EVOLUTE

Solution Architecture Framework and Methodology

Version 1.5

E
V
O
L
U
T
E

EVOLUTE

I keep six honest serving-men
(They taught me all I knew);
Their names are What and Why and When
And How and Where and Who.
I send them over land and sea,
I send them east and west;
But after they have worked for me,
I give them all a rest.

I let them rest from nine till five,
For I am busy then,
As well as breakfast, lunch, and tea,
For they are hungry men.
But different folk have different views;
I know a person small
She keeps ten million serving-men,
Who get no rest at all!
She sends em abroad on her own affairs,
From the second she opens her eyes
One million Hows, Two million Wheres,
And seven million Whys!

-- Rudyard Kipling, The Elephant's Child

E V O L U T E

Versions

Version	Date	Modified By	Modifications
1.0	07/01/2014	Peter De Kinder	Initial Version
1.1	23/05/2014	Peter De Kinder	Rebranding of template
1.2	30/06/2014	Peter De Kinder	Elaboration of technical design
1.3	17/02/2015	Peter De Kinder	PMBOK association, Infrastructure elaboration
1.4	04/11/2015	Peter De Kinder	Solution Arch Scope, test strategy elaboration
1.5		Peter De Kinder	

References

Document	Date	Description
NIKSPUR	2011	The Black Art of Software Estimation (Dmitry Nikelshpur)
DAMSGRD	2010	Seven Principles for Selecting Software Packages (Jan Damsgaard)

Prerequisites

Document	Description

E V O L U T E

Table of Contents

1 Introduction

When Rudyard Kipling published “The Elephant’s Child” in 1902, he scarcely could have imagined a hundred years later, it would be used as a frequent tool for describing whatever needs to be described. Its concepts can be found in numerous methodologies, the most famous amongst them being the Zachman Framework for Enterprise Architecture. The poem preaches that any approach to a situation elaborate on the “What”, the “Why”, the “When”, the “How”, the “Where” and the “Who”, and it should do so in such a way that they can only be interpreted in a unique way.

In the following chapters, our approach to Solution Architecture is explained in detail, both how to structure it and how to live it and make it part of the project culture. We define solution architecture as activity spanning over the project effort as well as the service management effort as a way of optimizing the chances of getting successful results.

1.1 The Big Picture

Illustration 1.1 – Phases of a Project

Every solution has its proper life cycle that runs through 4 separate phases, as shown in the illustration above. Each of these phases has its own activities that break up these phases into manageable parts each assigned to a group of stakeholders responsible for their resolution.

The Plan phase concerns itself with identifying and documenting the broad contour of the project. Coarse-grained requirements, usually focused on the triple constraint of project management, are penned down, and a global planning with milestones is drawn up. We mainly concern ourselves with the goals of the project in this phase, so that all parties know what we are working towards from the start.

The Build phase is where the action of the getting down to business starts. We elaborate the business needs into sets of measurable (SMART) requirements, and design architectures that cover all of these, which we in turn realize by implementing a solution, and testing it thoroughly. This is where the bulk of the work for the technical teams in charge of developing the solution, happens.

The Operate Phase is where we reap the benefits of the implementation, and the solution becomes a solution-in-use. The project switches from active development to maintenance and change management. This operational phase will be the true test to see whether the goals set forth by the stakeholder in the first and second phase have been met.

Every solution has a finite life before which it becomes obsolete. Either because the needs of the stakeholders have changed or the implementation cost of a replacement solution is starting to dive under the maintenance cost of the current solution. This final phase is about how the dispose of the current solution, while still retaining all relevant value the solution still possesses. Most of the time, this comes down to transferring all relevant information still present in the solution to the replacement solution, and repurposing the resources freed by the disposal.

Another poet of British origin once said “No Man is an Island, entire of Itself”, and the same can be proclaimed about Solution Architecture. In itself, it does not amount to much, needing the entirety of a solution development process to complement it. As far as the phases of a project goes, it is conceived either in the Build phase or the Plan Phase, serves as input and quality verification for both the Build and Operate Phase, and finally becomes a point of reference to launch the dispose phase.

1.2 Sphere of Knowledge

Just as the Solution Architecture is part of a bigger picture, so should the architect's skills not reside in a solitary realm. An architect should possess a knowledge base of varying depth in several different areas. As noted in the illustration, the most prominent of these are Project Management Frameworks (such PMBOK or Prince2), Testing Methodologies (such a TMAP), the Software Best Practices, Business Architecture, Functional Design and the Service Management Frameworks (such as ITIL).

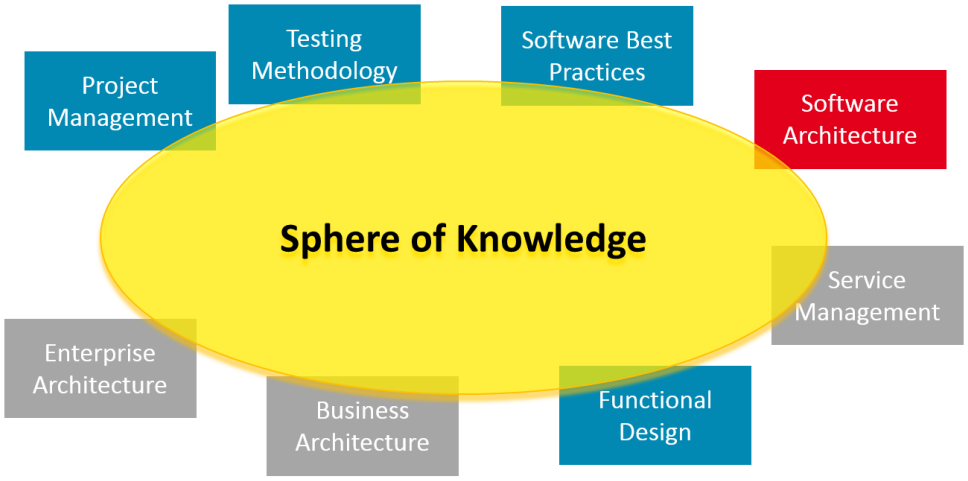


Illustration 1.2 – Sphere of Knowledge

The depth of the knowledge needed in each of these topics is described in a way similar to Bloom's Taxonomy of Learning for the Cognitive Domain (1956). We acknowledge 5 levels of depth, to be sequentially mastered per topic. These levels are listed in Illustration 1.3 (see below). Each of the topics mentioned in the previous paragraph can be attributed one of these levels to form the basis of a good architect.

