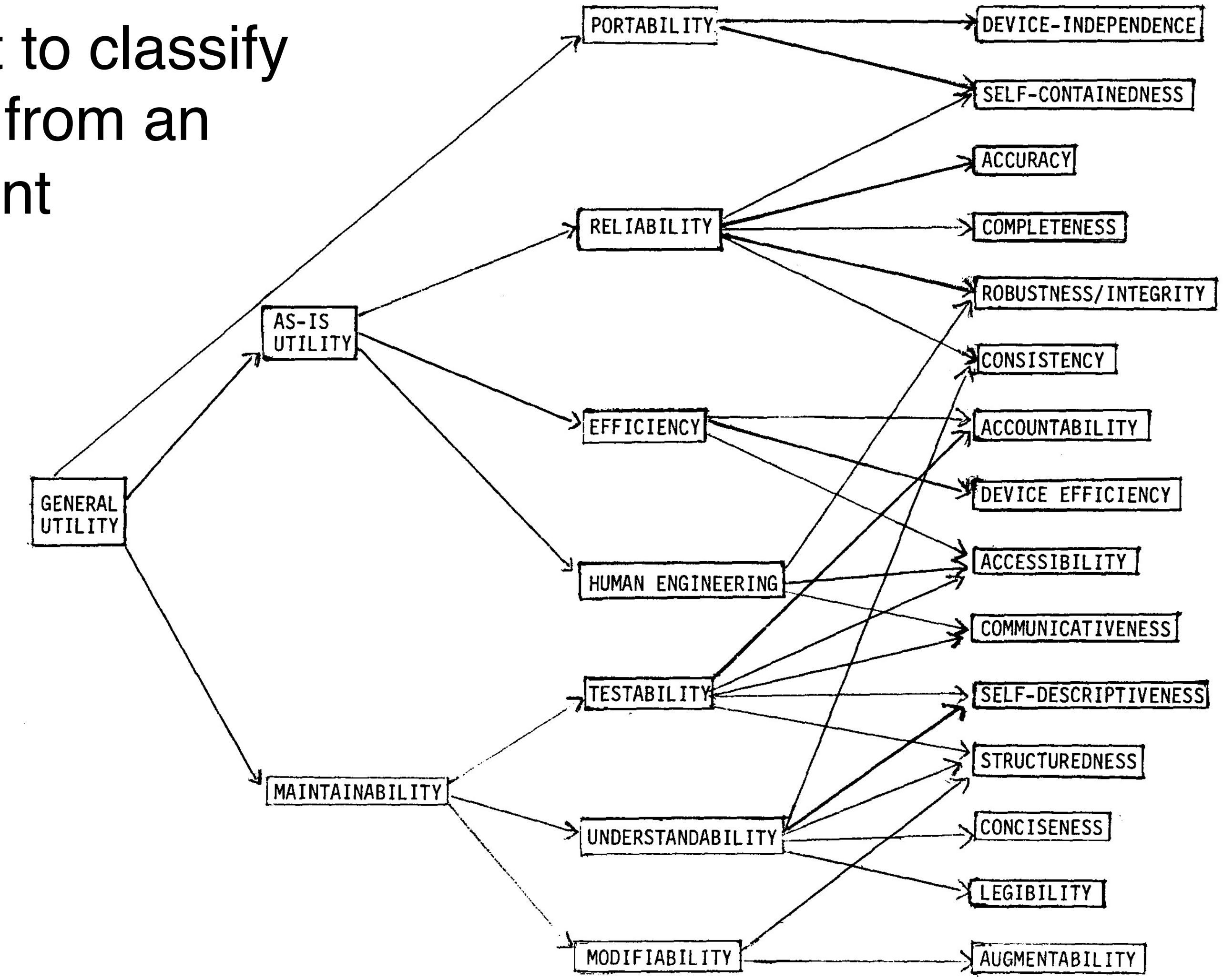
As there are many facets of external product quality, numerous approaches for creating taxonomies and frameworks have been made, for example

