Architecture of Distributed Systems

Quality Attributes 2019-2020

R. H. Mak

Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

Goals

- **Identify quality attributes (QA)**
- **Understand relation between QAs and architecture**
- **Learn how to express QAs in scenarios**
- **Learn how to achieve QAs through tactics, specifically**
  - **fault tolerance tactics for availability/reliability**
  - **tactics for modifiability**

Slides are based on Chapters 4, 5 & 7 from the book
"Software architectures in practice" (third edition)
Len Bass, Paul Clements and Rick Kazman

Also parts of Chapter 8 on Fault tolerance from vST

Motivation

- Systems are developed for a certain purpose, to satisfy *functional requirements*
  - Remember architectural context model (ISO/IEC/IEEE 42010)
- Main problems with, and reasons to redesign software and systems:
  - *Lack of functionality: new features needed*
    - *requires extensibility, evolvability*
  - *Difficult to maintain, port, scale*
  - *Lack of performance: too slow, too large resource footprint*
  - *Lack of dependability: maintainability, reliability, availability, safety, integrity*
  - *Lack of security: confidentiality, integrity, availability*
- Besides functional requirements, a good understanding of quality requirements of the system is essential
  - Quote from Eltjo Poort: " For functionality almost any architecture will do, but for quality requirements finding the right architecture is paramount."
  - 'quality use cases' to describe quality requirements
  - leads to a "design for ...."-approach
    - design for test, design for maintenance, design for security, design for extensibility

QAs: some definitions

Recall from lecture 1, there are many! A small selection:

- *Availability*
  - readiness for correct service (expressed as a probability)
- *Reliability*
  - continuity of correct service (expressed as a period of time)
- *Modifiability*
  - about how difficult it is to introduce desired changes
- *Performance*
  - timely response to service request events, throughput, jitter
- *Testability*
  - how difficult it is to verify the correctness of the system
- *Usability*
  - how user friendly the system is
- *Scalability*
  - various definitions exist, (see separate lecture)
- *Portability*
  - how difficult it is to make the system run on another platform

QAs: some groupings

Some qualities that are defined as an aggregate of other qualities, e.g.

Dependability

- the ability to deliver service that can justifiably be trusted
- dependability attributes
  - Availability
  - Reliability
  - Safety: absence of catastrophic consequences on users and environment
  - Integrity: absence of improper state alterations
  - Maintainability: ability to undergo repairs (and modifications), e.g. automatic recovery

Security

- the ability to resist unauthorized attempts to access data and services
- security attributes
  - Availability: for authorized actions only
  - Integrity: absence of unauthorized state alterations
  - Confidentiality: absence of unauthorized disclosure of information

QAs in design process

Types of QAs

- Qualities of the developed system (- our focus)
  - e.g. availability, modifiability, performance, security, testability, usability, ...
- Qualities of the architecture itself
  - e.g. conceptual integrity, consistency, relative completeness, buildability
- Qualities of development process
  - coding, testing, documentation & integration procedures, code reviews..
- Business qualities (mapped to system qualities)
  - e.g. time to market, cost/benefit

QAs in design process

Management of QAs

- **Scenario specification** (stating QA requirements)
  - define what is desired response in particular situation
  - specification details:
    - preconditions,
    - given inputs,
    - required outputs
- **Tactics definition** (achieving QAs)
  - how to enforce required quality
    - for given preconditions and inputs
  - tactics becomes part of the designed system

QAs in development process

- Every stage in development process further constrains space for QAs
- Architecture is on top level and thus most critical
- Architecture by itself cannot provide QAs
- QAs are achieved only by combination of:
  - architecture (big picture: patterns + tactics) and
  - non-architectural (details) choices, e.g. :

QA	*Architectural level*	*Design & implementation*
Usability of UI	*redo/undo support*	*GUI layout & coding*
Modifiability	*separation of concerns* *e.g. independent modules*	*coding techniques*
Performance	*usage of shared resources* *communication infrastructure*	*algorithms used* *optimization techniques*

Practical aspects of QAs

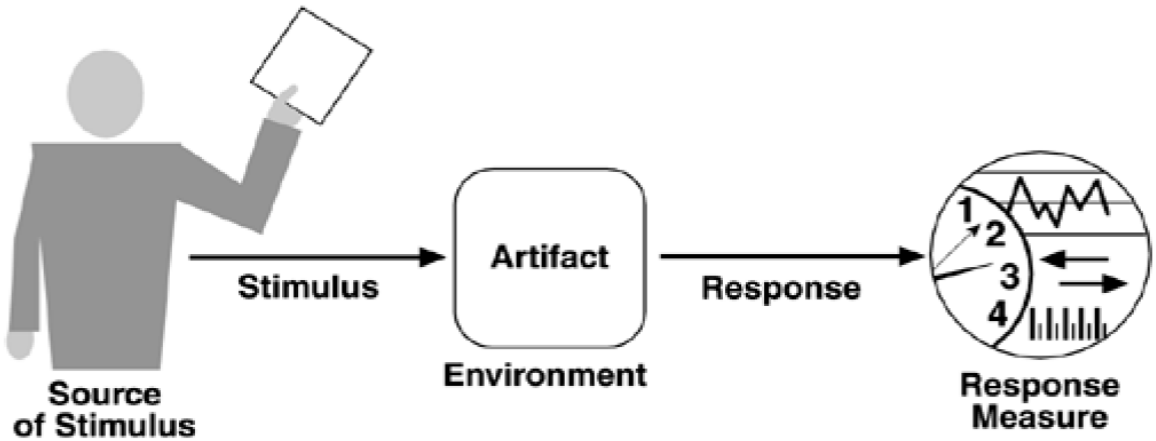
- Dependencies exist between QAs
  - Improvement of most QAs negatively influences performance
    - e.g., isolating dependencies to increase portability or
    - introducing redundancy to improve reliability
  - Replicated data needs to be (or eventually become) consistent
- Quality attributes must be dealt with in all phases of development:
  - The choices of an architect set the foundation for quality attributes
  - But to achieve qualities proper implementation is needed
- Limited man power and time-to-market results in a trade-off
  - Requires Pareto analysis
    - regard a design as a point in an multidimensional space
    - QAs + Functionality as dimensions (degrees of freedom)
- Not all QAs are equally important in particular system
  - It is important to prioritize them,
  - E.g. in safety critical systems, reliability is a key QA

Brewer's CAP-theorem

Example of QA dependencies and subsequent prioritization.

