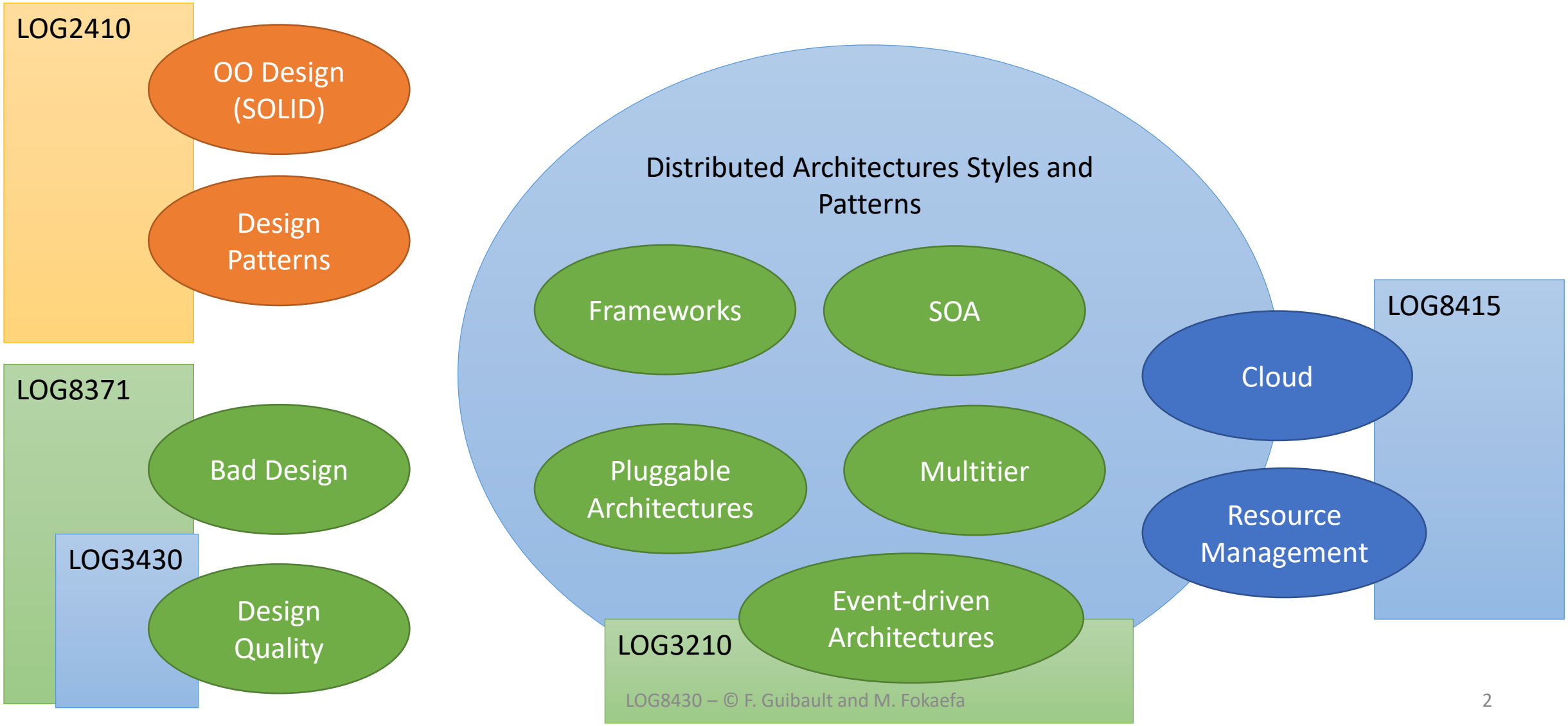
POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

