PLM and Innovation Excellence

Learning Campus

Your partner for Business Learning

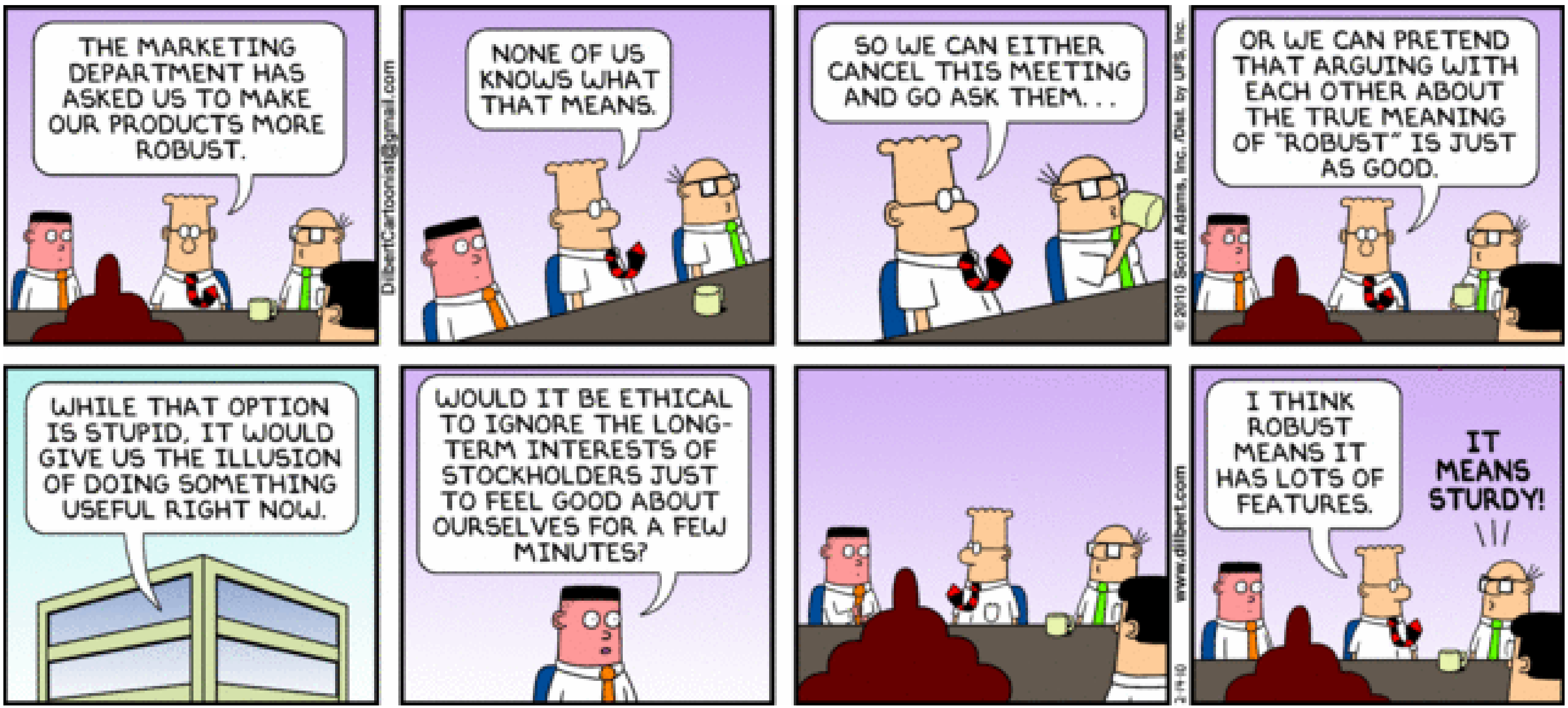
Siemens Core Learning Program

Dealing with Quality Attributes

Authors: Peter Zimmerer, CT | Rüdiger Kreuter, CT | Christian Hahn, CT | Sylvia Jell, CT

SIEMENS
Ingenuity for life

Dealing with quality attributes



Dealing with quality attributes

Learning objectives

- Understand the relevance of qualities
- Know basic implementation tactics for qualities
- Get to know design patterns as solutions to recurring problems
- Understand how to test qualities

Scope of *Dealing with Quality Attributes*

Remember WS1

As a Test Architect
You have one Role – but you are wearing two Hats

Test Expert
for the system under test (SUT)

- Design the test approach
- Apply innovative test technologies
- Drive the quality of the SUT

Software / System Architect
for the test system

- Design and realize the test architecture
- Apply innovative software technologies
- Drive the quality of the test system

Quality Attributes
of the SUT (production code)

Quality Attributes
of the test system (test code)

Dealing with Qualities

Agenda

Quality Attribute / Quality / "Non-Functional Requirement" (NFR)

SIEMENS
Ingenuity for life

There are many definitions for a "Quality Attribute"; a useful one is:

> **A requirement that specifies system properties,**
> **such as environmental and implementation constraints, performance, dependencies, maintainability, extensibility and reliability.**
> **A requirement that specifies physical constraints on a functional requirement.**
>
> Jacobson, Booch, and Rumbaugh

By contrast, a functional requirement defines the transport, processing, storage and control behaviors of a system, regarding material, energy and information.

Note: "Quality" or "Quality Attribute" or "NFR" are often used as synonyms.
In this course we use these terms in a synonym way!

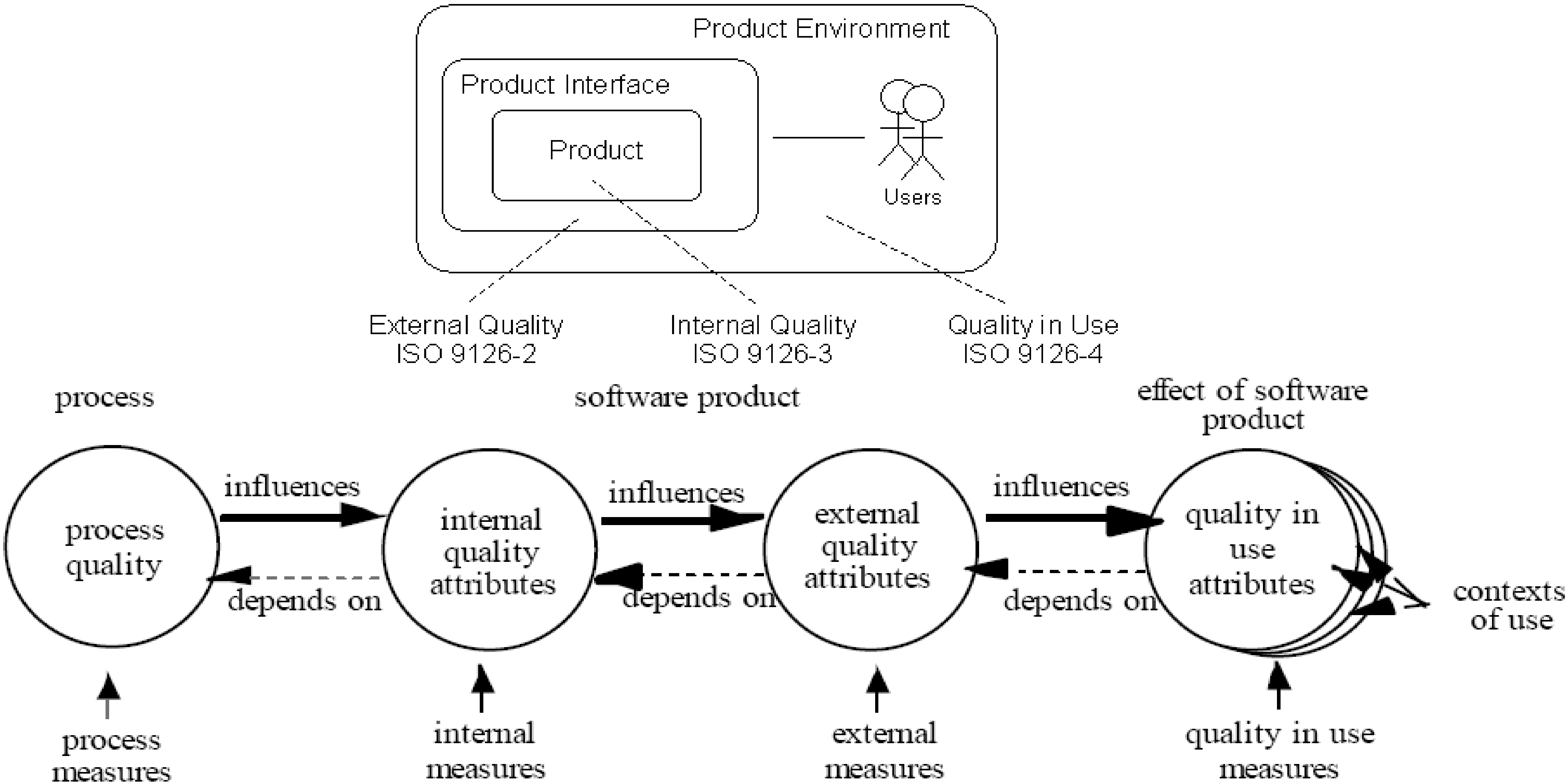
Some characteristics of quality attributes

- Inseparable from architecture design
- May be cross-cutting
- Cannot be discovered by looking at customer processes
- Main sources for discovering qualities are stakeholder requests
- Trace to higher-level goals of the system
- Many are due to constraints – regulatory, solution, environmental
- …

Typical issues with quality attributes

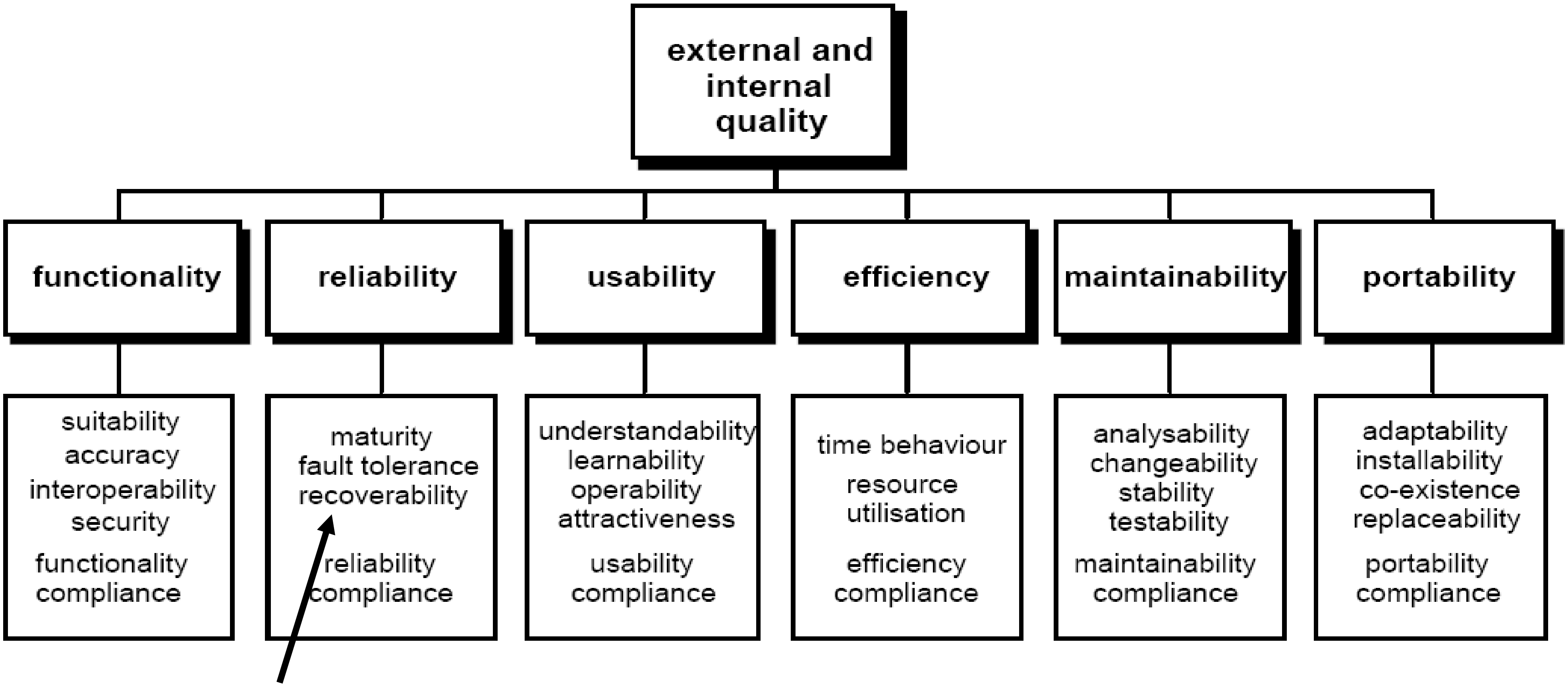
- No commonly agreed-upon vocabulary (semantics and syntax) among stakeholders
- Hidden and poorly understood dependencies
- Conflicting qualities
- Qualities cannot be discovered by looking at customer processes
- Incomplete stakeholder input
- Only indirect stakeholder input available, based upon existing or desired product quality and features
- Many qualities are derived from regulations, standards and norms; customers often expect them without naming them explicitly
- Qualities given for the system, but relate to features or functions; they may be cross-cutting though

ISO/IEC 9126 Quality model – Overview (1)



http://www.iso.org/

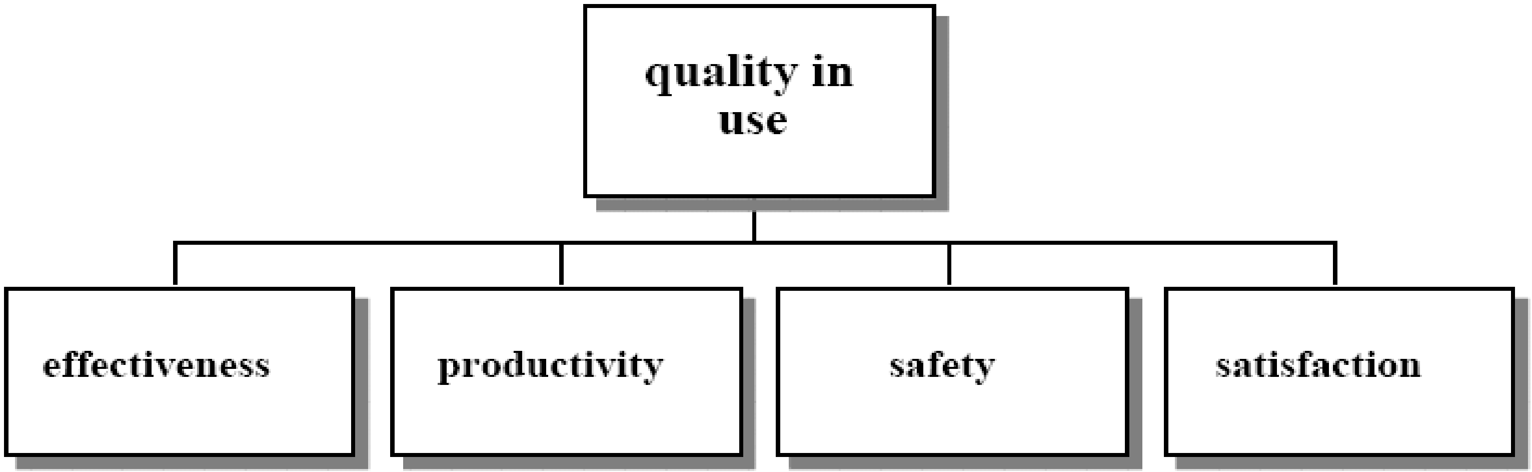
ISO/IEC 9126 Quality model – Overview (2)