Knowledge Domain	Level of Understanding
Project Management	EDUCATED
Testing Methodology	EDUCATED
Software Best Practices	EDUCATED
Service Management	UNDERSTOOD
Business Architecture	UNDERSTOOD
Enterprise Architecture	UNDERSTOOD
Functional Design	EDUCATED
Solution Architecture	COMMITED

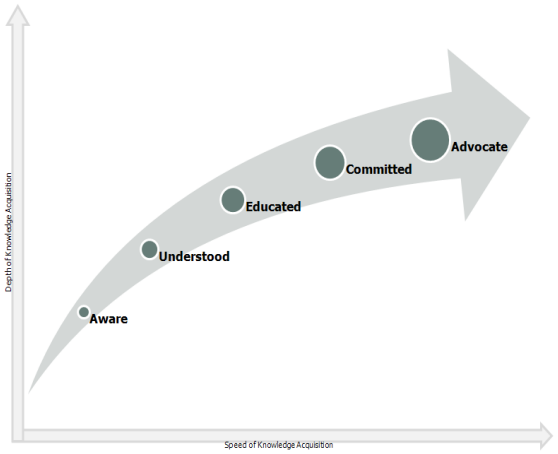


Illustration 1.3 – Levels of Understanding

2 What is Solution Architecture

2.1 The Definition

The most widely accepted definition of solution architecture comes from work done by the Software Architecture group of the Software Engineering Institute (SEI) at Carnegie-Mellon University in Pittsburgh:

> *The architecture of a software-intensive system is the **structure or structures** of the **system**, which comprise software **elements**, the externally visible **properties** of those elements, and the **relationships** among them.*

An architectural element (or just element) is a fundamental piece from which a system can be considered to be constructed. It is often referred to as **components** or **modules.** The nature of an architectural element depends very much on the type of system and the context you are considering. Programming libraries, subsystems, deployable software units (e.g., Enterprise Java Beans and Active X controls), hardware components (e.g., firewalls, proxy servers, routers…), reusable software products (e.g., database management systems), or entire applications may form architectural elements in an information system, depending on the system being built. Architectural elements are often known informally as components or modules, but these terms are already widely used with established specific meaning. For this reason, we deliberately use the term element to avoid confusion.

The external visible properties of an architectural element are a set of properties that owned solely by the element and identify the element. Often these properties are expressed as interfaces, contracts, specifications (functional and non-functional) of an element (component or module).

The inter-element relationships indicate how the elements are connected to each other. The most common relationships are expressed as connectivity, dependencies, ownerships, extensions, compositions, aggregations, usages, communications and/or constraints.

The illustration below lays out the architecture conception framework proposed by IEEE 1471-2000, which is often used during the architectural design of the software engineering process as a common language. We adapt these concepts to align with the industry standards and to attain a common language for architecture related discussions.

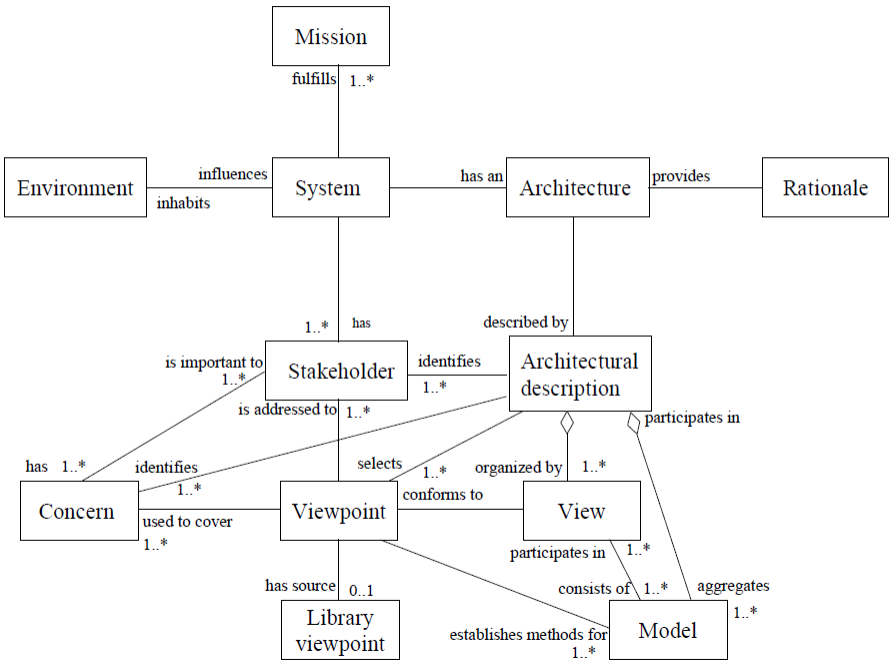


Illustration 2.1 - Conceptual model of architectural description from IEEE 1471-2000

A quick glossary of the different terms used in Illustration 2.1 is listed in the table below.

Mission	The business purpose of the project and architecture
System	A computer system, which is consisted of hardware and software (applications)
Environment	Environment determines the settings and circumstances of developmental, operational, political, and other influences upon that system. An environment determines the boundaries that define the scope of the system of interest relative to other systems.
Architecture	Architecture describes the structure of elements, which is used to build the system.
Rationale	The underlying principles and justification
Stakeholder	Concerned group of people or elements, which have impact or can be impacted on by the system
Architectural description	Written documents which describe the architecture of the system
View	An instance of representation of a whole architecture from the perspective of a related set of concerns
Viewpoint	The viewpoint defines the languages and methods to use when describing such views
Concern	Concerns are those interests which pertains to the system's aspects that are critical or otherwise important to one or more stakeholders

2.2 Formalized Deliverables

Architecture description is a complex task. Often architecture must answer multi-dimensional problems; must take into consideration the different concerns of the different stakeholders; must make trade-offs; and in most cases a single "right" architecture doesn't exist (often there are more than one "right" architecture).

A description of architecture has to present the essence of the architecture and has to be detailed enough at the same time. In other words, it must provide an overall picture that summarizes the whole system, but it must also decompose into enough details so it can be validated and the described system can be built. It is a major challenge to achieve the right balance between the essentials and the detailing in an architectural description.

To represent complex systems in a way that is manageable and comprehensible by a range of business and technical stakeholders, we use the only successful and widely used approach: we partition the architecture description into a number of separate but interrelated **views**, each of which describes a separate aspect of the architecture. Collectively, the views describe the whole system.