LOG8430E: Quality Attributes, part 1

Course Map

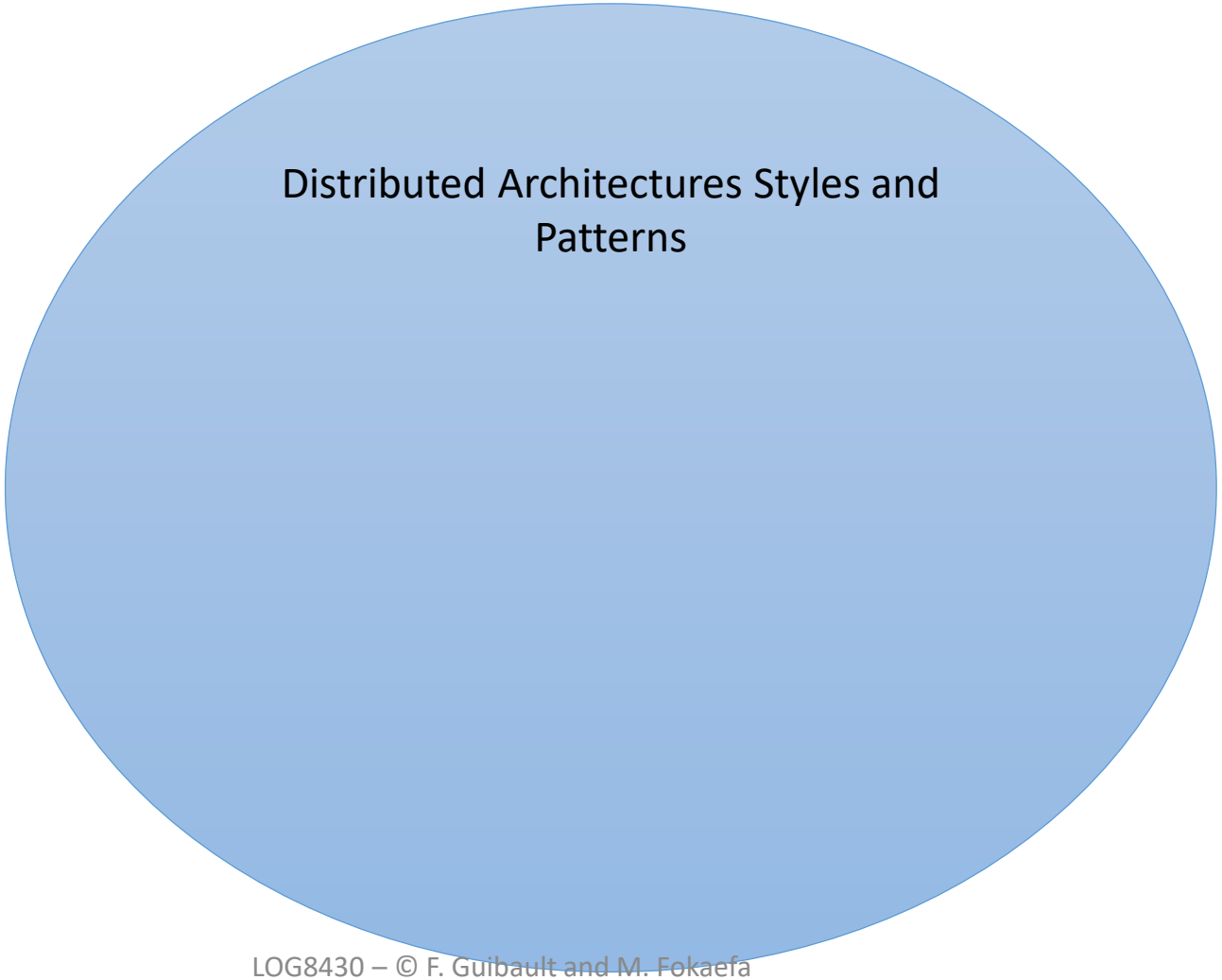

LOG2410
OO Design (SOLID)
Design Patterns
LOG8371
Bad Design
LOG3430
Design Quality
Distributed Architectures Styles and Patterns
Frameworks
SOA
Pluggable Architectures
Multitier
Event-driven Architectures
LOG3210
Cloud
Resource Management
LOG8415

So far…

LOG2410

Conception OO (SOLID)

Patrons de Conception



Today

LOG2410

OO Design (SOLID)