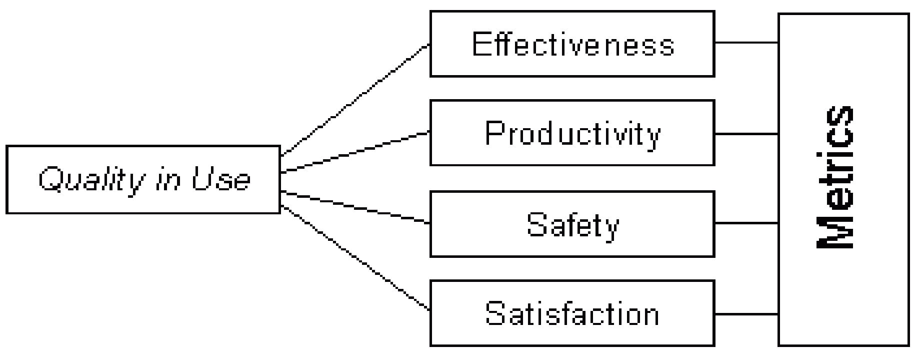


Availability as a combination of maturity, fault tolerance, and recoverability is included here.

http://www.iso.org/

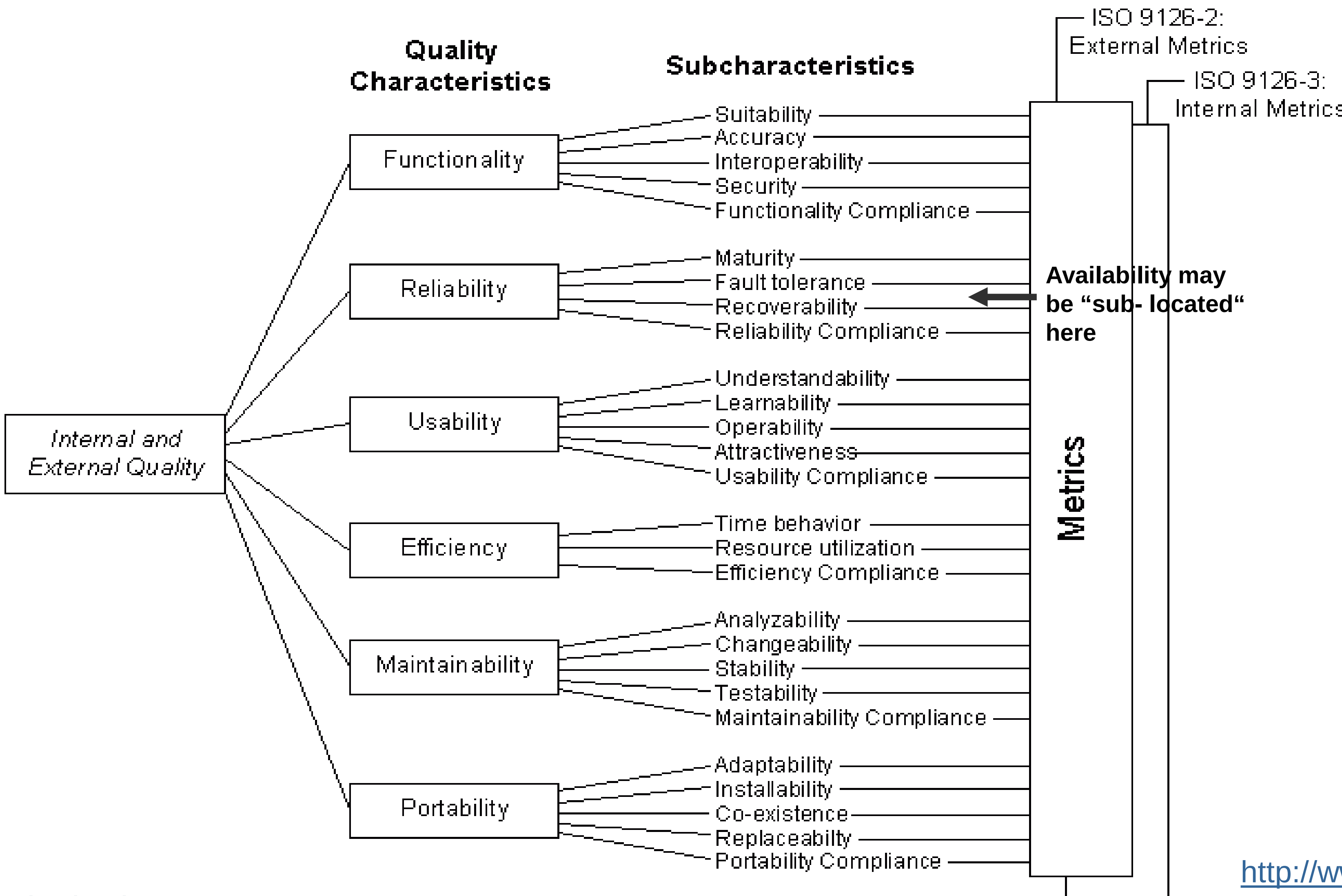
ISO/IEC 9126 Quality model – Overview (3)





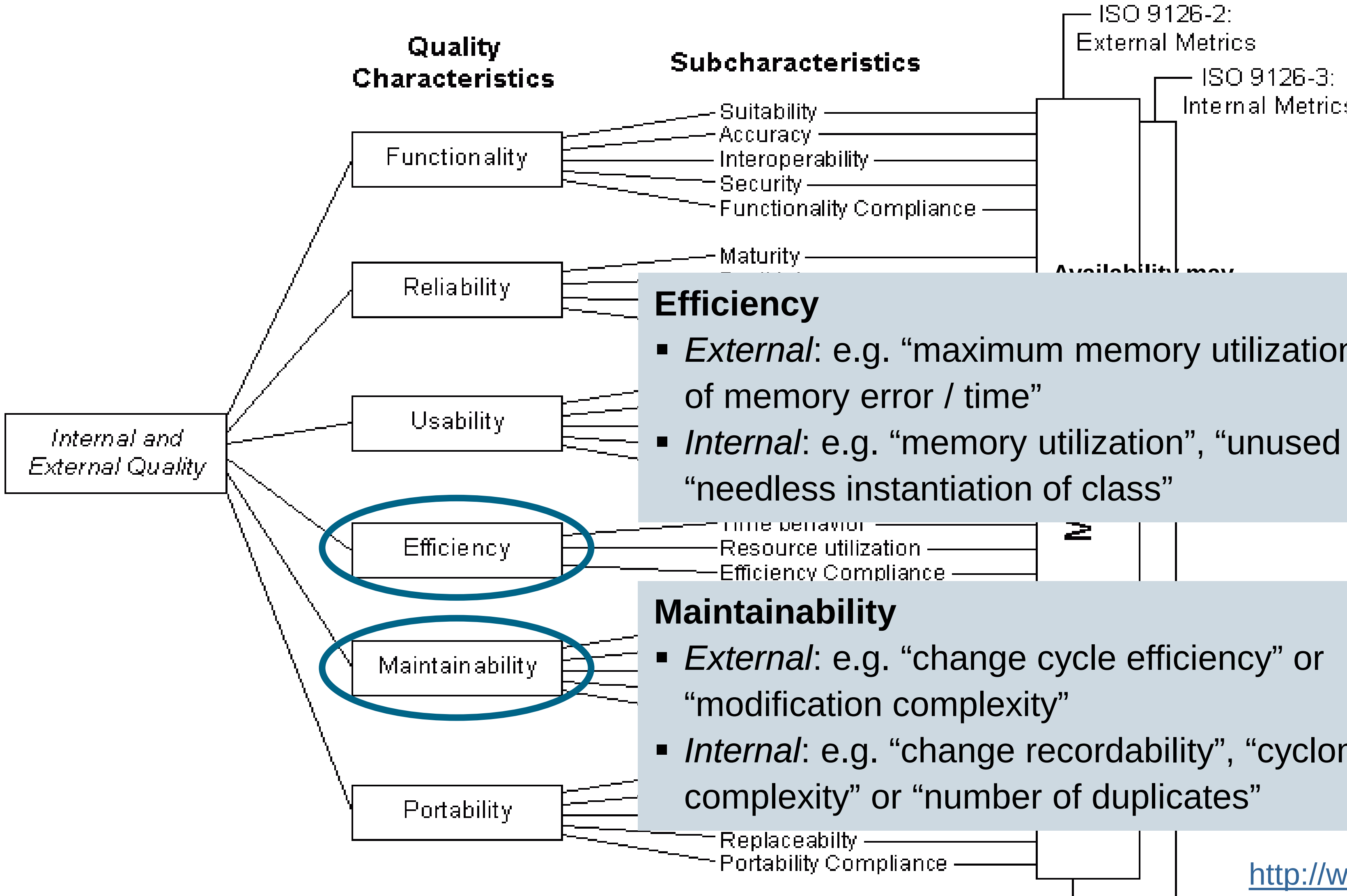
http://www.iso.org/

ISO/IEC 9126 Quality model – Internal and external quality (1)



http://www.iso.org/

ISO/IEC 9126 Quality model – Internal and external quality (2)



Efficiency

- *External*: e.g. “maximum memory utilization” or “ratio of memory error / time”
- *Internal*: e.g. “memory utilization”, “unused field” or “needless instantiation of class”

Maintainability

- *External*: e.g. “change cycle efficiency” or “modification complexity”
- *Internal*: e.g. “change recordability”, “cyclomatic complexity” or “number of duplicates”

http://www.iso.org/

Close relation between internal and external quality

Internal quality		External quality
Code, structure	↔	Behavior
Developer's view	↔	User's view, customer's view
Invisible root, fault	↔	Visible symptom, failure
Analyze the software	↔	Execute the software
Static	↔	Dynamic
White-box	↔	Black-box

Quality attributes

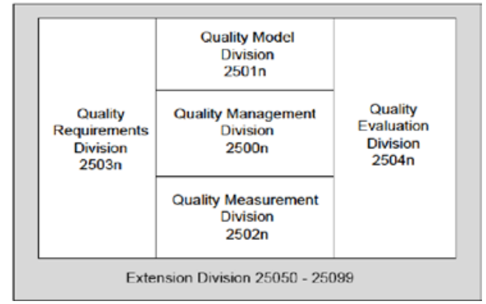
NFRs

Quality characteristics

Internal metrics measure the software itself, **external metrics** measure the behavior of the computer-based system that includes the software. (ISO/IEC 9126-1)

SIEMENS
Ingenuity for life

New ISO/IEC 25010 Product quality model

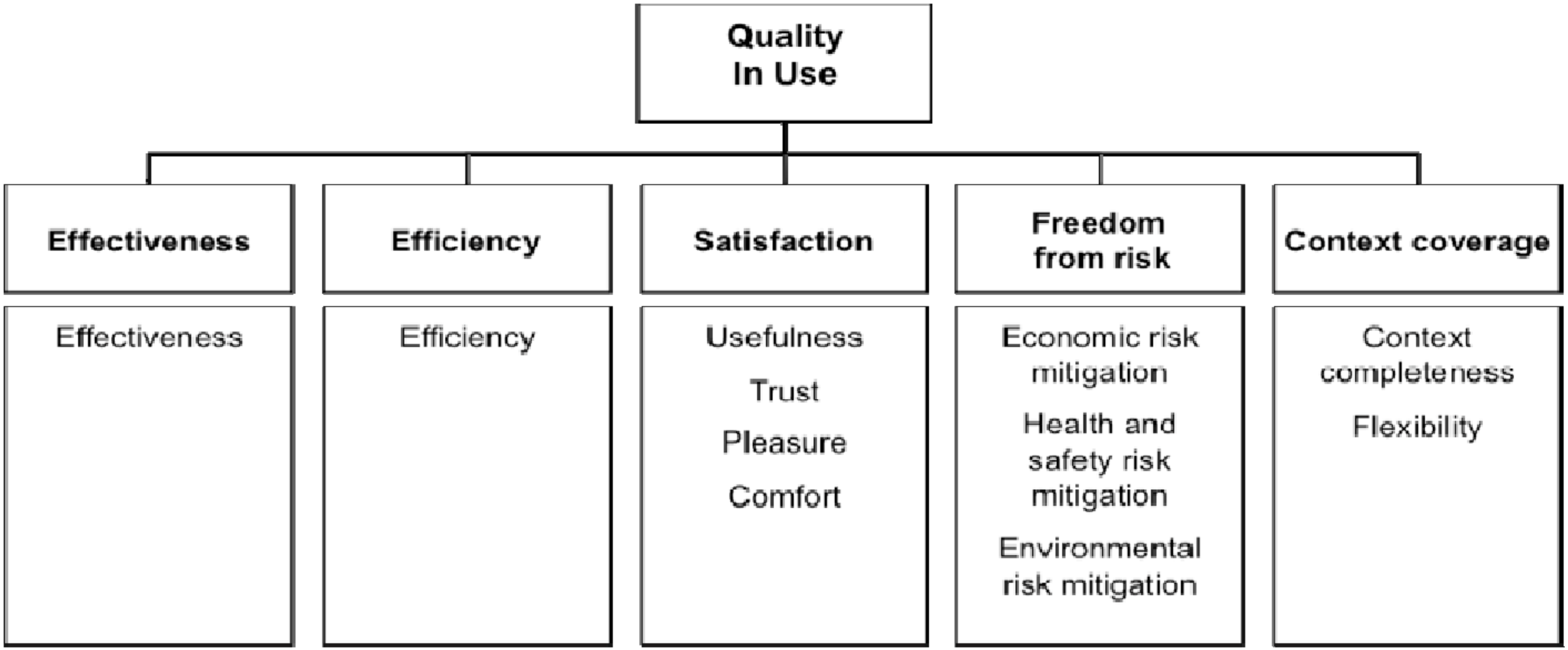


System/Software Product Quality

Functional Suitability	Performance efficiency	Compatibility	Usability	Reliability	Security	Maintain-ability	Portability
Functional completeness	Time-behaviour	Co-existence	Appropriateness recognisability	Maturity	Confidentiality	Modularity	Adaptability
Functional correctness	Resource utilisation	Interoperability	Learnability	Availability	Integrity	Reusability	Installability
Functional appropriateness	Capacity		Operability	Fault tolerance	Non-repudiation	Analysability	Replaceability
			User error protection	Recoverability	Accountability	Modifiability	
			User interface aesthetics		Authenticity	Testability	
			Accessibility				

Internal measures characterize software product quality based upon static representations of the software, **external measures** characterize software product quality based upon the behaviour of the computer-based system including the software, and **quality in use measures** characterize software product quality based upon the effects of using the software in a specific context of use. (ISO/IEC 25020) http://www.iso.org/

New ISO/IEC 25010 Quality in use model



http://www.iso.org/

Terminology examples (mainly from IEEE)

