POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

LOG8430E: Quality Attributes, part 2

Quality attributes – part 2

Definition of quality attributes

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

Quality attribute 2: Interoperability

The interoperability refers to the degree to which two, or more, systems can exchange useful information through interfaces in a given context.

This property implies the ability to correctly interpret the exchanged data (semantic interoperability).

A system cannot be interoperable in an isolated manner. The interoperability implies the identification of the context:

- With what other system?
- What data?
- In what circumstances?

Quality attribute 2: Interoperability

The information exchange between two system can take many forms:

- From more direct: program A calls program B by passing parameters,
- To more implicit: program A is written under the assumptions that program B will produce information in a given format or will have a given behaviour.

The systems or the components often have precise expectations with respect to the behavior of their partners during the exchange of information.

Quality attribute 2: Interoperability

In a context of interoperability, the concept of interface has a much more extended meaning than just a contract over the names of the methods and the types of the parameters.

The interface of an entity needs to include a set of assumptions that need to be ensured concerning this entity.

Example: need for periodic recalibration of an antimissile system used during the war in Iraq.

Quality attribute 2: Interoperability

Typical situations that require interoperability between systems:

- When we provide a service used by multiple systems that are unknown:
  - For example a map service (Google Maps) or a weather prediction service (Accuweather).
- When we compile a new functionality from multiple existing systems.
  - For example:
    1. A system responsible to acquire sensor data in an environment,
    2. A system responsible to process the raw data,
    3. A system responsible to interpret the data, and
    4. A system responsible to represent and distribute the data.
  - For example, a monitoring service for traffic routing.

Quality attribute 2: Interoperability

Two aspects important for interoperability:

- Discovery: the consumer of a service needs to discover (before or after execution) the location, the identity and the interface of a service.
- Response processing (3 possibilities):
  1. The service responds directly to the consumer,
  2. The service sends the response to another system,
  3. The service broadcasts the response to all interested parties.

These two aspects lead to scenarios and tactics for the interoperability.

Quality attribute 2: Interoperability

The systems of systems (SoS): a group of systems that collaborate to achieve a common goal.

Categories of SoS:

1. Directed: The objectives, centralised management, the financing and the authority of the entire SoS are in place. The systems are subordinate to the SoS.
2. Recognized: The objectives, centralised management, the financing and the authority of the entire SoS are in place. The systems conserve their own management, financing and authority, in parallel to those of the SoS.
3. Collaborative: No objectives, centralised management, financing or authority on the level of the SoS. The systems function voluntarily together to achieve common or shared interests.
4. Virtual: Like collaborative, but the systems do not know each other.

Quality attribute 2: Interoperability

In the directed and recognized SoSs, there is a deliberate tendency to create a SoS.

The key difference between the 2 categories resides in the degree of autonomy of the systems, greater in the case of a recognized system.

The collaborative and virtual SoSs are more ad hoc. Absence of a global authority or specific financing.

In the case of a virtual SoS, absence of knowledge of the participants and of the span of the entire SoS.

Quality attribute 2: Interoperability

The case of collaborative SoSs is very common:

- In the case of Google Maps
  - Google is the manager and the authority that finances the map service.
  - Each application using the service has its own management and source of financing.
  - No global management of all applications that use Google Maps.
  - The different applications collaborate to ensure that everything functions well.

A virtual SoS implies large systems and it is often ad hoc:

- Example of electrical network in USA
  - More than 3000 companies for electricity.
  - Public committee for electricity in each state.
  - The federal department of energy imposes certain policies and regulations.
  - Multiple systems of the network need to interact, but there is no central authority.

General scenario of interoperability

Source of stimulus

A system initiates a request.

Stimulus

A request to exchange information between systems.

Artifact

The systems that wish to interoperate.

Environment

The systems that wish to interoperate are discovered at runtime or are known beforehand.

Response

The request to interoperate results in an exchange of information. The information is understood by the recipient both syntactically and semantically. Otherwise, the request is rejected and the appropriate entities are notified. In all cases, the request can be registered.

Measure of the response

The percentage of the requests correctly processed or the percentage of the requests correctly rejected.