Design Patterns

LOG8371

LOG3430

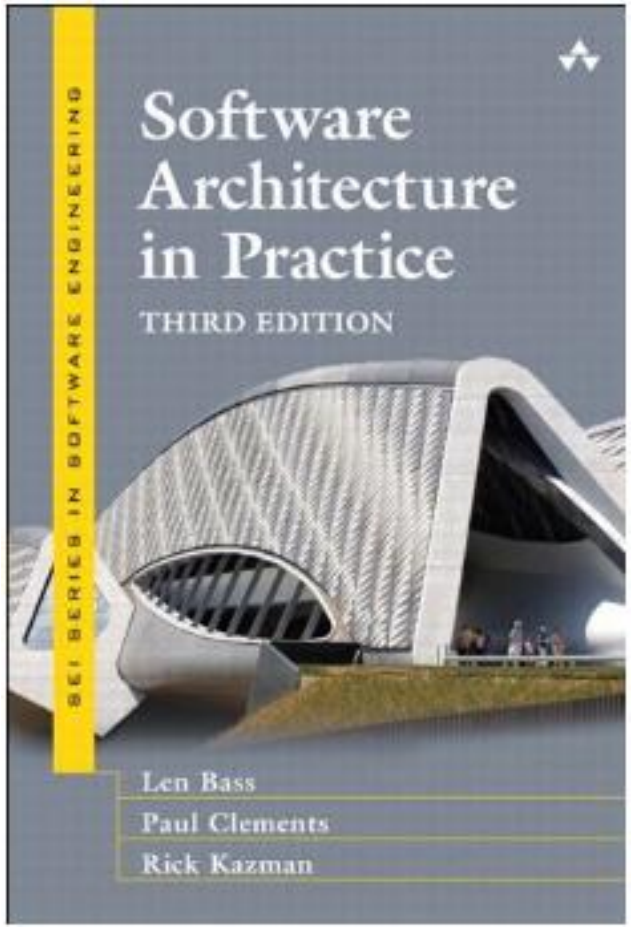
Design Quality

Distributed Architectures Styles and Patterns

Quality Attributes

The content of this chapter comes from the book:

Bass, L., Clements, P. et Kazman, R. *Software Architecture in Practice*. 3rd Edition. Addison-Wesley. 2013.



Quality Attributes – part 1

Definition of quality attributes

1. **Availability**
2. Interoperability
3. Modifiability
4. Performance
5. Security
6. Testability
7. Usability
8. Other quality attributes

Quality Attributes

As designers, turn your attention to non-functional quality attributes.

Software systems are often reimplemented for reasons that are not related to the functionality:

- Difficulties in maintenance
- Portability or scalability problems
- Security vulnerabilities
- Lateral migration of the functionality in the reimplementation process

Quality Attributes

A quality attribute is a measurable and verifiable (by a test) property of the system, which we can use to indicate up to what point the system satisfies the requirements of the stakeholders.

It's a value measure of a product according to an interest dimension for a stakeholder.

Two categories of quality attributes: 1) system properties at runtime, 2) system properties at design time.

The quality attributes are intimately connected to the system requirements.

Quality attributes – An example

In the case of an unavailable system due to a Denial-of-Service attack, what system quality is affected? Is it:

- An availability problem?
- A performance problem?
- A security problem?
- A usability problem?

Each community related to these different quality attributes could claim some ownership or responsibility for finding a solution the problem of managing a DoS attack.

Quality Attributes – An example

3 fundamental problems related to the usual definitions of quality attributes:

1. The definitions are generally hard to test.
2. The discussions focus more on the identification of the affected quality attribute during a situation rather than on how to adapt or modify the architecture to respond to it.
3. Each community has its own vocabulary to talk about a quality attribute.

A solution in 2 parts:

1. Use scenarios of quality attributes to characterize each attribute,
2. Concentrate on concerns underlying the quality attributes to illustrate the fundamental concepts of each attribute.

Specification of a quality attribute requirement

Quality attributes are defined in four parts:

1. Stimulus: description of an event that affects the system.
2. Source of stimulus: origin of the stimulus. The origin of the stimulus can modify the way we respond to it.
3. Response: description of the expected reaction of the system (execution) or of the developers (development).
4. Measure of the response: determine if the response is satisfying.

Two other important characteristics:

1. Environment: the set of circumstances in the which the scenario exists. It qualifies the stimulus.
2. Artifact: part of the system on which the requirement applies.

Example: a general scenario of availability

Source of stimulus

Internal/External: human, hardware, software, physical infrastructure, physical environment

Stimulus

Error: omission, crash, incorrect timing, incorrect response

Artifact

Processors, communication channels, persistent storage, processes

Environment

Normal operation, launch, stop sequence, repair mode, restrained operation mode, overload

Response

Prevent the error from causing a failure
Detection of the error: logging, warning
Recovery of the error: deactivation of the error source, become unavailable, repair/mask, enter restrained operation mode

Measure of the response

Time or interval of availability of the system
Percentage of availability
Time in restrained mode
Time to detect error
Time of repair
Proportion of errors processed by the system

Achieve quality attributes: tactics

Designing the system, how can we attain quality attributes?

By applying a set of specific techniques: architectural tactics.