- The CAP theorem states that any networked shared-data system can have at most two of the following three desirable properties:
  - consistency (C) equivalent to having a single up-to-date copy of the data;
  - high availability (A) of that data (for updates); and
  - tolerance to network partitions (P).
- Some recent data stores, such as Amazon's Dynamo, sacrifice consistency for high availability
  - See Werner Vogels' blog
    - All Things Distributed: Eventually Consistent – Revisited
- Facebook, on the other hand, sacrifices availability
  - See Jeff Johnson's Qcon 2014 presentation
    - How Facebook Scales Big Data Systems

QA scenarios (specifying requirements)



Goal: specify required outputs for a particular relevant situation

inputs:
- stimulus
- source of stimulus

required outputs:
- response
- response measure

preconditions:
- environment of the scenario (state of system)
- resources (artifacts) that are subject of the QA

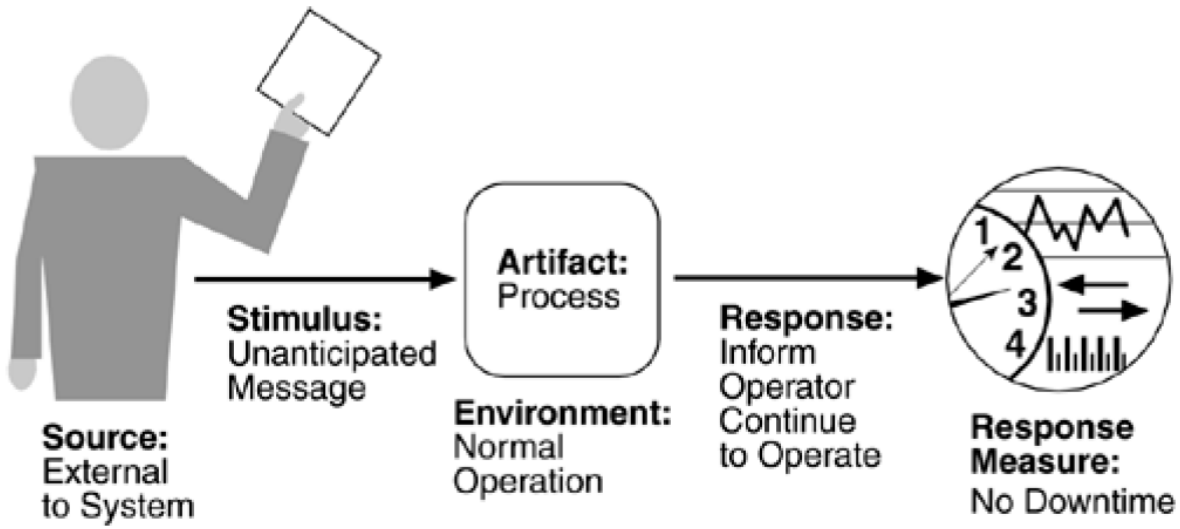
QA scenarios (some remarks)

- We distinguish between *general scenarios* that are QA-specific, but hold for any system, and *specific scenarios* that hold for a particular system (the one to who's AD they belong)
  - A case of template (type) versus instance
  - For any system, there can be many specific scenarios per QA
- Models may be needed to capture the stimuli
  - Fault models for availability
  - Attack models for security
  - Resource models for performance
- For quantifiable responses, metrics need to be defined to specify the desired level / intensity of the response
  - Mind the units, e.g., throughput in Mb/sec
    - accompanied by bandwidth also in Mb/sec in the precondition
  - The more quantitative, the better

Generic availability scenario

- *Source*
  - system external / system internal (other subsystem)
- *Stimulus*
  - fault: in external input, failure in subsystem, e.g. omission / crash / timing failure / wrong input
  - exception raised when fault is detected
- *State of system (environment)*
  - for example: normal / degraded operation, overloaded system
- *Affected resources (artifacts)*
  - process, processor, server, storage, communication… or entire system
- *Response & tactics*
  - prevent, detect, recover: log, notify, disable, isolate, continue (normal/degraded), repair, replicate, ...
- *Response measure*
  - repair time, availability, available/degraded time interval...

Specific availability scenario



inputs:
- stimulus: unanticipated message
- source: external

required outputs:
- response: continue
- response measure: no downtime

preconditions:
- state of system: normal operation
- affected resource: process that handles messages

Assumptions and models

- ‘Assumptions about the stimuli
  - consider all behaviors of the inputs to a component, i.e., behaviors of its environment
    - typical loads, (mis)handling, attacks, ...
    - include those considered extreme, erroneous
  - for a component these can be
    - both internal (from within the system)
    - and external (from outside the system)
  - the architecture needs a model of the environment of a component
    - RW: context model for the entire system

- Each QA has its own model aspects, and artifacts to which the QA pertains
  - failure model (reliability, internal /external components),
  - attack model (security, external/internal?),
  - load or usage model (performance, internal /external), ...

Assumptions and models

- An architecture addresses QA's only to the extent to which they are incorporated in the given models
  - touches upon the completeness of the system specification
    - in fact, the specification has been extended, by the QA's, to be 'more total'
  - rationales for entities, qualities, models and decision also help to make explicit what is left out (abstracted from)
  - mind the 'Black Swan', disruptive dependence on unexpected behavior
    - N.N. Taleb, *The Black Swan – the impact of the highly improbable*, Random House, New York, 2007.

example of
a link failure

https://www.youtube.com/watch?v=VVJlKJi9FWU

QA metrics: availability/reliability

- *Availability*
  - Readiness for correct service (expressed as a probability)
- *Reliability*
  - Continuity of correct service (expressed as the expected (mean) time to failure)

Question: can a system be

– highly available but unreliable?
– highly reliable but unavailable?

Availability is concerned with system's *failure* to provide specified services.

– failure is observable by the service user
  - another system or human operator
– availability metric: MTTF / (MTTF + MTTR)
  - MTTR: time to repair is the time till the failure becomes not observable
  - MTTF: mean time to failure (= reliability)
  - MTBF = MTTF + MTTR: mean time between failures

Tactics definition (achieving QAs)

Goal: design tactics that guarantee required outputs for given inputs

inputs that originate from system:
- environment (state of system)
- affected resources (artifacts)

inputs that originate from stimulus:
- stimulus
- source of stimulus

Tactics

guaranteed outputs:
- response
- response measure

Availability - example of QA tactics

inputs that originate from stimulus:
- stimulus: unanticipated message
- source : end-user

inputs that originate from system:
- state: normal operation
- artifact: process handling messages

discard the message

guaranteed outputs:
- response: continue
- response measure: no downtime

Example:

- Availability and reliability of storage
  - failure model (assumptions):
    - crash of at most one disk
      - probability dependent on #transactions performed
      - which depends on the usage (external)
    - disk crash can be observed
  - solution:
    - replicate disk
  - what if the disk could produce faulty information?
    - use majority vote and three disks
    - examine RAID solutions (levels 1--5)

- Maintainability
  - a model of expected changes, and repair scenarios
    - i.e., corrective maintenance in contrast to proactive (preventive) maintenance

Scenario: Highly available file-i/o

- **Stimulus:**
  - invocation of i/o operations at a certain rate over an extended period of time
- **Source of stimulus:**
  - OS, on behalf of user programs
- **Environment:**
  - normal operation
- **Resources:**
  - storage infrastructure: disks, disk controllers, …
- **Response:**
  - Correct and timely execution of the bulk of these operations
- **Response measure:**
  - *MTTF, MTTF/MTBF* (≥ 0.999)

(as in SLA)

Alternative: Take a crash as the stimulus and uninterrupted file i/o as response. Details DIY.

