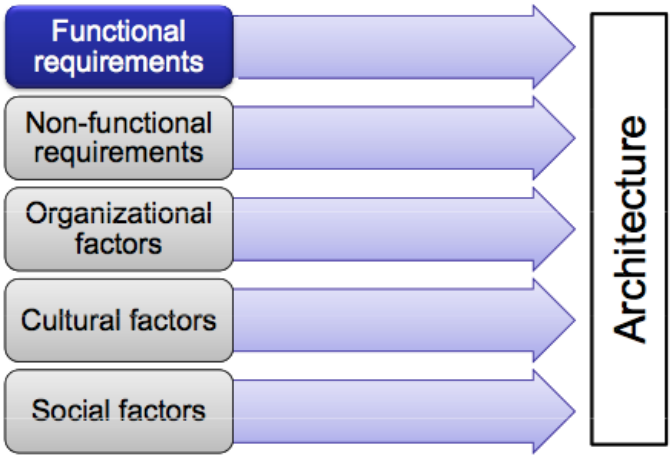
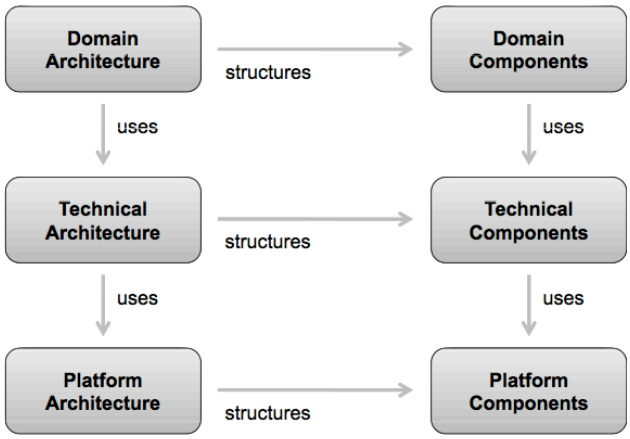
*Influencing Factors:*




*Domain Architecture:*

- reflects the problem space for which the system is developed
  - driven by the character of the domain
  - driven by the functional requirements



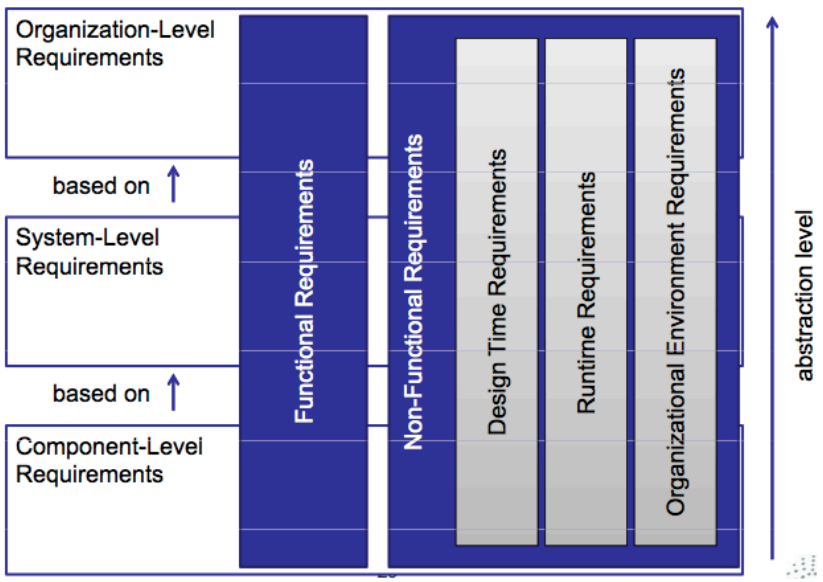

*Technical Architecture:*

- is domain-neutral and deals with realizing non-functional requirements (logging, security, auditing, data consistency, …)
- Technical components use services of the platform

*Platform Architecture:*

- provides services for executing software components

## Kinds of Architectural Requirements



*Functional Requirements:*

- define the functionalities needed (what a system is supposed to accomplish)

*Non-Functional Requirements:*

- define expectations or criteria that can be used to judge the operation of a system
- NFRs with direct implications: quality attributes (performance, reusability, …)
- NFRs with indirect implications: Standards, parameters or conditions that must be considered or reached (Budget, Regulations, Legislatives, Business Policies, …)

Relevance of NFRs:

- Fulfilling the NFRs is essential to the acceptance of the functionality of the system
- Neverthelesse NFRs are often neglected (Architects' task: Sensitize the stakeholders for the importance of the NFRs)

## Architectural Principles

Coupling:

- measures the dependencies **among** components of an architecture
- characterizes the interactions of the components

*Principle of Loose Coupling:*
The coupling of compontents should be kept **as low as possible**.
1st Goal: **Keep the complexity of structures low** (the less a component is coupled with other components, the easier it is understandable on its own)
2nd Goal: **Increase the changeability of the architecture** (the less components are affected by a change in a component, the easier it is to do a change locally in a component without looking at the environment)

Cohesion:

- measures the dependencies **within** a component

*Principle of High Cohesion:*
The cohesion of a component should be **as high as possible.**
The goal is to support understandability and changeability of components. (if a component contains all elements that are needed to understand or change it, it can be understood or changed without looking at other components)

Interdependence: the higher the cohesion of individual components, the lower the coupling between them.

*Principle of Design for Change:*
means to plan for foreseeable changes when designing an architecture. Requirements that are likely to arise can be considered during planning.
Risks: More development time, higher costs, more implementation effort
More flexible architectures often consume more resources (memory, performance)

*Separation of Concerns Principle:*
Different aspects of a problem should be separated from each other, and each aspect of the problem should be treated on its own.

Main Kind of Separation of Concerns: Modularization
Other Kinds: Separation of Requirements, Separation of a complex architecture model into views, separation of organizational responsibilities

*Information Hiding Principle:*
Present to the client only the necessary parts of the whole information and hide the rest.
Goal: Reduction of the complexity of a SW architecture
Example: Hiding implementation details behind an interface

*Other Important Architecture principles:*

- Abstraction Principle: use abstractions to make a complex problem understandable
- Modularization Principle
- Traceability Principle
- Self Documentation Principle
- Incremental Evolution Principle

## Architectural Styles And Patterns

- An architectural style describes **a fundamental structure of a software systems and its properties**
- An architectural style consists of the following elements:
  - **Component types** that fulfill specific functions at runtime
  - A **topology** of these components
  - **Connectors** that manage the communication and coordination between the components
  - **Semantic constraints** that determine how the components and connectors can be composed
- Patterns are **principal solutions** for **recurring problems**
  - A pattern can be used not only for one specific problem, but a **whole range** of concrete problems
  - It gives clear advice **how** to solve the problem
  - But in a concrete design situation the generic solution in the pattern must be **tailored** to the problem at hand
- The architect should know the important design and architectural patterns in his area to **avoid "reinventing the wheel"**
- Patterns can be used by architects to **name** recurring architectural structures
  - They are also used for documentation and communication
- Patterns don't describe new ideas, but **time-proven solutions**
  - For every pattern (at least three) **known uses** should exist in real systems

In SW architecture, both design patterns and architectural patterns are important. Design patterns describe rather **specific, local design decisions.** Architectural patterns describe problems and solutions the architect would judge as **principal design decisions**.

A single pattern only describes a solution for one recurring problem. In many cases, we face **multiple design problems** that often occur together and have **relationships** to each other. Hence, patterns are often not presented in isolation.

A **pattern language** is a **collection of patterns** that systematically explains **how to apply a number of patterns in combination**. Patterns in a pattern language are applied

in an incremental refinement process. The decision making in this process is based on pattern's forces.

## Important Architectural Styles And Patterns

### Patterns for Layered Decomposition, Data Flow and Data Repositories

*Layers Pattern:*
The system is structured into layers so that each layer provides a set of services to the layer above and uses the services of the layer below. Within each layer all constituent components work at the same level of abstraction and can interact through connectors. Between two adjacent layers a clearly defined interface is provided.

*Indirection Layer Pattern:*
An indirection layer is a layer between the accessing component and the instructions (e.g.: API, public interfaces) of the sub-system that needs to be accessed. The indirection layer wraps all accesses to the relevant sub-system and should not be bypassed. It can perform additional tasks such as converting or tracing the invocations.

*Batch Sequential Pattern:*
In a Batch Sequential architecture the whole task is sub-divided into small processing steps, which are realized as separate, independent components. Each step runs to completion and then calls the next sequential step until the whole task is fulfilled. During each step a batch of data is processed and sent as a whole to the next step.