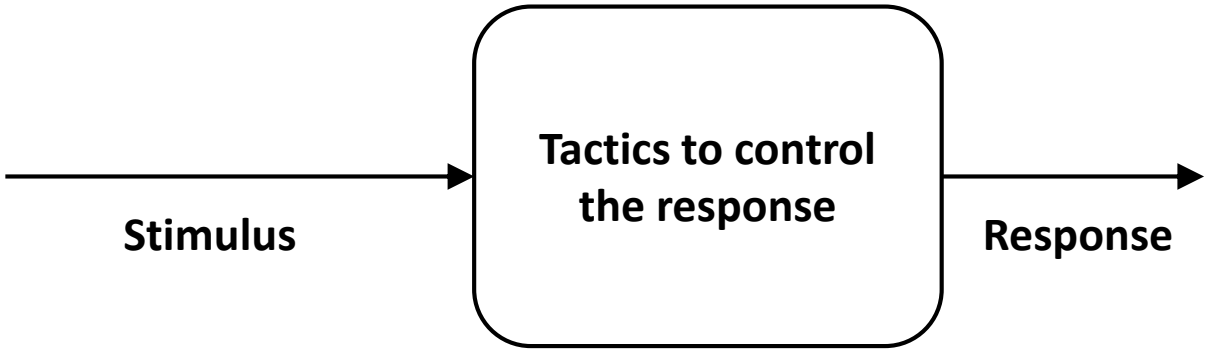
An architectural tactic is a design decision which influences the achievement of a response of a quality attribute.

A tactic affects directly the response of a system to a stimulus.

Achieve quality attributes: tactics

A tactic focuses on a single quality attribute.

- No tradeoff with other attributes.
- It's to the designer to consider/control the tradeoffs.
- Different than an architectural pattern, which includes by construction the notion of tradeoff.
- The tactics are proven techniques that captures developed by software architects over the years.



Guide the design decision towards quality

A classification of architectural design decisions

1. Allocation of responsibilities
2. Coordination model
3. Data model
4. Resource management
5. Correspondence between architectural elements
6. Binding time
7. Technology choice

Design decisions: Allocation of responsibilities

The decisions around the allocation of responsibilities include:

- Identification of important responsibilities: system functions, architectural infrastructure, satisfaction of quality attributes.
- Determination of the way to allocate responsibilities to non-executables and executables elements (modules, components, connectors).

Strategies to make this type of decisions include:

- Functional decomposition
- Modeling of real-world objects
- Reorganization based on important operation modes
- Reorganization based on similar quality requirements

Design decisions: Coordination model

A software system functions through the interaction of its elements with mechanisms designed for this goal: the set of these mechanisms form the coordination model.

The decisions allowing to decide on a coordination model include:

- Identification of systems elements that need or do not need to assume a coordinating role.
- Determination of the coordination properties: precision, concurrency, completeness, correctness and consistency.
- Choice of communication mechanisms: stateful or stateless, synchronous or asynchronous, guaranteed or not, performance properties (throughput, latency, …).

Design decisions: Data model

All software systems need to internally represent data. The data model is a set of data representation and the way to interpret them.

The decisions on the data model include:

- Choice of principals abstractions, their operations and their priorities: mechanisms for creation, for initialization, for access, for persistence, for manipulation, for conversion and for destruction.
- Compilation of necessary metadata at the correct interpretation of data.
- Data organisation: database, object collection or both. Correspondence between the representations in the database and the objects.

Design decisions: Resource management

It is on an architectural level that it is decided how a system will use shared resources.

The decisions related to resource management include:

- Identification of resources that need to be management and determination of their limits/capacity.
- Determination of elements responsible for the management of each resource.
- Determination of the sharing rules and arbitration strategies in case of shortage.
- Determination of the impact of saturation of different resources. For example, CPU saturation vs memory saturation.

Design decisions: Correspondence between architectural elements

An architecture needs to establish two types of correspondences:

1. Correspondence between elements of different types within the architecture: e.g., correspondence between a development unit (a module) and an execution unit (a process).
2. Correspondence between software elements and elements of the environment: e.g., correspondence between processes and specific CPUs.

Useful correspondences include:

- The correspondence between modules and execution elements.
- The correspondence between execution elements and processors.
- The correspondence between items of data models and the data repositories.
- The correspondence between modules and execution elements, and delivery units.

Design decisions: Choice of moment to establish a link

Each variable aspect of the system needs to be fixed so as to render the system executable. The moment at which a variable aspect is linked to its value is an architectural decision.

The decisions associated with the choice of moment when a link is established include:

- The set of choices,
- The point in the lifecycle,
- The mechanism to establish the link.

Design decisions: Choice of moment to establish a link

The decisions in the other six categories alla have a moment when a link has to be established.