Tactic: Replicate disk (one spare)

Consists of a number of stimulus-response pairs

- Stimulus: write operation,
- System: normal, both disks operational
- Response: write on both disks (primary and spare)

- Stimulus: read operation
- System: normal, both disks operational
- Response: retrieve data from primary

- Stimulus: i/o-operation
- System: primary crash
- Response: perform operation on spare without delay;
  - replace crashed disk and bring it in a consistent state

Fault tolerance

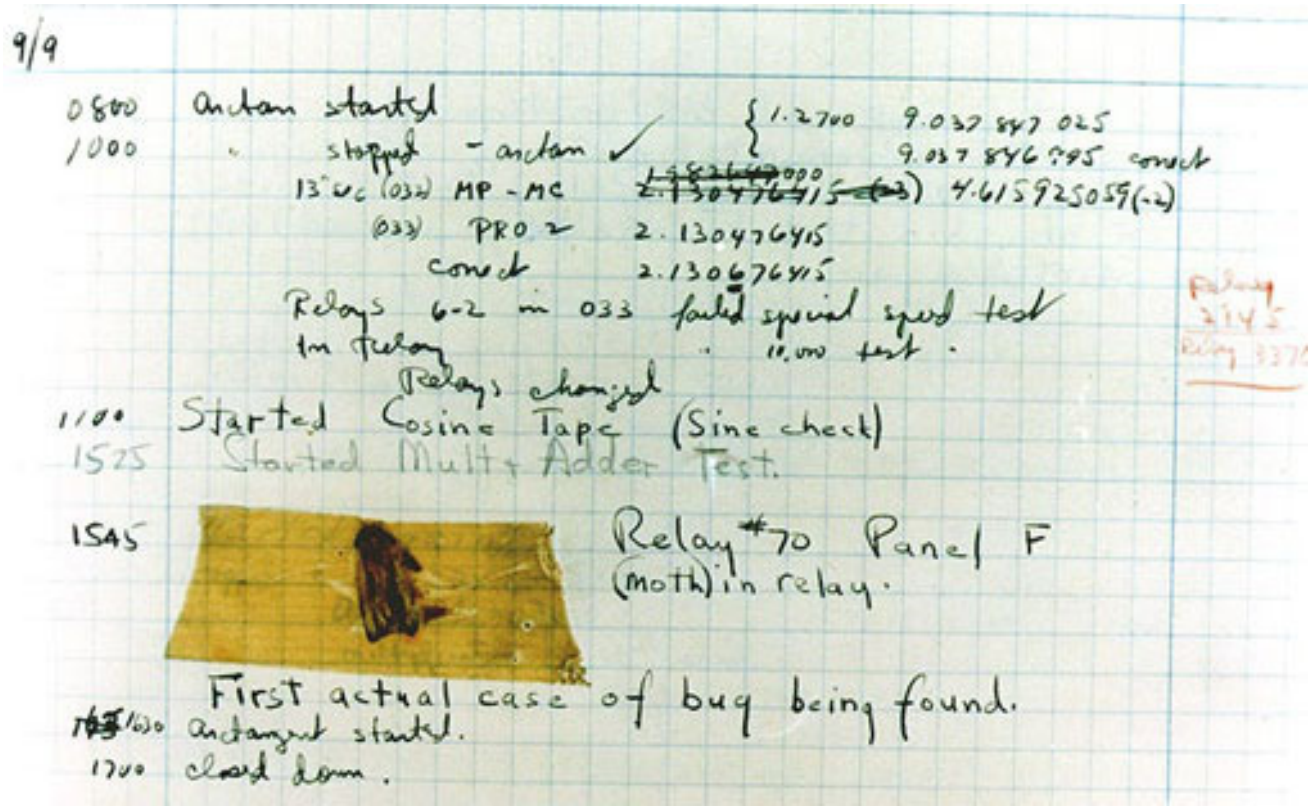
- **Fault tolerance is a quality attribute of a system**
  - the ability of a system to deal with faults
  - the extent to which a system behaves correctly, i.e. according to specification, in the presence of faults
    - graceful degradation of service
- **Making a system fault tolerance is crucial to obtain dependable systems**
- **Faults may be accidental or by malicious intent**
- **Faults may originate from within the system (internal) or from its environment (external)**
  - in the first case, often in the shape of “software bugs”

Fault, errors and failures

Fault tolerance is about preventing faults to become (system) failures

- **Failure: not meeting the specification**
  - externally observable deviation from expected correct behaviour, i.e., the service to be delivered at the agreed level
- **Error: system state that may lead to failure**
  - need not be externally observable
  - system may recover before leading to failure
- **Fault: cause of an error**
  - can be the failure of another (sub)system

The first bug report



Page from the Harvard Mark II electromechanical computer's log

Some serious software faults

Consequences of unreliable software can severe. See

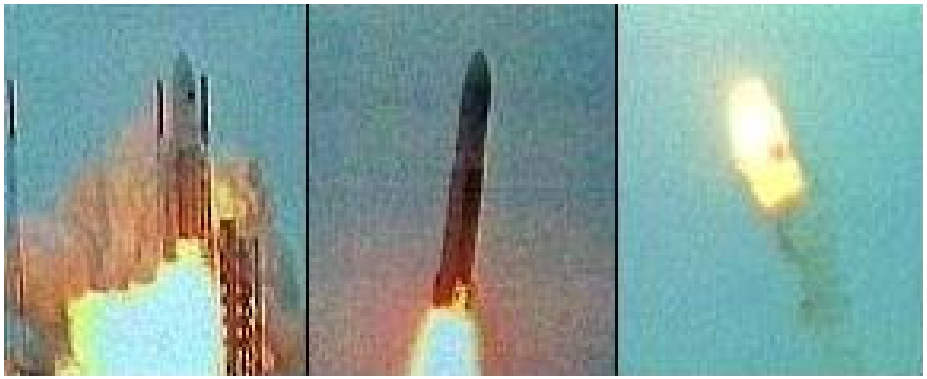
1. http://royal.pingdom.com/2009/03/19/10-historical-software-bugs-with-extreme-consequences/
2. http://www.intertech.com/Blog/15-worst-computer-software-blunders/
3. http://www.scientificamerican.com/article/pogue-5-most-embarrassing-software-bugs-in-history/
4. http://www.computerworld.com/article/2515483/enterprise-applications/epic-failures--11-infamous-software-bugs.html/

Examples in these lists overlap. List 4 contains references to original failure reports

Ariane 5, flight 501

- Disintegrated in air
- June 4, 1996, 32 seconds after lift-off from Kourou, French Guiana
- 10y development, $10 billion
- Cargo+rocket: $0.5 billion

- Failure report made up by inquiry board chaired by prof. J.L. Lions

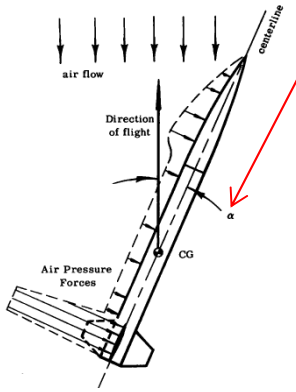


See also: http://www.esa.int/esapub/bulletin/bullet89/dalma89.htm
And http://spaceflightnow.com/ariane/va219/launchtimeline.html
for a proper launch timeline

From inquiry report ...