Example of concrete scenario of interoperability

Source of stimulus

Information system of vehicle

Stimulus
Current position sent

Artifact
Surveillance system of traffic

Environment
System known before execution.

Response
The traffic surveillance system combines the current position with other information, superimposes it on Google Maps and retransmits the information.

Measure of the response
The information about the position of the vehicle is correctly included 99.9% of time.

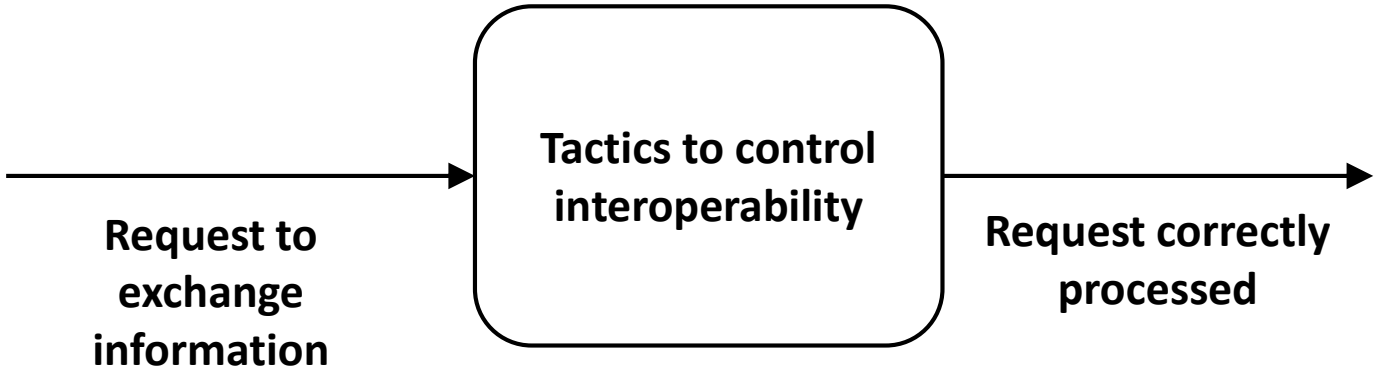
Quality attribute 2: Interoperability

Two modern technologies to allow web applications to interoperate:

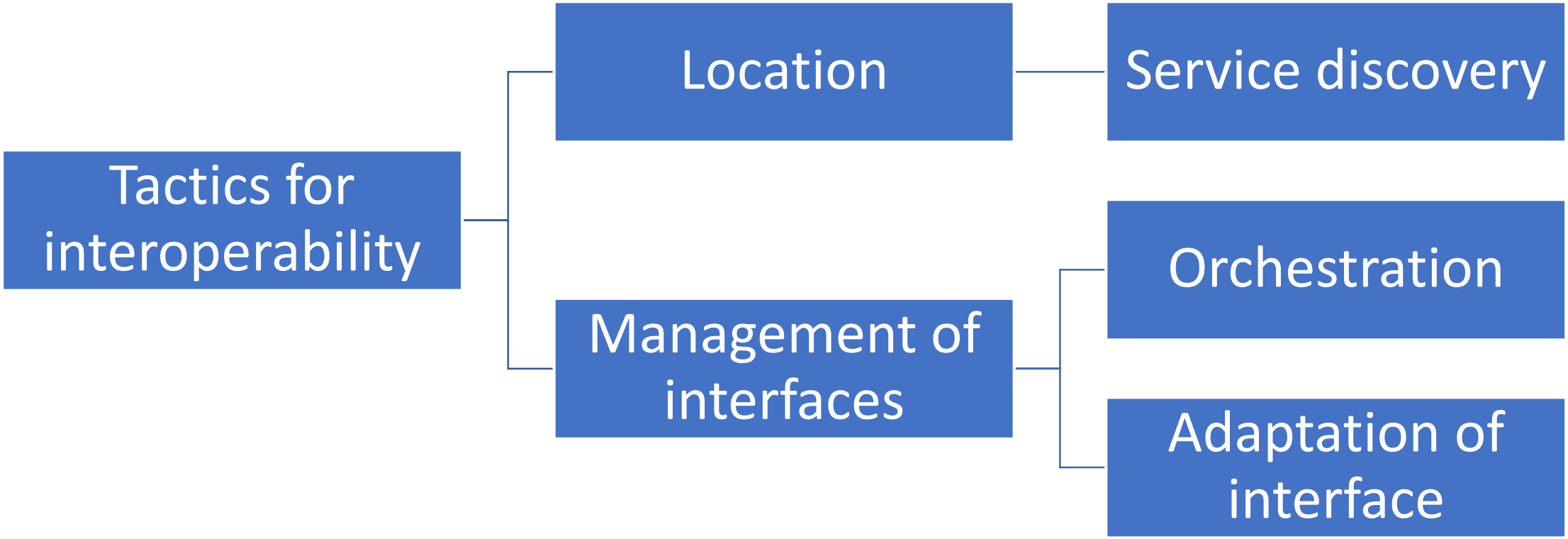
1. SOAP: protocol to exchange data based on XML.
   - Definition of multiple standards for the composition of services, the definition of transactions, the discovery of services, the reliability…
2. REST: protocol to access resources based on the semantics of the basic operations of the HTTP protocol.
   - Operations for creation, read, update and destruction of resources (CRUD).
   - Simple addressing method based on URIs (Universal Resource Identifier).
   - Semantic interoperability not guaranteed by the interface.