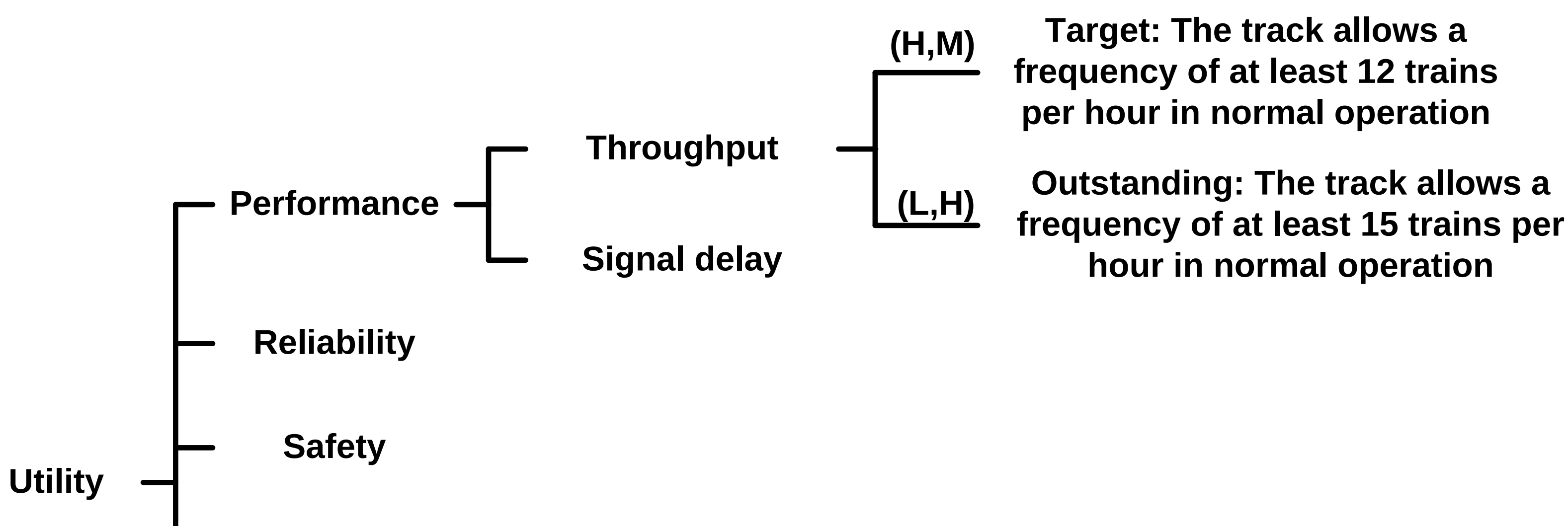
- **Availability (`Uptime / (Uptime + Downtime)`; `MTBF / (MTBF + MTTR)`):**
  Is it usable?
  - The degree to which a system or component is operational and accessible when required for use.
  - The probability that a system is functional at a given time in a specified environment. (John D. Musa)
  - A failure may be acceptable as long as the system is up and running again quickly e.g. telephone system.
- **Reliability ($e^{-\lambda t} = e^{-(t/MTBF)}$ where λ is the failure rate; $\lambda = 1 / MTBF$):**
  Is everything working fine?
  - The ability of a system or component to perform its required functions under stated conditions for a specified period of time (does not account for any repair actions).
  - Probability a system functions without failure for a specified time in a specified environment. (John D. Musa)
  - In general, reliability goals are operation based or time based.
- **Robustness (≈ "stability" (≈ reliability and availability) under unusual situations)**
  - The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environment conditions.
- **Fault tolerance**
  - The ability of a system or component to continue normal operation despite the presence of hardware or software faults.
- **Stability**
  - The capability of the software product to avoid unexpected effects from modifications of the software (bears on the risk of unexpected effect of modifications).

Structuring quality attributes using a "Utility Tree"

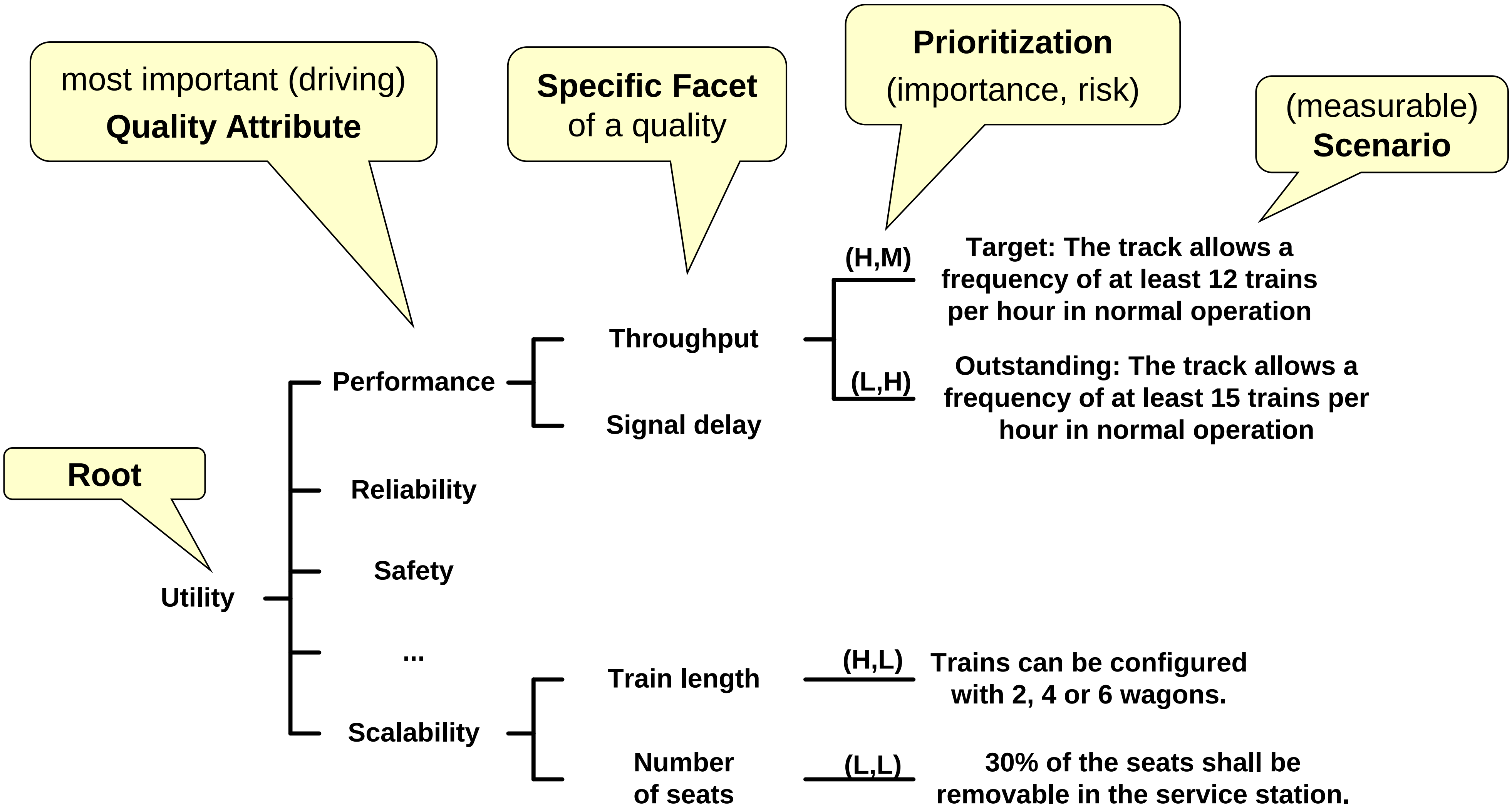
Quality attributes are not atomic. They usually have multiple facets.

A **Utility Tree**

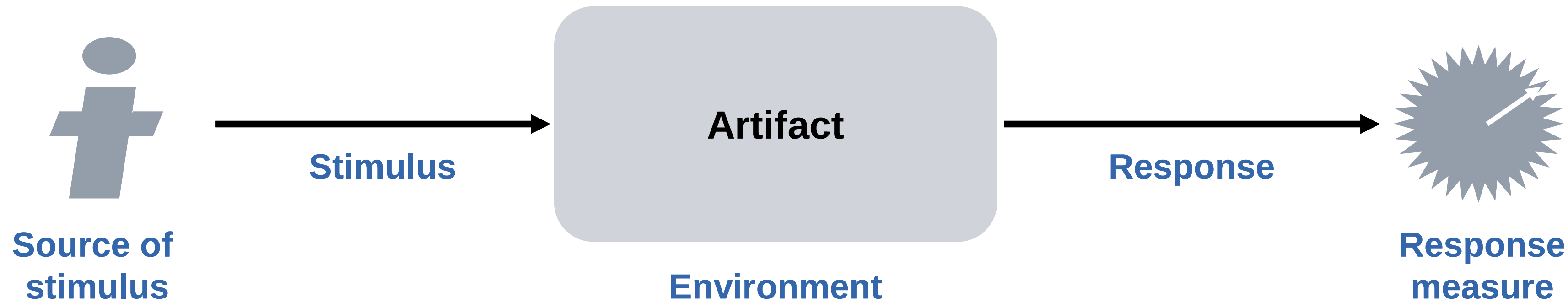
- describes the most important quality attributes (in a project)
- details each quality attribute in relevant facets
- relates scenarios, described in requirements, to quality attributes or their facets and thus allows to prioritize them



Attributes of the "Utility Tree"



Utility tree
Scenarios



***Scenarios* are used to**

- Represent *stakeholders*' interests
- Understand and clarify quality attribute requirements

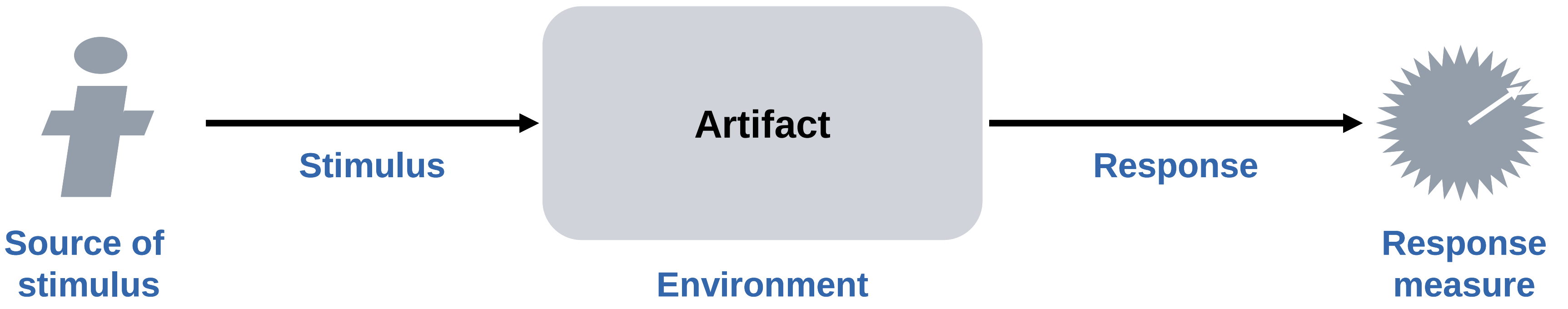
A good scenario makes clear what the stimulus is that causes it and what responses are of interest

Scenarios should cover a range of

- Anticipated uses (use case scenarios)
- Anticipated changes (growth scenarios)
- Unanticipated stresses (exploratory scenarios)

towards the system in focus

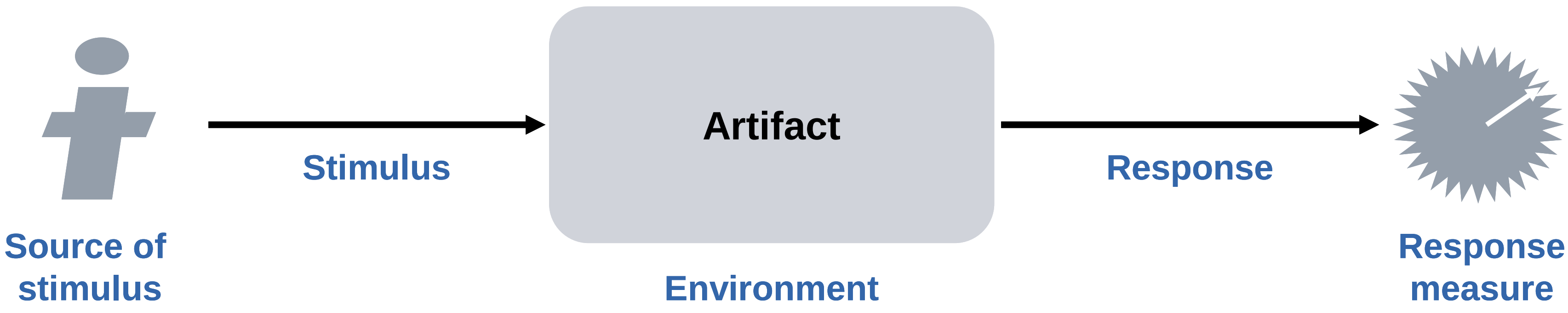
Scenario description Template



Source of stimulus	Who/what initiates the scenario.
Stimulus	Which periodic, stochastic or sporadic event initiates the scenario.
Artifact	What is the relevant unit; e.g. a (part of a) system or a feature.
Environment	What is the environmental condition for this scenario; e.g. normal, startup / shut down, maintenance, emergency, overload, etc.
Response	How does the artifact react to the event in the given environment. This may cause an environment change (e.g. from normal to shutdown mode).
Response measure	How can the response be measured, using indicators like: ▪ the time it takes to process the event (latency or deadline); or the variation in time (jitter) ▪ the amount of data, material or energy that can be processed in a particular time interval (throughput) ▪ or a characterization of the events that cannot be processed (e.g. miss rate, data / energy / material loss)

Scenario description
Examples

SIEMENS
Ingenuity for life



Stimulus for the event that concerns the quality attribute, e.g. function invoked, failure, modification	Relevant assumptions about the environment and the relevant conditions	Precise statement of quality attribute response, e.g. response time, difficulty of modification
One of the CPUs fails	Normal operation in a redundant system	hand-over to mate in <0.1s
X-Ray tube fails	Official lifetime end not yet reached	remote alarm to service center; tube replaced in < 24h
Database is changed from MySQL to Oracle	SW modification during development	Change implemented in 20 work days

Scenario description
Example “Performance of IT Systems”

Source:
One of a number of independent sources

Stimulus:
Periodic/ stochastic/ sporadic events arrive

ARTIFACT:
IT System

Environment:
Normal mode, overload mode, emergency mode, standalone mode

Response:
Response generated within time constraints, changed level of service

Response measure:
Latency deadline, throughput, jitter, miss rate, data loss

Scenario description

Clarifying quality attribute scenarios instead of just quality attributes, will more likely reduce ambiguity and ensure testability (→ xTDD)

It is certainly easy and fast to just say the system should be “fast” or “secure” …

… but this also certainly produces ambiguity!