- The launcher started to disintegrate at about H0 + 39 seconds because of high aerodynamic loads due to an angle of attack of more than 20 degrees that led to separation of the boosters from the main stage, in turn triggering the self-destruct system of the launcher.

- This angle of attack was caused by full nozzle deflections of the solid boosters and the Vulcan main engine.



- These nozzle deflections were commanded by the On-Board Computer (OBC) software on the basis of data transmitted by the active Inertial Reference System (SRI 2). Part of these data at that time did not contain proper flight data, but showed a diagnostic bit pattern of the computer of the SRI 2, which was interpreted as flight data.

Cnt'd

- The reason why the active SRI 2 did not send correct attitude data was that the unit had declared a failure due to a software exception.
- The OBC could not switch to the back-up SRI 1 because that unit had already ceased to function during the previous data cycle (72 milliseconds period) for the same reason as SRI 2.
- The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This resulted in an Operand Error. The data conversion instructions (in Ada code) were not protected from causing an Operand Error, although other conversions of comparable variables in the same place in the code were protected.

Cnt'd

- The error occurred in a part of the software that only performs alignment of the strap-down inertial platform. This software module computes meaningful results only before lift-off. As soon as the launcher lifts off, this function serves no purpose.

- The alignment function is operative for 50 seconds after starting of the Flight Mode of the SRIs which occurs at H0 - 3 seconds for Ariane 5. Consequently, when lift-off occurs, the function continues for approx. 40 seconds of flight. This time sequence is based on a requirement of Ariane 4 and is not required for Ariane 5.

- The Operand Error occurred due to an unexpected high value of an internal alignment function result called BH, Horizontal Bias, related to the horizontal velocity sensed by the platform. This value is calculated as an indicator for alignment precision over time.

- The value of BH was much higher than expected because the early part of the trajectory of Ariane 5 differs from that of Ariane 4 and results in considerably higher horizontal velocity values.

Recommendations by the inquiry board

- **R1** Switch off the alignment function of the inertial reference system immediately after lift-off. More generally, no software function should run during flight unless it is needed.

- **R2** Prepare a test facility including as much real equipment as technically feasible, inject realistic input data, and perform complete, closed-loop, system testing. Complete simulations must take place before any mission. A high test coverage has to be obtained.

- **R3** Do not allow any sensor, such as the inertial reference system, to stop sending best effort data.

Recommendations: cntd

- **R4** Organize, for each item of equipment incorporating software, a specific software qualification review. The *Industrial Architect* shall take part in these reviews and report on complete system testing performed with the equipment. All restrictions on use of the equipment shall be made explicit for the Review Board. Make all critical software a Configuration Controlled Item (CCI).

- **R5** Review all flight software (including embedded software), and in particular :
  - Identify all implicit assumptions made by the code and its justification documents on the values of quantities provided by the equipment. Check these assumptions against the restrictions on use of the equipment.
  - Verify the range of values taken by any internal or communication variables in the software.
  - Solutions to potential problems in the on-board computer software, paying particular attention to on-board computer switch over, shall be proposed by the project team and reviewed by a group of external experts, who shall report to the on-board computer Qualification Board.

Recommendations: cntd

- **R6** Wherever technically feasible, consider confining exceptions to tasks and devise backup capabilities.
- **R7** Provide more data to the telemetry upon failure of any component, so that recovering equipment will be less essential.
- **R8** Reconsider the definition of critical components, taking failures of software origin into account (particularly single point failures).
- **R9** Include external (to the project) participants when reviewing specifications, code and justification documents. Make sure that these reviews consider the substance of arguments, rather than check that verifications have been made.
- **R10** Include trajectory data in specifications and test requirements.
- **R11** Review the test coverage of existing equipment and extend it where it is deemed necessary.

Recommendations: cntd

- **R12** Give the justification documents the same attention as code. Improve the technique for keeping code and its justifications consistent.

- **R13** Set up a team that will prepare the procedure for qualifying software, propose stringent rules for confirming such qualification, and ascertain that specification, verification and testing of software are of a consistently high quality in the Ariane 5 programme. Including external RAMS experts is to be considered.

- **R14** A more transparent organisation of the cooperation among the partners in the Ariane 5 programme must be considered. Close engineering cooperation, with clear cut authority and responsibility, is needed to achieve system coherence, with simple and clear interfaces between partners.

Main ingredients of the Ariane failure report

- **An analysis of what happened:**
  - In the form of a detailed chain of events leading to the catastrophic disintegration of the space
    - tracing the failure back to the fault (set of faults)
  - Containing a set of lessons to be learned

- **A set of recommendations:**
  - Not only aimed at the prevention of this specific chain of events in future Ariane 5 flights, but also of a more generic nature, addressing reliability in general
  - In other words, identifying a set of fault tolerance tactics, targeting the development of (software for) systems for delivery of satellites in orbit

Lessons to be learned

- Components are dependent, and so is their reliability
  - The overall failure required a whole series of components to ‘cooperate’
  - One component’s failure is another ones fault
  - Note that the language (ADA) was specifically designed for safety-critical systems and has the facilities to properly deal with the error
- Small errors, big effects
  - Small errors can have big impact
    - dependency cause/consequence is not linear
  - Several simultaneous small errors can result in unexpected (emergent) behavior
- Assumptions need to hold
  - Solutions for problems are based on assumptions
  - Assumptions are often not explicitly stated in documentation or simply disregarded when using a component

Recommendations

- The recommendations are generic tactics to improve reliability
  - Minimize risk by keeping running code minimal during flight
  - Provide realistic test environment
  - Do not stop measurements (no ‘error exit’)
  - Provide backup facilities
  - Provide scenarios for handling failures of software
- Recommendations also focus on procedures
  - Review every component (sw+hw)
  - Review all software (identify implicit assumptions, missing scenarios..)
  - External reviewers requested (ones not involved in development)
  - Improve testing procedures
  - Proper documentation is as important as code
  - Transparent cooperation between partners needed

Recommendation(s) that would have caught the problem

- R5
  - Would have prevented the out-of-range fault.
- R6
  - Would have kept the effects of the fault confined to the SRI module
- R10
  - Would have created a test that used Ariane 5 data and reveal the occurrence of the out-of-range value
- R12
  - Would have spotted that there is no reason to have the code running after take-off.