- For resource allocation, we can use a selection of modules at the moment of construction through a makefile.
- To choose a coordination model, we can implement a mechanism for the negotiation protocol at runtime.
- For the resource management, we can design connection mechanism for new peripherals at runtime, including downloading and installing drivers.
- To choose technologies, we can construct a marketplace which automatically selects the appropriate version of an application based on the type of the client device.

Design decisions: Choice of technology

All architectural decisions eventually need to be implemented using a specific technology.

If the technology options are not fixed a priori, the decisions related to the choice of technology include:

- Decide which technologies are available to implement the decisions made in the other categories.
- Determine what tools are available to support the choice of technology and if these tools are adequate.
- Determine the level of internal familiarity and of the available external support for the technology and if this level is sufficient.
- Determine the effects related to the choice of technology.
- Determine the compatibility of a new technology with the existing technological environment.

Quality Attributes

1. **Availability**
2. Interoperability
3. Modifiability
4. Performance
5. Security
6. Testability
7. Usability
8. Other quality attributes

Quality Attribute 1: Availability

The availability refers to the property of a software system to be there and ready to perform tasks at the moment when we have need of it.

We often refer to this property in the same manner as reliability, but availability includes additional considerations related among others to periodic interruptions for maintenance.

The availability adds to the notion of reliability that of recovery.

The availability refers to the capacity of a system to mask or correct its errors so that the cumulative duration of the service interruption does not surpass a specified value over a given time interval.

Quality attribute 1: Availability

The availability is closely related to security:

- A DoS attack aims directly at rendering a system unavailable.

The availability is closely related to performance:

- It is very difficult to distinguish between a system that is being very slow and a system that is unavailable.

The availability is closely related to safety:

- A highly available system is capable of detecting when it enters an unstable or dangerous state and of limiting the damages if they occur.

Quality attribute 1 : Availability

A failure and a deviation of the system from its specifications that is observable by a human.

The cause of a failure is called a fault.

The intermediate states between the appearance of a fault and the appearance of the failure are called errors.

The availability is to minimise the duration of service interruptions by mitigating the faults.

Quality attribute 1: Availability

The principal considerations to examine with respect to faults include:

- How to detect faults?
- At what frequency do faults appear?
- What happens when a fault appears?
- How much time is the system permitted to stay out of service?
- When can faults and failures happen in a secure way?
- How can faults be avoided?
- What kind of warnings are useful when a fault occurs?

One must also consider the level of the system capacity after a fault: reduced operation mode.

Quality attribute 1: Availability

The availability of a hardware system can be calculated as the probability that the system provides specific services within a prescribed margin for a specific durations

$$\frac{MTBF}{(MTBF + MTTR)}$$

where *MTBF=Mean Time Between Failures, and MTTR=Mean Time To Repair*.

For software systems, one has to interpret this formula by evaluating what can constitute a failure of the system, what is the probability that it happens and how much time we need to repair it.

Quality attribute 1: Availability

The level of availability of a software system is often expressed through the service-level agreement (SLA).

A SLA defines the guaranteed level of availability and penalties related to a violation of the level of service by the provider.

Availability	Duration of stoppage/90 days	Duration of stoppage/year
99.0%	21 hours, 36 minutes	3 days, 15.6 hours
99.9%	2 hours, 10 minutes	8 hours, 0 minutes, 46 seconds
99.99%	12 minutes, 58 seconds	52 minutes, 34 seconds
99.999%	1 minute, 18 seconds	5 minutes, 15 seconds
99.9999%	8 seconds	32 seconds

Quality attributes 1: Availability

Plan for failures

The best way to design a highly available system is to analyse the sources of failures and manage them correctly.

One needs to understand the types of failures, which the system is especially subject to and evaluate the consequences of each.

Quality attribute 1: Availability

Analyse the danger. We can categorise the dangers that can occur during operation of the system. The standard DO-178B in the aerospace industry proposes the following categories, according to the impact on the device, the equipment and the passengers:

- Catastrophic: can cause crushing, loss of critical functions.
- Dangerous: great negative impact on security or performance. Can cause an unacceptable workload for the equipment and serious or fatal injuries to the passengers.
- Major: important negative impact. Can cause a significant workload for the equipment that affects security or an important discomfort to passengers.
- Minor: minor impact that can be perceived, but it does not cause any inconvenience to equipment or passengers.
- No impact: no impact for the security of the device, the equipment or the passengers.