An **architectural view** is a way to portray those aspects or elements of the architecture that are relevant to the concerns that the view intends to address, and by implication, the stakeholders for whom those concerns are important. Formally, addressed by IEEE 1471, quoted hereunder:

> *A view is a representation of one or more structural aspects of an architecture that illustrates how the architecture addresses one or more concerns held by one or more of its stakeholders.*

We describe the architecture with the following views:

- **Requirements view**
  The architecturally significant requirements, both functional and non-functional, are listed, ranked and matched to the descriptions in later views. This view acts as a checklist and justifies correctness and completeness of the architecture by measuring the coverage of the functional and non-functional requirements.

- **Logical view**:
  The architecture is described by using a decomposition of the system into logical layers, subsystems and components of the system in the logical view. This does not only include the internal subsystems, components and the relations between them, but also the dependencies to the external systems and components.

- **Implementation view**
  This view describes how the technological architecture supports and constraints the software development process. It consolidates decomposition of the logical presentation of the architecture to modules, standardization of design, development and testing. It translates the conceptual model of the system into something that can be realized as a concrete implementation by mapping it to implementation solutions.

- **Physical view**
  We discuss all the environments into which the system will be deployed, concerning the constraints of hardware, third-party software, network, and system topology, as well as the cooperation between hardware and software. Additionally, we stipulate the different protocols used to communicate with the outside world.

- **Operational view**
  The operational view of the architecture justifies the correctness and completeness of the architecture against the operational requirements. Where the previous views describe the tools available for service management teams, this view describes the procedures needed to perform maintenance, upgrades, monitoring, etc. This view makes the link with the service management discipline.

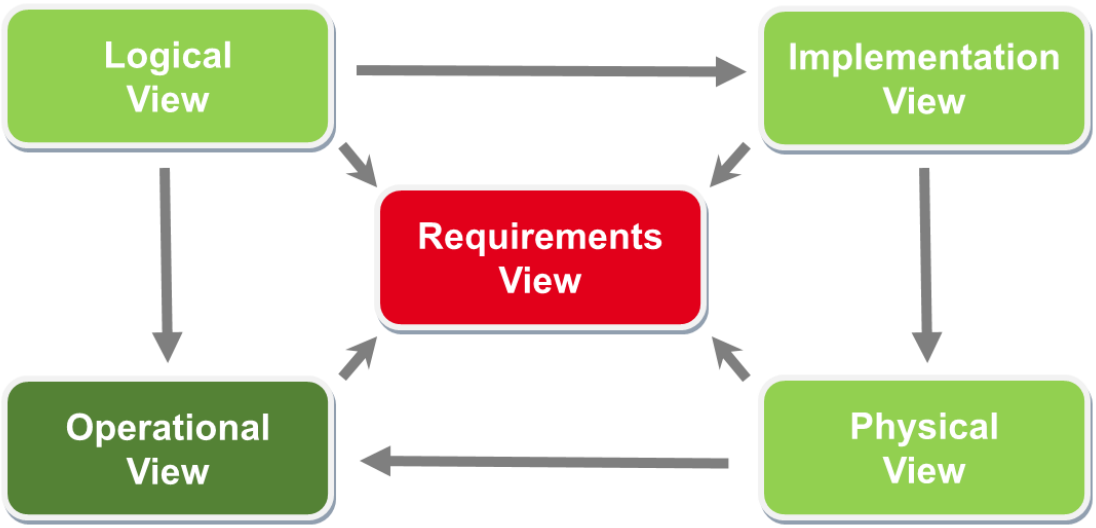


Illustration 2.2 – "4+1" View Model

As illustrated above, the **logical view**, **implementation view** and **physical view** cover the system's conceptual model and describe how this is realized and mapped to the implementation and to the operational solution. This answers the concerns of the stakeholders from the development team, the project management, the infrastructure team and the maintenance team. The **requirements view** is for sponsors as stakeholder to verify the architecture (introduced by **logical view**) against their concerns. The **operational view** provides the answers to the concerns from the maintenance team and infrastructure team. By using this view based approach, we collectively cover the whole architecture of the whole system.

This approach was based on the article Philippe Kruchten wrote in 1995, titled *Architectural Blueprints - The "4+1" View Model of Software Architecture*.

E V O L U T E

3 To whom it may Concern

3.1 Stakeholder Overview

Stakeholders are those who have an interest in or concerns about the realization of the architecture. To quote Mason and Mitroff when they introduced stakeholder analysis into business practice in 1981:

> *Stakeholders are all those claimants inside and outside the firm who have a vested interest in the problem and its solution.*

The most common architecture stakeholders are from the following categories:

- Business Users
- Project Management
- Development and Quality Assurance
- Infrastructure and maintenance

Some of these are:

- Project/Program Management (PMO)
- Sponsors
- End users
- Development team (technical project leads, architect, developers, analysts, …)
- Sparring architect
- Infrastructure team
- Maintenance team
- Quality Assurance team (QA)

A complete stakeholder analysis from different perspectives is needed in any project. A business architecture has an elaborate analysis, as well as the requirements view in the solution architecture from the perspective of the technological decisions.

Architects themselves will assume any of three roles in the project: **Solution Architect**, **Sparring Architect** or **Architect Reviewer.**

The **Solution Architect** is the main actor in the architect phase. He is the driving force behind designing the system architecture. He has the following responsibilities:

- Participate in the coordination of the functional and technical requirements gathering and analysis
- Validate (accept or reject) the functional and non-functional requirements documents.
- Construct the architecture.
- Describe the architecture.
- Organize the architecture review meeting and present the architecture
- Revise the architecture and architecture document according to the architecture review if necessary.
- Hand over to the technical design phase.

The **Sparring Architect** is the soundboard for the application architect during the architect phase. He has the following responsibilities:

- Assist the solution architect and technical analyst by providing the suggestions and guidance.
- Review and validate architecture.
- Furthermore, the sparring architect is responsible for the technical review and sparring to the technical team during the whole technical development (architect, technical design and construct).
- Report and escalate to the project and competence management

The **Architecture Reviewer** is another role involved in the architect phase. The stakeholders of the corporate architecture team could for example fulfill this role. The architecture reviewers have the following responsibilities:

- Review the architecture by asking the questions, pointing out the incorrectness and incompleteness of the architecture in order to allow the revision of the architecture
- Validate the architecture by confirming the sufficiency, correctness and completeness of the architecture from the specific perspective as a stakeholder.