Scenario for “minimize running code”

- Stimulus: lift-off event, e.g. acceleration > 0
- Source of stimulus: acceleration sensor

- Precondition: Ariane is on the launch platform, and count-down procedure completed in normal fashion.
- Affected systems: on-board computer software

- Response:
  - switch of all functionality that is not necessary during flight.
    - in particular, switch off the horizontal alignment functionality BH of the inertial reference system (SRI).
- Measure:
  - status flag value / condition, e.g.,
    - Running[BH] = Off, or
    - BH $\notin$ Active Task List

Fault types

Recall

- Failure: not meeting the specification
- Error: system state that may lead to failure
- Fault: cause of an error.

Fault types:

- transient: exists and then disappears
- intermittent: occurs repeatedly, disappears in between
- permanent: remains until repair (fault handling)

Note: faults can occur at all system components

- processes, channels, machines, data, code, …

and may be accidental or by malicious intent

Fault model and metric

- **Fault/failure model**
  - Identifies the system components, where faults may occur
    - typically entities that provide computation or communication
  - Classifies the type of faults that can occur
    - or failures that may be observed
  - Defines the notion of correct behavior

- **Resilience**
  - Metric to measure fault tolerance
  - A t-resilient system can tolerate up to and including t faulty components
    - as specified by the fault model
  - Depending on the fault model there are theoretical bounds on the achievable resilience (t_{max})

(Process) Failure types (vST)

Type of failure	Description of server's behavior
Crash failure	A server halts, but is working correctly until it halts Halting is observable by other processes (clients)
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	The server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure ("byzantine")	A server may produce arbitrary responses at arbitrary times

The processes are blamed, even if the network is the cause, e.g.,.

- omission failures instead of message loss
- arbitrary server responses instead of man-in-the-middle attack
- distinction often not relevant to the application

Synchronous systems (Hadzilacos and Toueg)

A *synchronous* distributed system is one in which:

- The execution time of each process step has known lower and upper bounds
- Each message transmitted over a link is received within a known bounded time
- Each process has a local clock with known bounded drift rate

In synchronous system is possible to observe timing failures, and hence build a service, called a *failure detector*, that reports to (the) other processes that a fellow process has crashed

- by using timeouts on *alive messages*

Fault models for processes

General framework.

- **Processes execute a *distributed algorithm*, i.e., a *set of local algorithms* (one per process) that**
  - cooperate to achieve a global goal
  - are (usually) identical
  - perform (usually) infinite computations
- **Each local algorithm consists of *sending and receiving messages* and of *local computation steps*, thus maintaining the local part of the distributed state.**
- **A process fault model indicates how a process may deviate from the correct execution of its local algorithm.**
  - due to its distributed nature, a distributed algorithm, and each of its local components in isolation, may exhibit many distinct executions.

Fault models for processes I

- **Initially dead**
  - a process does not execute any step of its local algorithm
- **Crash-stop**
  - a process executes its local algorithm correctly up to some moment in time, where after it stops.
  - the crashed process is no longer considered part of the system.
  - a process is correct when it can execute an infinite number of steps
  - in combination with the existence of a failure detector, this gives the crash-failure model on slide 41
- **Omission**
  - a process deviates from its local algorithm by not sending or receiving a message when it was supposed to do so.

Fault models for processes II

- **Crash-recovery**
  - upon recovery a process is aware that it has crashed and restores its state before it resumes its local algorithm
    - for the latter a persistent log can be used
  - a process is correct, if it never crashes or crashes and subsequently recovers only a finite number of times.
- **Byzantine (arbitrary)**
  - a process may exhibit arbitrary behavior, i.e., behave according to any of the above fault models, and/or executes non-specified steps and/or sends non-specified messages.
  - often malicious behavior, e.g., caused by a virus.
  - requires cryptographic tactics to deal with, e.g., authentication techniques to establish the identity of the sender of a message, etc.

Link models/abstractions

- **are usually obtained by**
  - a basic model that captures reasonable hardware behavior
  - a software stack implemented as a layered hierarchy of components executed by a single process at both ends of the link
    - each layer of the stack provides a different abstraction
    - so link models refer to logical connections, not to physical network links
    - in this context links are also known as channels
- **consider three types of message failures**
  - message loss, message duplication, and message creation
- **are distinguished by the requirements they impose on the occurrences of each type of these failures**
  - in addition there are assumptions about the process model of the processes executing the stack

Message loss

Message loss is usually expressed as lack of delivery (to the client of the messaging service).

The following requirements are usually considered:

- Fair-loss delivery (FLD)
  - If a correct process p infinitely often sends a message m to a correct process q, then q delivers m an infinite number of times.
- Reliable delivery (RD)
  - If a correct process p sends a message m to a correct process q, then q eventually delivers m.

Message duplication

Can be caused by the link hardware, but usually by the software stack, e.g., in case of recovery after crash by process when the process is amnesic to what has been delivered.

The following requirements are usually considered:

- Finite duplication (FD)
  - If a correct process p sends a message m a finite number of times to a correct process q, then m cannot be delivered an infinite number of times by q.
- Stubborn delivery (SD)
  - If a correct process p sends a message m once to a correct process q, then q delivers m an infinite number of times.
- No duplication (ND)
  - No message m is delivered by a process more than once.

Message creation

Mostly caused by malicious intent, e.g., in case the link hardware or the software stack has been attacked. Its prevention usually requires cryptographic techniques.

The following requirements are usually considered:

- No creation (NC)
  - If some process q delivers a message m with sender p, then m was previously send to q by p.
- Authenticity (AU)
  - If some correct process q delivers a message m with sender p and process p is correct, then m was previously send to q by p.

Note that it is assumed that each message carries a tag that uniquely identifies (but not necessarily truthfully) its sender.

Link models (combination of requirements)

	Requirements			Process model
	loss	**dupl**	**crea**	**assumed**
Link model				
Fair-loss links	FLD	FD	NC	Crash-stop
Stubborn links	--	SD	NC	Crash-stop
Perfect links	RD	ND	NC	Crash-stop
Authenticated perfect links	RD	ND	AU	Byzantine

A fair-loss link can be seen as the minimal transport service on top of which more reliable communication primitives can be build.

UDP guarantees FLD
TCP guarantees RD

System failure models

A combination of a process model, a link model and some failure detection capability. Distributed algo-rithms are designed for a specific system model!

	Process model	Link model	Detection
System model			
Fail-stop	Crash-stop	Perfect links	Perfect
Fail-noisy	Crash-stop	Perfect links	Eventual perfect
Fail-silent	Crash-stop	Perfect links	None
Fail-recovery	Crash-recovery	Stubborn links	Eventual perfect
Fail-arbitrary	Byzantine	Auth-perfect links	None

An *eventual perfect* failure detector reports processes as *suspected* to have crashed. It may *revoke* its suspicion, when a message arrives with information to the contrary. When the same system state persists, event-ually no correct process is suspected, and all faulty process are.

