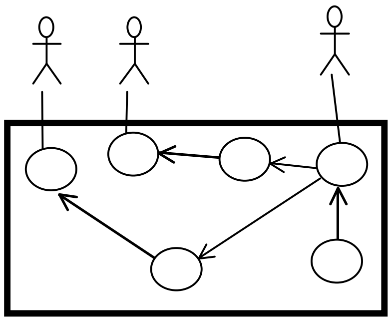
Lecture 16:
Non-Functional Requirements (NFRs)

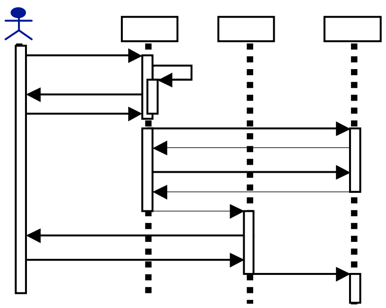
- **Refresher:**
  - Modeling notations we've met
- **What are NFRs?**
  - Quality factors, design criteria; metrics
  - Example NFRs
- **Product-oriented approaches to NFRs**
  - Making quality factors specific
  - Example: Reliability
- **Process-oriented approaches to NFRs**
  - Softgoal analysis for design tradeoffs

We've seen these UML diagrams…



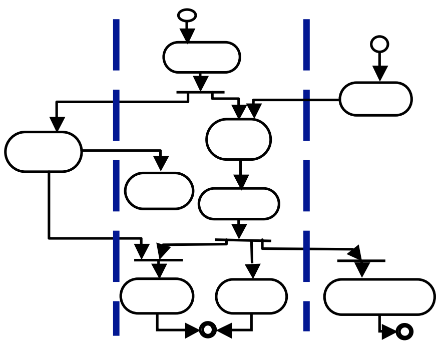
Use Cases

- user's view
- Lists functions
- visual overview of the main requirements



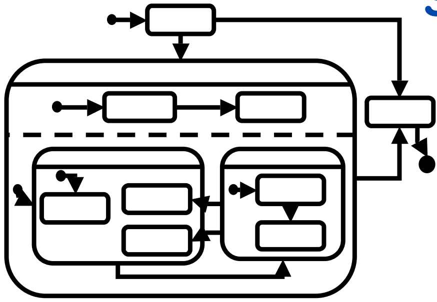
Sequence Diagrams

- individual scenario
- interactions between users and system
- Sequence of messages



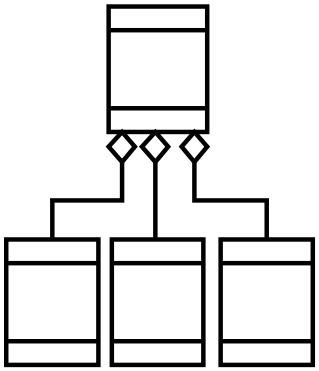
Activity diagrams

- business processes
- concurrency and synchronization
- dependencies between tasks



Statecharts

- responses to events
- dynamic behavior
- event ordering, reachability, deadlock, etc



Class Diagrams

- information structure
- relationships between data items
- modular structure for the system

…and the following non-UML diagrams:

- Goal Models
  - Capture strategic goals of stakeholders
  - Good for exploring 'how' and 'why' questions with stakeholders
  - Good for analysing trade-offs, especially over design choices
- Fault Tree Models (as an example risk analysis technique)
  - Capture potential failures of a system and their root causes
  - Good for analysing risk, especially in safety-critical applications
- Strategic Dependency Models (i*)
  - Capture relationships between actors in an organisational setting
  - Helps to relate stakeholders's goals to their organisational setting
  - Good for understanding how the organisation will be changed
- Entity-Relationship Models
  - Capture the relational structure of information to be stored
  - Good for understanding constraints and assumptions about the subject domain
  - Good basis for database design
- Mode Class Tables, Event Tables and Condition Tables (SCR)
  - Capture the dynamic behaviour of a real-time reactive system
  - Good for representing functional mapping of inputs to outputs
  - Good for making behavioural models precise, for automated reasoning

What are Non-functional Requirements?