Quality attribute 1: Availability

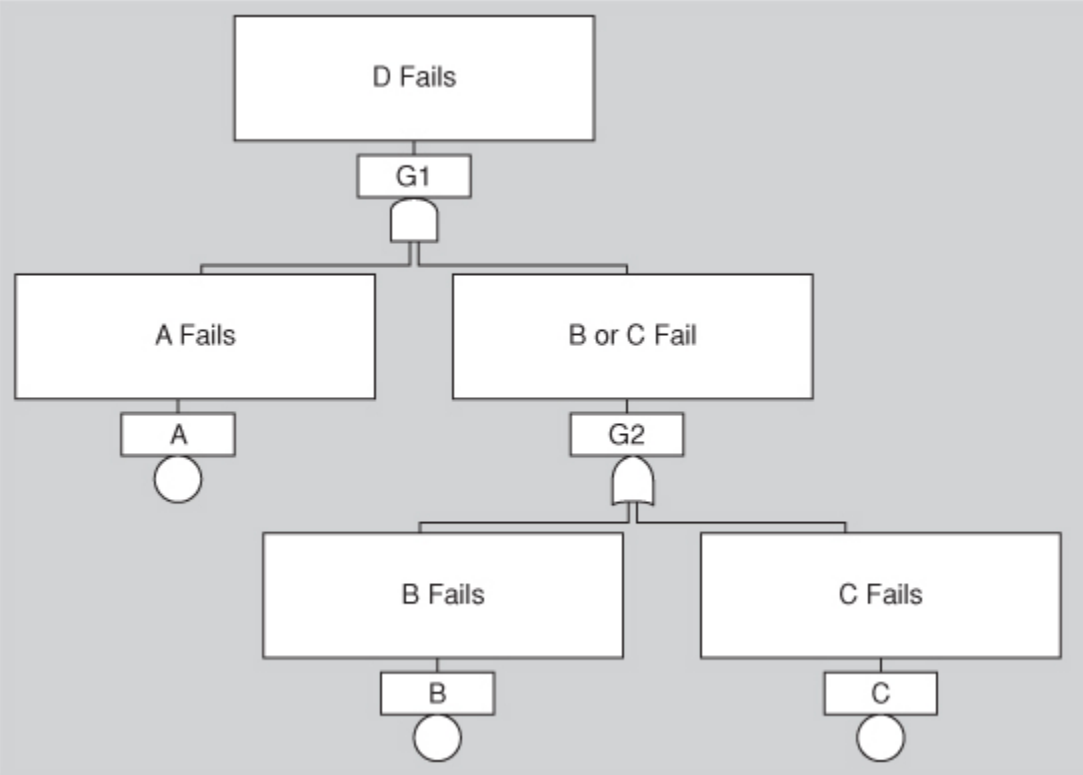
Fault tree analysis.

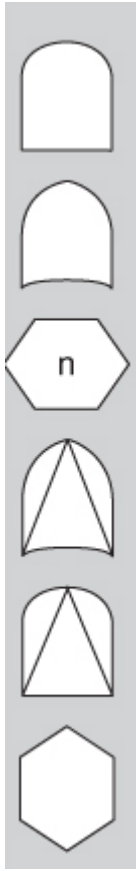
Analytical technique that specifies a state of the system, which can have an impact on the safety or the reliability and which analyses the context and the operations of the system to determine all the ways in which this state can appear.

Method based on the graphical construction of a tree that serves to identify all the parallel and sequential series of faults that can lead to a non-desirable state of the system.

Quality attribute 1: Availability

Fault tree analysis.





AND

OR

COMBINATION of n inputs

XOR

AND WITH PRIORITY

INHIBIT

Quality attributes 1: Availability

Fault tree allows for performing different types of analyses:

- The sets of minimal cuts allow for finding the subset of minimum faults that can cause a failure.
- All sets of minimal cuts indicate all the ways in which faults can combine themselves to cause a failure.
- A set of minimal cuts that has a single element allow to identify a unique point of failure.
- Probabilities can be assigned to different faults to obtain a probability of appearance for the studied failure.