- Boehm et al. (1976)
- McCall and Matsumoto (1980)
- ISO/IEC 9126 (first published in 1991, revised in 2001, superseded by ISO/IEC 25010 in 2011)
- Quamoco (2011)

Boehm’s quality model

[Boehm, Brown and Lipov 1976]

The first attempt to classify software quality from an external viewpoint



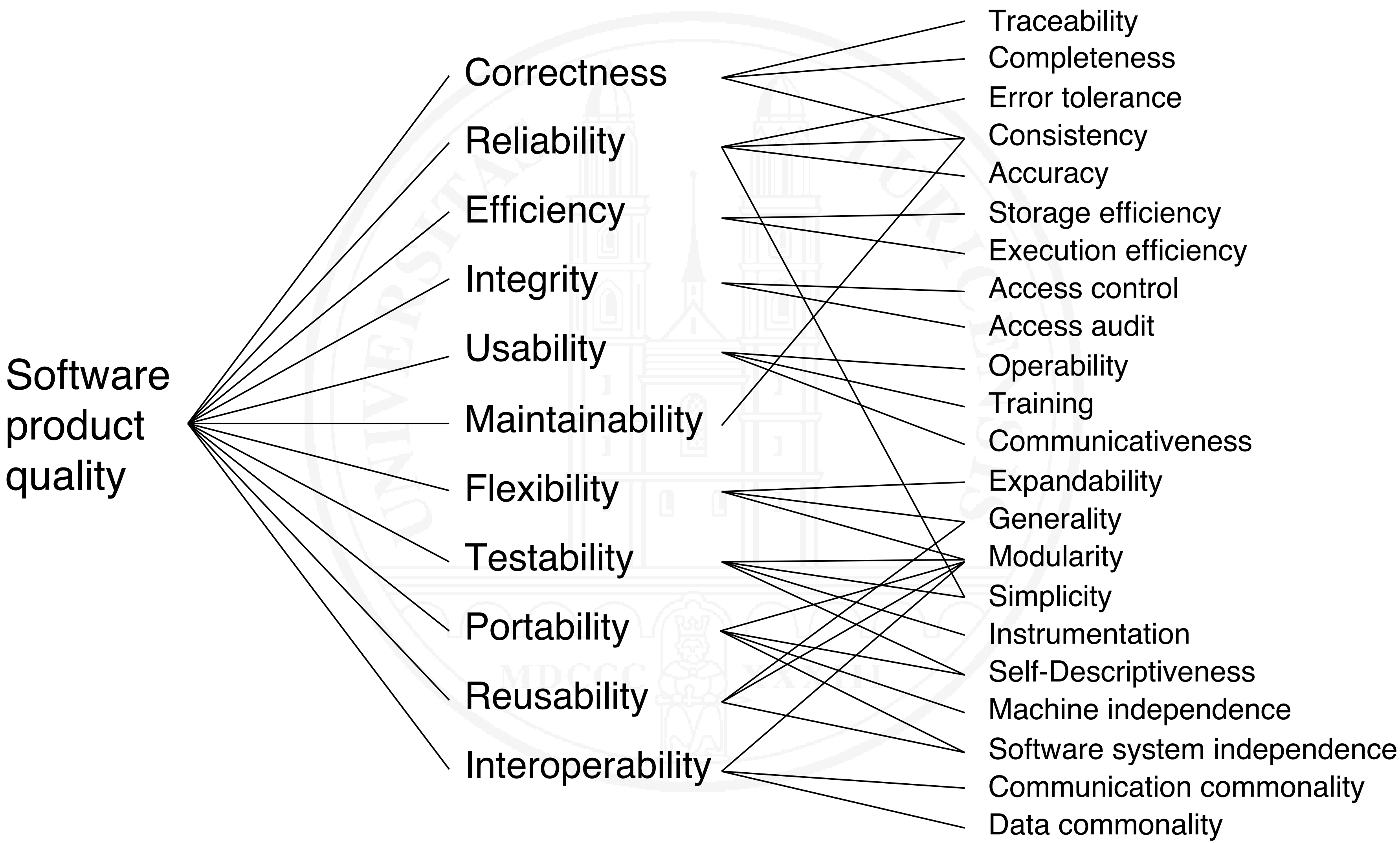
The quality model by McCall and Matsumoto