*Pipes and Filters Pattern:*
In a Pipes and Filters architecture a complex task is divided into several sequential subtasks. Each of these sub-tasks is implemented by a separate component, a filter, which handles only this task. Filters have a number of inputs and a number of outputs. They are connected flexibly using pipes. Each pipe realizes a stream of data between two components. Filters consume and deliver data incrementally. Pipes act as data buffers between adjacent filters.

*Shared Repository Pattern:*
Shared Repository is used as a central data store, accessed by all other independent components:

- It offers suitable means for accessing the data, for instance, a query API or language
- It is scalable to meet the clients' requirements
- It must ensure data consistency
- It must handle problems of resource contention, for example by locking accessed data
- It might also introduce transaction mechanisms

An Active Repository is a shared repository variant in the sense, that it **informs a number of subscribers** from its **client registry** through appropriate **notification mechanisms** of specific events that happen in the shared repository.

A complex task is divided into smaller sub-tasks for which deterministic solutions are known. The Blackboard is a shared repository variant that uses results of its clients for **heuristic computation** and **step-wise improvement** of the solution. Each client can access the Blackboard to see if new inputs are presented for further processing and to

deliver results after processing. A control component monitors the blackboard and coordinates the clients according to the state of the blackboard.

### Adaption Infrastructure Patterns

*Microkernel Pattern:*
A Microkernel realizes services that all systems, derived from the system family, need and a plug- and-play infrastructure for the system-specific services. Internal servers (not visible to clients) are used to realize version-specific services and they are only accessed through the Microkernel. External servers offer APIs and user interfaces to clients by using the Microkernel and are the only way for clients to access the Microkernel architecture.

*Reflection Pattern:*
In a Reflection architecture all structural and behavioral aspects of a system are stored into meta-objects and separated from the application logic components. Thus Reflection allows a system to be defined in a way that allows for coping with unforeseen situations automatically.

*Plugin Pattern:*
Plugins resolve both rebuilding and redeploying problems by providing a centralized configuration that can be performed at runtime. Plugin components also introduce an interface, but at a single, central point so that configuration can be easily managed. They can be connected and disconnected at runtime rather than during compilation.

*Interceptor Pattern:*
An Interceptor is a mechanism for transparently updating the services offered by the framework in response to incoming events. An application can register with the framework any number of Interceptors that implement new services. The framework facilitates this registration through dispatchers that assign events to Interceptors.

### Language Infrastructure Patterns

*Interpreter Pattern:*
An Interpreter for a language provides both parsing facilities and an execution environment. The program that needs to be interpreted is provided in form of scripts which are interpreted at runtime. These scripts are portable to each platform realization of the Interpreter.

*Virtual Machine Pattern:*
A Virtual Machine defines a simple machine architecture on which not machine code but an intermediate form called the byte-code can be executed. The language is compiled into that byte-code. The Virtual Machine redirects invocations from a byte- code layer into an implementation layer for the commands of the byte-code. The Virtual Machine can be realized on different platforms, so that the byte-code can be portable between these platforms.

*Rule-Based System Pattern:*
A Rule-Based System offers an alternative for expressing logical problems in a system. It consists mainly of three things: **facts**, **rules**, and an engine that acts on them. **Rules** represent knowledge in form of a condition and associated actions. **Facts** represent data. A Rule-Based System applies its rules to the known facts via its rule engine. The actions of a rule might assert new facts, which, in turn, trigger other rules.

## Interaction Decoupling Patterns

*Model-View-Controller Pattern:*
The system is divided into three different parts:

- a **Model** that encapsulates some application data and the logic that manipulates that data, independently of the user interfaces
- one or multiple **Views** that display a specific portion of the data to the user
- a **Controller** associated with each View that receives user input and translates it into a request to the Model

The users interact strictly through the Views and their Controllers, independently of the Model, which notifies all user interfaces about updates.

*Explicit Invocation Pattern:*
An Explicit Invocation allows a client to invoke services on a supplier, by coupling them in various respects. The decisions that concern the coupling (e.g. network location of the supplier) are known at design-time. The client invokes the supplier using the service name and parameters. The Explicit Invocation mechanism performs the invocations and delivers the result to the client.