All three of these roles could evidently also be performed by a coordinated team of people as well.

3.2 Levels of Participation

As specified in the previous chapter, one purpose of the architecture document is to be a medium for the communication with the different stakeholders. This can be greatly enhanced by specifying in each chapter what its relevance is to the stakeholders listed in the project details. For this, a commonly used method is applying the **RACI Matrix**. Following the PMBOK® Guide, it assigns a level of participation for each stakeholder. These levels can be found in Illustration 3.1.

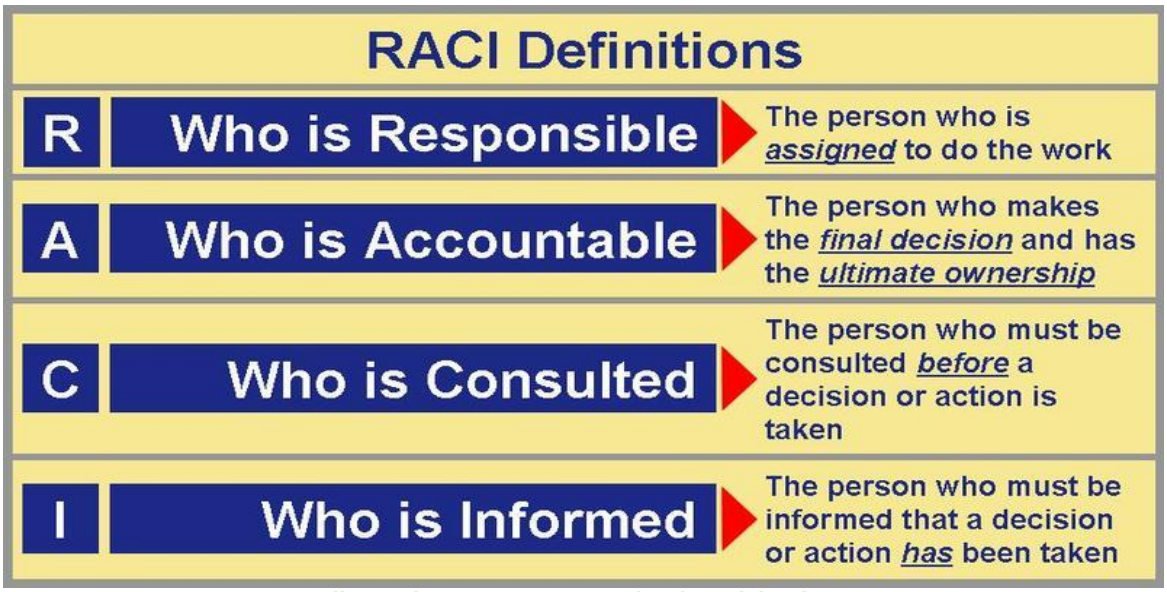


Illustration 3.1 – RACI Levels of Participation

Translated to our document, this will include a matrix such as the one in Illustration 3.2, which details one or more of these levels per relevant stakeholder. If a defined stakeholder has no level to attribute, he should not be included in the matrix.

Role	Level of Participation
Business Analyst	R, A
Functional Analyst	R, A
Business Expert	C
Solution Architect	C, I

Illustration 3.2 – Example RACI Matrix

An extended matrix RACI-VS (VS is for Verification and Sign off) is also sometimes used when the RACI levels of participation are not sufficient. Not described in the PMBOK® Guide, these levels are reserved more for elaborate projects needing several levels of accountability (for example in higher CMMI levels). Some common sense rules to consider:

- For many projects, the Sign off role might also be assigned to the Accountable person to provide them an opportunity to ensure project standards are met and processes are followed.
- Too many Sign offs can cause delay as the work product is routed for review.
- Verification should be independent (e.g. an architect who created a change shouldn't verify the change) where possible to insure the highest quality.
- Verification often designates the quality assurance or project scope verification role.
- The verification and sign off roles may be in addition to other roles.

In fact, a plethora of variants on this approach exist, each identifying one or more different levels that might be in need of stipulating. Some of these are CAIRO, where we add an "Out-of-the-Loop"-level, RASCI, where we add a "Support"-level, and DACI, which stands for Driver, Approver, Contributors and Informed.

Other ways of indicating stakeholder interest include the **Responsibility Assignment Matrix** (RAM) and the **Linear Responsibility Chart** (LRC). We won't go into detail about these methods.

4 Ours is to Reason Why

If there are no valid reasons for creating a Solution Architecture, why bother doing it at all? The reasons for making a Solution Architecture are almost as many as there are stakeholders. We elaborate on those in the next chapter. We group all the different stakeholders into four categories described in Illustration 4.1 (see below), based on the take they have on the Architecture.

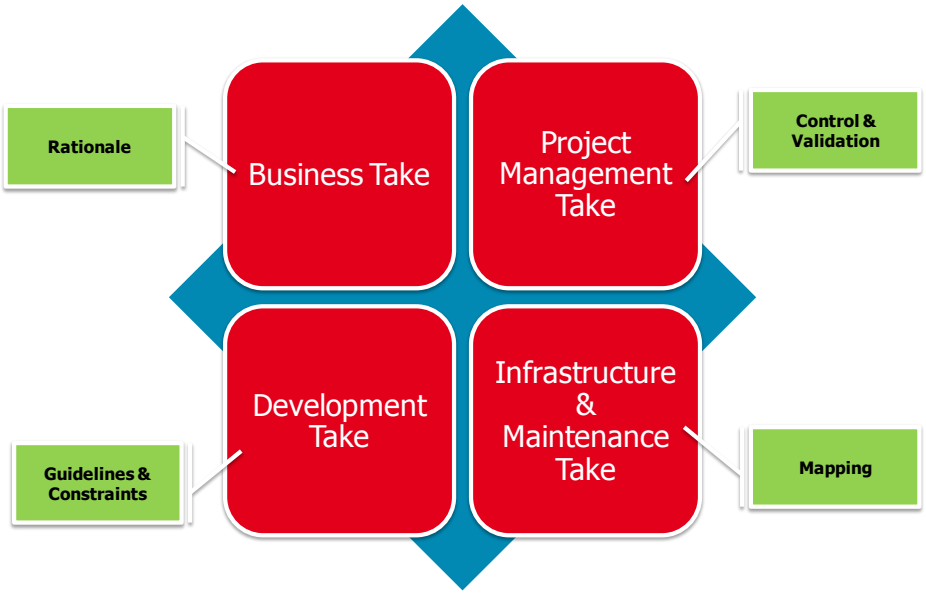


Illustration 4.1 – Stakeholder Takes on the Architecture

From the **business viewpoint**, the Architecture is a rationale of how a solution realizes the requirements. The architecture should reassure the recipients of the solution that their needs are met.