→ Functional vs. Non-Functional

- Functional requirements describe what the system should do
  - functions that can be captured in use cases
  - behaviours that can be analyzed by drawing sequence diagrams, statecharts, etc.
  - … and probably trace to individual chunks of a program
- Non-functional requirements are global constraints on a software system
  - e.g. development costs, operational costs, performance, reliability, maintainability, portability, robustness etc.
  - Often known as software qualities, or just the "ilities"
  - Usually cannot be implemented in a single module of a program

→ The challenge of NFRs

- Hard to model
- Usually stated informally, and so are:
  - often contradictory,
  - difficult to enforce during development
  - difficult to evaluate for the customer prior to delivery
- Hard to make them measurable requirements
  - We'd like to state them in a way that we can measure how well they've been met

Example NFRs

- **Interface requirements**
  - how will the new system interface with its environment?
    - User interfaces and "user-friendliness"
    - Interfaces with other systems
- **Performance requirements**
  - time/space bounds
    - workloads, response time, throughput and available storage space
    - e.g. "the system must handle 1,000 transactions per second"
  - reliability
    - the availability of components
    - integrity of information maintained and supplied to the system
    - e.g. "system must have less than 1hr downtime per three months"
  - security
    - E.g. permissible information flows, or who can do what
  - survivability
    - E.g. system will need to survive fire, natural catastrophes, etc
- **Operating requirements**
  - physical constraints (size, weight),
  - personnel availability & skill level
  - accessibility for maintenance
  - environmental conditions
  - etc
- **Lifecycle requirements**
  - "Future-proofing"
    - Maintainability
    - Enhanceability
    - Portability
    - expected market or product lifespan
  - limits on development
    - E.g development time limitations,
    - resource availability
    - methodological standards
    - etc.
- **Economic requirements**
  - e.g. restrictions on immediate and/or long-term costs.

Approaches to NFRs

- Product vs. Process?
  - Product-oriented Approaches
    - Focus on system (or software) quality
    - Capture operational criteria for each requirement
    - … so that we can measure it once the product is built
  - Process-oriented Approaches
    - Focus on how NFRs can be used in the design process
    - Analyze the interactions between NFRs and design choices
    - … so that we can make appropriate design decisions
- Quantitative vs. Qualitative?
  - Quantitative Approaches
    - Find measurable scales for the quality attributes
    - Calculate degree to which a design meets the quality targets
  - Qualitative Approaches
    - Study various relationships between quality goals
    - Reason about trade-offs etc.

Software Qualities

- Think of an everyday object
  - e.g. a chair - how would you measure it's "quality"?
    - construction quality? (e.g. strength of the joints,…)
    - aesthetic value? (e.g. elegance,…)
    - fit for purpose? (e.g. comfortable,…)
- All quality measures are relative
  - there is no absolute scale
  - we can sometimes say A is better than B…
    - … but it is usually hard to say how much better!
- For software:
  - construction quality?
    - software is not manufactured
  - aesthetic value?
    - but most of the software is invisible
    - aesthetic value is a marginal concern
  - fit for purpose?
    - Need to understand the purpose