There are two main variants of Explicit Invocations:

- **Synchronous, explicit invocations:** The client blocks until the result is available
- **Asynchronous, explicit invocations:** The client continues with its work immediately. The result is delivered at a later point and afterwards computed.
    - **Fire and Forget:** best effort delivery semantics but doesn't convey results or acknowledgments
    - **Sync with Server:** describes invocation semantics for sending an ACK back to the client once the operation arrives on the server side, but doesn't convey results
    - **Poll Object:** describes invocation semantics that allows clients to poll/query for the results of asynchronous Invocations.
    - **Result Callback:** describes invocation semantics that allow the client to receive results. In contrast to Poll Object the client is actively notified.

*Implicit Invocation Pattern:*
The invocation is not performed explicitly from client to supplier, but indirectly through a special mechanism that decouples clients from suppliers, such as:

- Publish-subscribe
- Message queuing
- Broadcast

## Patterns for component interaction and distribution

*Client-Server Pattern:*
The Client-Server pattern distinguishes two kinds of components: **clients** and **servers**. The client requests information or services from a server. It requires an ID or an address of the server and the server's interface. The server responds to the requests of the client, and processes each client request on its own. It does not know about the ID or address of the client before the interaction takes place. Clients are optimized for their application task, whereas servers are optimized for serving multiple clients.

3-Tier Architecture:

- a **client tier**, responsible for the presentation of data, receiving user events, and controlling the user interface
- an application **logic tier**, responsible for implementing the application logic (also known as business logic)
- a **backend tier**, responsible for providing backend services, such as data storage in a data base or access to a legacy system

*Peer-to-Peer Pattern:*
Each component has equal responsibilities, in particular it may act both as a client and as a server. Each component offers its own services (or data) and can access the services in other components. The Peer-to-peer network consists of a dynamic number of components. Before a component can join a network, it must get an initial reference to this network. This is solved by a bootstrapping mechanism, such as providing public lists of dedicated peers or broadcast messages (using Implicit Invocation) in the network announcing peers.

*Publish-Subscribe Pattern:*
Publish-Subscribe allows event consumers (**subscribers**) to **register for specific events**, and event producers to **publish** (raise) **specific events** that reach a specified number of consumers. The Publish-Subscribe mechanism is triggered by the event producers and automatically executes a callback operation to the event consumers. The mechanism thus takes care of decoupling producers and consumers by transmitting events between them.

*Broker Pattern:*
A Broker separates the communication functionality of a distributed system from its application functionality. The Broker hides and mediates all communication between the objects or components of a system. Brokers are the main architectural concept used to realize Middleware-Architectures. A **middleware** is a platform which provides **services for all aspects of distribution.** It **realizes the distributed connections of components,** using e.g. Client/Server, Peer-to-Peer, Publish/Subscribe.

## Modeling Architectures

To be able to **automatically process model data**, it must be **described precisely** in a formal language. To achieve this, models are typically described through models, which are then called **meta-models**. A model is hence the **instance** of its meta-model. The cascade of abstraction by creating a meta- model for models can be continued arbitrarily, leading to a number of **modeling-levels.**

Meta-models are also used to **describe modeling languages**. Meta-models define the structure or **abstract syntax** of the modeling language. In addition a **concrete syntax** is needed. Finally, the modeling languages require **semantics** that describe the meaning of the models.

*Architecture Description Language (ADL):*
An ADL is a modeling language specifically designed to describe software and/or system architectures. Properties of ADLs:

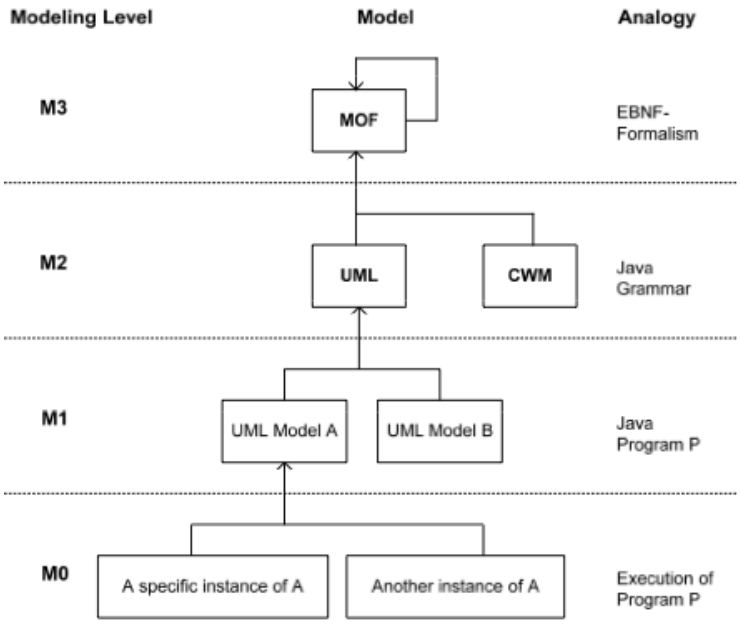
- **Formal representation** of the architecture using a textual or graphical notation on a very high abstraction level
- **Readable by humans and machines**