From the **project management viewpoint**, the Architecture is a control mechanism, to validate whether a system of such structure is sufficient, feasible, efficient, maintainable, risks free (meaning that all risks have been mitigated), and which promises a system with high quality.

From the **development viewpoint**, the Architecture is about the guidelines and constraints on the technical design of the system elements and relationships between these elements, and how the structure of the system is constructed with these elements.

From the **infrastructure and maintenance viewpoint**, the Architecture explains how the system is mapped to the infrastructure and how the system is operated and maintained. The architecture is often used to understand how the infrastructure and operation process shall be adapted and applied to support the system.

Clearly, all stakeholders need a common reference that illustrates the solution's architecture from each of their viewpoint, yet contributing to the architecture as a whole. Hence, the book *Software Architecture In Practice, Second Edition*, *Addison-Wesley* summarizes the purpose of documenting architecture as follows:

> Documenting software architecture facilitates
> • **Communication** between stakeholders
> • Documents early **decisions** about high-level design
> • Allows **reuse** of design components and **patterns** between projects

However, this is only part of the story. A final participant in the architectural exercise is of course the architect himself. His viewpoint can be viewed as part of the development viewpoint. His job is to translate the requirements into an architectural model. He does this for numerous reasons:

- Guiding thought in himself
- Guiding thought in others (as indicated in the development viewpoint)
- Being able to answer questions asked of the architect
- Being able to examine the results of requirement gathering

The architect doesn't guide action, he guides thought. Normally, thought precedes action, but this is by no means an absolute truth. Additionally, the role of the architect is to suggest action, as well as oversee that action to ensure that it achieves its goal (quality assurance). So the architectural process serves to support a reasoning process (guiding thought). The architect walks through the reasoning process, either for his own decisions or for the decisions he requires from others.

If the purpose of an architectural model is to support a reasoning process, then, by extension, the purpose of an architectural framework is to support a particular theory of organizational evolution. For example, the architectural process during the requirements gathering phase may lead to them requiring a SOA model to cover all needs, and thus the SOA model will become the set of standards guiding the transformation of how that business works.

By using the 4+1 views approach, the architect will strengthen the quality of the solution. Not only will the architect be able to guide his team to achieve fewer defects, but under the umbrella of quality, the view usage strives for the following results:

- **Separation of concerns:** When attempting to include all aspects of a solution into a single representation, the communications done about these aspects are rendered more difficult. The aspects of the solution become more closely entangled, and obscure clear vision on individual aspects. Separating these aspects in distinct, but related descriptions alleviates this concern.
- **Communication:** As indicated earlier in this chapter, communication is an important reason for architecture work. However, the needs of the different stakeholder groups can be quite different, making a single representation of the architecture excessive and obfuscating. Communication is a task that is already difficult enough without a complicating medium. Division in viewpoints also allows for the usage of notation standards for each of these stakeholders.
- **Complexity Management:** By separating the architecture in viewpoints, the architect focuses on each aspect in their proper description, dealing with individual complexity of the components without having to worry about the complexity of their combination.

In essence, all of these architectural views and their goals, boil down to the need to reduce exposure to risk. The additional cost of the architecture phase far outweighs the risks we avoid in later stages of the development lifecycle. It is an early detection system where we attempt to cover as many bases as possible before venturing into the phases where the cost of change is exponentially higher.

5 When it is used

We define the solution architecture scope as the glue between the capabilities and standards governed at the enterprise level, and the technical designs of individual components. As shown in the illustration below, solution architecture is positioned at the project scope, unifying the guidelines set out by the enterprise scope with the capabilities and standards part of shadow IT, and thus not yet part of those governed by the IT department.

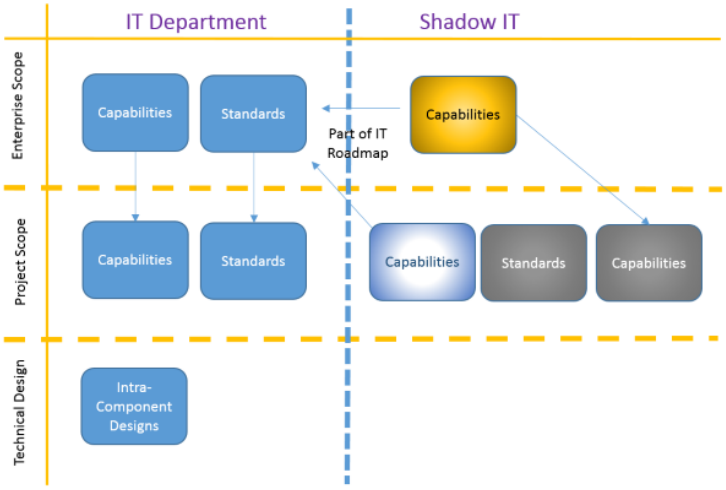


Illustration 5.1 – Enterprise Capabilities vs Shadow IT

Afterwards, these project level capabilities and standards might become enterprise scoped by establishing the proper governance surrounding them. This is normally done as part of an IT roadmap for an organization.

5.1 Enterprise Architecture

There are several frameworks when it comes to Enterprise architecture, and since this is not the focus of this course, we will only briefly discuss 3 of the more generally adopted ones, and where a solution architecture fits in these. The EA frameworks we will consider are Zachman, TOGAF and FEAF.

John Zachman stipulates in his approach that an enterprise architecture should be approached from 6 foci, as shown in illustration 5.2, namely the data, the function, the network, the people, the time and the motivation. These foci are to be designed in a manner that is traceable through the different layers of granularity, from “Scope” to “Detailed Representation”. If we consider this framework, solution architecture would be a patchwork of descriptions dispersed over the Business Model, System Model and Technology Model Layers.