Fitness

Source: *Budgen, 1994, pp58-9*

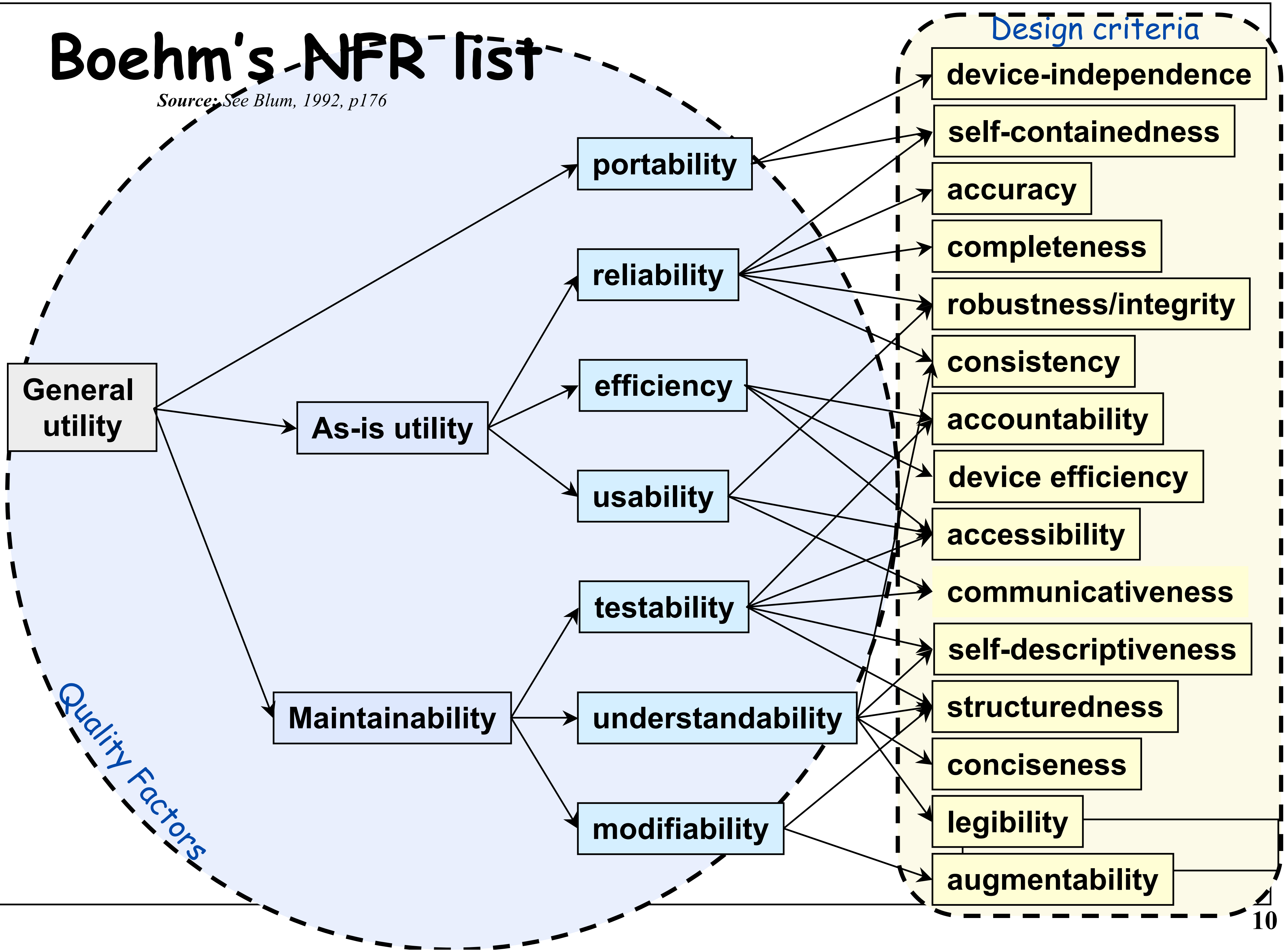
- **Software quality is all about fitness to purpose**
  - does it do what is needed?
  - does it do it in the way that its users need it to?
  - does it do it reliably enough? fast enough? safely enough? securely enough?
  - will it be affordable? will it be ready when its users need it?
  - can it be changed as the needs change?
- **Quality is not a measure of software in isolation**
  - it measures the relationship between software and its application domain
    - cannot measure this until you place the software into its environment…
    - …and the quality will be different in different environments!
  - during design, we need to *predict* how well the software will fit its purpose
    - we need good quality predictors (design analysis)
  - during requirements analysis, we need to *understand* how fitness-for-purpose will be measured
    - What is the intended purpose?
    - What quality factors will matter to the stakeholders?
    - How should those factors be operationalized?