- Can by **analyzed** with regard to architectural properties, such as consistency, completeness, performance, etc.
- Sometimes, support for **code generation**

The main concepts of ADLs are:

- **Components**: Syntactic or semantic **specification of functional or non-functional aspects** of an architectural elements using interfaces. The **exported** and **imported** interfaces of a component are described.
- **Connectors**: Components communicate via connectors that define **how components interact with each other**.
- **Architectural Configurations**: Describes the architectural structure by defining **how components and connectors are connected** with each other.

*Unified Modeling Language – Meta Object Facility / UML – MOF:*
The MOF provides basic meta-model elements to build meta-models. The Object Management Group's (OMG) meta-modeling architecture defines 4 modeling levels:




The UML specification itself is split into the **UML Infrastructure** and the **UML Superstructure** specifications. The **UML infrastructure** defines elements used in both the meta-meta-model of UML (MOF) and the superstructure. The **UML meta-model** (i.e. the language definition) is defined in the **UML superstructure**. The infrastructure is merged into the superstructure.

UML supports modeling the **component & connector view,** import and export of interfaces, architectural configurations. **Little support for specific architectural properties**, e.g. for analysis, as in ADLs. **Little support for modeling system integration** because system integration concepts are missing.

According to the UML standard there are two ways to extend the language:

- the hard extension produces an **extension of the language meta-model**, i.e., a new member of the UML family of languages is specified
- the soft extension results in a **profile**, which is a set of **stereotypes, tag definitions,** and **constraints** that are based on existing UML elements with some extra semantics according to a specific domain

- A **stereotype** can extend any element (**meta-class**) of the meta-model (new types of classes, relationships, …).
- (OCL) **Constraints** can be used to formally define the semantics of the stereotyped meta-classes.
- Stereotypes can have a **custom image** for the concrete syntax.

## Model-Driven Design and Development

**Domain-Driven Design and Development** means to make software development **domain-related** and not (just) SW-technology-related.

- Goal: Make software development in a certain domain more efficient
- This is achieved by mapping (business) domain concepts into software artifacts using a **domain model**

A domain model establishes a **common model between the (business) domain and IT stakeholders**.

*Domain-Specific Languages (DSLs):*
A DSL is a tailor-made (computer) language for a specific problem domain. DSLs are often not Turing complete and only provide abstractions suitable for one particular problem domain. This specialization results in significant gains in expressiveness and ease of use.

An **Embedded DSL** (also called internal DSL) is defined as an extension to an existing General-Purpose-Language (GPL; e.g. Java, Python, Tcl, …):

- It uses the syntactic elements of the underlying GPL
- It can directly access all features and tools of the host language
- Instead of using a generator, often embedded DSLs are interpreted
- Typical examples of embedded DSLs are DSLs in scripting languages and dynamic languages, such as Ruby, Python, Tcl, or Perl

An **External DSL** is defined in a different format than the intended target language(s):

- It can use all kinds of syntactical elements
- DSL designers may define any possible syntax, be it textual or graphical
- An external DSL is not bound to a certain host language or platform
- It can be mapped to target platforms via transformations