⇨ If possible ask your stakeholders or customers what they mean
⇨ Help them to understand any ambiguities you see

It’s seems hard to quantify softer quality attributes, like maintainability or usability, but possible; e.g. measurement may be done by specifying test cases and percentage of "good" ratings

Scenario refinement table
Example from SEI Quality Attribute Workshop (QAW)

SIEMENS
Ingenuity for life

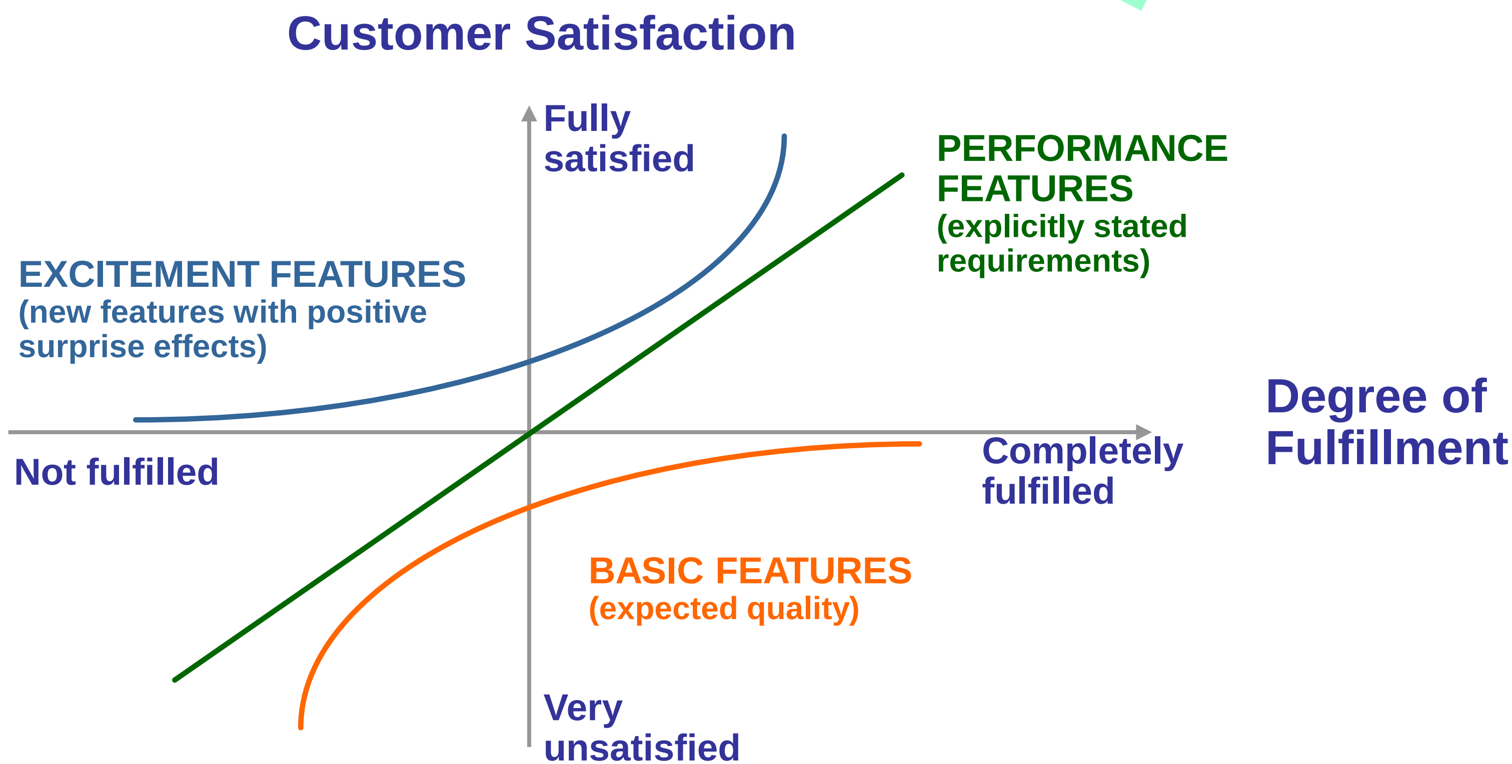
Scenario Refinement for Scenario N		
Scenario(s):		When a garage door opener senses an object in the door's path, it stops the door in less than one millisecond.
Business Goals:		safest system; feature-rich product
Relevant Quality Attributes:		safety, performance
Scenario Components	Stimulus:	An object is in the path of a garage door.
	Stimulus Source:	object external to system, such as a bicycle
	Environment:	The garage door is in the process of closing.
	Artifact (If Known):	system's motion sensor, motion-control software component
	Response:	The garage door stops moving.
	Response Measure:	one millisecond
Questions:		How large must an object be before it is detected by the system's sensor?
Issues:		May need to train installers to prevent malfunctions and avoid potential legal issues.

CMU/SEI Quality Attribute Workshops (QAWs), Third Edition, Technical Report CMU/SEI-2003-TR-016, August 2003
http://www.sei.cmu.edu/reports/03tr016.pdf

Quality attributes and Kano model
Where are we with our Quality?

Remember WS1

SIEMENS
Ingenuity for life



KANO Model

Considering qualities

- Basic requirements must be fulfilled
- Quality often is expected. Where on the curve are we?
- **Can a Quality be a Unique Selling Proposition (USP)?**

Dealing with Qualities

Agenda

Quality Attribute Requirements

Design Strategies & Tactics

Design Patterns

Testing Quality Attributes

Summary

Quality attributes and constraints drive structure!

It is very important to know the relevant quality attributes for a system, and to support them already in the initial concept!

If the system does not support relevant quality attributes initially then this will (usually) lead to high restructure costs!

Imagine …

- to build in Chinese language support in a legacy German train control system
- to increase a system's reliability by factor 10 / 100 / …
  (e.g. an IT system, an Espresso Machine, a train or a car)
- to improve a mechanic optimized for mild climate, so that it works in arctic climate too
- …

Design alternatives for a quality attribute
Strategies & Tactics

SIEMENS
Ingenuity for life

Quality Attribute

Strategy 1	Strategy n
• Tactic 1.1 • Tactic 1.2 • … • Tactic 1.x	• Tactic n.1 • Tactic n.2 • … • Tactic n.y

The Quality in focus

What could be done

How could it be done

Strategies & Tactics "Performance of IT Systems"

Performance of IT Systems

Resource demand	Resource management	Resource arbitration
• Increase computation efficiency • Reduce computational overhead • Manage event rate • Control frequency of events / raw data	• Introduce concurrency • Maintain multiple copies of data/services • Adapt HW resources (processor, memory, network)	• Scheduling policy • Distribution • Localization

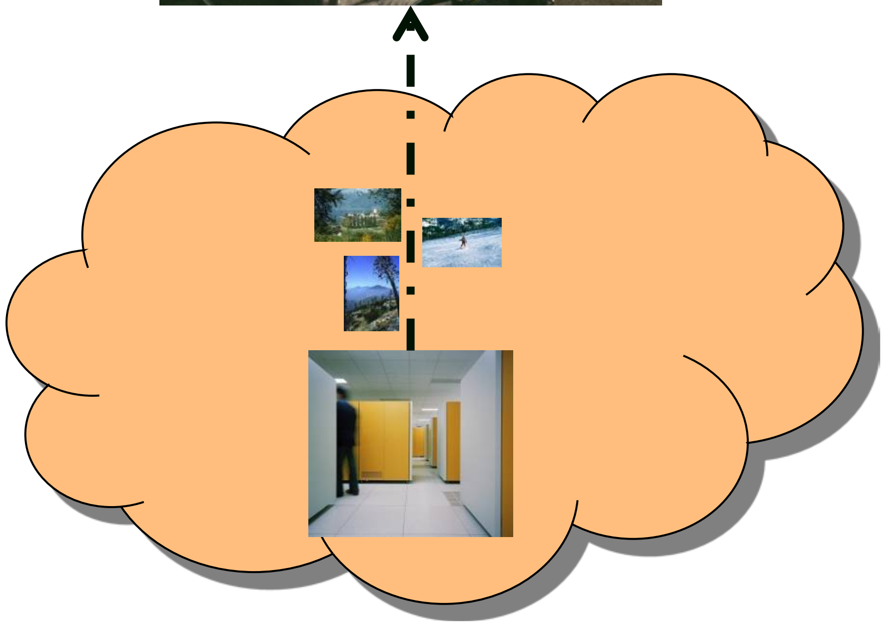
Case study: Image processing

Let us make a deep dive for handling Performance

Users can use a RTC (Rich Thin Client) to access images stored in a network archive (cloud). Also the business logic runs on a server in the cloud.

Users may search for images, browse them (thumbnail presentation), view them in full resolution, and process them.

Workflow support is implemented, e.g.:
Search → Browse → Select → View → Process → Store or Print



Fast enough?

A typical requirement:

Searches for images should be very fast.

An improved alternative: Quality attribute scenario

Whenever a user searches for images in the User Interface (using keywords or advanced search criteria), finding and displaying the results should never take longer than 3 seconds.

Utility tree

Utility	Performance	Data Latency	(H,M)	Storage latency of image archive < 10 msec
		Throughput	(H,H)	One image should be loaded per second
			(M,H)	Even complex image searches should return results within 3 sec
		Perceived Performance	(M,L)	Image storing or printing should not lead to any perceivable delays for other user activities

One example scenario: Search

Source:
User

Stimulus:
Search images

ARTIFACT:
Web-based Image Processing System

Environment:
Normal mode

Response:
Locate images in storage cloud with requested search criteria and always return results (thumbnails and refs) immediately and incrementally to user

Response measure:
latency
< 3 sec

Strategies & Tactics
"Performance of Image Processing"

SIEMENS
Ingenuity for life

Performance of Image Processing

Resource demand

- Use advanced image processing algorithms (if possible GPU based)
- Introduce caching strategies
- Either allow upper bound of clients or scale-out mechanisms depending on resources

Resource management

- Introduce a pool of worker threads for background loading, storing, printing
- Don‘t copy mass data but use meta files and refs for image processing/copying
- Use thumbnail images for browsing
- Only use full resolution when images are selected
- Apply eager loading

Resource arbitration

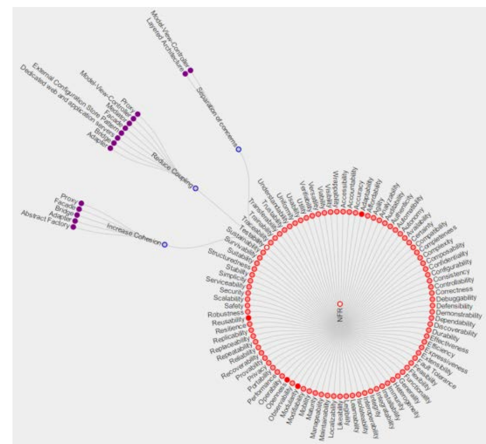
- Schedule resources preferably for processing visible images

There are various patterns for implementing some of the tactics such as
- Caching
- Lazy Evaluation
- Coordinator
- Eager Loading
- Evictor & Activator
- Half Sync / Half Async
- Command Processor

Design strategies and tactics for quality attributes
References

- Bass, Len / Clements, Paul / Kazman, Rick
  Software Architecture in Practice, 3rd edition
  Addison-Wesley, 2012
  ISBN: 978-0321815736
  2nd edition:
  http://www.ece.ubc.ca/~matei/EECE417/BASS/index.html

- NFR Engineering Repository
  https://wse02.siemens.com/content/P0009144/Wiki



Dealing with Qualities

Agenda

Quality Attribute Requirements

Design Strategies & Tactics

Design Patterns

Testing Quality Attributes

Summary

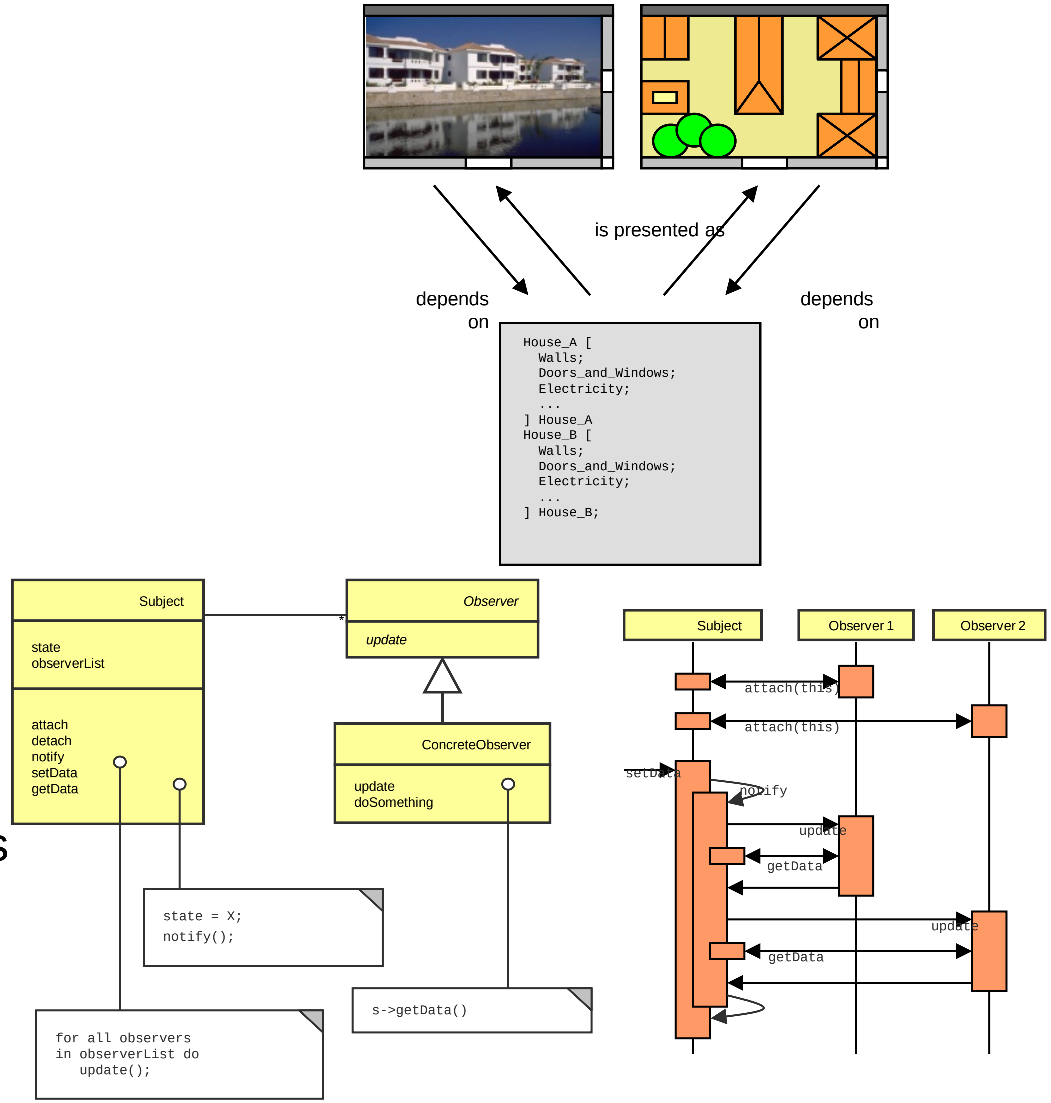
SIEMENS

Ingenuity for life

Pattern – A solution to a recurring problem

A pattern

- Presents a solution for a recurring design problem
- Documents proven design experience; is an ***aggressive disregard of originality*** [Brian Foote]
- Specifies a spatial configuration of elements and the behavior that happens in this configuration
- Provides a common vocabulary and concept understanding
- Addresses additional quality properties of the problem's solution



Caching (POSA) (1)

Context

Systems, that repeatedly access the same set of resources and need to optimize for performance.