Factors vs. Criteria

- Quality Factors
  - These are customer-related concerns
    - Examples: efficiency, integrity, reliability, correctness, survivability, usability,...
- Design Criteria
  - These are technical (development-oriented) concerns such as anomaly management, completeness, consistency, traceability, visibility,...
- Quality Factors and Design Criteria are related:
  - Each factor depends on a number of associated criteria:
    - E.g. correctness depends on completeness, consistency, traceability,...
    - E.g. verifiability depends on modularity, self-descriptiveness and simplicity
  - There are some standard mappings to help you…
- During Analysis:
  - Identify the relative importance of each quality factor
    - From the customer's point of view!
  - Identify the design criteria on which these factors depend
  - Make the requirements measurable

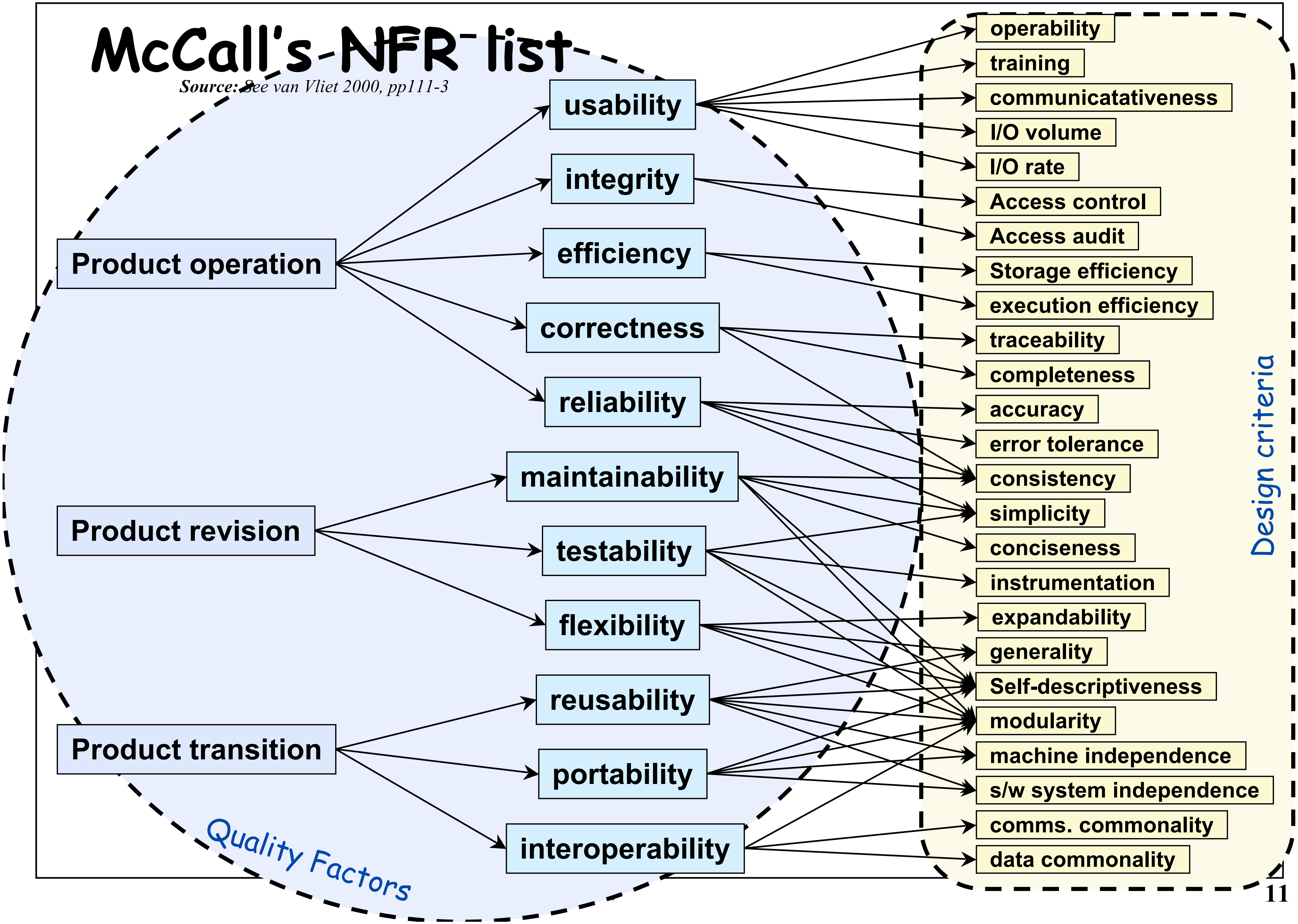
Boehm's NFR list

Source: See Blum, 1992, p176



McCall's NFR list

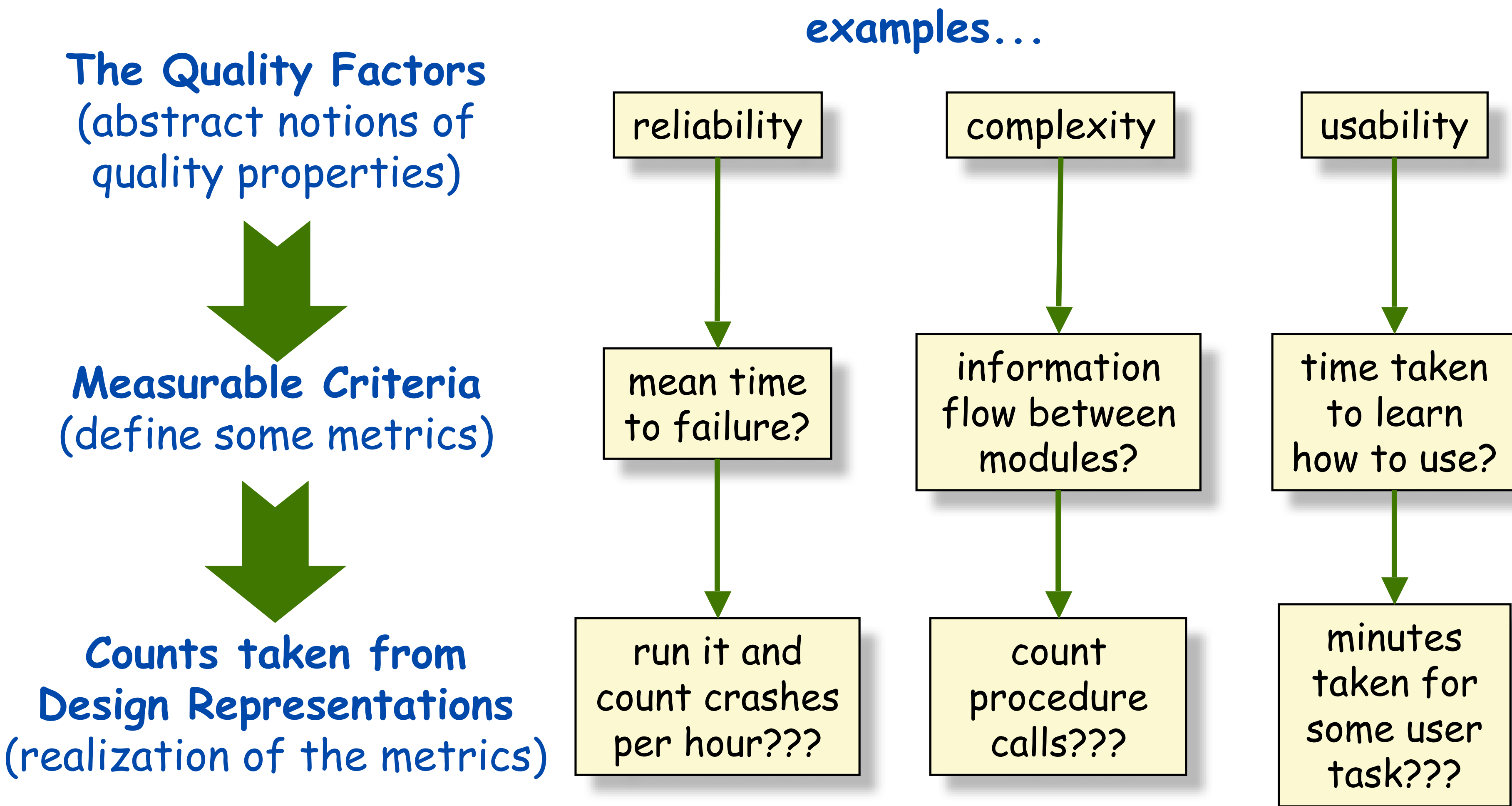
Source: See van Vliet 2000, pp111-3



Making Requirements Measurable

Source: *Budgen, 1994, pp60-1*

→ We have to turn our vague ideas about quality into measurables



Example: Measuring Reliability

- **Example Definition**
  - the ability of the system to behave consistently in a user-acceptable manner when operating within the environment for which it was intended.