Reliable unicast communication

A one-to-one communication (unicast) service is *reliable* when it satisfies requirements for a perfect link, i.e., RD + ND + NC

In this context requirement RD is referred to as

- *Validity:* any message sent is eventually delivered

The combination of requirements ND + NC is called

- *Integrity:* the message received is identical to the one sent, and no messages are duplicated

Note that RD, ND, and NC are formulated without specifying the domain of processes p and q. For unicast communication their identity is unique and therefore left anonymous in the formulation of validity and integrity above.

Process groups

To create a single service that is

- highly performant and scalable
- highly available and reliable (fault-tolerance)
  - "***In a fault-tolerant process group, each correct process executes the same commands in the same order, as every other correct process***"

This requires that there is *consensus*

- on the *group view*, i.e., the members of the group
  - time is divided into epochs in which the group view is constant
  - communication is *view-synchronous (virtually synchronous)*
    - each message received in an epoch was send in the same epoch
- on the next command to be executed
  - commands need to be disseminated via *atomic multicast*, i.e., delivered to *all or no* group member, and in the *same order*

Reliable multicast communication

In group communication each messages uniquely identifies both its sender and the group to which it is sent. A group communication (multicast) service is *reliable* when it satisfies requirements

- *Validity:* if a correct process multicasts message m, then it will eventually deliver m (to itself).
- *Integrity:* a correct process delivers message m at most once and if it delivers a message with sender *s*, then it belongs to the group of m, and m was previously sent in a multicast operation by *s*.
- *Agreement:* if some correct process delivers m, then all other correct processes in the group will eventually deliver m.

These requirements are equivalent to RD + ND + NC under the additional assumption that all processes mentioned belong to a single group.

Consensus problem

In the context of faults the set of correct processes need to arrive at the same decision (agreement)

- the decision may depend on initial state
- binary decisions or numerical values

Typical applications

- selecting a leader (process)
- committing or aborting a transaction
- establishing a total order of operations
  - repeatedly deciding on the next operation

Consensus service interface

- accepts a proposal event $\langle\, consensus, \text{Propose} \mid v \,\rangle$
- returns a delivery event $\langle\, consensus, \text{Deliver} \mid v \,\rangle$

Consensus guarantees

- **Termination**
  - every correct process eventually decides some value
- **Validity**
  - if a process decides v, then v was proposed by some process
- **Integrity**
  - no process decides twice
- **Agreement**
  - no two correct processes decide differently
- **Uniform agreement**
  - no two processes decide differently

Fault tolerance tactics (for availability/reliability)

- *Fault masking*
  - Error detection
  - Error diagnosis (identifies the fault(s) causing the error)
  - Error/fault containment (isolating from spreading further)
  - Error recovery (replacing erroneous state with error-free state)

- *Redundancy*
  - Hot, warm, cold spare

- *Fault prevention*
  - Sound development methodologies:
    - coding standards, test procedures, design patterns, ...
  - Transactions
  - Temporary *removal from service*
    - e.g. occasionally restarting component to prevent memory leaks accumulating to the point of causing failure

Availability: fault/error detection

- *Acceptance test*
  - checking correctness of results
- *Exceptions*
  - raised when error is detected
- *Watchdog timer*
  - timing error when result is not available in specified time
- *Ping/echo*
  - a component issues ping and measures the time till it receives an echo response
  - lack of echo indicates fault
- *Heartbeat*
  - one component emits heartbeat messages periodically and another component listens and measures the time (e.g. using watchdog)
  - lack of expected heartbeat message indicates a fault (e.g. node/network failure)

Mars Rover Pathfinder

- July 4, 1997, landing on Mars
- After a few days into the mission random resets occurred
  - after start of gathering meteorological data
- Attributed to ‘software glitches’ or ‘system overload’; OS VxWorks

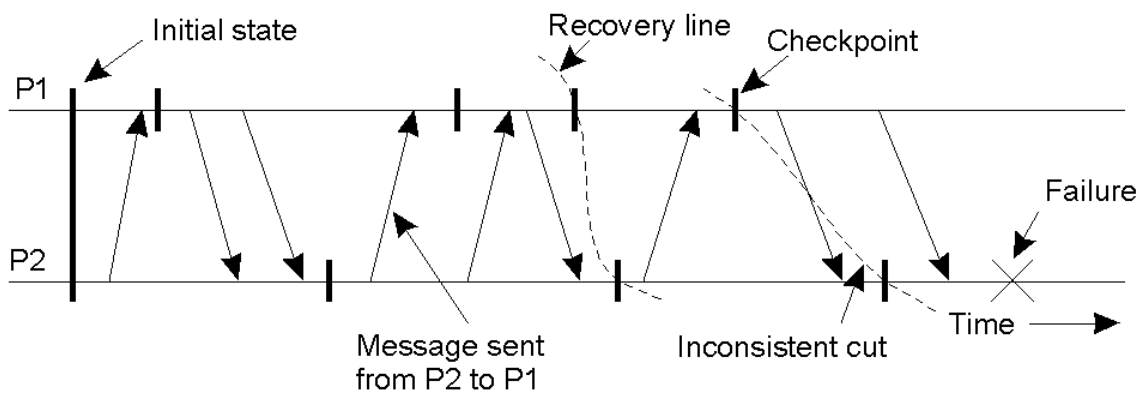
What really happened on Mars
http://knusbaum.com/mars/Authoritative_Account

- Cause: shared resource: information bus
  - high priority task: moves important information
  - low priority meteorological data gathering task: gets interrupted by a middle priority, unrelated task
  - a watchdog reset is triggered upon a long delay of the high priority task

  - .... priority inversion
  - .... forgot to declare the correspondent semaphore as ‘priority ceiling’ (or inheritance)