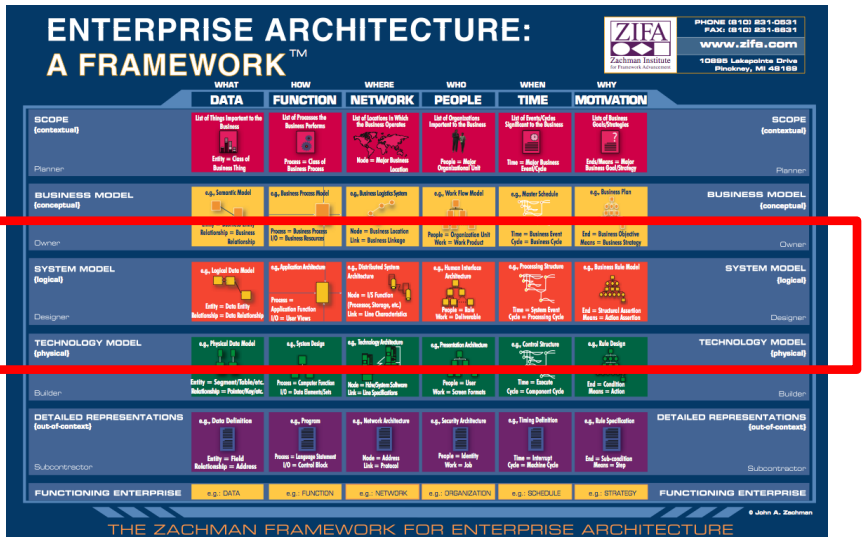


Illustration 5.2– Zachman Framework for Enterprise Architecture

The Open Group Architecture Framework (TOGAF) is a framework that structures EA descriptions by discerning relevant viewpoints and modeling techniques. It also indicates elements that should be part of an architectural model and describes the process leading to this architecture. It does so by utilizing six components, one of which is the Architecture Development Method or ADM for short. This method consists of 10 phases, as shown in illustration 5.3.

These phases can be grouped into 4 sections:

1. Getting an organization committed & involved (Preliminary and phase A)
2. Getting the architecture right (phase B, C and D)
3. Making the architecture work (phase E, F and G)
4. Keep the process running (phase H and Req. Management)

The solution architecture process is handled primarily in phases C and D, the Information Systems and Technology Architecture. However, in TOGAF, it will influence all subsequent phases, as well as influence the requirements management, possibly mandating an update of previous phases.

Illustration 5.3 – TOGAF ADM

The Federal Enterprise Architecture Framework (Illustration 5.4) is an attempt of the US federal government at an enterprise architecture spanning all of its divisions in order to create value. It is built using reference models that develop a common taxonomy and ontology for describing IT resources in order to assure that all stakeholders use the same language when developing a strategy for the enterprise. These models are:

- The performance reference model (PRM)
- The business reference model (BRM)
- The service component reference model (SRM)
- The data reference model (DRM)
- The technical reference model (TRM)

The solution architecture, as with all other enterprise architectural frameworks, plays in all of these models, but mainly focuses its activities in the TRM.

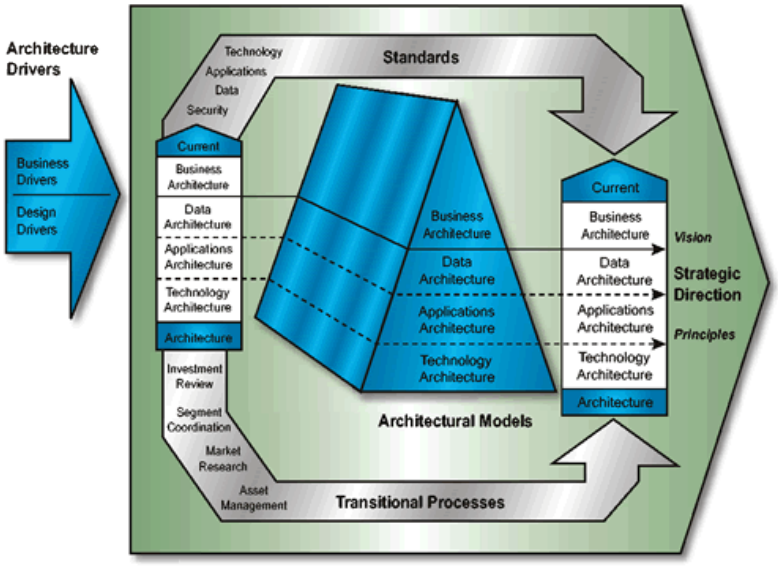


Illustration 5.4 – Federal Enterprise Architecture Framework

5.2 Project Architecture Effort

The architecture effort within the project spans over several phases, as indicated by the illustration below. It starts when the global analyses starts in the Plan phase. It goes along with the entire design portion of the Build phase and extends further till the architecture is constructed, documented, validated and accepted. The effort ends when the solution enters the last step in its lifecycle and enters the Dispose phase. Its main activities do however change depending on which phase of the project is currently happening.

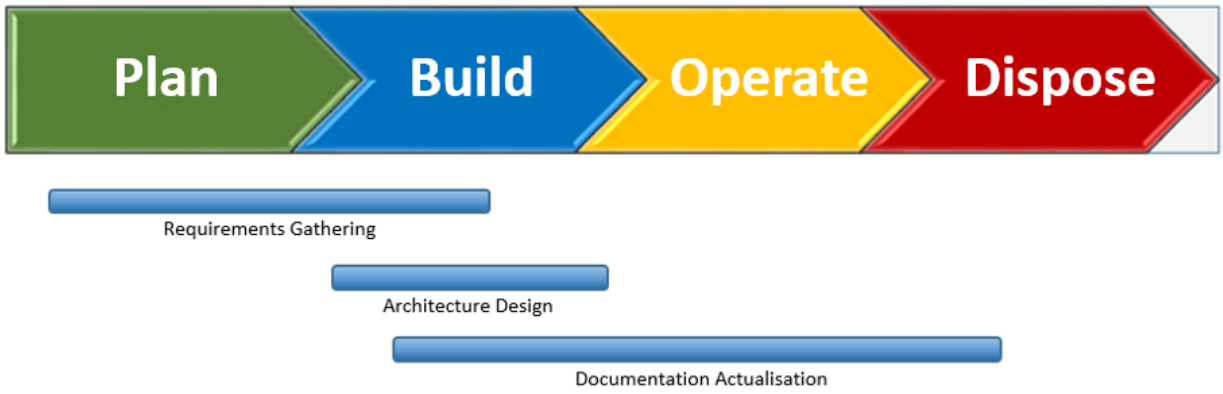


Illustration 5.5 – Architecture in a Project

The first activities of the architecture effort focus mainly on gathering all relevant requirements that might influence the design. During the Requirements Gathering effort of the Plan Phase, and their elaboration during the Build Phase, an Architect will perform the following tasks:

- The architecture construction starts with the gathering and analyzing of the system requirements. These requirements can be divided into functional and non-functional requirements, where the latter can be divided into technical (such as integration, quality, and infrastructure requirements) and operational requirements (such as documentation, training, and managed services requirements), transforming them in to measurable statements.