Quality attribute 2: Tactics for interoperability

Systems interoperate if they can form requests to exchange information between them, which can be understood by each one.



Quality attribute 2: Tactics for interoperability



Quality attribute 2: Tactics for interoperability: **Location**

A single tactic for the location of services, used to find a service at runtime:

- Service discover: use of a service with known location.
- Can involve multiple levels of indirection.
- Identification of service by:
  - Type,
  - Name,
  - Location,
  - Other criteria.

Quality attribute 2: Tactics for interoperability: **Management of interfaces**

Two tactics for the management of interfaces:

- Orchestration:
  - Put in place by a mechanism to coordinate, manage, and sequence the calls to particular services.
  - Used to allow services to interact in complex fashion.
  - The workflow management engines are examples of this tactic.
  - The Mediator pattern can assume this function in these simple cases.
  - Specification of complicated orchestrations with the aid of BPEL (Business Process Execution Language)
- Adaptation of interfaces:
  - Add or remove capabilities to/from an interface.
  - Capabilities such as translation, push in memory buffer.
  - Conceal certain functions from non-reliable users.
  - The Decorator and Adapter patterns are examples of this strategy.

Quality attribute 2: Interoperability Standards and their limits

In order to agree on the structure and the function concerning the exchange of information, the industry tends to adopt standards.

The standards play a central role to render systems interoperable.

The standards are useful and often indispensable, but the expectations generated by the adoption of standards are often unrealistic.

Quality attribute 2: Interoperability Standards and their limits

Challenges associated with the use of standards:

1. Implementation: each organisation needs to implement its own version of a standard. These implementations are different and often not perfectly compatible with each other.
2. Extensibility: the standards often contain points of extension, which the organisations can use to distinguish themselves, but which can also render the contained information in these extensions incompatible.
3. Lifecycle: the standards evolve and change. Different versions of a standard are not necessarily compatible with each other. Each organisation needs to decide what version of the standard to support at which moment.

Quality attribute 2: Interoperability Standards and their limits

Challenges associated with the use of standards:

4. Specification: the specifications of standards are imperfect. Certain standards are underspecified, others are overspecified, unstable or insignificant.
5. Competition: distinct standards often cover the same information domain. Choosing the best standard is often equivalent to choosing sides within the industry.
6. Delays in innovation: very strict or limited standards reduce the flexibility and expressiveness, which can delay developments.

Quality attribute 2: Interoperability Standards and their limits

The standards cannot help to decide the architecture of a system.

The architecture of a system needs to be decided in the beginning. The supported standards are decided after.

The standards can evolve without affecting the global architecture of the system.