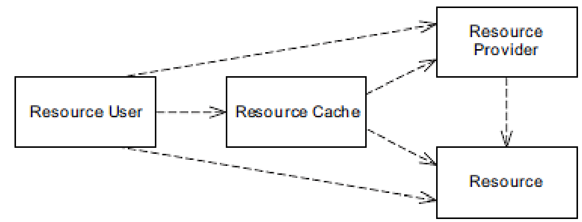
Problem

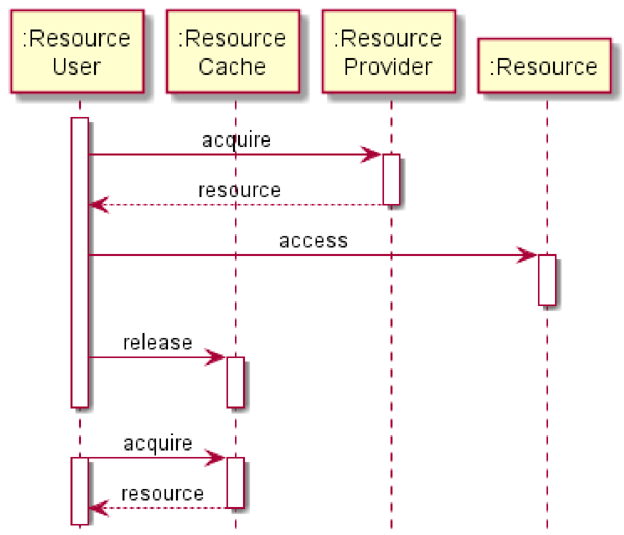
Repetitious acquisition, initialization, and release of the same resource causes unnecessary performance overhead. To address the problem the following forces need to be resolved:

- **Performance**: Cost of repetitious resource access must be minimized.
- **Complexity**: Solution should not make resource access more complex.
- **Availability**: Solution should allow resource accessibility even when resource providers are temporarily unavailable.
- **Scalability**: Solution should scale w.r.t. number of resources.

Solution

Temporarily store the resource in a fast-access buffer called a cache. Subsequently, when the resource is to be accessed again, use the cache to fetch and return the resource instead of acquiring it again from the resource provider.





Frank Buschmann et al., *Pattern-Oriented Software Architecture (POSA)*

Caching (POSA) (2)

Benefits

- **Fast** access to frequently used resources
- Improved **scalability**
- No increased **usage complexity**
- Increased **availability** of resources
- Greater system **stability** due to reduced number of resource releases/reacquisitions

Liabilities

- Depending on the type of resource increased complexity due to **synchronization** needs (update and invalidation strategies to ensure data consistency)
- Reduced **durability** as changes to the cached resource can be lost when system crashes
- Increased run-time **footprint** of the system as possibly unused resources are cached.

Example

Network management system monitoring the state of many network elements where middle tier implements a cache of connections to network elements.

- On user access of a specific network element, the connection is acquired.
- Connection added to the cache when no longer needed by the application.
- For new requests acquisition is done from the cache thus avoiding high acquisition cost.
- Subsequent connections to other network elements established when the user first accesses them.
- Connection is put back into the cache when user context switches to another element.
- If a user accesses the same network element, the connection will be reused.
- No delay will occur on access of reused connections.

Frank Buschmann et al., *Pattern-Oriented Software Architecture (POSA)*

NFR Intent of design patterns

- Many design patterns usually provide solutions to satisfy:
  - Functional requirements (FR-intent) as well as
  - Non-functional requirements (NFR-intent)
- NFR intent not always explicit and clear
- For design decisions it is also important to consider the quality contribution of the pattern

Examples:

Pattern	FR-intent	NFR-intent
Observer	A subject object can notify all related objects (called observers) when it changes state	Without knowing types of the observers → **Extensibility**
Strategy	An object uses an algorithm to resolve a specific problem	Easier to replace the algorithm with a new one → **Replaceability**
Iterator	A client object navigates an aggregate object	Without knowing its internal structure → **Exchangeability**

Pattern example
"Performance of Image Processing"

SIEMENS
Ingenuity for life

Performance of Image Processing

Resource demand

- Use advanced image processing algorithms (if possible GPU based)
- Introduce caching strategies
- Either allow upper bound of clients or scale-out mechanisms depending on resources

Caching

Resource management

- Introduce a pool of work… threads for backgrou… loading, storing, p…g
- Don‘t copy mass data but use meta files and refs for image processing/copying
- Use thumbnail images for browsing
- Only use full resoluti…n when images ar… …ed
- Apply eage…

Pooling

Eager Acquisition

Resource arbitration

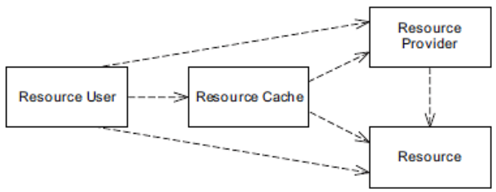
- Schedule resources preferably for processing visible ima…s

Resource Lifecycle Manager

Pattern examples
"Performance of Image Processing"

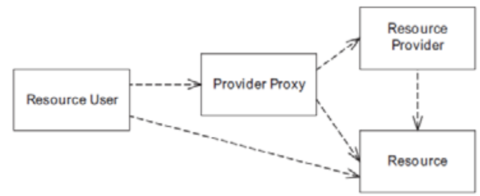
Caching

- Context: Systems that repeatedly access the same set of resources and need to optimize performance.
- Solution: Resources stored temporarily in fast-access buffer (cache); subsequent access through cache instead of resource provider.



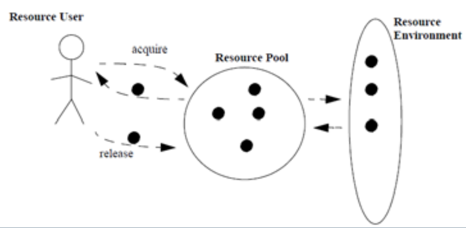
Eager Acquisition

- Context: Systems that must satisfy high predictability and performance in resource acquisition time.
- Solution: Resources eagerly acquired before their actual use by a provider proxy; resources then kept in an efficient container; requests intercepted by the proxy.



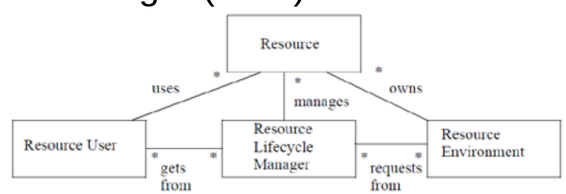
Pooling

- Context: Systems that continuously acquire and release resources of same/similar type.
- Solution: Multiple instances of one type of resource managed in a pool; released resources are put back into the pool.



Resource Lifecycle Manager

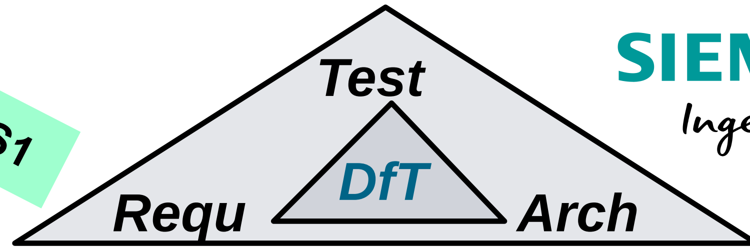
- Context: Systems that require simplified management of the lifecycle of their resources.
- Solution: Resource usage separated from resource management through introduction of a Resource Lifecycle Manager (RLM).



Michael Kircher; Prashant Jain, *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management,* John Wiley & Sons, 2004

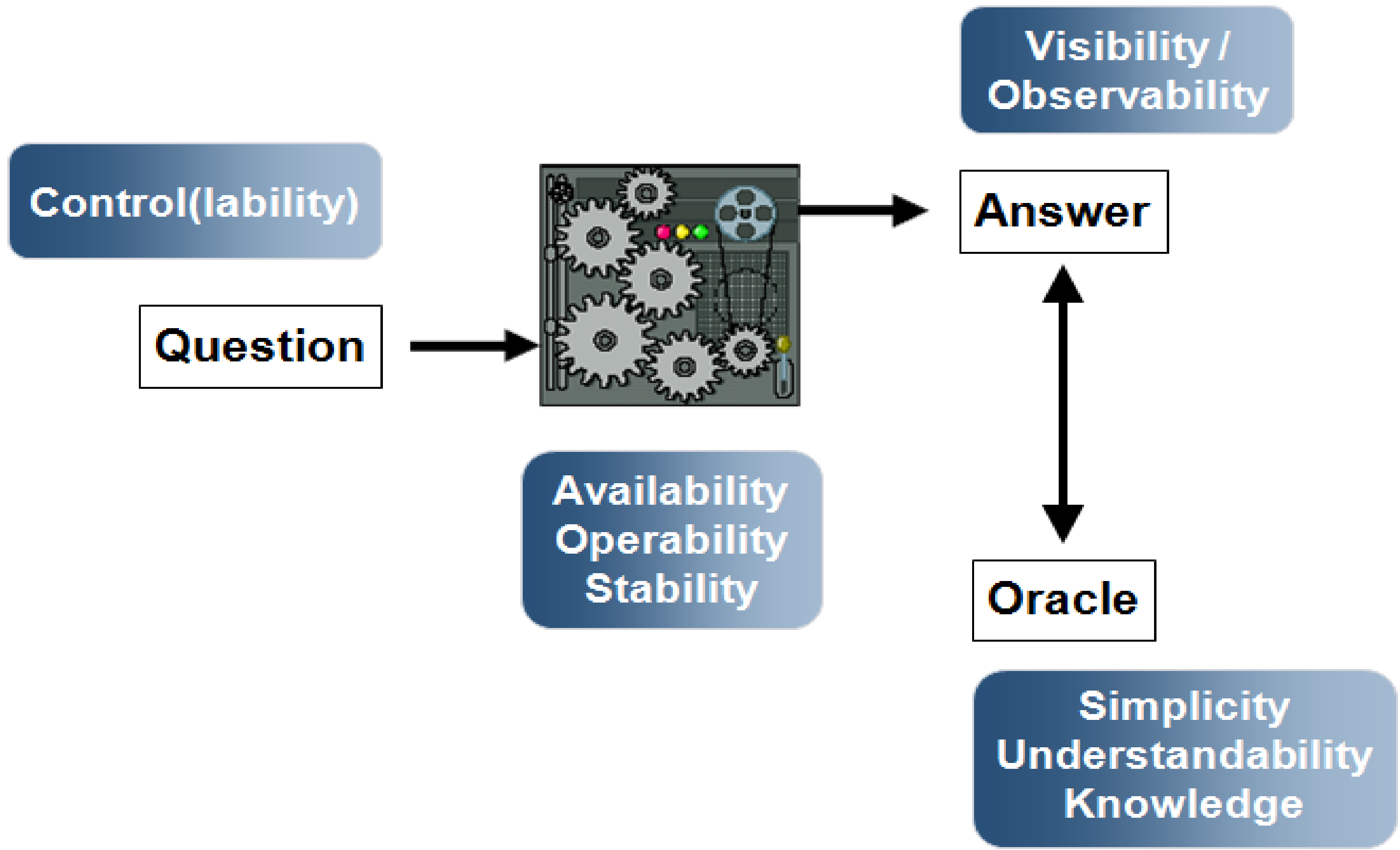
Testability and Design for testability (DfT)

Remember WS1



SIEMENS
Ingenuity for life

- **What?**
- **Why?**
- **Who?**
- **How?**



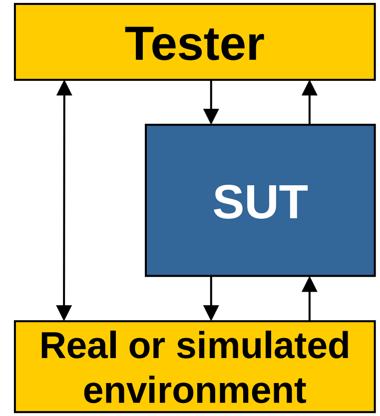
→ ***Strategy for Design for Testability (DfT) over the lifecycle***

Remember WS1

SIEMENS
Ingenuity for life

Architectural and design patterns (1)

Use layered architectures, reduce number of dependencies



Use good design principles

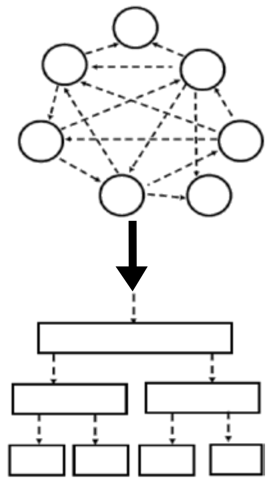
- High cohesion, loose coupling, separation of concerns

Component orientation and adapters ease integration testing

- Components are the units of testing
- Component interface methods are subject to black-box testing

Avoid / resolve cyclic dependencies between components

- Combine or split components
- Dependency inversion via callback interface (observer-observable design pattern)



Example: MSDN Testability guidance

- *Using the Model-View-Presenter (MVP) Design Pattern to enable Presentational Interoperability and Increased Testability*

http://blogs.msdn.com/b/jowardel/archive/2008/09/09/using-the-model-view-presenter-mvp-design-pattern-to-enable-presentational-interoperability-and-increased-testability.aspx

Remember WS1

SIEMENS

Ingenuity for life

Architectural and design patterns (2)

Provide appropriate test hooks and factor your design in a way that lets test code interrogate and control the running system

- Isolate and encapsulate dependencies on the external environment
- Use patterns like dependency injection, interceptors, introspective
- Use configurable factories to retrieve service providers
- Declare and pass along parameters instead of hardwire references to service providers
- Declare interfaces that can be implemented by test classes
- Declare methods as overridable by test methods
- Avoid references to literal values
- Shorten lengthy methods by making calls to replaceable helper methods

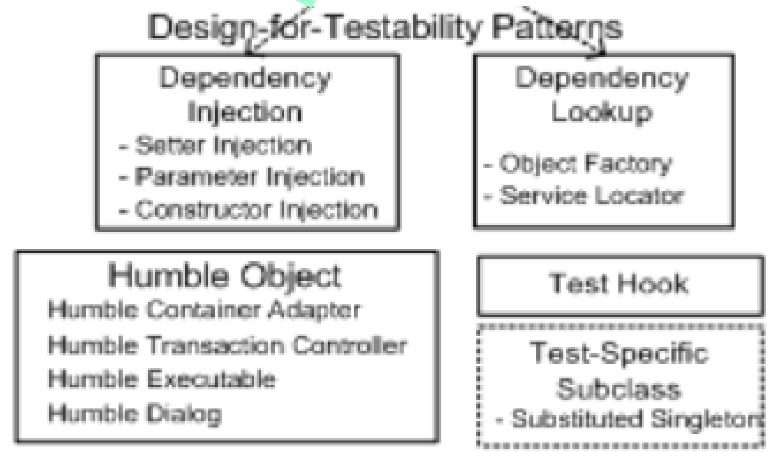
Promote repeatable, reproducible behavior

- Provide utilities to support deterministic behavior (e.g. use seed values)

→ That's just good design practice!