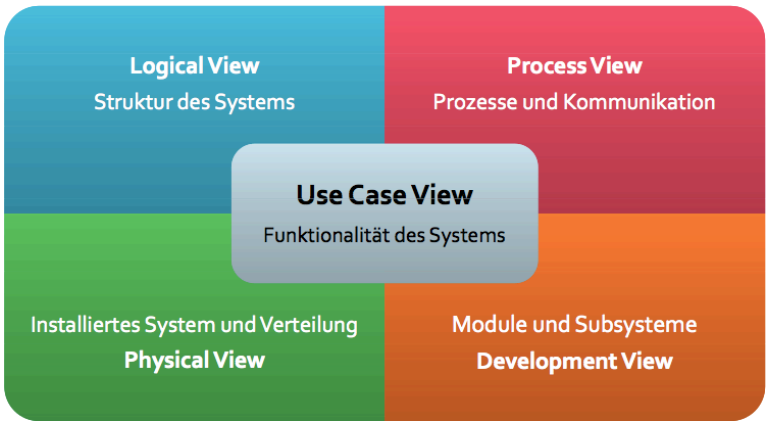
In Model-Driven-Software-Development (MDSD) a developer/designer creates **models** based on a meta-model, that are often expressed using a DSL. **Code generation** is used to transform the model into executable code. MDSD requires creating an infrastructure consisting of modeling tools, generators, platforms, etc and a lot of effort for domain analysis. These **efforts usually not pay off for using the MDSD infrastructure only once**, multiple incarnations of an MDSD infrastructure are more feasible. This often leads to **software product lines**.

Benefits of a Custom Architecture Meta-Model/DSL:

- **Communicating the architecture among stakeholders** can be specifically supported and eased
- It can support **architecture documentation**:

- The custom architecture meta-model can serve as a **glossary of architecture concepts** used
- Its models can serve as a **glossary of the components**, **interactions**, etc.

- The act of formalizing the architecture often means to **clarify** the architecture, too.




- The approach lends toward **technology-neutral architecture definition**.

## Architectural Views

*4+1 View Model:*
Use **Case View**: Central in the 4+1 View Model

- All architectural decisions are based on use cases and scenarios of the system in focus
- Foundation of all other architectural views
- Used for validating the other views
- This view can be described using case diagrams and use case specifications

The **logical view** describes the (object-oriented system) system in terms of abstractions, such as classes and objects. The **development view** describes the structure of modules, files and/or packages in the system. The **process view** describes the processes of the system and how they communicate with each other. The **physical view** describes how the system is installed and how it executes in a network of computers.

## Architectural Decisions

Often strategic and operational decisions are distinguished. Architectural decisions have a strategic nature, as they have a long-term and significant influence on the system. Architecture is the **result of a series of decisions**:

- Architect must decide which functional or non-functional requirements are **prioritized**
- Architecture is always a **compromise**

Steps during decision making:

- **Prepare** decision
- **Make** decision
- **Communicate** decision
- **Realize** decision
- **Evaluate** decision

Decision Modeling:

Model architectural decision explicitly using a **decision template** or **meta-model.** Capture the key design issues and the rationale behind the decision:

- **Conscious design decisions** concerning the architecture
- Consider the **impact on nonfunctional requirements** and **quality factors**

## Architecture in the Organization

The organization culture defines values and norms. These are a normative frame for designing an architecture. They define how individuals behave among each other and which expectations the organization has for its members. Often an architecture fails because it does not reflect certain values and norms of the organization. Organization hence influences architecture as a discipline. Sometimes organization cultures vary in sub- organizations (Internationalization, outsourcing).

*Influences on the Organization Structure:* Conway's Law:

- …organizations which design systems … are constrained to produce designs which are copies of the communication structures of these organizations

Organizational distance makes communication harder, in particular: The integration of a number of systems into one system, especially in large organizations, is difficult for many project teams. Often the influence of the organization is not considered (enough) in architectures. A number of approaches help by focusing on the human in the organization and seeing humans as a motivated individuals: **Organizational Patterns and Principles**, **Agile Software Development** and Project Management Methods (Scrum, XP, …)

Architects must have social competences, because every project member has its own individual background, understanding and preferences. Thus, architects must understand the behavior and interactions of individuals. The composition of the group is essential for success. The group should be led by an experienced, cooperative moderator, team members should be appointed according to their skills. One or two creative team members who produce new ideas are usually absolutely necessary.

Architects requires skills and experiences in many different areas. The TOGAF Architecture Skills Framework includes the following main categories of skills:

- **Generic Skills**: leadership, teamworking, inter-personal skills, etc
- **Business Skills and Methods**: business cases, business process, strategic planning, etc
- **Enterprise Architecture Skills**: modeling, building block design, applications and role design, systems integration, etc.
- **Program or Project Management Skills**: managing business change, project management methods and tools, etc.
- **IT General Knowledge Skills**: brokering applications, asset management, migration planning, SLAs, etc.
- **Technical IT Skills**: software engineering, security, data interchange, data management, etc.
- **Legal Environment**: data protection laws, contract law, procurement law, fraud, etc.