Verification list for interoperability
Allocation of responsibilities

- Determine which responsibilities of the system need to interoperate with other systems.
- Ensure that responsibilities are added to detect the requests that need to interoperate with other systems, known or unknown.
- Ensure that responsibilities are related to:
  - Accepting a request
  - Exchanging information
  - Rejecting a request
  - Notifying the appropriate entities
  - Registering requests (to ensure the non-repudiation in a non-reliable environment, the registration of requests is essential)

Verification list for interoperability
Coordination model

- Ensure that coordination mechanisms can meet the requirements for critical quality attributes. This includes the consideration of the following aspects:
  - The amount of network traffic created by both the systems under your control and the systems that are not under your control.
  - The temporal accuracy of the messages sent by your system.
  - The temporal variance (jitter) in the arrival of messages.
  - Verify that all systems under your control make assumptions about the protocols and network infrastructures that are consistent with the systems that are not under your control.

Verification list for interoperability
Data model

- Determine the syntax and the semantics of all important data abstractions that can be exchanged between the interoperable systems.
- Ensure that all these abstractions are consistent with the data coming from systems, which our system interoperates with. This implies the application of transformations on the data model if it needs to remain confidential.

Verification list for interoperability

Correspondence between architectural elements

- For the interoperability, the critical correspondence is that between software components and processors.
- Moreover, ensure that the components that need to interoperate with other systems are installed on processors that can communicate with the network. Principal considerations include security, availability and performance of the communication.

Verification list for interoperability
Resource management

- Ensure that the interoperability with other systems can never exhaust a critical resource of the system.
- Ensure that the load imposed on resources by the communications related to the interoperability are acceptable.
- Ensure that if the resources need to be shared between participating systems, an adequate arbitration policy is in place.

Verification list for interoperability
Choice of moment to establish a link

- Determine the systems that can interoperate and the moment when they become known to each other.
- For each system on which you have control:
  - Ensure that it has a policy to manage the links to other external systems, known or unknown.
  - Ensure that it has a mechanism to reject unacceptable link requests and to register these requests.
  - In case of dynamic binding, ensure that the mechanisms will support the discovery of services and of appropriate protocols or the emission of information according to the chosen protocols.

Verification list for interoperability
Choice of technology

- For each of the technologies that you have chosen, is it visible at the interface level of the system? If yes, what are the effects on interoperability? Does it support, prevent or is it neutral when interoperability scenarios are applied to your system? Ensure that these effects are acceptable.
- Consider technologies that support the interoperability such as web services. Can they be used to satisfy interoperability requirements for systems under your control?

Quality attribute 3: Modifiability

The modifiability refers to how easy it is for a system to be modified, with what level of risk and at what cost.

A number of studies show that the largest part of the cost of a software system is associated with its maintenance and the changes after the system has been deployed.

To plan in the face of modifiability, an architect should consider four questions:

- What can change?
- What is the probability of change?
- When is the change applied and by whom?
- What is the cost of change?

Quality attribute 3: Modifiability
What can change?

EVERYTHING!

- The operations of the system,
- The platform (hardware, operating system, middleware) on which the system runs,
- The environment in which the system operates:
  - The systems with which it needs to interoperate,
  - The used protocols.
- The quality attributes that the system possesses:
  - Its performance, its reliability and its capacity can be modified!
- Its processing capacity:
  - Number of supported users, number of simultaneous operations.

Quality attribute 3: Modifiability
What is the probability of change?

It is not possible to plan for all types of changes.

Despite the fact that everything can change, the architect needs to make difficult decisions concerning the probable changes, and which can be facilitated, and the improbable ones.

Quality attribute 3: Modifiability
When is the change applied and by whom?

The most frequent changes are the ones happening in the source code.

Today, the question of when a change is made is intertwined with who makes the change.

Changes can happen:

- In the source code,
- In the compilation (e.g., using conditional compilation statements),
- In the construction (e.g., by choosing a library),
- In the configuration (e.g., by changing parameter values),
- In the execution (e.g., by parameters, plugins, etc.).

These changes can be made by developers, users or by system administrators.

Quality attribute 3: Modifiability
What is the cost of change?

Rendering a system easier to be modified implies 2 kinds of cost:

- The cost of introducing a mechanism to render a system easier to be modified.
- The cost to apply the modification using this mechanism.

Quality attribute 3: Modifiability
What is the cost of change?

The simplest change mechanism is to wait until a change request is made to modify the source code:

- No cost to introduce a modification mechanism.
- The cost of the change is that to modify the source code, to test and to deploy the new version.

At the other extreme, we can develop an application generator (e.g., a generator for user interfaces):

- The generator receives as input a modified specification of the system and generates new source code.
- The cost of change comprises of: the cost to develop the generator, the cost to modify the specification, the cost to execute the generator, to test and to deploy the new version.

Quality attribute 3: Modifiability
What is the cost of change?

To justify the introduction of a modification mechanism, the costs related to its development and its usage need to be inferior to those necessary for the modification of the system without the mechanism.

The frequency of changes is a key factor to introduce a modification mechanism.

The prediction of the frequency of changes remains a difficult estimation of high uncertainty:

- If fewer medication than predicted happen, the mechanism will not be justified.
- The necessary time for the development of modification mechanisms could be invested elsewhere (addition of functionality, optimisation, …)

General scenario of modifiability

Source of stimulus

Identity of the one who makes the change:

- Developer
- System administrator
- User

Stimulus

Specification of the change. Addition/change or removal of functionality. Change to a quality attribute of the system (performance, capacity, etc.). Change of technology.

Artifact

Specification of what needs to change: component or module, platform, interface, environment

Environment

Moment when the change can be applied: at design, at compilation, at construction, at installation or at runtime.

Response

Apply the change, test it, deploy it.

Measure of the response

Normally, the time and the money necessary for the change.
The extend of the changes (number of modules, classes, etc. that were modified) or the number of new faults introduced. The effect on quality attributes.

Example of concrete scenario of modifiability

Source of stimulus

Developer

Stimulus
Wish to change the user interface

Artifact
Source code

Environment
At design-time

Response
Change performed and unit tests passed.

Measure of response
In 3 hours.

Quality attribute 3: Tactics for modifiability

Les tactiques pour contrôler la modifiabilité visent à contrôler la difficulté d'effectuer des changements, le temps nécessaire et le coût.

Quality attribute 3: Tactics for modifiability: Control coupling and cohesion

The modules of a system assume responsibilities.

When a change aims at changing a module, this change modifies its responsibilities in some way.

A change that affects a single module is generally simpler and cheaper than a change that affects multiple modules.

If the responsibilities of two modules overlap, a change on one module greatly risks affecting the other one.

We measure the overlap by the probability that a change on one module also implies a change on the other one: that is coupling.

Quality attribute 3: Tactics for modifiability: Control coupling and cohesion

Cohesion measures the degree of relationship that exists among the responsibilities of a module.

Cohesion measures the unity with respect to a goal in a module:

- Related to the number of change scenarios of the module,
- Probability that a change scenario, which affects one responsibility, will also affect another responsibility.
- The more elevated the cohesion, the smaller the probability that a change affect multiple responsibilities.

Quality attribute 3: Tactics for modifiability: Control coupling and cohesion

Parameters justifying the tactics of modifiability:

- Size of a module: the tactics that divide modules will reduce the cost of a modification on a module, to the point where the division reflects the types of changes that are probable.
- Coupling: reducing the strength of coupling between two modules reduces the modification cost for one of the modules. The tactics that reduce coupling introduce intermediaries between the module.
- Cohesion: if a module has low cohesion, we can improve the cohesion by removing responsibilities that are not affected by anticipated changes.

Quality attribute 3: Tactics for modifiability

We also need to consider the moment, in the development lifecycle of a software, when a change is applied.

If we ignore the cost of preparing the architecture to allow for a change, we prefer that the change is applied as late as possible in the software lifecycle.

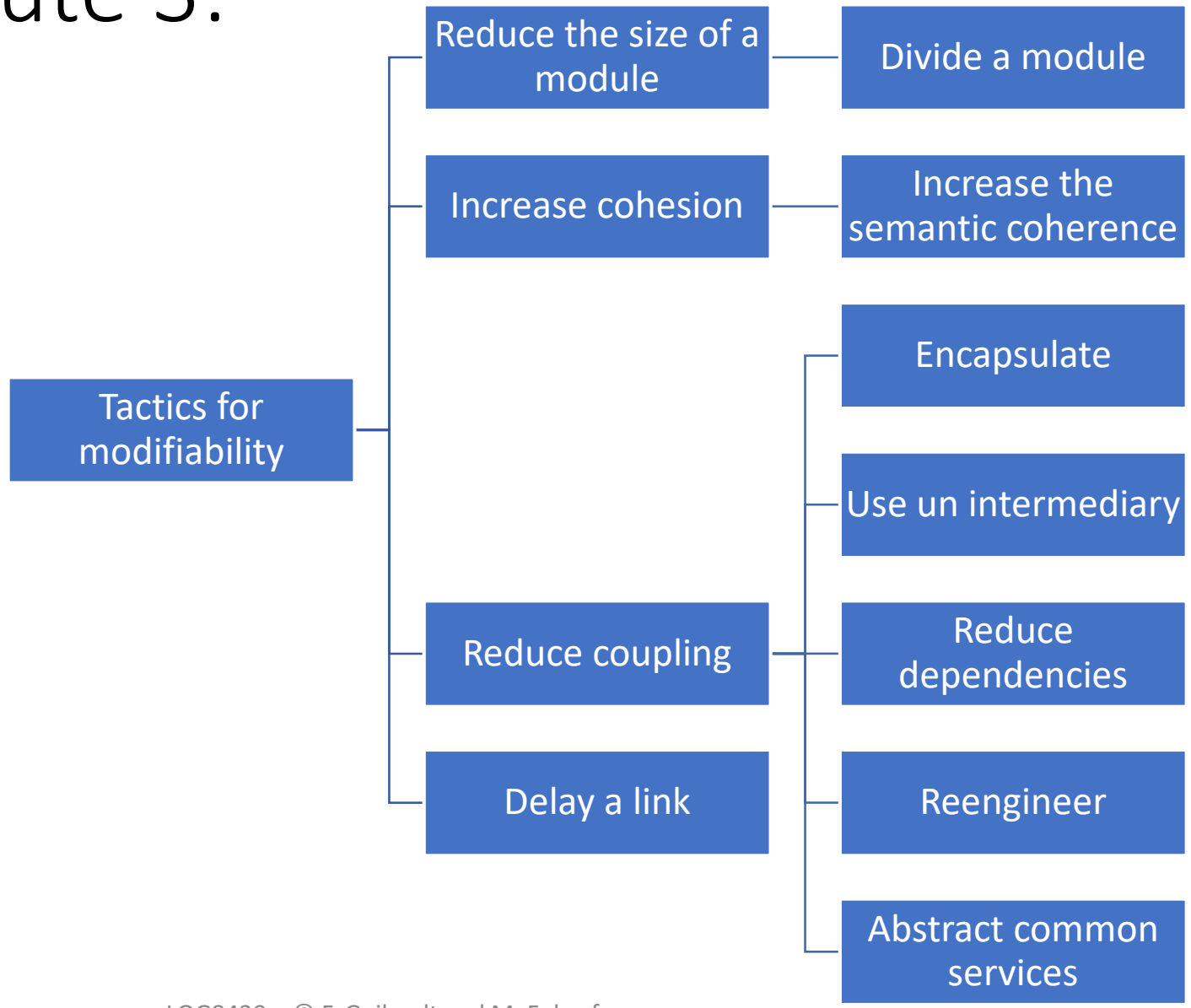
Changes can be easy only if the system architecture has been prepared to allow for the change.

Quality attribute 3: Tactics for modifiability

Another parameter that justifies the tactics for modifiability is the moment when the change is introduced:

- An architecture correctly equipped to allow for late changes in the software lifecycle costs, on average, less than an architecture which forces changes to be applied earlier.
- The preparation of the system is conducted in a way so that cost for late modifications is very low or zero.
- Nevertheless, this ignores the cost to prepare the architecture.

Quality attribute 3: Tactics for modifiability



Quality attribute 3: Tactics for modifiability: **Reduce the size of a module**

If a module that needs to be modified includes a lot of distinct functionalities, the cost of modification risks of being too elevated.

Dividing the module in multiple smaller modules should permit to reduce the cost of future changes.

Quality attribute 3: Tactics for modifiability: Increase cohesion

Many tactics imply the move of responsibilities between modules. The objective being to reduce the probability to introduce undesirable effects in case of modification.

Increase semantic coherence: if two responsibilities in the same module do not have the same goal/objective, these responsibilities should be placed in distinct modules.

- Moving the responsibilities may require the creation of a new module, or to move a responsibility to an existing module,
- To identify the responsibilities to be moved, we can seek that which is most probable to change. If the responsibilities are not affected by these changes, then they should probably be moved.

Quality attribute 3: Tactics for modifiability: **Reduce coupling**

Encapsulate: Introduce an explicit interface for a module.

- The interface includes an API and the associated responsibilities (data transformations, etc.)
- The encapsulation reduces the probability that a change to a module will propagate to other modules.
- The coupling with the module is transferred to the interface, but often in a reduced manner since the interface limits the ways to interact with the module.
- The external responsibilities can only interact with the module through its interface.
- The designed interfaces to improve modifiability should be abstract with respect to the details of the module that will change. These details should be hidden.

Quality attribute 3: Tactics for modifiability: **Reduce coupling**

Use an intermediary: an intermediary hides the dependencies.

- The type of intermediary depends on the type of the dependency:
  - An intermediary of type observer (publish-subscribe) eliminates the data producers' knowledge of the consumers.
  - A shared data repository eliminates the data readers' knowledge of those who write the data.
  - In a SOA, where services are discovered at run-time by dynamic search, the service repository is an intermediary.

Quality attribute 3: Tactics for modifiability: **Reduce coupling**

Restrain the dependencies: restrain the modules with which a module can interact or on which it depends.

- Consists of reducing the visibility of a module and of implementing accessors.
- This tactic is employed by layered architectures. A layer can only access inferior layers and often only immediately inferior layers.
- Use of a wrapper, where the entities can only depend on the wrapper and not on that which is wrapped.

Quality attribute 3: Tactics for modifiability: **Reduce coupling**

Reengineering: stage of cleaning the code in order to improve quality.

- An important practice in development approaches that are based on agile methodologies.
- Ensure that there is no code duplication or complex code.
- The concept can be also applied to the system architecture:
  - Common responsibilities (and the associated code) are extracted from modules where they belong and they are assigned to a module of their own.
- By regrouping common responsibilities, the architect can reduce the coupling.

Quality attribute 3: Tactics for modifiability: **Reduce coupling**

Abstract common services: construct a more abstract module to fuse two very similar, but not identical, modules.

- When two modules provide similar services, but identical, it can be more efficient from a cost perspective to implement the service once in a more general way.
- Modifications to the service should happen only to one place.
- A modern approach to introduce an abstraction consists of parameterizing the description and the implementation of the activities of the module.
- The parameters can be as simple as values for certain key variable or as complex as expressions in a special language to be interpreted.

Quality attribute 3: Tactics for modifiability: **Delay a link**

Designing artifacts using a certain degree of flexibility, and then using this flexibility, is generally cheaper than manually coding a specific change.

Parameterization is the most current mechanism to add flexibility.

When we choose the value of a parameter during a phase in the software lifecycle different than the one in which the parameter was introduce, we apply the tactic of **delaying a link**.

Quality attribute 3: Tactics for modifiability: **Delay a link**

The later a value can be linked in the lifecycle, the more flexible the system will be.

Introducing mechanism to delay links has a cost: so, we want to link values as late as possible if the introduction of the mechanism to do so is profitable.

“Outsourcing” is a change to allow a third party (user or installer) to make a change without changing the code.

Quality attribute 3: Tactics for modifiability: **Delay a link**

Tactics to establish a link

At compilation

- Replacing components (build script or makefile)
- Parameterization at compilation
- Aspects

At deployment

- Configuration files/Wizards

- At launch or at initialization
  - Resource files

At runtime

- Registration at runtime
- Dynamic search (e.g., services)
- Interpreted parameters
- Dynamic binding (e.g., dll)
- Name servers (DNS)
- Pluggable architectures
- Publish-subscribe (observer)
- Shared repositories
- Polymorphism

Verification list for modifiability
Allocation of responsibilities

Determine what changes or categories of change are susceptible of occurring concerning technical, legal, social, business or client stakeholders which affect the system:

- Determine the responsibilities that should be added, modified or removed to apply the change.
- Determine which responsibilities are affected by the change.
- Determine an allocation of responsibility for modules that places, as much as possible, the responsibilities that are expected to change (or affected by the change) together in the same module, and those that will change at different times in separate modules.

Verification list for modifiability
Coordination model

- Determine what functionality or quality attribute can change at runtime and in what way this affects coordination.
  - For example, do the communicated information or the protocol of communication change at runtime? If yes, ensure that these changes affect only a small set of modules.
- Determine what devices, protocols and communication channels used for coordination are susceptible to change.
  - For these devices, protocols and communication channels, ensure that these changes affect only a small set of modules.
- For the elements for which modifiability is at stake, use a coordination model that reduces coupling, like publish-subscribe, defers the link, like an enterprise service bus or restrains the dependencies, like an emission mechanism (broadcast).

Verification list for modifiability
Data model

- Determine what changes to data abstractions, their operations or their properties are possible and probable.
- Determine what changes to these abstractions will involve their creation, initialisation, storage, manipulation, translation or destruction.
- Determine, for each type of change, if they will be applied by users, system administrators or developers.
- For changes applied by users or system administrators, ensure that the necessary attributes are visible and that the users have necessary privileges to modify data, operations or their properties.

Verification list for modifiability
Data model

- For each type of change identified:
  - Determine what data abstractions should be changed,
  - Determine what operations over the data should be changed,
  - Determine what other data abstraction will be affected,
  - Ensure that the allocation of data minimises the number and the severity of modifications to abstractions.
- Design the data model in a way so that items allocated to each model element are changed together.

Verification list for modifiability
Correspondence between architectural elements

- Determine if it is desirable to change the way in which functionality is allocated to computational elements (processes, execution queues, processors) at runtime, at compilation, at design-time, or during construction.
- Determine the amplitude of modifications necessary to allow for adding, removing or modifying a function or a quality attribute. This can involve the determination, for example, of:
  - The dependencies at runtime,
  - The assignment of databases,
  - The assignment of execution elements to processes, execution queues or processors.
- Ensure that these changes are affected by mechanisms that use late binding correspondence decisions.

Verification list for modifiability
Resource management

- Determine how the addition, removal or modification of a responsibility or a quality attribute can affect the utilisation of a resource. For example:
  - Determine what changes can introduce new resources, remove old ones or affect the utilisation of existing resources,
  - Determine what limits over the resources will change and how.
- Ensure that the resources after the modifications are sufficient to meet the requirements.
- Encapsulate all resource managers and ensure that the policies implemented by these managers are also encapsulated and the bindings are delayed as much as possible.

Verification list for modifiability
Choice of moment to establish a link

- For each type of change:
  - Determine the latest possible moment when a change can be applied,
  - Choose a moment to delay the binding that provides the appropriate capacity for the chosen moment,
  - Determine the cost to introduce a mechanism and the cost to apply a change using the mechanism. Verify the profitability.
  - Do not introduce binding mechanisms so that the changes are prohibited due to complex or unknown dependencies between the choices.

Verification list for modifiability
Choice of technology

- Determine what modifications are rendered simpler or more difficult by your choices of technology:
  - Will your choices of technology help to apply, test and deploy the changes?
  - How easy will it be to change your technology choices (if certain technologies change or become obsolete)?
- Choose your technologies to support the most probable changes:
  - For example, the introduction of an enterprise service bus can facilitate to change how the elements are connected, but possibly create a dependency to a specific commercial provider.