[McCall and Matsumoto 1980]

Three-level model:

- Factors, representing a management-oriented view of software quality
- Criteria for every factor, representing software-oriented attributes that provide software quality
- Metrics, i.e., quantitative measures of those attributes

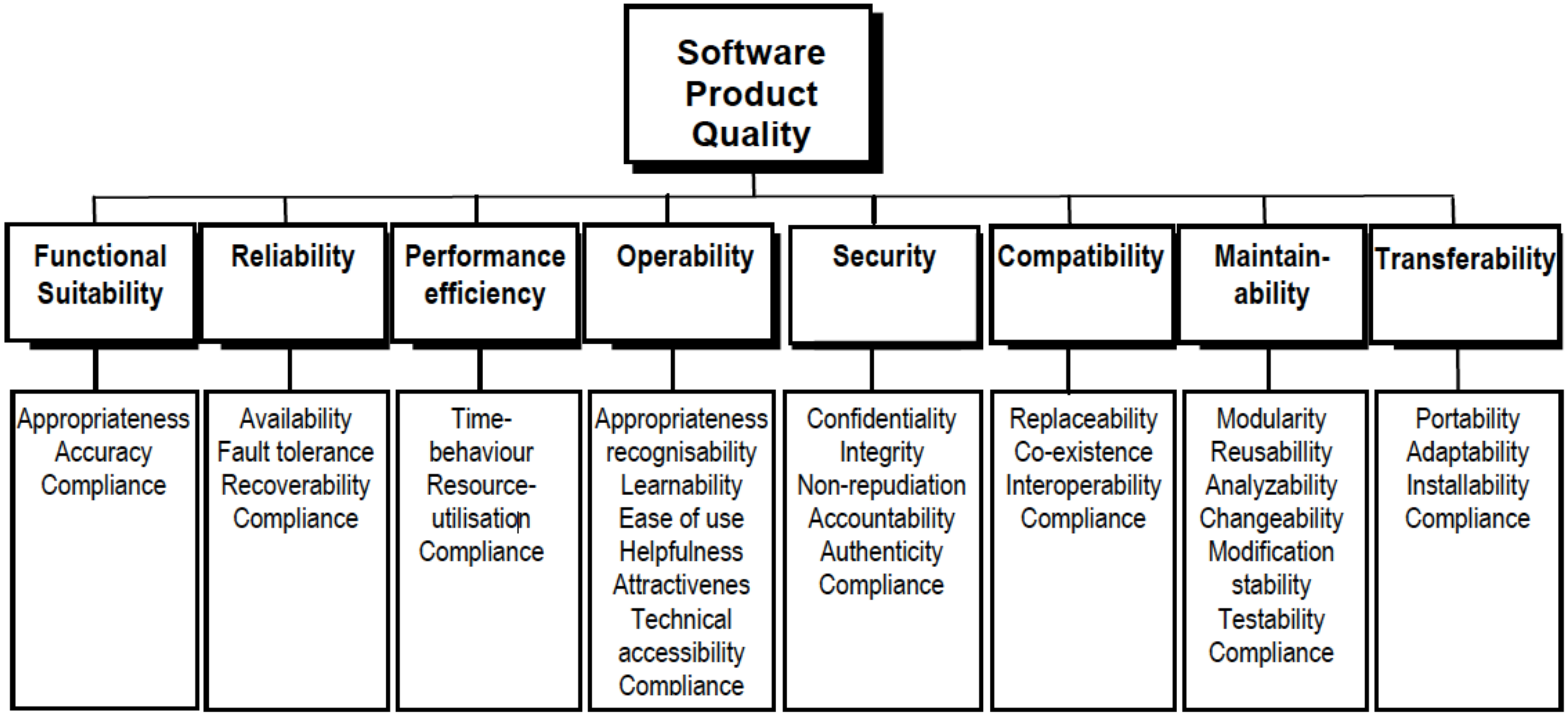
Mc Call and Matsumoto: Factors and criteria



