Architecture of Distributed Systems
Quality Attributes 2018-2019
R. H. Mak

TU/e 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
  • 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
Types of qualities

In the design process:

- Qualities of the developed system (QAs, system qualities – 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
Some serious software faults

Consequences of unreliable software can be large. See


Examples in these lists overlap. List 4 contains references to original failure reports
The first bug report

• Causes of failures can be small:

Page from the Harvard Mark II electromechanical computer’s log

• Page from the Harvard Mark II electromechanical computer’s log
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
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.
• 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.
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.
• **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.
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 one’s fault (see later)
  - 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

- Which recommendation(s) would have caught the problem?
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.
‘Assumptions about the environment’

- consider all behaviors of the inputs to a component: 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), ....
Examples

- 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 maintenance
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

https://www.youtube.com/watch?v=VVJIKJi9FWU
Practical aspects of QAs

• Dependencies exist between QAs
  • Improving most QAs negatively influence 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
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

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.:

<table>
<thead>
<tr>
<th>QA</th>
<th>Architectural level</th>
<th>Design &amp; implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usability of UI</td>
<td>redo/undo support</td>
<td>GUI layout &amp; coding</td>
</tr>
<tr>
<td>Modifiability</td>
<td>separation of concerns e.g. independent modules</td>
<td>coding techniques</td>
</tr>
<tr>
<td>Performance</td>
<td>usage of shared resources communication infrastructure</td>
<td>algorithms used optimization techniques</td>
</tr>
</tbody>
</table>
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
Tactics definition (achieving QAs)

Goal: design tactics that guarantee required outputs for given inputs

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

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

Tactics

guaranteed outputs:
- response
- response measure
Recall from the Ariane 5 example

- 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
    - which is what we are discussing now

- 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

- Question: can you give corresponding scenario’s?
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
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
Aggregate qualities

Some qualities that are defined as a collection other qualities, e.g.

**Dependability**
- the ability to deliver service that can justifiably be trusted
- dependability attributes
  - Availability
  - Reliability
  - Safety
  - Integrity
  - Maintainability

**Security**
- the ability to resist unauthorized attempts to access data and services
- security attributes
  - Availability
  - Integrity
  - Confidentiality
System qualities: metrics

- **Availability**
  - Readiness for correct service (expressed as a probability)
- **Reliability**
  - Continuity of correct service (expressed by the expected time to failure – in an infinite run of repeated starts, the mean time between failures)

**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 definition: \( \frac{MTTF}{MTTF + MTTR} \)
  - \( MTTR \): time to repair is the time till the failure becomes not observable
  - \( MTTF \): mean time to failure (= reliability)
  - \( MTBF = MTTF + MTTR \)
Notions

• Failure: not meeting the specification
  – 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
• Fault: cause of an error. Fault types:
  • transient: exists and then disappears
  • intermittent: repeatedly; disappears in between
  • permanent: remains until repair (fault handling)

• Note: faults can occur for all system components
  • processes, channels, machines, data, ...

• Fault tolerant system
  • A system that can continue to provide required or degraded service despite the presence of faults
## (Process) Failure types

<table>
<thead>
<tr>
<th>Type of failure</th>
<th>Description of server’s behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash failure</td>
<td>A server halts, but is working correctly until it halts. Halting is observable by other processes (clients)</td>
</tr>
</tbody>
</table>
| 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        | A server may produce arbitrary responses at arbitrary times.                                              |

**Note:** It is hard to distinguish some differences just by observation.
Availability - QA scenarios

- **Source**
  - system external / system internal (other subsystem)

- **Stimulus**
  - fault: in external input, failure in subsystem, e.g. omission / crash / timing error / 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, continue (normal/degraded), repair, isolate, replicate, ...

- **Response measure**
  - repair time, availability, available/degraded time interval…
Availability - example of QA scenario

inputs:
gulp stimulus: unanticipated message
(gulp source: external

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

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

- **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)
Availability: error recovery

- **Forward error recovery**
  - finding a new error-free state that will allow system to continue
  - 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
Availability: 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 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

Client

Primary server for item x

Backup server

Data store
Example: triple modular redundancy
Availability 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
Availability: summary

- Availability and reliability improvements comprise mainly:
  - Improvement of reliability
    - restricting the (probability of) occurrence of faults/errors
    - dealing systematically with partial failures
  - Restricting down time

- 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 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
**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
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)
Modifiability - example of QA tactics

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

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)
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

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

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