Remember WS1

Design for testability (DfT) patterns



Dependency injection

The client provides the depended-on object to the SUT

Dependency lookup

The SUT asks another object to return the depended-on object before it uses it

Humble object

We extract the logic into a separate easy-to-test component that is decoupled from its environment

Test hook

We modify the SUT to behave differently during the test

Reference: Gerard Meszaros: xUnit Test Patterns: Refactoring Test Code, Addison-Wesley, 2007
http://xunitpatterns.com/

Testability
General scenarios

Source:
Developer,
Tester,
Integrator,
User

Stimulus:
Completion/
integration of
increments/
(sub)systems

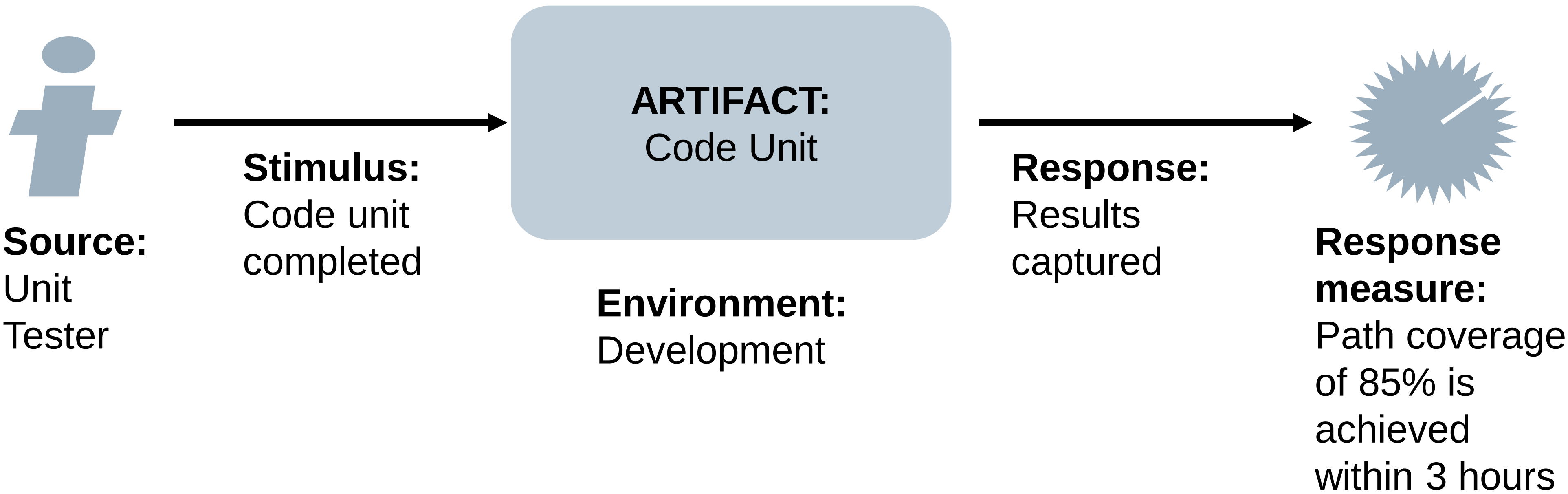
ARTIFACT:
Portion of system
being tested

Environment:
Design time,
Development time,
Compile time,
Integration time,
Deployment time,
Run time

Response:
Results from
execution;
System state
values;
Activity
before fault

Response measure:
Coverage,
Probability,
Effort,
Time

Testability
Sample concrete scenario



Sample testability scenario:
A unit tester performs a unit test on a completed system component
that provides an interface for controlling its behavior and observing its output;
85% path coverage is achieved within three hours.

Len Bass, Paul Clements, Rick Kazman; *Software architecture in practice*, Addison-Wesley

Strategies & Tactics "Testability of SW Systems"

SIEMENS
Ingenuity for life

Testability of SW Systems

Input/execution control

- Provide specialized interfaces to control variable values
- Localize state storage
- Use Capture/Replay
- Support injection

Monitoring

- Logging facilities
- Tracing mechanisms
- Provide specialized interfaces to access internal parts
- Use executable assertions

Reduced complexity

- Limit structural complexity
- Limit non-determinism

Selected Testability Design Patterns

Controllability

- Fault Injector [3]
- Injector [3]
- Interceptor [5]

Observability

- Data Logger [1]
- Interceptor [5]
- Logging Façade [4]
- Monitor [3]
- Poll Monitor [1]

Reduced Complexity

- **Increased Cohesion**
  - Abstract Factory [1] / Object Mother [2]
  - Adapter [1]
  - Bridge [1]
  - Façade [1]
  - Proxy [1]
- **Reduced Coupling**
  - Adapter [1]
  - Bridge [1]
  - External Configuration Store [1]
  - Façade [1]
  - Mediator [1]
  - Model-View-Controller [1, 2]
  - Proxy [1]
- **Separation of Concerns**
  - Layered Architecture [1]
  - Model-View-Controller [1, 2]
  - Transporter [2]

[1] NFR Engineering Repository https://workspace.cee.siemens.com/content/00000102/Wiki

[2] Misha Rybalov, *Design Patterns for Customer Testing*
http://www.autotestguy.com/archives/Design%20Patterns%20for%20Customer%20Testing.pdf

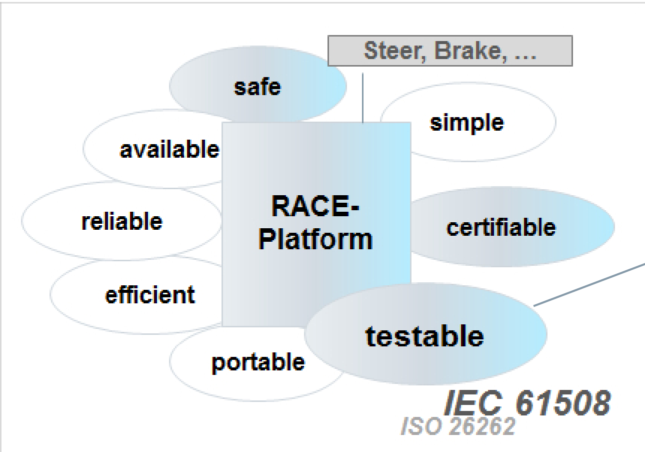
[3] Nelson G. M. Leme, Eliane Martins, Cecília M. F. Rubira, *A Software Fault Injection Pattern System*
http://www.ic.unicamp.br/~eliane/JACA/Jaca-PLoP2001_ngmleme3_3.pdf

[4] *Simple Logging Facade* http://www.slf4j.org/#1085793439

[5] Frank Buschmann et al., *Pattern-Oriented Software Architecture (POSA)*

Example RACE

SIEMENS
Ingenuity for life



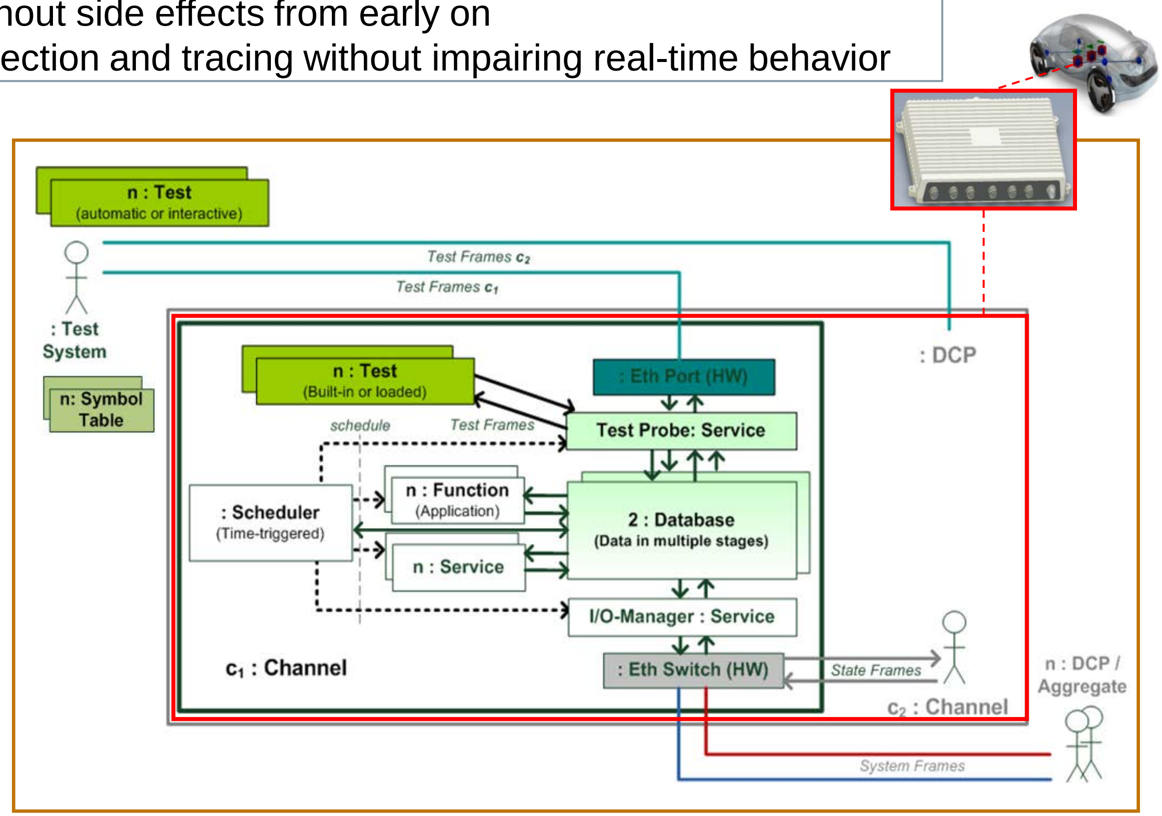
RACE **R**eliable **A**utomation and **C**ontrol **E**nvironment
A high-integrity platform for automotive applications

race

Enabling non-intrusive tests for hard-real-time systems
- Test without side effects from early on
- Fault injection and tracing without impairing real-time behavior

A bunch of coordinated measures providing system testability:
- Time triggered system
- Built-in test probe per channel
- Test probes send heart-beats
- Database per channel
- Test frames fixed in length, frequency

Design patterns:
- Watchdog
- Blackboard
- Fixed Size Buffer



Test Automation Design Patterns

- Test patterns to increase the quality of tests
- Main focus: black-box testing
- Collected by Feudjio and Schieferdecker

Separation of Test Design Concerns	Traceability of Test Objectives to Requirements	One-on-One Test Architecture
▪ Context: Generic organizational ▪ Intent: Proper organization of file structure of test artifacts	▪ Context: Test planning ▪ Intent: Linking test objectives to requirements or features of the SUT	▪ Context: Test architecture ▪ Intent: Test architecture design for sequential non-concurrent behaviour
Prioritization of Test Objectives	**Traceability of Test Objectives to Fault Management**	**Centralized Test Coordinator for Concurrent Test Components**
▪ Context: Test planning ▪ Intent: Prioritization scheme for test objectives w.r.t. resource constraints	▪ Context: Test planning ▪ Intent: Linking fault management system entries and testing process elements	▪ Context: Test architecture ▪ Intent: Test architecture design for parallel/distributed processing

Test Automation Design Patterns for Reactive Software Systems, EuroPLoP2009
http://ceur-ws.org/Vol-566/E6_BlackBoxTesting.pdf

Test Automation Design Patterns

Test Planning	Generic
▪ Prioritization of test objectives ▪ Traceability of requirements to test artifacts (and back) ▪ Traceability of test objectives to fault management	▪ Separation of test design concerns ▪ Grouping of testing concerns ▪ Apply naming convention

Organizational

Technical

Test Data	Test Architecture	Test Behaviour
▪ Purpose-driven test data design ▪ Flexible test data design ▪ Dynamic test data pool	▪ One-on-One test architecture ▪ Proxy test component ▪ Centralized test coordinator	▪ Assertion-driven test behaviour design ▪ Test component factory ▪ Time constraints on test events ▪ Architecture-driven test behaviour design

Test Automation Design Patterns for Reactive Software Systems, EuroPLoP2009
http://ceur-ws.org/Vol-566/E6_BlackBoxTesting.pdf

Unit Test Patterns

Pass/Fail Patterns	Data Driven Test Patterns	Data Transaction Patterns
▪ Simple Test ▪ Code Path ▪ Parameter Range	▪ Simple Test Data ▪ Data Transformation Test	▪ Simple Data I/O ▪ Constraint Data ▪ Rollback
Collection Mgmt Patterns	**Process Patterns**	**Simulation Patterns**
▪ Collection Order ▪ Enumeration ▪ Collection Constraint ▪ Collection Indexing	▪ Process Sequence ▪ Process State ▪ Process Rule	▪ Mock Object ▪ Service Simulation ▪ Bit Error Simulation ▪ Component Simulation
Multithreading Patterns	**Stress Test Patterns**	**Presentation Layer Patterns**
▪ Signaled ▪ Deadlock Resolution	▪ Bulk Data Stress ▪ Resource Stress ▪ Loading	▪ View-State ▪ Model-State

Unit Test Patterns, Mark Clifton http://www.codeproject.com/Articles/5772/Advanced-Unit-Test-Part-V-Unit-Test-Patterns

Dealing with Qualities

Agenda

Quality Attribute Requirements

Design Strategies & Tactics

Design Patterns

Testing Quality Attributes

Summary

Exercise / Discussion
“Performance of Image Processing – Let’s Test”

SIEMENS
Ingenuity for life

We built our quality attribute scenario…

Whenever a user searches for images in the User Interface (using keywords or advanced search criteria), finding and displaying the results should never take longer than 3 seconds.

… derived our utility tree …

- Utility
  - Performance
    - Data Latency
      - (H,M) Storage latency of image archive < 10 msec
    - Throughput
      - (H,H) One image should be loaded per second
      - (M,H) Even complex image searches should return results within 3 sec
    - Perceived Performance
      - (M,L) Image storing or printing should not lead to any perceivable delays for other user activities

Now let’s test this!

- What **test types** are needed?
- Which **test levels** should be used?
- What potential **obstacles** for writing test cases do you see?
- Which additional **input** is necessary? Which **stakeholders** can provide it?
- How can you – as Test Architects – **support** quality attribute testing activities?

… and selected our design strategies and tactics

Performance of Image Processing

Resource demand

- Use advanced image processing algorithms (if possible GPU based)
- Introduce caching strategies
- Either allow upper bound of clients or scale-out mechanisms depending on resources

Resource management

- Introduce a pool of worker threads for background loading, storing, printing
- Don't copy mass data but use meta files and refs for image processing/copying
- Use thumbnail images for browsing
- Only use full resolution when images are selected
- Apply eager loading

Resource arbitration

- Schedule resources preferably for processing visible images

There are various patterns for implementing some of the tactics such as
- Caching
- Lazy Evaluation
- Coordinator
- Eager Loading
- Evictor & Activator
- Half Sync / Half Async
- Command Processor

SIEMENS
Ingenuity for life

Did you achieve the quality attributes?

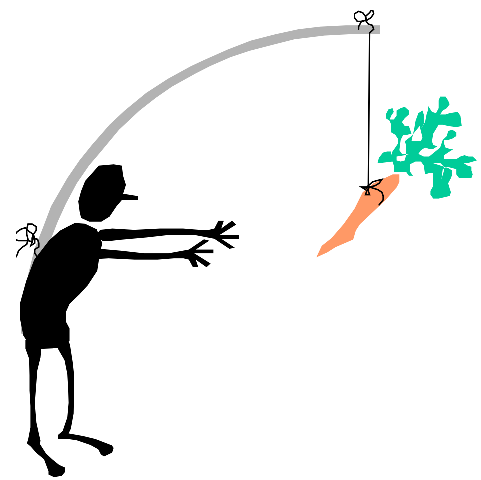
How can you know that you achieved the quality?

By testing – testing provides information

What do you need to be able to evaluate and assess?

Precisely and quantitatively described attributes

But what is a precisely described attribute?