  The application architect works together with domain specialists, both business and technical, to guide and constrain the business and technical analyses from a technical perspective and should assist the analysts by informing them of technical information and possibilities.

  In this step, the business knowledge is acquired; high level business and technical requirements are produced. Meanwhile, the application architect constructs the high level architecture (structure of the architecture elements).

  These requirements do not only feed the technical effort, but other disciplines such as business architecture and testing as well.

- This high level architecture is written down to the architecture document version 1.

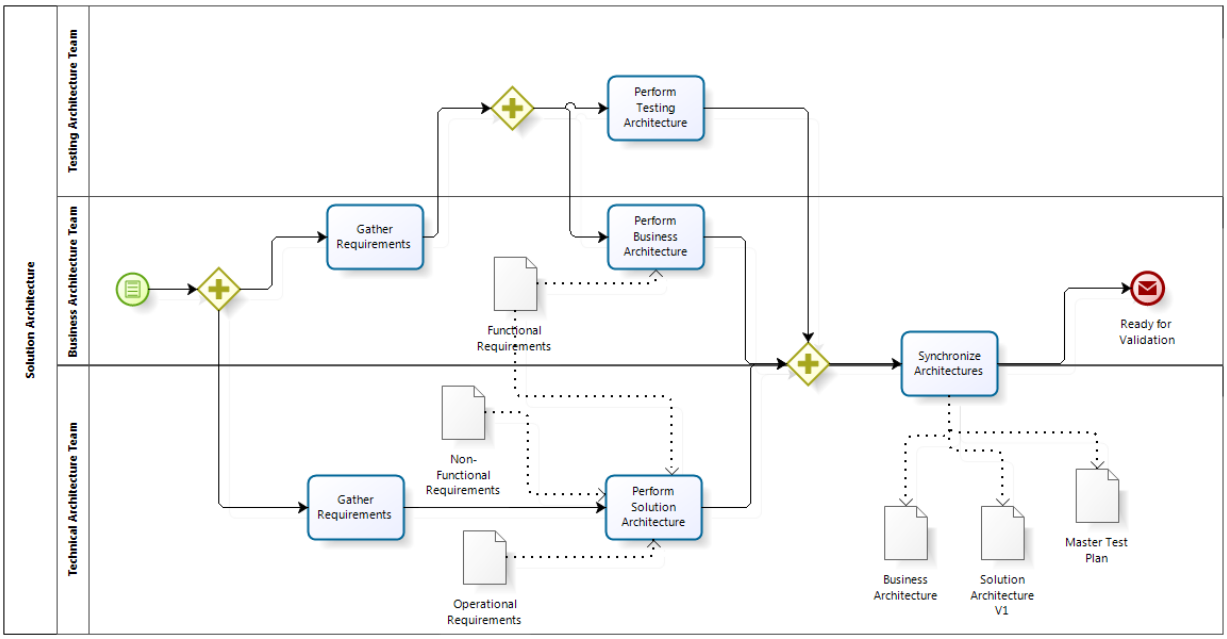


Illustration 5.6 – Solution Architecture Document (version 1)

For an outsourced project, the first version of a Solution Architecture (and its corresponding document) could even be constructed before the Plan phase, during a presales effort, when a first set of requirements becomes known, and an attempt at setting the scope ensues as part of a request for information (RFI). However, in the project lifecycle the first architecture version of some maturity, happens after the initial activities of Build Phase have happened, and we have a first attempt at an architecture design. As with most deliverables of a project, the solution architecture document will mature well into the Operate phase and even a bit of the Dispose phase with activities to keep the documentation up to date with reality.

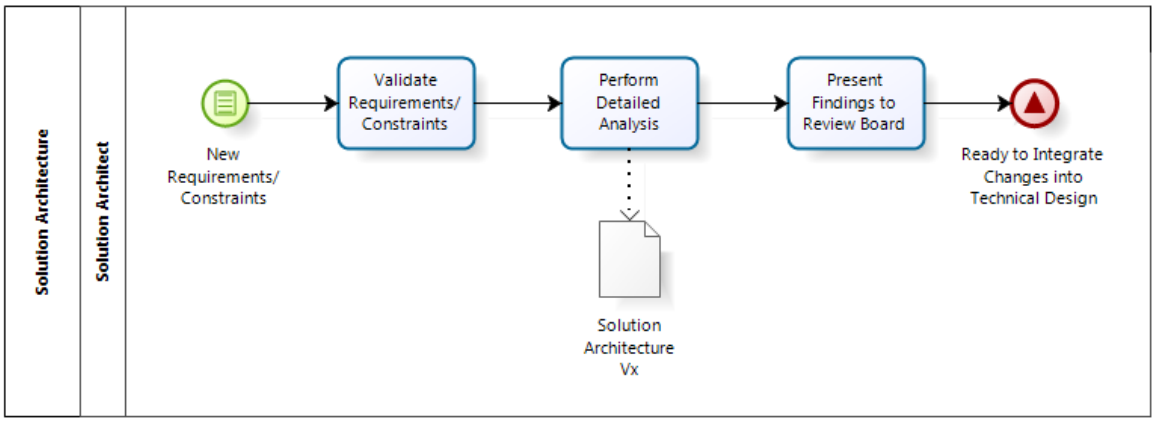


Illustration 5.7 – Revisitation of the Architecture Design

The resulting first version of the Solution Architecture will be reassessed several times during the architecture effort. Each time new requirements are detected, or new constraints are introduced, the architecture needs to go through a cycle of validation of the new requirements/constraints, which feeds into a new version of the architecture design, followed by an architecture presentation and review event, as shown in illustration 5.7. In most cases, these new requirements are derived from proofs of concept (POC), which have been executed following an earlier version of the architecture, and have exposed gaps in the solution. The following activities will be undertaken by the architect to achieve a more and more mature architecture document:

- Based on the previous architecture version, the detailed analysis will be executed by the analysts. The application architect has further the responsibility to streamline the correlation between this detailed analysis and his architecture. The application architect provides technical information to the analysts; influences and aids the analysts' decision making.

The application architect gathers further information concerning the business and technical requirements. The application architect thinks together with the business analysts and technical analysts in order to make the architectural decision, which can have immediate consequence on the analyses.