- **Comments:**
  - Reliability can be defined in terms of a percentage (say, 99.999%)
  - This may have different meaning for different applications:
    - Telephone network: the entire network can fail no more than, on average, 1hr per year, but failures of individual switches can occur much more frequently
    - Patient monitoring system: the system may fail for up to 1hr/year, but in those cases doctors/nurses should be alerted of the failure. More frequent failure of individual components is not acceptable.
  - Best we can do may be something like:
    - "...No more than X bugs per 10KLOC may be detected during integration and testing; no more than Y bugs per 10KLOC may remain in the system after delivery, as calculated by the Monte Carlo seeding technique of appendix Z; the system must be 100% operational 99.9% of the calendar year during its first year of operation..."

Measuring Reliability…

- **Example reliability requirement:**
  - "The software shall have no more than X bugs per thousand lines of code"
  - ...But is it possible to measure bugs at delivery time?
- **Use bebugging**
  - Measures the effectiveness of the testing process
  - a number of seeded bugs are introduced to the software system
    - then testing is done and bugs are uncovered (seeded or otherwise)

$$\text{Estimated number of bugs in system} = \frac{\text{\# of seeded bugs} \times \text{\# of detected bugs}}{\text{\# of detected seeded bugs}}$$

  - ...BUT, not all bugs are equally important!

Example model: Reliability growth

Source: Adapted from Pfleeger 1998, p359

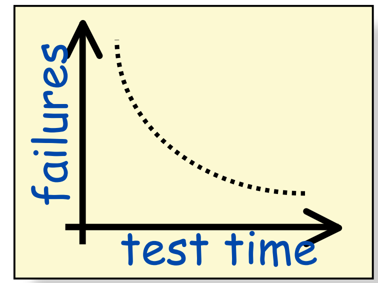
→ Motorola's Zero-failure testing model

- Predicts how much more testing is needed to establish a given reliability goal
- basic model:

empirical constants

$$\text{failures} = ae^{-b(t)}$$

testing time



→ Reliability estimation process

- Inputs needed:
  - fd = target failure density (e.g. 0.03 failures per 1000 LOC)
  - tf = total test failures observed so far
  - th = total testing hours up to the last failure
- Calculate number of further test hours needed using:

$$\frac{\ln(fd/(0.5 + fd)) \times th}{\ln((0.5 + fd)/(tf + fd))}$$

- Result gives the number of further failure free hours of testing needed to establish the desired failure density
  - if a failure is detected in this time, you stop the clock and recalculate
- Note: this model ignores operational profiles!