Example: a general scenario of availability

Source of stimulus

Internal/External: human, hardware, software, physical infrastructure, physical environment

Stimulus

Error: omission, crash, incorrect timing, incorrect response

Artifact

Processors, communication channels, persistent storage, processes

Environment

Normal operation, launch, stop sequence, repair mode, restrained operation mode, overload

Response

Prevent the error from causing a failure
Detection of the error: logging, warning
Recovery of the error: deactivation of the error source, become unavailable, repair/mask, enter restrained operation mode

Measure of the response

Time or interval of availability of the system
Percentage of availability
Time in restrained mode
Time to detect error
Time of repair
Proportion of errors processed by the system

Example of concrete scenario of availability

Source of stimulus

Heartbeat monitor

Stimulus
Non-responding server

Artifact
process

Environment
Normal Operation

Response
Inform an operator
Continue normal operations

Measure the response
No service interruption

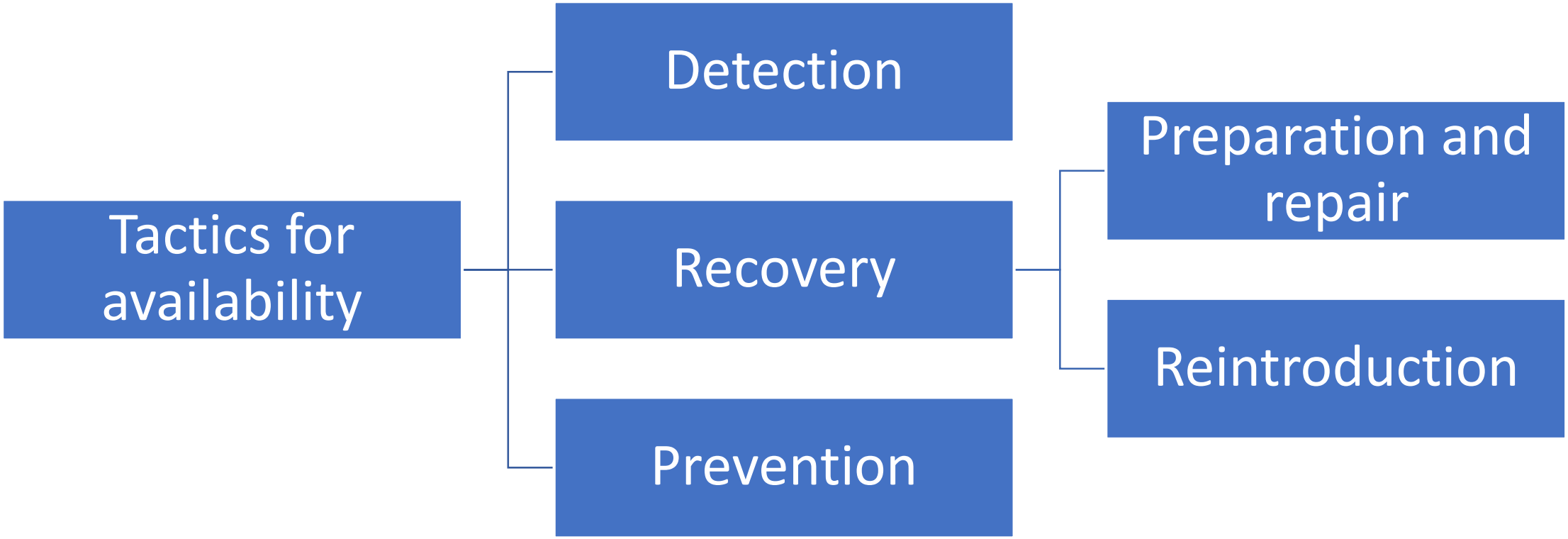
Quality attribute 1: Tactics for availability

A failure occurs when a system no longer provides a service that is consistent with its specifications. This failure is observable by the actors of the system.

The tactics for the availability are designed to allow the system to manage faults and avoid failures.

The tactics aim at preventing faults to cause failures or limiting the effects of the fault and allowing for repair.

Quality attribute 1: Tactics for availability



Quality Attribute 1: Tactics for availability: **Detection**

Tactics to detect or anticipate the presence of faults

- Ping/Echo
- Monitoring
- Heartbeat
- Timestamp
- Verification of coherence
- State monitoring
- Voting: replication, functional redundancy, analytical redundancy
- Detection of exceptions: system exceptions, parameter barrier, parameter types, timeout
- Auto-verification

Quality attribute 1: Tactics for availability: **Recovery by preparation and repair**

Tactics to recover from a fault by preparation and repair

- Active redundancy
- Passive redundancy
- Replacement
- Exception processing
- Return to previous state
- Software update
- Retry
- Ignored behavior
- Degradation
- Reconfiguration

Quality attribute 1: Tactics for availability: **Recovery by reintroduction**

Tactics to recover from a fault by reintroduction

- Operation in shadow mode
- Resynchronisation of state
- Stepwise restart
- Retransmission without interruption

Quality attribute 1: Tactics for availability: **Prevention of faults**

Tactics to prevent faults

- Retire the service
- Transactions
- Predictive model
- Prevention of exceptions
- Increased skill set