Error recovery

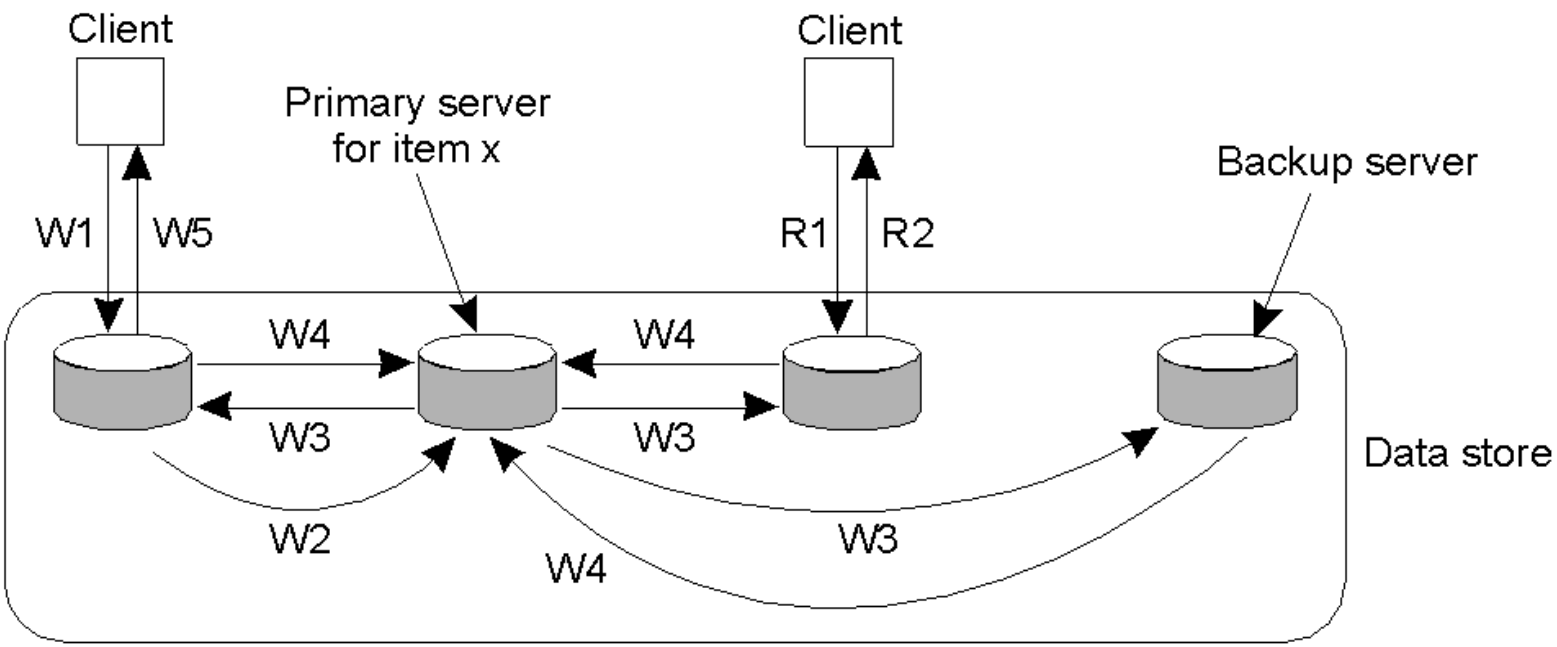
- *Forward error recovery*
  - finding a new error-free state that will allow system to continue
    - e.g. a self-stabilizing algorithm
  - often based on redundancy
  - predictable in terms of time and memory => applicable in real-time systems
- *Backward error recovery*
  - restoring previous error-free state
  - suited for handling transient faults
  - "checkpointing" and restoring from a consistent cut is most widely used mechanism
    - consistent cut: a *possible* distributed state obeying causality of message passing



Redundancy

- *Redundancy* (replication for fault tolerance!)
  - information: add extra bits
  - time: repeat failing operation (only useful for transient or intermittent faults)
  - physical redundancy: multiply hardware and/or software components
- *Active redundancy*
  - all replicas performing calculations in parallel
    - Beware this is different from storing a copy of a state update
  - choice to deliver: first obtained (correct) result, result of preferred component (primary replica), or to perform “majority voting”
  - “synchronization” of replicas needed
- *Passive redundancy*
  - a replica activated only when previous replica fails
    - detected by watchdog (failure detector) or via an acceptance test
    - example: a standby spare configured to jump in; during initialization reads last valid state from persistent memory (rollback)

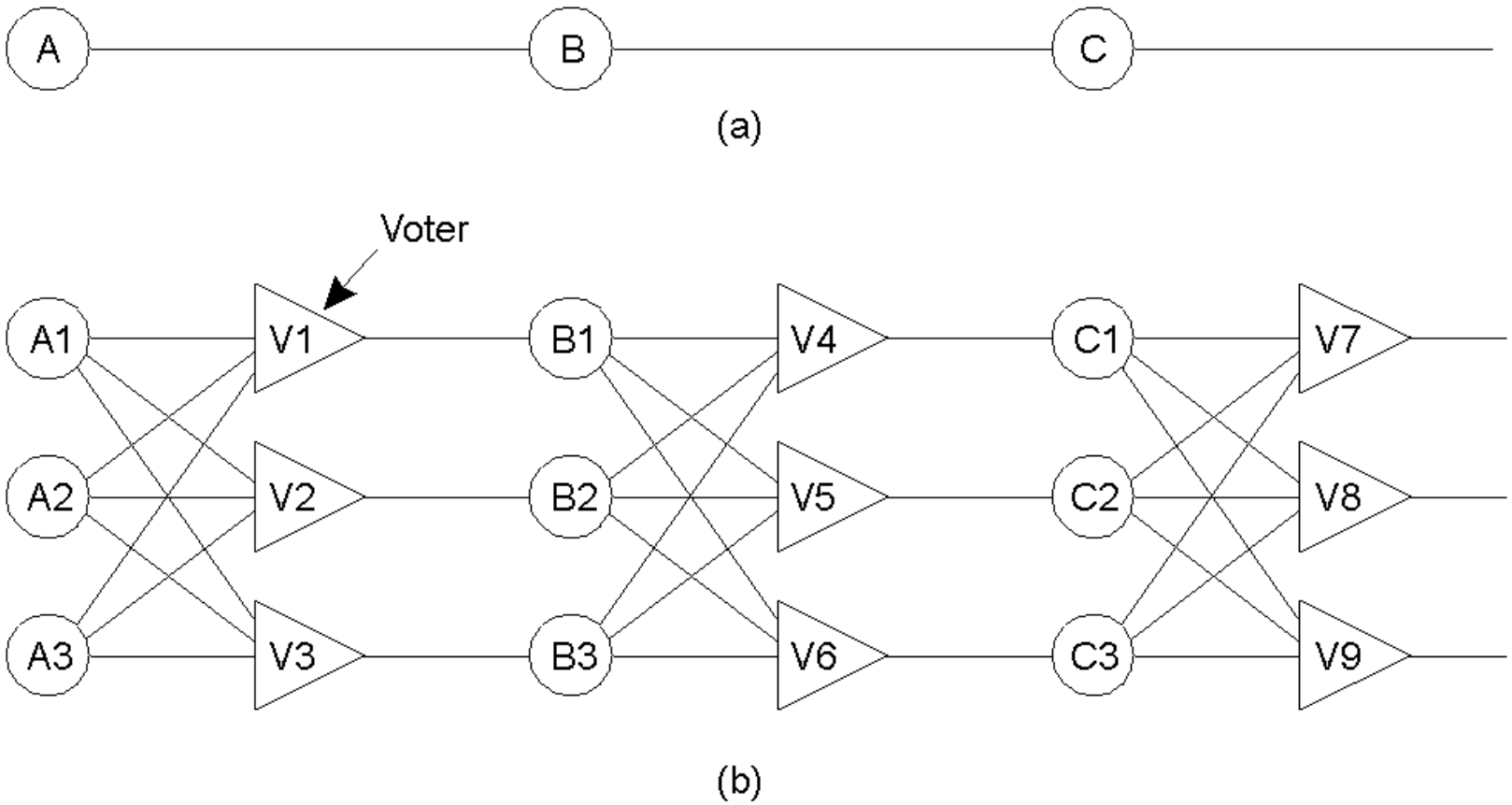
Example: Primary backup



W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

Example: triple modular redundancy



Fault prevention: transactions

- Transaction: sequence of actions, combined into a single operation
- with the ACID properties