Testable quality attributes

Functional testing *tests* what a product/system does

Non-functional testing *tests* how well a product/system operates

- Quality attributes shall at least satisfy two characteristics
  - Quality attributes must be specified objective
  - Quality attributes must be specified testable (measurable)
- Making requirements measurable
  - Define ‘fit criteria’ for each requirement
  - Give the ‘fit criteria’ alongside the requirement
- ISO/IEC 25010 classifies the quality attributes in scope in a structured set of characteristics and sub-characteristics and provides metrics to make the attributes measurable.

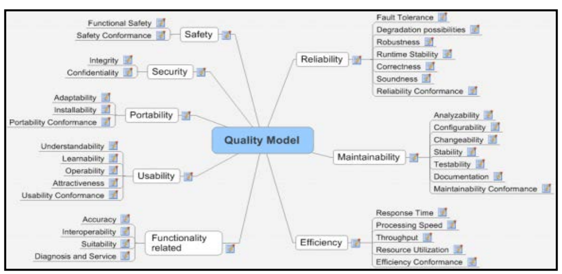
What do you really mean by …???
***Understanding* is the first step for testing NFRs …**

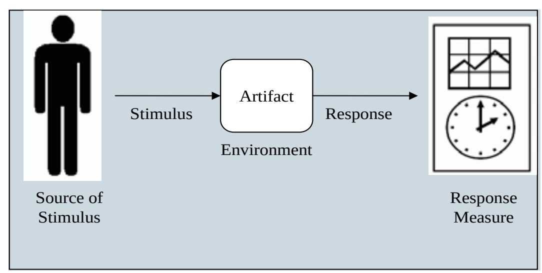
Differentiate and clarify the understanding of terms

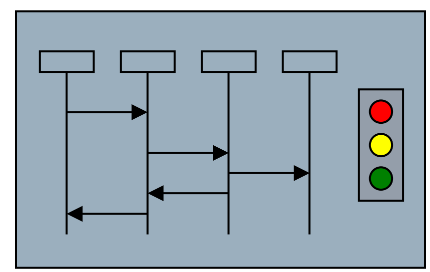
- Availability ≠ Reliability
- Reliability ≠ Robustness ≠ Fault tolerance
- Stability?
- Performance ≠ Scalability

It starts with choosing the right set of metrics and consistently measure against those metrics by picking the right operations and usage scenarios (operational / user profiles) …

- Quality models (quality trees) provide business and user needs as well as priorities
- Quality attribute scenarios (design tactics) for test case design







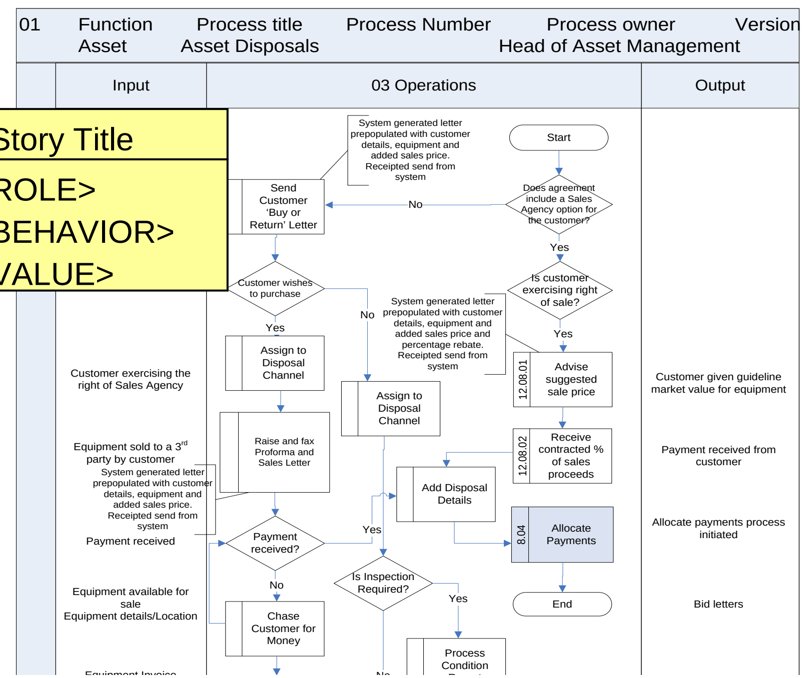
Test basis

- **All kind of specifications (formal, semi-formal, informal)**
  - UML, SysML, domain specific languages (DSL)
  - Especially interesting: dynamic behavior
  - Functional requirements
  - Non-functional requirements (performance, reliability, etc.) }

You have to test for these magic things!!!

- **Many other sources**
  - Standards, norms
  - User manual, online help, UI prototypes
  - Design guidelines, interface guidelines
  - Workflow models, business rules
  - Use cases, scenarios, epics
  - User stories, story boards, story maps
  - Data models, data use descriptions
  - The system under test or code itself
    → what we know or see during testing
  - Any heuristics / experience, ask stakeholders …

Story #	Story Title
As a	<ROLE>
I want	<BEHAVIOR>
so that	<VALUE>

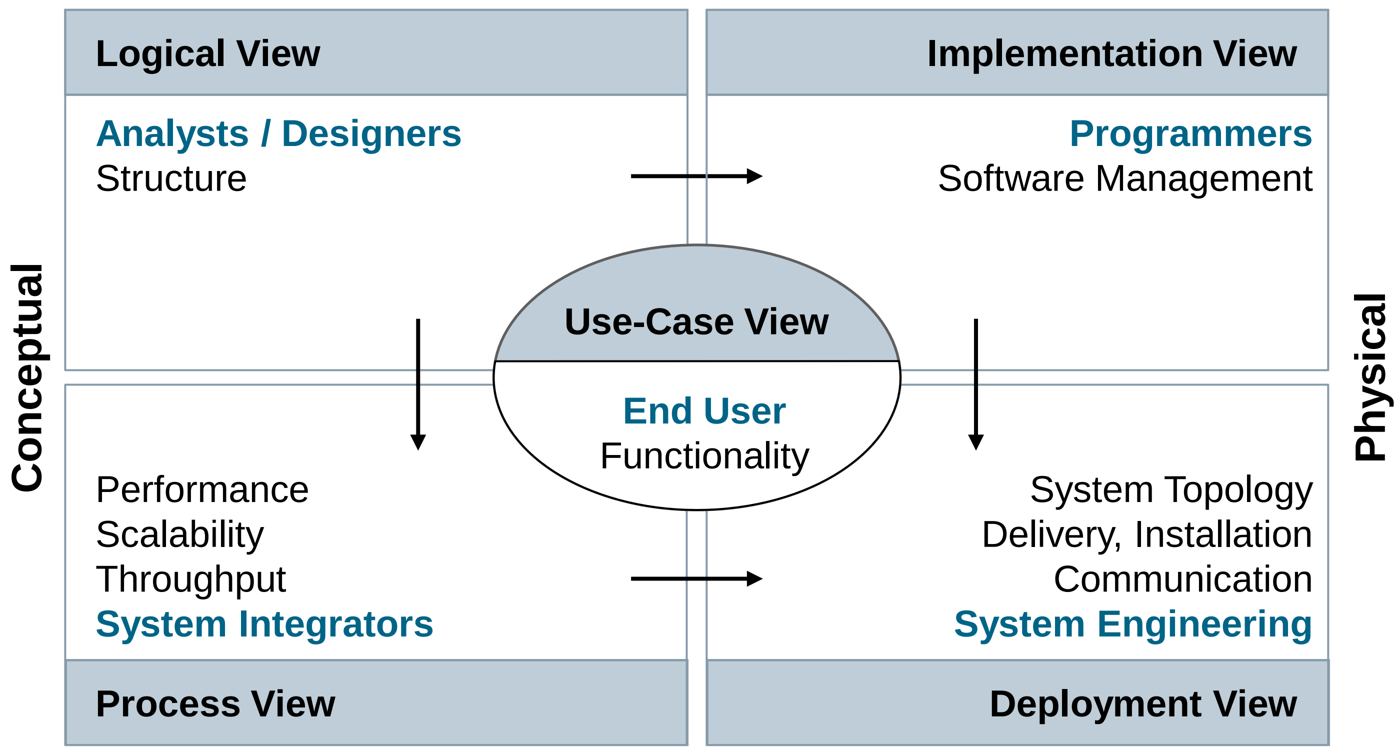


- **ISO/IEC/IEEE 29119-1**
- **Test basis** – body of knowledge used as the basis for the design of tests and test cases

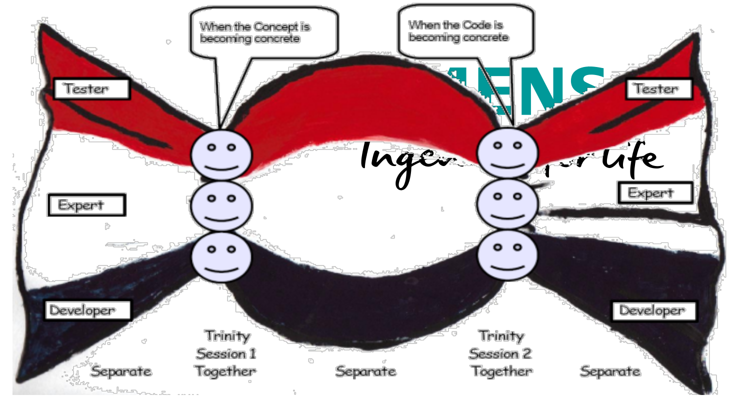
SIEMENS
Ingenuity for life

Test basis for integration testing – Example

- **S**oftware **A**rchitecture **D**escription document
- **4+1 View Model of Software Architecture by Philippe Kruchten**




SAD_PRODUCTNAME.doc



Collaborative work for a better test basis

The job of the testers is

- to *actively* review the specifications
- to *actively* participate early in the creation and to ask questions: **J**oint **A**pplication **D**evelopment, **J**oint **R**equirement **P**lanning, **Feature Teams** @ Microsoft, **Trinity Testing** @ Google, **A-TDD**, **Power of Three** @ agile testing, **3 Amigos** @ SbE, **Shift Left**
- to enhance the specifications for testability issues
- to improve the specifications concerning quality issues (prevention)

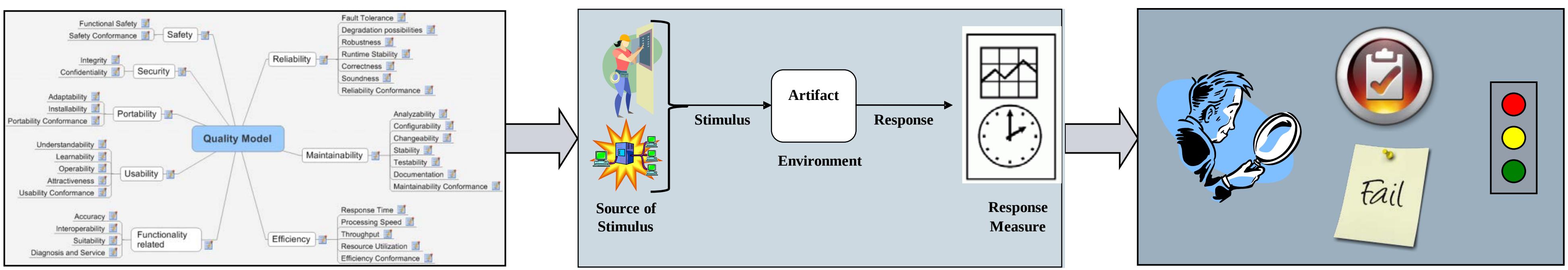
Focus on *using* requirements not only on *perceiving* requirements

- Reviews are often too passive – requirements are only augmented but not questioned
- ***Quality is a result of usage***
- Describing / Specifying a test (even better: more tests …) for a requirement ~~will help you~~ **is a precondition** to really understand the requirement

Testing quality attributes Approach

SIEMENS
Ingenuity for life

Operational Level	Operational Quality Requirements	Customer Acceptance Level / Criteria	based on	Acceptance, Field Test
System Level	Utility Tree	Quality Facets Scenarios Risks	based on	Scenario Test (Black Box)
Architecture Level	Design Strategies & Tactics	Arch. Patterns Building Blocks Interactions Deployment	based on	Integration Test (Grey Box)



Goals and inputs for quality attribute tests

	Testing Goals	Testing Inputs	Examples
Operational Level	Validate that the system ▪ allows the user to achieve his goals ▪ with all explicitly required and implicitly expected qualities ▪ in any relevant execution environment	▪ Stakeholder Requests ▪ Load Profiles ▪ Threat Profiles ▪ Environment Profiles ▪ SLAs	▪ Perform a certain number of concurrent reading workflows within a given time period on different system deployments. ▪ Check reliability of the system during field test
System Level	Check quality / correctness of individual ▪ System functions ▪ Building blocks ▪ Features under defined conditions	▪ Quality Facets ▪ Scenarios ▪ Risks ▪ External Interfaces	▪ Execution time ▪ Latency / Throughput ▪ User Authentication ▪ Friendly Hacking
Architecture Level	Validate the ▪ correct implementation of chosen design tactics ▪ by checking the system's building blocks ▪ against the architectural specification	▪ Design Strategies ▪ Design Tactics ▪ Realization concept ▪ Architecture specification (building blocks, interactions, states, deployment, …)	▪ Component interactions (protocols) ▪ Message encryption ▪ Segregation ▪ Data formats

Selected quality attribute testing approaches

Security

- Fuzz Testing
- Vulnerability assessment
- Vulnerability testing
- Penetration test
- Security audit
- Security review
- Network scanning

Performance

- Load testing
- Stress testing
- Spike testing
- Endurance/Soak testing
- Benchmarking

Conformance

- Protocol tests
- Radiated immunity testing
- Radiated emissions testing
- Conformity assessment

Safety

- Hazard and operability analysis/study (HAZOP)
- Defect history tracking
- Statistical testing
- Probabilistic risk assessment (PRA)
- Failure mode and effect analyses (FMEA)
- Fault tree Analysis (FTA)
- Reliability Testing

Usability

- Hallway testing
- Remote usability testing
- (Automated) Expert review

Others

- Recovery testing
- Volume testing

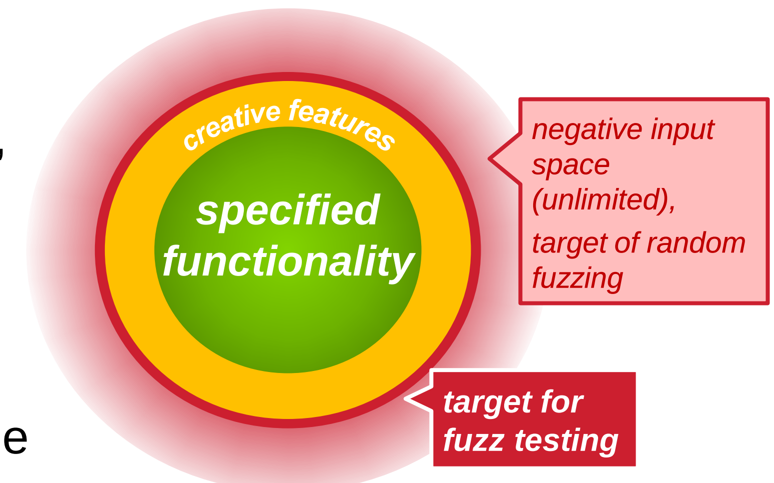
Additionally – on Architecture Level many test design techniques can be used, too

Security testing
Fuzz testing approach

- Fuzzing originally describe the generation of randomly generated test vectors (Miller et. al. in the early 1990s)
- **Random fuzzing:** has close to zero awareness of the tested interface
- **Mutation-based fuzzing:** mutate existing data samples to create test data, breaks the syntax of the tested interface into blocks of data, which it semi-randomly mutates

Model-based fuzzing

- uses models of the input domain (protocol models, e.g. context free grammars), for generating systematic non-random test cases
- in security testing purposes, the models are augmented with intelligent and optimized anomalies that will trigger the vulnerabilities in code
- finds defects which human testers would fail to find



See also: [TAK08]

Security testing
Categorization of fuzzers

- **Random-based fuzzers** generate randomly input data. They don't know anything about the SUT's protocol.

  fuzzed input: **HdmxH&k dd#**&%**

- **Template-based fuzzers** uses existing traces (files, …) and fuzzes some data.

  template: `GET /index.html`

  fuzzed input: `GE? /index.html`, `GET /inde?.html`

- **Block-based fuzzers**

  break individual protocol messages down in static (grey) and variable (white) parts and fuzz only the variable part.

GET	/index.html

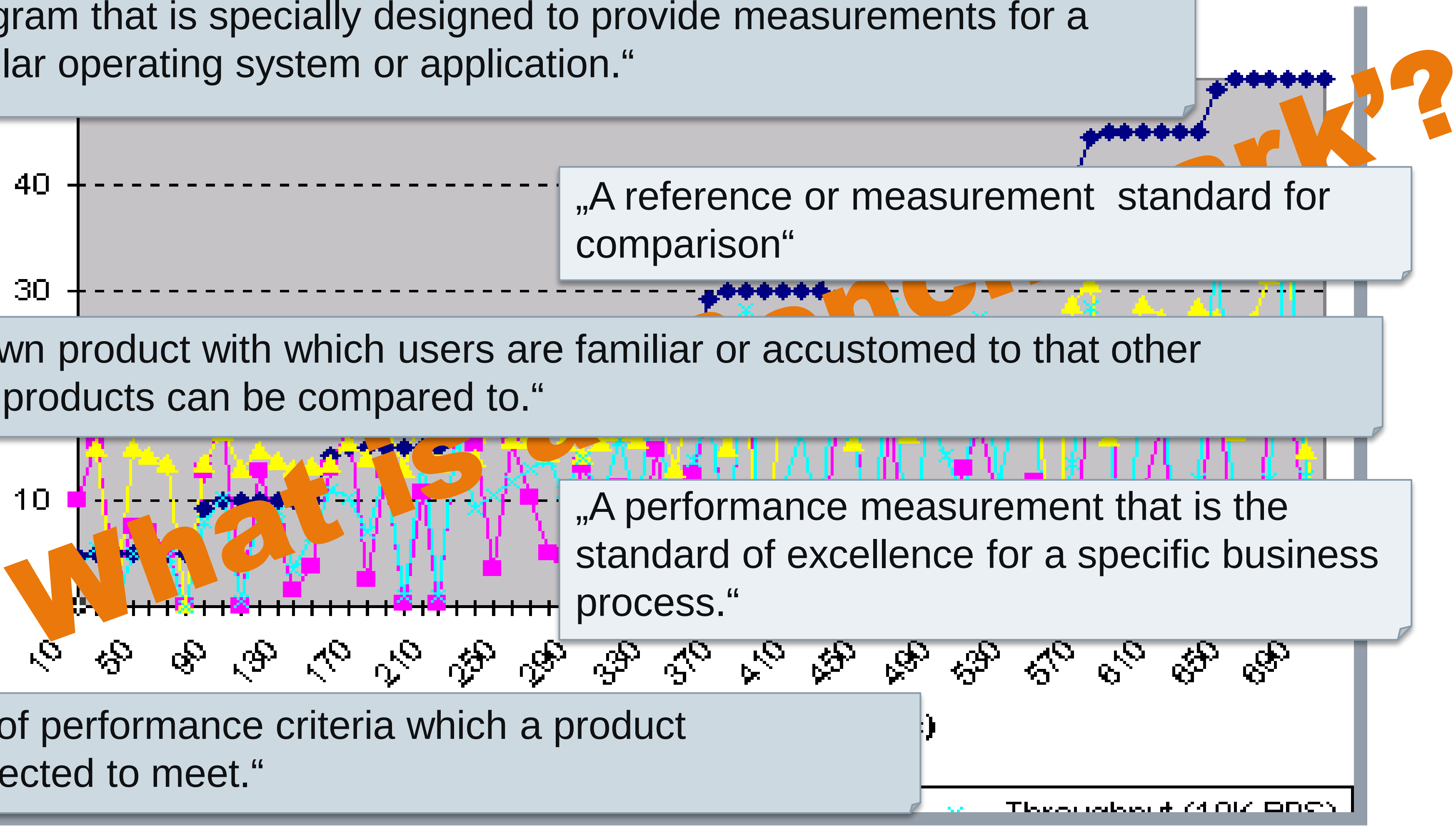
  only the (white) part gets fuzzed

  fuzzed input: `GET /inde?.html, GET /index.&%ml`

- **Dynamic Generation/Evolution-based fuzzers** learn the protocol of the SUT from feeding the SUT with data and interpreting its responses, for example using evolutionary algorithms. While learning the protocol these fuzzers run implicit fuzzing.

Performance testing
Benchmarking approach

SIEMENS
Ingenuity for life



„A program that is specially designed to provide measurements for a particular operating system or application.“

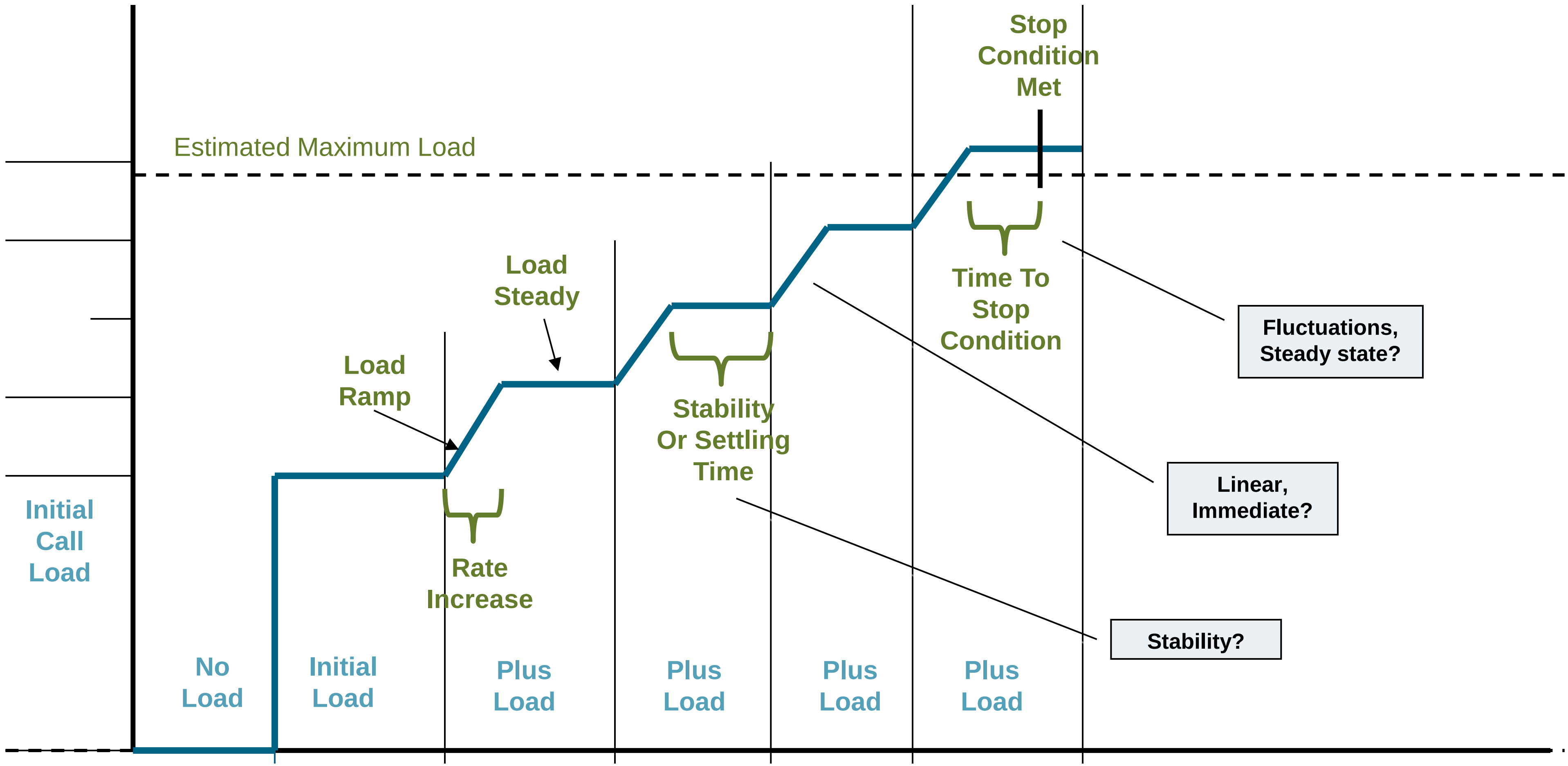
„A reference or measurement standard for comparison“

„A known product with which users are familiar or accustomed to that other newer products can be compared to.“

„A performance measurement that is the standard of excellence for a specific business process.“

„A set of performance criteria which a product is expected to meet.“

Performance testing
Benchmarking procedure



[DIN09]

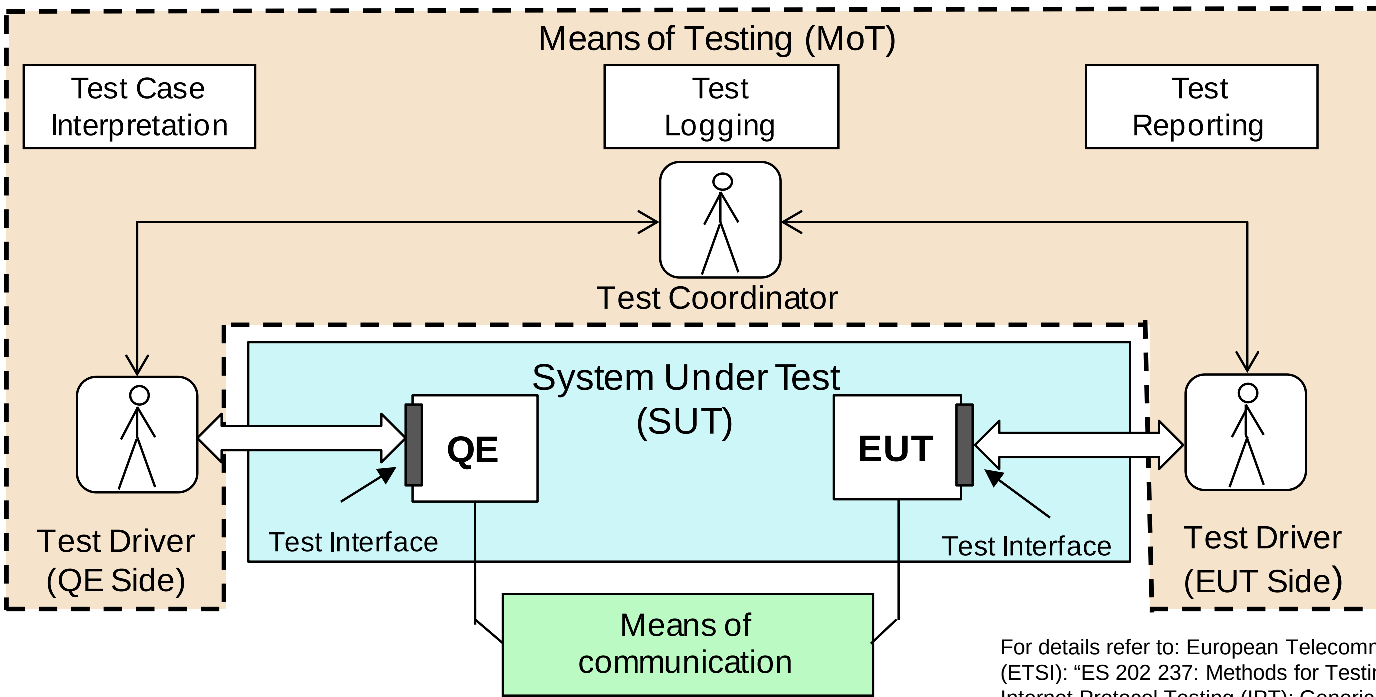
Interoperability testing approaches

Interoperability testing can be conducted by…

- Using reference implementations of some components and test drivers associated to all components
- Using monitoring and proxy components
- Replacing some components of the system with test simulators

Interoperability testing with GAIT

- Generic Approach to Interoperability Testing (GAIT) v1.2.1 (2010)
- Defined for software interoperability testing, without focusing on a specific software domain.
- QE – Qualified Equipment



For details refer to: European Telecommunications Standards Institute (ETSI): “ES 202 237: Methods for Testing and Specification (MTS); Internet Protocol Testing (IPT); Generic Approach to Interoperability Testing”, version 1.2.1, 2010.

Interoperability testing
Classification of IOP checking levels

Interaction Scenario

- **what**: the required sequence of messages is validated
- **how**: timeout events indicate that some messages are not sent according to the specified sequence

Message Type

- **what**: check that IOP *message structure profile* constraints are fulfilled
- **how**: type checking upon receipt of a message from SUT according to IHE IOP integration profiles

Fields Conditionality

- **what**: the conditionally constraints across the fields within the same message is validated (e.g., if field1 is present, then field2 must be also present)
- **how**: special checking functions are used

Message Content

- **what**: the content of messages is inspected against expected values, code sets (tables), values imposed by standards, etc.
- **how**: message content checking functions

Semantic Correlations

- **what**: correlation of pieces of information *across different messages* within the flow has to be verified (e.g., for updating a patient, a Patient ID used in a previous step has to be used)
- **how**: using message tuning functions with semantic parameters or using semantic checking functions

Final recommendations for good quality testing

Always keep in mind

- **Address each Quality on multiple test levels!**
- **Derive your test cases from well-formulated, measurable qualities!**
- **Use scenarios as input for test case design!**
- **Resolve conflicts between qualities early!**
- **Never forget that there are many implicit quality expectations, too!**
- **Take all relevant environmental conditions into account!**
  - Deployment, network (topology), concurrent user activities, …

Dealing with Qualities

Agenda

Quality Attribute Requirements

Design Strategies & Tactics

Design Patterns

Testing Quality Attributes

Summary

What we have learned

You need precisely described and testable (measurable) quality scenarios.

Utility trees can be used for structuring and detailing quality attributes/scenarios.

You know possible tactics considering testability.

You know design patterns to implement testability tactics.

You know how to test quality attributes.

Departing thought

Photograph by NASA

We cut down to six ounces (of water) each per day, a fifth of normal intake, and used fruit juices; we ate hot dogs and other wet-pack foods when we ate at all.

[Apollo 13 Commander James A. Lovell]

SIEMENS
Ingenuity for life

Further readings

Use the SSA Wiki :
https://wiki.ct.siemens.de/x/fReTBQ

and check the “Reading recommendations”:
https://wiki.ct.siemens.de/x/-pRgBg

- Architect's Resources:
  - Competence related content
  - Technology related content
  - Design Essays
  - Collection of How-To articles
  - Tools and Templates
  - Reading recommendations
  - Job Profiles for architects
  - External Trainings
  - ... more resources