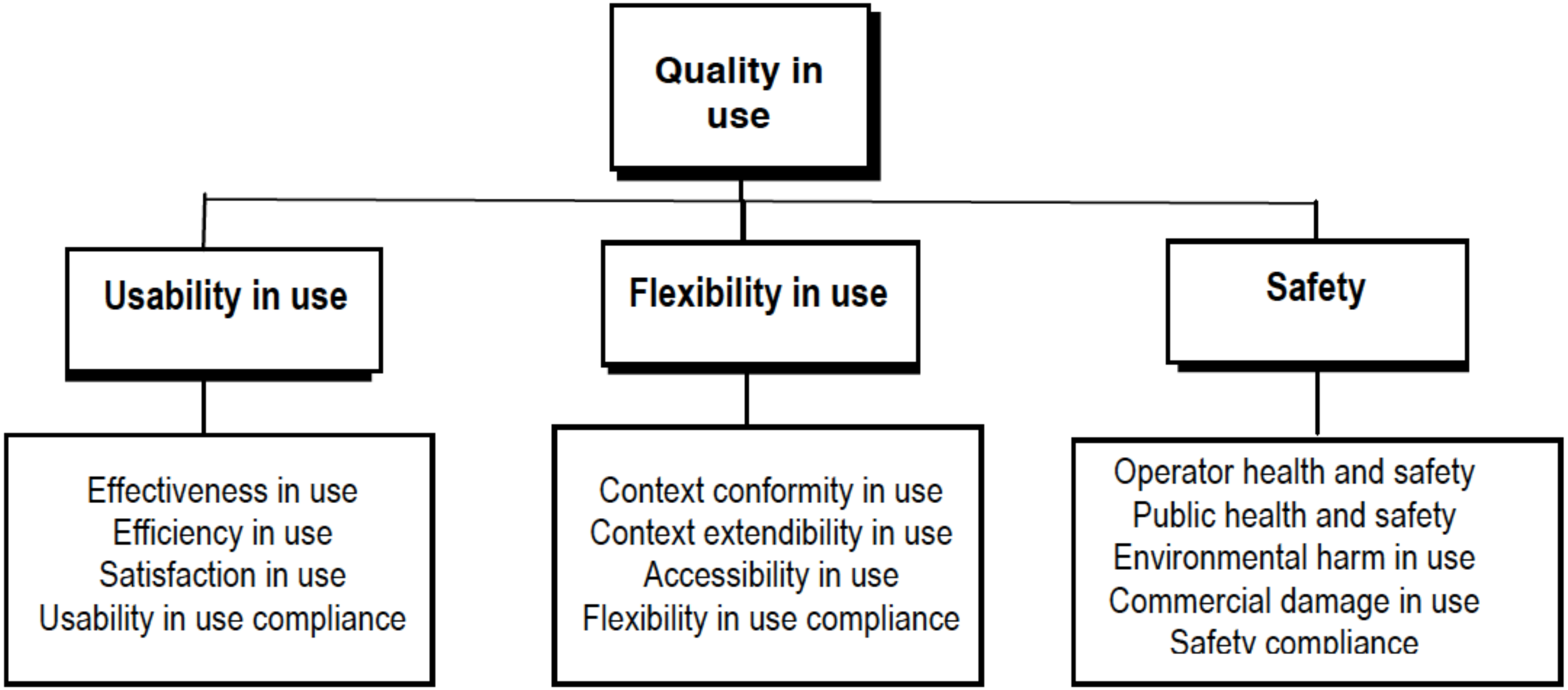
The ISO/IEC 25010 quality model

- Differentiates between
  - Product quality model
  - Quality in use model
- External and internal quality have a specific meaning in the ISO/IEC 25010 framework:
  - External quality assesses the characteristics of the product quality model by black-box measurement
  - Internal quality assesses the characteristics of the product quality model by glass-box measurement, i.e. measuring system properties based on knowledge about the internal structure of the software

The ISO/IEC 25010 product quality model



The ISO/IEC 25010 quality in use model



Problems with ISO/IEC 25010

- Basing the distinction of external and internal quality on the type of measurements is counter-intuitive: the very same characteristic can denote external quality or internal quality or both, depending on the metrics used to measure it
- No convincing rationale for classifying characteristics as product quality or quality in use characteristics, for example:
  - Security is a product quality characteristic, while safety is a quality in use characteristic
  - Learnability and Ease of use are product quality sub-characteristics, although they pertain to using the product

Quality models are in the eye of the beholder

- Availability is missing from the McCall-Matsumoto model
- Storage efficiency may be highly relevant in some context and irrelevant in another context
- Assessing performance might include transmission rate behavior, while this is not included in the ISO/IEC 25010 model

Factors of a modern product quality model

Usage-oriented factors

- Functionality
- Usability
- Efficiency
- Reliability
- Security
- Safety
- Dependability

Product-oriented factors

- Maintainability
- Portability
- Compliance

The factors explained

[partially adapted from ISO/IEC 9126]

Functionality – The capability of a software system to provide functions which meet stated and implied needs when the software is used under specified conditions

Usability – The capability of a software system to be understood, learned, used and attractive to the user, when used under specified conditions

Efficiency – The capability of a software system to provide appropriate performance, relative to the amount of resources used, under stated conditions

Reliability – The capability of a software system to maintain a specified level of performance when used under specified conditions

The factors explained – 2

Security – The capability of a software system to protect information so that unauthorized agents cannot access them and authorized agents are not denied access to them

Safety – The capability of a software system to achieve acceptable levels of risk of harm to people or any other entities in a specified context of use

Dependability – The trustworthiness of a software system such that reliance can justifiably be placed on the service it delivers

The factors explained – 3

- Maintainability – The capability of a software system to be changed and to evolve by correcting, adapting and improving the software
- Portability – The capability of a software system to be transferred from one environment to another or be adapted to some changed or new environment
- Compliance – The capability of a software system to comply to given standards, procedures, legal regulations or other constraints

Assessing external quality

- Measurement
  - No direct measures available in most cases
  - Typically predicting quality from measuring measurable quality indicators
- Testing
  - For example, for assessing functionality, efficiency or reliability
- Inspection
  - Manual assessment by a group of experts
- Monitoring and feedback
  - Monitoring relevant indicators during system operations
  - Encourage and systematically evaluate user feedback

6.4 Dependability

Definition

Dependability – The trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers.

- Can pertain to both functionality and system properties
- Dependability is different from
  - Reliability
  - Availability
  - Security
  - Safety

Threats

Loss of dependability by

- System failures
  - Requirements correctly interpreted, but implementation is faulty
  - Requirements are faulty or wrongly interpreted
- Hidden unwanted system properties
- Problems in the environment of a system

Loss may happen

- Accidentally
- Negligently
- Deliberately (typically with criminal intent)

Problems in the system environment (context)

- Errors in the system environment
  - Errors caused by failing devices or neighboring systems
  - Operating errors
  - Unexpected external events

- Violation of assumptions
  - Unexpected input data or events
  - Unexpected reactions to system outputs
  - Manipulation by non-authorized persons
  - Abuse by authorized persons

Measures for assuring dependability

- Prevent errors
- Identify and correct errors
- Tolerate errors
- Demonstrate and assure absence of errors

- Trade-off cost vs. benefit
- Maybe establish dependability for critical components only

Means

- Achieve dependability of software in use by
  - Frequent Use
  - Self-monitoring systems
- Achieve dependability prior to deployment
  - Analytically, in particular thorough testing and static analysis
  - Constructively by
    - Verification
    - Model Checking
    - Assurance (dependability cases)
  - Rigorous processes
- Simplification by modularization

Testing

- System test: not sufficient for establishing dependability
- Preferred means: Random testing based on usage profile
  - Allows statistically sound predictions
  - Problem: Determining the usage profile(s)
  - Requires a large number of test cases (only feasible when test is automated)
- Make sure that the system environment is included in the test (end-to-end testing)

Verification and Model Checking

- Verification
  - In most cases impossible for entire systems → only critical components can be verified
  - Covers the system only, not its environment
  - Verification involves humans who design the proofs → errors in proofs can happen

- Model Checking
  - Full state space of full system is typically too large
    - State space abstractions required
    - actually no verification, but systematic automated test
  - Covers the system only, not its environment

Assuring dependability

- Determine the required dependability properties
  - The less, the easier and cheaper
- Build dependability cases
  - Constructing end-to-end arguments for the required properties
  - using any available techniques (test, verification, etc.)
  - Identify assumptions required for a dependability case to hold
  - Document these assumptions (for example, in a user manual)
- Build dependability cases prior to development
- Orient development towards satisfying dependability cases

Dependability needs a dependable foundation

- Suitable programming languages
  - for example, languages featuring strong type checking
- Dependable hardware
- Dependable operating system
- Dependable communications infrastructure
- Build upon existing dependable systems
  - However: dependability cases need to be re-validated!
- Otherwise the effort for demonstrating / proving the validity for a dependability case can grow infinitely

Dependable software is crucial

- Safety-critical and security-critical systems are becoming pervasive
- Software systems control non-software technical systems we need to rely on (e.g. in transportation, communication, or power generation and distribution)
- Due to networking interdependencies, seemingly uncritical systems are becoming critical

➥ We crucially need dependable software systems

Reading assignment

Read the following article:

B. Nuseibeh, C. B. Haley, and C. Foster (2009). Securing the Skies: In Requirements We Trust

It is about making end-to-end arguments for the security of a system, which ultimately contributes to its dependability.

References

C. Bird, N. Nagappan, P. Devanbu, H. Gall, B. Murphy (2009). Does Distributed Development Affect Software Quality?: An Empirical Case Study of Windows Vista. *Communications of the ACM* **52**(8):85–93.

B. Boehm, J.R. Brown, and M. Lipow (1976). Quantitative Evaluation of Software Quality. *Proceedings of the 2nd International Conference on Software Engineering*, San Francisco. 592–605.

ISO/IEC (2001). *Software Engineering – Product Quality – Part 1: Quality Model*. International Standard ISO/IEC 9126-1:2001.

ISO/IEC (2011). *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. International Standard ISO/IEC 25010:2011

D. Jackson (2009). A Direct Path to Dependable Software. *Communications of the ACM* **52**(4):78–88.

J.C. Laprie (1985). Dependable Computing and Fault Tolerance: Concepts and terminology. *Proc. 15th IEEE International Symposium on Fault-Tolerant Computing*. 2–11.

J.A. McCall, M.T. Matsumoto (1980). *Software Quality Measurement Manual*, Vol. II. Rome Air Development Center, RADC-TR-80-109-Vol-2.

N. Nagappan, T. Ball, A. Zeller (2006). Mining Metrics to Predict Component Failures. *Proceedings of the 28th international conference on Software engineering (ICSE 2006)*. 452–461.

B. Nuseibeh, C. B. Haley, and C. Foster (2009). Securing the Skies: In Requirements We Trust. *IEEE Computer* **42**(9):64–72.

J. H. Saltzer, D. P. Reed, D. D. Clark (1984). End-to-End Arguments in System Design. *ACM Transactions on Computer Systems* **2**(4). 277–288.

References – 2

J. Sliwerski, T. Zimmermann, A. Zeller (2005).When do Changes Induce Fixes? On Fridays. *Proc. International Workshop on Mining Software Repositories (MSR)*, St. Louis, Missouri, USA, May 2005.

S. Wagner et al. (2012). The Quamoco Product Quality Modelling and Assessment Approach. *Proc. 34th International Conference on Software Engineering (ICSE 2012)*, Zurich, Switzerland. 1133–1142.

C. Weinstock, J. Goodenough, and J. Hudak (2004). *Dependability Cases.* Technical Note CMU/SEI-2004-TN-016). Pittsburgh, PA: Software Engineering Institute. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=6919

T. Zimmermann, P. Weißgerber, S. Diehl, A. Zeller. Mining Version Histories to Guide Software Changes (2005). *IEEE Transactions on Software Engineering* **31**(6):429–445.