Verification list for availability Allocation of resources

- Determine which system responsibilities need to be highly available.
- Among these responsibilities, ensure that the additional responsibilities were added to detect omissions, crashes, incorrect logging or incorrect responses.
- Ensure that responsibilities are in place to:
  - Maintain a trace of error
  - Warn the appropriate entities
  - Deactivate the sources of events causing the faults
  - Be temporarily unavailable
  - Operate in degraded mode

Verification list for availability Coordination model

- For the system responsibilities that must be highly available, ensure that:
  - The coordination mechanisms can detect omissions, crashes, incorrect logging or incorrect responses. For example, examine if the guaranteed delivery of a message is necessary. Can the coordination work in case of degraded operation conditions?
  - The coordination mechanisms allow to maintain a trace of errors, warn the appropriate entities, deactivate the sources of events that cause the faults, repair or mask the faults, or operate in degraded mode.
  - The coordination model supports the replacement of used artifacts (processors, communication channels, persistent storage and processes). For example, does the replacement of a server allows the system to continue working?
  - The coordination will work under degraded communication conditions during launch or stop, in repair mode or on overload. How much loss of information can the coordination model tolerate?

Verification list for availability
Data model

- For the portions of the system that need to be highly available, determine what data abstractions, with their operations and their properties, could cause a fault by omission, a crash, incorrect logging or an incorrect response.
- For these abstractions, ensure that they can be deactivated, be temporarily unavailable or be repaired or masked in case of a fault.
- For example, ensure that write requests are stored in cache if a server is temporarily unavailable and are executed when the server is back online.

Verification list for availability
Correspondence between architectural elements

- Determine what artifacts (processors, communication channels, persistent storage or processes) can cause a fault: by omission, crash, incorrect logging, or incorrect response.
- Ensure that the correspondence between architectural elements is sufficiently flexible to allow for recovery in case of fault.

Verification list for availability Correspondence between architectural elements

- Consider for example:
  - Which process being executed on a defective processor need to be reassigned at runtime.
  - Which processors, data repositories or communication channels can be active or reassigned at runtime.
  - How the data in defective processors or data repositories can be rendered available by replacement units.
  - At what speed can the system be reinstalled based on the provided delivery units.
  - How to reassign at runtime elements to processors, to communication channels and to data repositories.
  - When redundancy tactics are used, the correspondence of modules with redundant components is important. For example, it is possible to write a module that contains code suitable for both the active and the backup components.

Verification list for availability Resource management

- Determine what critical resources are necessary to monitor operations in the presence of a fault: omission, crash, incorrect logging or incorrect response.
- Ensure that there are sufficient available resources in case of a fault to:
  - Maintain a trace of the fault,
  - Warn the appropriate entities,
  - Deactivate the sources of the events that caused the fault,
  - Be temporarily unavailable,
  - Repair or mask the fault/failure,
  - Operate normally during launch, stoppage, in repair mode, in degraded operation mode or in overload.

Verification list for availability Resource management

- Determine the time of availability for critical resources. What critical resources need to be available during what time interval, during which the critical resources can operate in degraded mode, and the time of repair for critical resources. Ensure that critical resources are available during these time intervals.
- For example, ensure that an input queue is sufficiently large to accumulate expected messages if a server fails, in a way that we do not permanently lose messages.

Verification list for availability
Choice of moment to establish a link

- Determine when and how architectural elements are fixed.
- If the elements are fixed at runtime to allow to choose among different components that can themselves be the source of faults (e.g., processes, processors, communication channels), ensure that the chosen availability strategy is sufficient to cover faults introduced by all sources.

Verification list for availability
Choice of moment to establish a link

- For example:
  - If the choice between different artifacts, like processors, is established at runtime and these artifacts need to process or can generate faults, are the mechanisms to detect and recover from faults going to be executed for all possible options?
  - If the choice to define a threshold or a tolerance of what constitutes a fault is fixed at runtime (for example, for how much time a process can run without response and without assuming that there is a fault), is the chosen recovery strategy able to process all cases? For example, if a fault is signaled after 0.1ms, but the recovery time is 1.5s, it constitutes an unacceptable delay.
  - What are the availability characteristics of the selection mechanism itself? Can it fail too?

Verification list for availability
Choice of technology

- Identify available technologies that can (help to) detect the faults, recover from faults or reintroduce components that have failed.
- Identify available technologies that can help to respond to a fault (e.g., a trace registry).
- Identify the availability of characteristics of the technologies themselves: what faults can they recover from? What faults can they introduce in the system?