  - atomic, i.e., taken ('committed') or not taken, indivisible
    - e.g. reserving an airline seat in a multi-flight trip
  - consistent, maintaining system invariants
    - e.g. no seats are lost (free + reserved = # seats in airplane)
  - isolated, concurrent transactions appear serialized – transient states not observable
    - e.g. it becomes my seat or your seat; system determines the outcome
  - durable, a *committed* transaction is persistent
    - system failure after commit cannot undo the result of the transaction

- NOTE: not completely independent notions

Summary

- Availability and reliability improvements comprise mainly:
  - Extending the up-time
    - restricting the (probability of) occurrence of faults/errors
    - dealing systematically with partial failures
    - improves reliability and availability (when repair time stays constant)
  - Restricting down time
    - facilitating fast repair/recovery
    - improves availability

- Together these techniques are referred to as *fault tolerance* techniques
  - Techniques that prevent a fault from becoming a failure

System qualities: overview

- ...
- *Modifiability*
  - about how difficult it is to introduce desired changes
- *Performance*
  - timely response to service request events, throughput, jitter
- *Security*
  - ability to resist unauthorized attempts to access data and services
- *Testability*
  - how difficult it is to verify the correctness of the system
- *Usability*
  - how much the system is user friendly
- *Scalability*
  - see before
- *Portability*
  - how difficult it is to port system to another platform

Modifiability

Modifiability is concerned with the cost of system change:

- with the extent of the modification..

- *Directly affected modules*
  - their responsibilities need adjustment
- *Indirectly affected modules*
  - need to change due to cooperation with directly affected modules

- ... and with the time and effort to do it

- Objective: Reduce time to deploy
  - modification done by developer requires testing and distribution process
  - time lag between making change and making it available to the user

- ....allow late change or even non-developers to make changes

- Derived quality attributes
- usability/configurability – modifiability offered to the user in user interface
- portability – modifiability of the platform

Modifiability - QA scenario elements

- *Source*
  - end user / developer / operator
- *Stimulus*
  - change request concerning function / quality / capacity
- *State of system (environment)*
  - during analysis / at design time / compile time / build time / initiation time, runtime / before release/ after release / during testing…
- *Affected resources (artifacts)*
  - code / data /interfaces / components / configuration …
- *Response & tactics*
  - normal design sequence of specify, change, test, deploy
  - limit dependencies, select appropriate styles
- *Response measure*
  - all response activities cost time and money which can be measured (manpower spec e.g. 3 man-years

Cost factors: in relation to responsibilities

- Extent of a single responsibility
  - more complex responsibility implies higher cost to modify
- *Coupling:* the degree of dependence *between* two components
  - stronger coupling implies higher cost to modify
- *Cohesion:* the degree to which the responsibilities of a single component (module, system) form a meaningful unit
  - stronger cohesion implies lower cost to modify
- Dependencies between components or modules
  - arise from assumptions about peer components regarding
    - (existence of) data/service, QoS of these
    - syntax/semantics
    - data/control flow
    - interfaces
    - location
    - resource behavior
- Terms: see e.g. M. Page-Jones, *The practical guide to structured system design*, Prentice-Hall, 1988.

Coupling and Cohesion

- Cohesion (coherence) is somewhat subjective
  - coherent *with respect to what*?
    - e.g., highly dependent sets of responsibilities, part of a more abstract responsibility
    - optimally, 1-1 mapping between responsibilities and modules
    - so, modified by redistribution of responsibilities

- Classical distinction (next slide)
  - functional,
    - ideal
  - sequential, informational / communicational
    - good
  - procedural, temporal, logical, coincidental
    - bad

Cohesion types

cohesion type	*description*	*example*
functional	single independent function	ComputeOrbit
sequential	one function's output is the next one's input	Stages of a compiler: tokenize, scan, parse, generate code, optimize code
informational	operations using the same data, with separate entry points	GetName(Customer), GetAddress(Customer)
communicational	unrelated operations on the same data /devices	Update(Customer) and WriteBack(Customer)
procedural	operations in component describe some workflow	Cooking recipe, checking file permission before opening the file
temporal	operations put together because they happen at a certain stage	(re)initialization of global variables; operations upon termination
logical	similar category operations	do all input
coincidental	arbitrary grouping	

Modifiability tactics: *"Localize modification"*

- Maintain semantic coherence: reduce coupling and increase cohesion
  - the 'good' cohesion
- Abstract common services
  - special case of maintaining semantic coherence
  - idea is to abstract common services, so that changing them is done on a single place e.g. in application frameworks and middleware
- Anticipate and isolate expected changes
  - for each anticipated change limit the influenced modules
  - It is not possible to anticipate all changes => assume that expected changes will be in semantically coherent modules
- Generalize the module
  - more general module is more resilient to changes
    - e.g. exchanging data via XML files yields more resilience to the changes of actual data format
    - e.g. the interpreter style idea is that changes will be in provided scripting language programs and not in interpreter
- Limit options

Modifiability tactics: *"Prevent ripple effects"*

- Hide information
  - isolate possible changes by keeping them away from interfaces
  - related to "anticipate expected changes" tactic

- Maintain existing interfaces
  - interface stability is achieved by clear separation between interface and implementation
    - Parnas' principle,
  - "adding interfaces" instead of changing them
  - restrict communication paths

- Insert intermediary between modules
  - special case of maintaining existing interfaces
  - e.g. blackboard and MVC architectural styles
  - e.g. façade, bridge, mediator, strategy, proxy and factory design patterns

Example: modifiabilty tactic

inputs that originate from stimulus:
- source : end-user
- stimulus: changes in GUI (expected to occur frequently)

inputs that originate from system:
- state: development
- artifact: GUI application

use MVC (model-view-controller) architectural style

Actually a col-lection tactics

guaranteed outputs:
- response: views can be changed independently and without need to rewrite model
- response measure: manpower needed for a change (e.g. 5 man-month)

Modifiability tactics: “Defer binding time”

- Runtime registration
  - plug-and-play operation
- Configuration files
  - used to setup parameters at startup rather than putting them in code
- Polymorphism
  - late bindings of method calls
- Component replacement
  - load time binding
- Runtime binding of independent services
  - allowed by adherence to defined protocols and interfaces
  - orchestrated by special (framework service)
  - spontaneous through choreography

Modifiability: summary

- Modifiability is addressed

- within the development view
  - modules, interfaces, decomposition into subsystems and components: cohesion/coupling
- within the process view
  - binding, configuration
- through the applied styles and frameworks
  - each style defines a particular structure of building blocks, i.e., modules with assigned responsibilities and prescribes interaction patterns