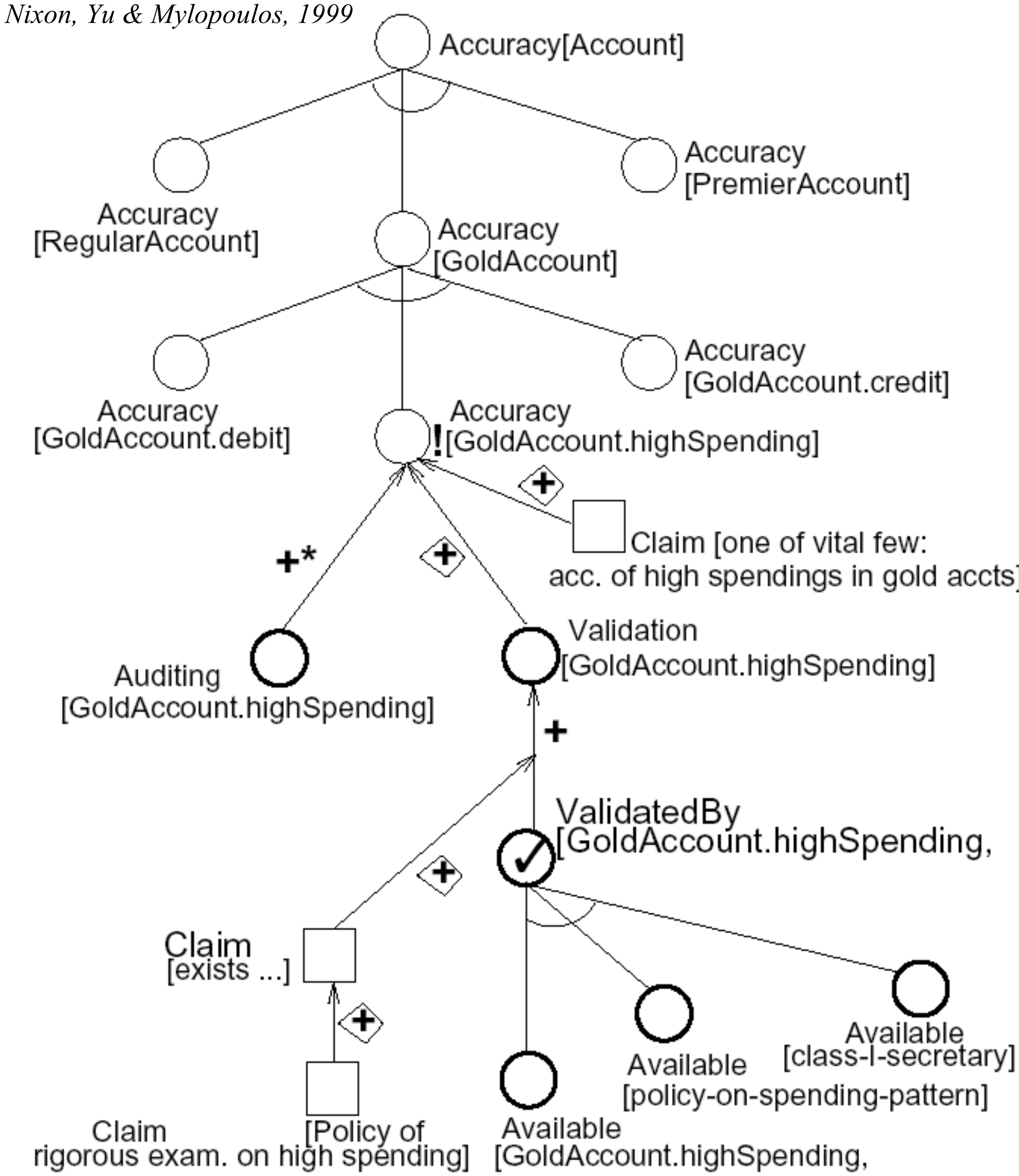
Making Requirements Measurable

- Define 'fit criteria' for each requirement
  - Give the 'fit criteria' alongside the requirement
  - E.g. for new ATM software
    - Requirement: "The software shall be intuitive and self-explanatory"
    - Fit Criteria: "95% of existing bank customers shall be able to withdraw money and deposit cheques within two minutes of encountering the product for the first time"
- Choosing good fit criteria
  - Stakeholders are rarely this specific
  - The right criteria might not be obvious:
    - Things that are easy to measure aren't necessarily what the stakeholders want
    - Standard metrics aren't necessary what stakeholders want
  - Work with stakeholders to find good fit criteria
- Proxies
  - Sometimes the quality is not directly measurable. Seek indicators instead:
    - E.g. "Few data entry errors" as proxy for Usability
    - E.g. "Loose coupling" as a proxy for Maintainability

Using softgoal analysis

Source: *Chung, Nixon, Yu & Mylopoulos, 1999*

- → **Goal types:**
  - Non-functional Requirement
  - Satisficing Technique
    - e.g. a design choice
  - Claim
    - supporting/explaining a choice
- → **Contribution Types:**
  - AND links (decomposition)
  - OR links (alternatives)
  - Sup links (supports)
  - Sub links (necessary subgoal)
- → **Evaluation of goals**
  - Satisficed
  - Denied
  - Conflicting
  - Undetermined



NFR Catalogues

Source: *Cysneiros & Yu, 2004*

- **Predefined catalogues of NFR decomposition**
  - Provides a knowledge base to check coverage of an NFR
  - Provides a tool for elicitation of NFRs
  - Example: