
WIT 2016 ITA Module

Architecture Principles Lecture Group #3 - Part 1



Lecture Group #3 - Part 1

Architecture Principles

WIT 2016 ITA Module
Lecture Group #3 - Part 1
Architecture Principles

Fundamental Design Principles



Prescriptive Heuristics

A heuristic technique, often called “heuristic”, is any approach to problem solving, learning, or discovery that employs a practical method not guaranteed to be optimal or perfect, but sufficient for the immediate goals. Heuristics are used to speed up the process of finding a satisfactory solution. Heuristics can be mental shortcuts that ease the cognitive load of making a decision.

Architectural Styles and Patterns are heuristics for treating complex, unbounded problems. Architectural Styles and Patterns are abstractions of experience that help the transition from descriptive (problem space) to prescriptive (normative, solution space) - i.e. provide a bridge between concept and implementation, synthesis and design, system to subsystem thinking.

Styles and Patterns are codified, succinct expressions from lessons learned through community experience. It is a key modeling and learning technique for designers architects.



"Patterns of Play"

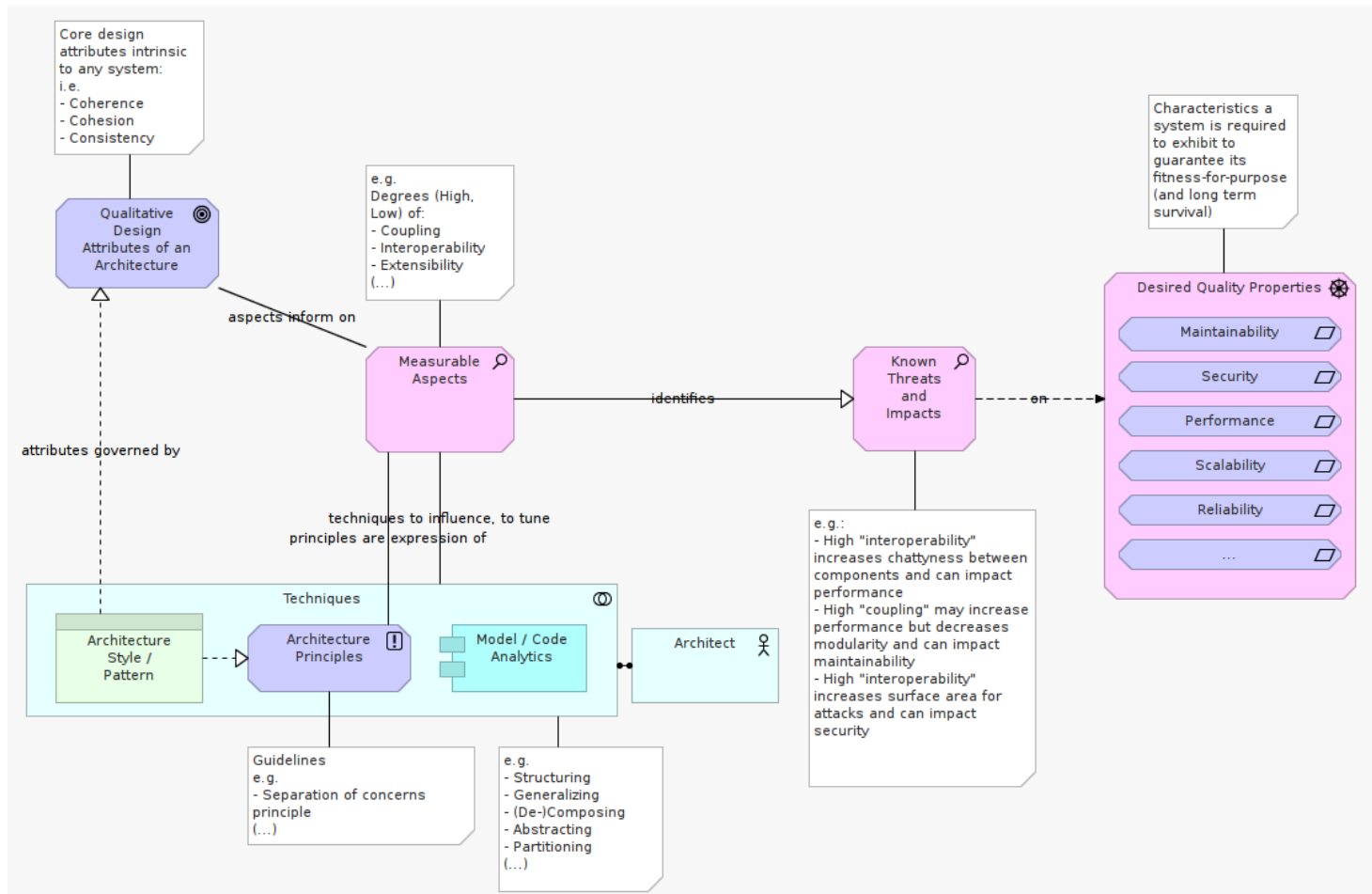
A Chess Master does not think all possible moves ahead, they see a "pattern of play" or "pattern of positions" and add insight and experience to know a probable outcome.

An architect judges of the eligibility of a candidate Architectural Style from the environmental constraints, and the principles and guidelines that support the application of the Style.

An Architect is good at selecting a Style by matching desired intrinsic design attributes to desired quality properties of the architecture, and figuring out from ASPECTS that are relevant from details that are irrelevant.



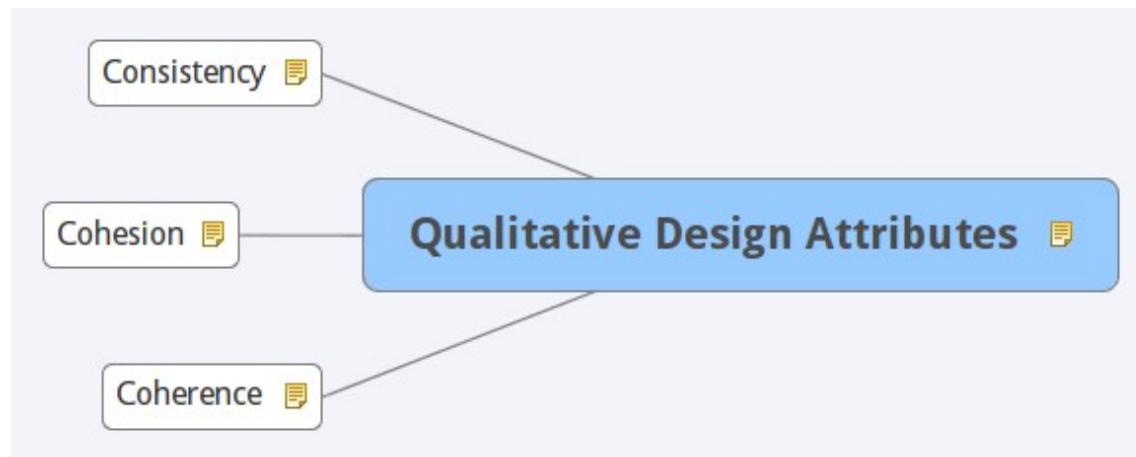
“Patterns of Play”



Qualitative Design Attributes

Coherence, Cohesion and Consistency are three fundamental design attributes of any system. These attributes are implicitly enforced in architecture styles.

An architecture style has a predictable form and way of functioning, at all levels throughout the design, helping reviewers to know what to expect without necessarily having to delve into the documentation, or reverse engineer design from code.



Coherence

Coherence is about FORM.

Within a coherent architecture, elements are organized and ordered in a logical structure according to their relationship to one another.

Coherence brings a visible pattern of organization in the structure of elements working together to form PARTS of a bigger whole.

- Smaller, granular parts, are more generic and re-usable (down to common technical building blocks or elements with a specific utilitarian focus).
- Bigger, coarse-grain parts, are more specific to a bounded context of use (aggregation of building blocks in a domain model, or coordination of use case flows for example).

Coherence is visible in the modeling of compositions and dependencies between elements.

A coherent design is easier to understand and easier to build.

A coherent component or system is easier to extend.



Cohesion

Cohesion is about FUNCTION.

It is about making sure each component does one thing and does it well, avoiding overlap of concerns and duplication of functionality.

Within a cohesive architecture, elements are brought together (and given authority to) to perform a set of operations realizing a defined (bounded) function.

- Example of high component cohesion: A customer maintenance component implementing CRUD operations on Customer data entities, and notifying about changes pertaining to customer details.
- Example of low component cohesion: Same as above but overloaded with CRUD operations on order/purchases performed by a customer, but that have nothing to do with customer maintenance.
- A component exhibits high cohesion when all its functions/methods are strongly related in terms of function.

A cohesive design is easier to figure-out and simulate.

A cohesive component or system is easier to debug and test.



Consistency

Consistency is about INTERACTION MECHANISMS.

Within a consistent architecture, mechanisms for information exchange between elements are of a similar nature.

There is a visible pattern in the way elements interact.

The same set of interaction mechanisms is applied consistently throughout the architecture.

- A set of defined contract interfaces governs the outputs and inputs of interacting elements.
- A set of defined policies govern the way by which elements communicate with each other.

Consistency is visible in the modeling of element conversations and means of data exchange.

A consistently designed and implemented system is much easier to build, test, operate, and evolve.

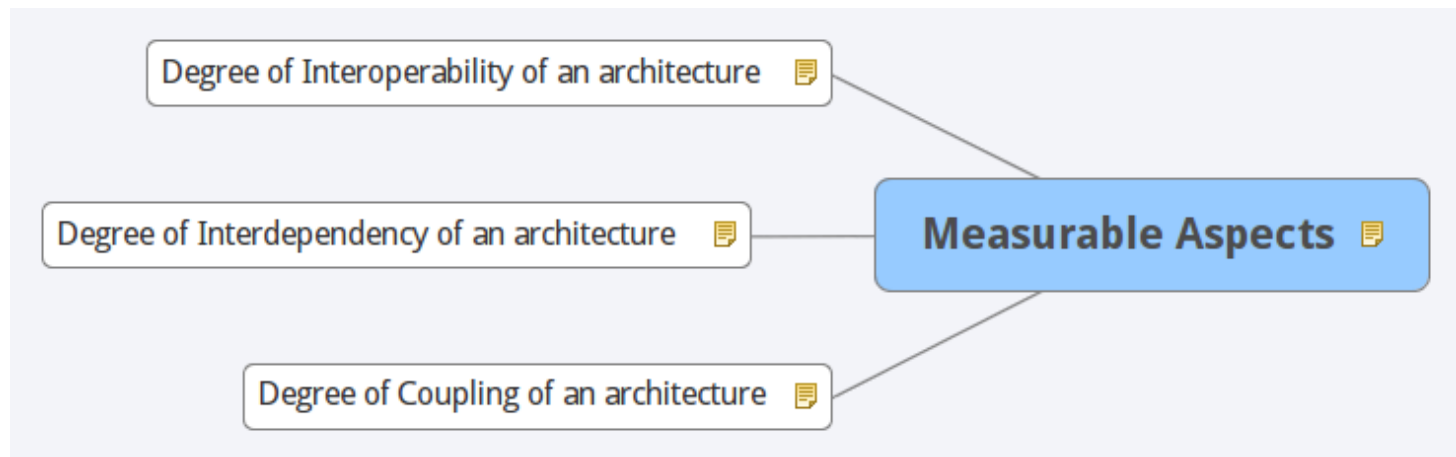


Measurable Aspects

In Software, components are units of delivery that can be independently replaced and upgraded. This is a remarkable feature of software architecture, which constitutes a fundamental principle which must not be broken.

All measurable aspects of Coherence, Cohesion and Consistency pertain to this principle.

Measurable aspects are comparable to the symptom of a sick patient. For example, being forced to redeploy an entire set of system in a 3 days long integrated release because a minor change in the system needs to be deployed.



Degree of Coupling of an architecture

Measures of Coupling:

- % of components that can be deployed independently without impacting other components
- % of component that can change without affecting other already deployed components



Aspects of high coupling

In a tightly-coupled (highly-coupled) architecture, each component and its associated components must be present in order for code to be deployed.

Examples:

1 component relies on the inner workings of another component

2 components share the same global data (via an in-memory global variable)

1 component controls the flow of another component, by passing data on what to do (the more control parameters causing the target component to take a different action/path, the tighter is the binding)

2 modules are tied together into one module because of a timing dependency

Any situation of bi-directional or circular dependencies within the code of an application i.e library dependencies are valid examples of tight-coupling.



Aspects of low coupling

In a decoupled (loosely coupled, or weakly coupled) architecture, a change in one component never (or rarely) necessitate a change in the others.

For example, components can operate completely separately and independently using services running in separated processes, and communicating with networking mechanisms such as HTTP. Doing so, each components can remain autonomous and let middleware software to manage communication between them.

Examples:

- 2 component share data through parameters. Each piece of data is elementary, what the target strictly needs to function, and the only shared data.
- 2 components share a composite data structure and use only a part of it, possibly a different part
- 2 component are bound together via a shared data context (data store, database)
- 2 modules collaborate but are neither tied nor bundled together into one deployable unit, or process.



Informs/Impacts

Coupling measures & metrics inform about the Cohesion and Coherence of an architecture.

Example of Low coupling impact:

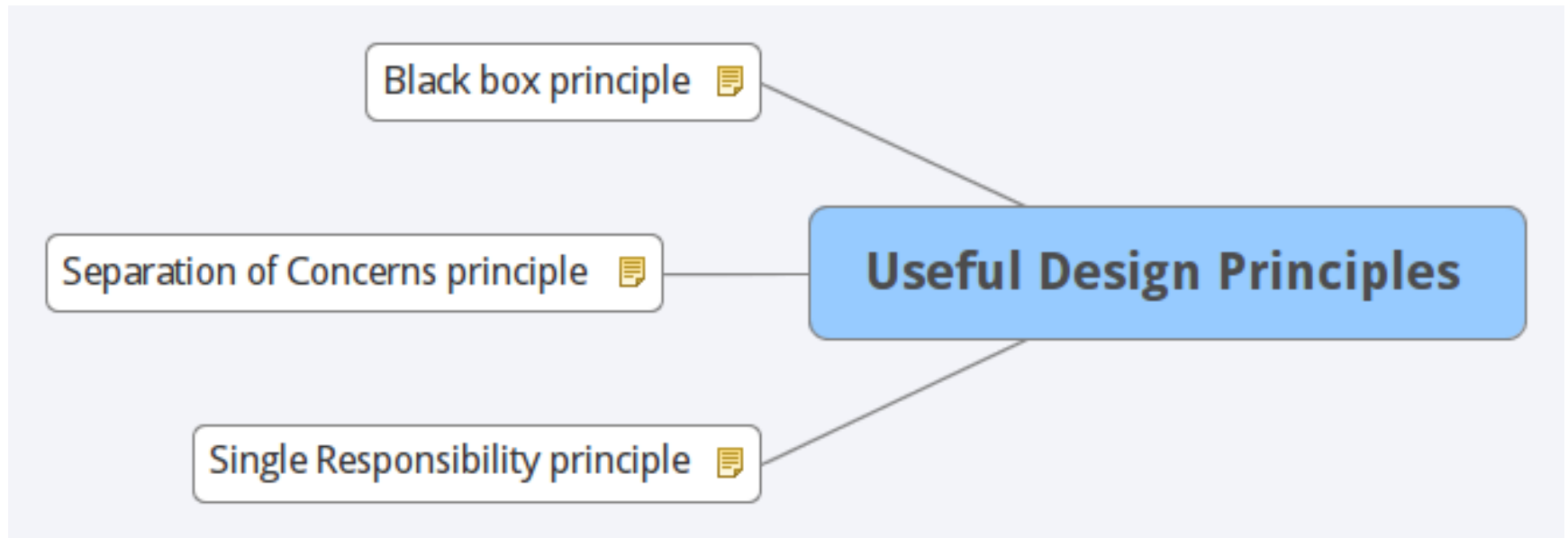
- Performance ■: Communicating between certain types of element can be more an order of magnitude more expensive in terms of processing time and elapsed time. Loosely coupled systems are often easier to build, support and enhance but may suffer from poor efficiency compared with a monolithic approach.

Example of High coupling impacts:

- Performance ■: Function calls in same process, less interdependency, less coordination, less information flow.
- Maintainability ■: Changes cause ripple effect within the code.
- Scalability ■: Force a “big-bang” monolithic component deployment every time a change is introduced in the application.



Useful Design Principles



Single Responsibility principle

Each component or module should be responsible for only a specific feature or functionality, or aggregation of cohesive functionality.

You should only need to specify intent in one place.

Specific functionality should be implemented in only one component; the functionality should not be duplicated in any other component.



Separation of Concerns principle

Break your application into distinct features that overlap in functionality as little as possible.

The main benefit of this approach is that a feature or functionality can be optimized independently of other features or functionality.

If one feature fails, it will not cause other features to fail as well, and they can run independently of one another.

This approach also helps to make the application easier to understand and design, and facilitates management of complex interdependent systems.



Black box principle

A component should not know about internal details of other components.

A component or an object should not rely on internal details of other components or objects.

Each component or object should call a method of another object or component, and that method should have information about how to process the request and, if appropriate, how to route it to appropriate subcomponents or other components.

This helps to create an application that is more maintainable and adaptable.



Degree of Interdependency of an architecture

Measures of Interdependency:



- % point-to-point connections between components of an architecture
- % of components dependent of another component to achieve its intent
- % of components required to perform the complete scope of an operation
- % of components going over a threshold of size, conciseness



Informs/Impacts

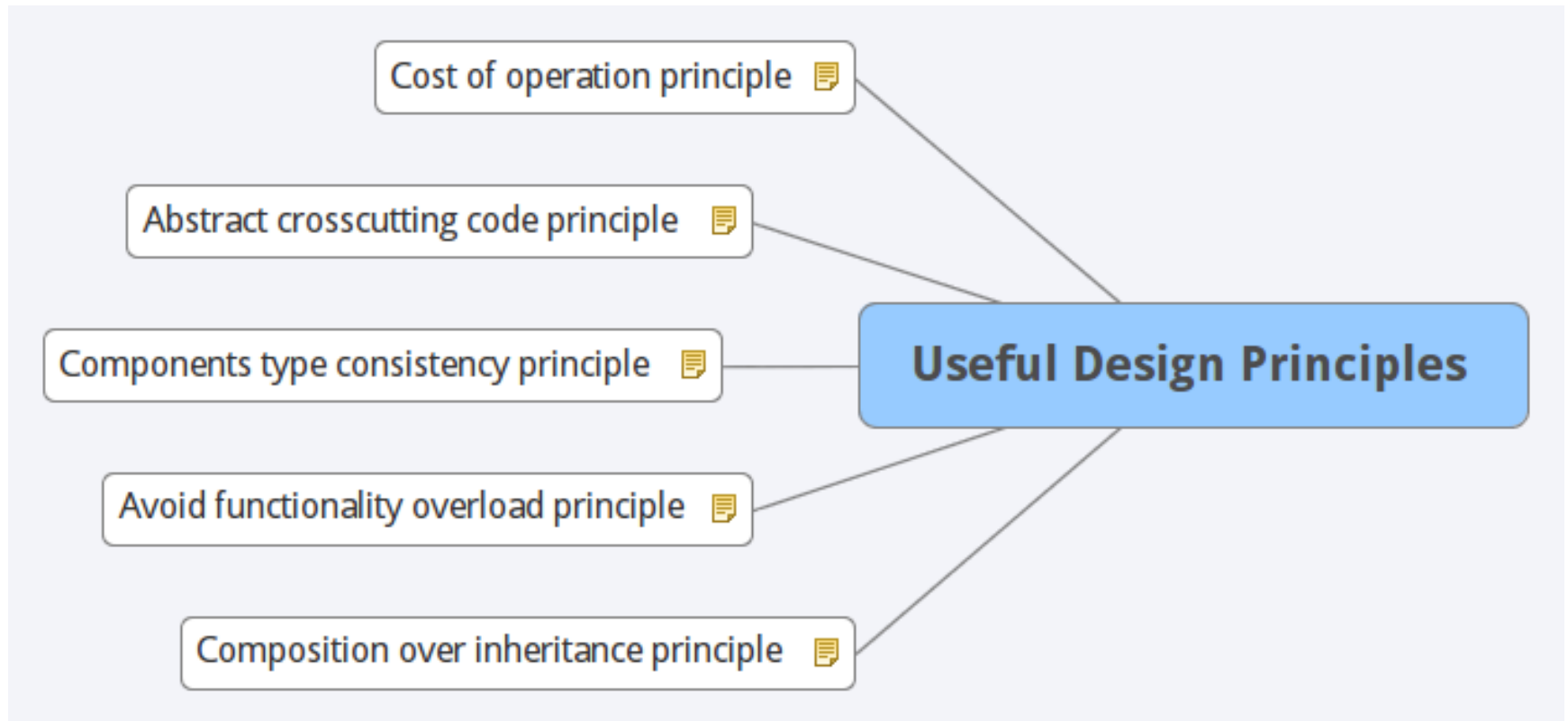
Interdependency measures & metrics inform about the Coherence of an architecture.

Examples of High interdependency impacts:

- Reliability : Reduce brittleness of an application since operations an operation within a same element / process.
- Maintainability : Changes in the component interface require changes to already implemented and deployed components.



Useful Design Principles



Composition over inheritance principle

Use composition over inheritance when reusing functionality.

Inheritance increases the dependency between parent and child classes, thereby limiting the reuse of child classes.

This also reduces the inheritance hierarchies, which can become very difficult to deal with.



Avoid functionality overload principle

A UI processing component should not contain data access code or attempt to provide additional functionality.

Overloaded components often have many functions and properties providing business functionality mixed with crosscutting functionality such as logging and exception handling.

The result is a design that is very error prone and difficult to maintain.



Components type consistency principle

Start by identifying different areas of concern, and then group components associated with each area of concern into logical layers. For example, a UI layer should not contain business processing components, but instead should contain components used to handle user input and process user requests.



Abstract crosscutting code principle

Keep crosscutting code abstracted from the application business logic as far as possible. Crosscutting code refers to code related to security, communications, or operational management such as logging and instrumentation. Mixing the code that implements these functions with the business logic can lead to a design that is difficult to extend and maintain.



Cost of operation principle

Consider the operation of your application. Determine what metrics and operational data are required by the IT infrastructure to ensure the efficient deployment and operation of your application. Designing your application's components and sub-systems with a clear understanding of their individual operational requirements will significantly ease overall deployment and operation.



Degree of Interoperability of an architecture

Measures of interoperability:

- % of components dedicated to communicate/exchange data with other components (e.g. by means of service facade)
- % of components that use the information that has been (or to be) exchanged (e.g. marshalling, un-marshalling)
- % of processing logic of 1 component involving interaction with other components



Informs/Impacts

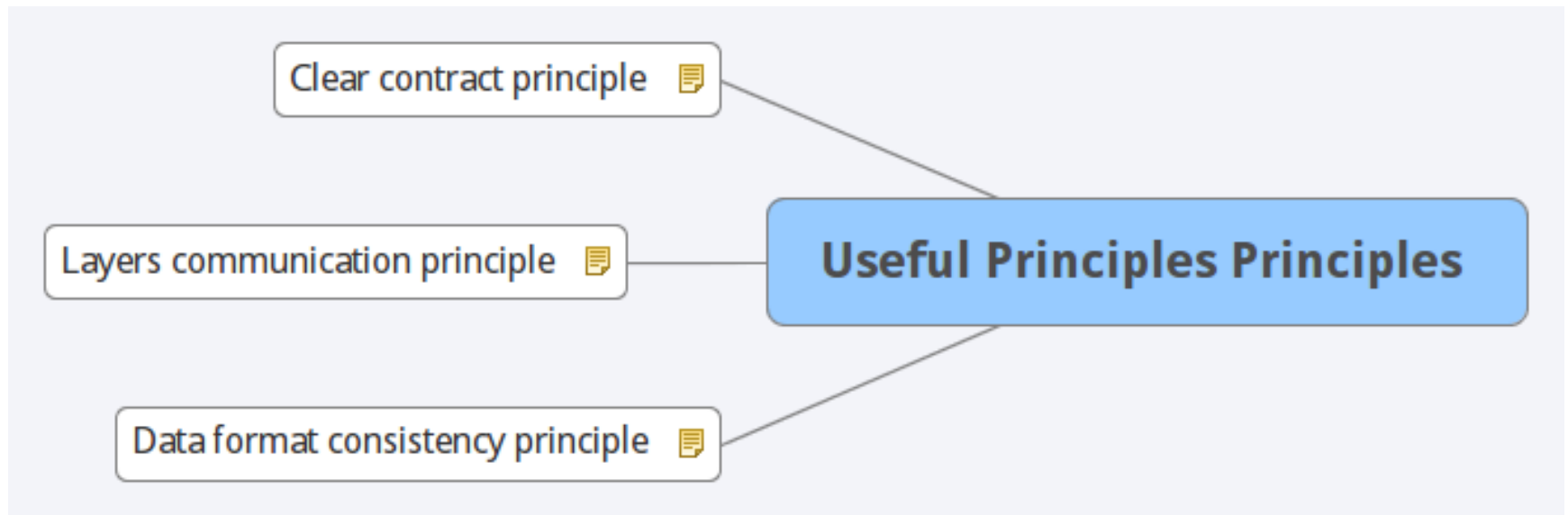
Interoperability measures & metrics inform about Cohesion and Consistency.

Examples of High interoperability impacts:

- Maintainability ■: Extensibility points in the architecture allow the system to perform new functions in the future.
- Performance ■: Chatty-ness between components over the network is increased
- Reliability ■: Brittleness of whole system is increased, less fault tolerant, difficult to find the source of a problem, or where a problem can occur
- Reliability ■: Teams may make wrong assumptions about the behavior of collaborating components they didn't develop
- Security ■: The surface area of the architecture is increased for each service provided, multiplying angles of attack.



Useful Principles Principles



Data format consistency principle

Keep the data format consistent within a layer or component. Mixing data formats will make the application more difficult to implement, extend, and maintain. Every time you need to convert data from one format to another, you are required to implement translation code to perform the operation and incur a processing overhead.



Layers communication principle

Be explicit about how layers communicate with each other. Allowing every layer in an application to communicate with or have dependencies upon all of the other layers will result in a solution that is more challenging to understand and manage. Make explicit decisions about the dependencies between layers and the data flow between them.

Understand how components will communicate with each other. This requires an understanding of the deployment scenarios your application must support. You must determine if all components will run within the same process, or if communication across physical or process boundaries must be supported—perhaps by implementing message-based interfaces.



Clear contract principle

Define a clear contract for components. Components, modules, and functions should define a contract or interface specification that describes their usage and behavior clearly. The contract should describe how other components can access the internal functionality of the component, module, or function; and the behavior of that functionality in terms of pre-conditions, post-conditions, side effects, exceptions, performance characteristics, and other factors.



Useful Design Activities

Generalizing: Are the mechanisms of the architecture and decisions made general enough as they are also practicable?

Partitioning: To what extent each internal element is responsible for a distinct part of the system's operation? To what extent is common processing performed in only one place?

Structuring: Is a visible structure of groupings, packages for component emerging from your architecture?

Composing: How are components combined a coarse-grained level? What are the rules of composition of your design?

Decomposing: How are components breaking down the design into several smaller and re-usable entities?

Abstracting: Is your architecture hiding detail to isolate the coarse-grained parts from underpinning infrastructure elements?

Interfacing: What are the types of connectors available in your architecture to connect or integrate components? Are these connectivity mechanisms sufficient in terms of speed?