- The application architect is responsible for the validation and acceptation of the final requirements documents. The acceptation of the requirements means the application architect agrees that the requirements documents are relevant, correct, complete and unambiguous not only for architectural decisions, but also later for design and construction phases.
- The architect is also responsible for the follow-up of any Proof of Concepts that are to be performed as validation for the decided architecture. Based on the results of these POCs, a new version of the Architecture Document will be written out.
- After the validation and acceptation of the requirements, and any POCs needed performing, the application architect describes the architecture into the architecture document version 2.
- The application architect organizes the presentation and the review session for the proposed architecture. The presentation and review session can be omitted upon the agreement and decision from a technical project manager, sparring architect and project manager.

These revisitations of the solution architecture should however be limited as much as possible in order to avoid 'scope creep'. And if they are not avoidable, efforts should be made to detect these changes as soon as possible in order to limit the impact these changes will have on the existing solution.

Another change to the software architecture (or more precisely to its describing document) can arise in the form of limitation. From efforts during the Build, Operate or even the Dispose phase, the need can arise for the Solution Architecture Document to be actualized to the current situation without the need for a ripple effect on the solution. This actualization is vital in order to limit the impact of future changes, as well as input for the lessons learned once the project has run its course.

In summary, an architect will not have a full workload during all phases of the project. At the beginning of the project, the architect works together with the business and technical analysts to coordinate and guide the requirements gathering and analyses, resulting in a first mature version of the architecture. Further on, the technical analysis will be based on this version of the architecture document, and will consolidate all the requirements in detail under the architect's vigil. The Architect oversees the detailed technical designs and organizes any POCs that are to be performed. Recurring actualization efforts are coupled with quality assurance of the implemented designs.

6 Partners in Crime

As evidenced by the life cycle of a solution, architecture goes hand in hand with several other disciplines, such as project management and service management. This chapter will elaborate more on the osmosis between the solution architecture effort and these disciplines.

6.1 Project Management

There are several methods to performing project management in a proper way such as Prince2 or Agile Project Management. We won't go into detail for any of these ways, but each requires a slight tweak on the insights shared in this chapter. We will employ the structure and guidelines as described in PMBOK[1], which is up to its fifth edition at the time of this writing. The PMBOK specifies 10 knowledge areas to keep in mind, of which several have an interface with solution architecture.

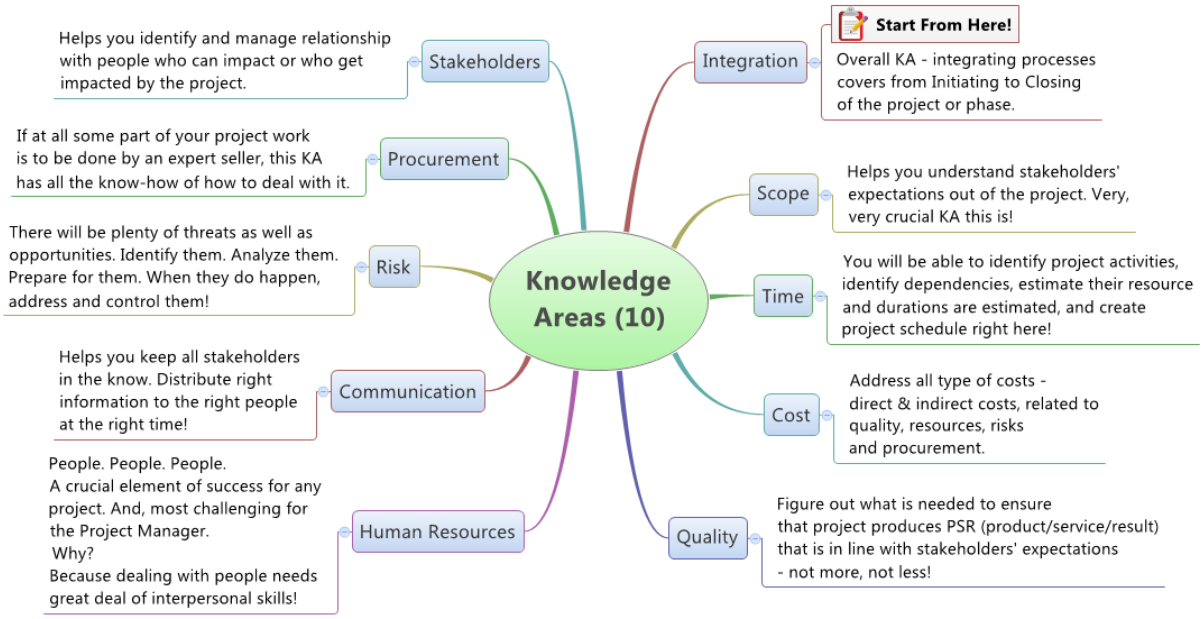


Illustration 6.1 – PMBOK Knowledge Areas

For some areas we will go into deeper detail in the following chapters, but the main concept of Project Management is called the "Triple Constraint", formed by the competing forces of time, cost and scope. Where the project manager stands guard over the budget (being time and money), the solution architect will guard the scope. This tension between the two will result in a quality delivery. Almost all knowledge areas affect at least one aspect of this. Either by the estimates given by the solution architect, or the constraints put on scope for budgetary reasons by the project manager.

6.1.1 Scope Management

One of the processes of Scope Management is what PMBOK calls "Collect Requirements": the process of determining, documenting, and managing stakeholder needs and requirements to meet project objectives. As elaborated in thee chapter on the Requirements View, this view forms the framework for the exchange of information about requirements between the architecture and the project manager.

> *Software requirements engineering is the science and discipline concerned with establishing and documenting software requirements. It consists of requirements elicitation, analysis, specification, verification, and management.*
>
> *-Thayer and Dorfman (IEEE), Software Requirements Engineering, 2nd Ed., 1997*

[1] A Guide to the Project Management Body of Knowledge (PMBOK Guide) – Project Management Institute

Requirements are the most misunderstood part of systems development, and yet the most crucial. The ISO 9000 standard uses the following definition for a requirement:

> *A requirement is a need, expectation, or obligation. It can be stated or implied by an organization, its customers, or other interested parties. A specified requirement is one that has been stated (in a document for example), whereas an implied requirement is a need, expectation, or obligation that is common practice or customary.*

When the transition towards a definite system requirement is made (implying that the system will cover the need), the IEEE Standard 1233-1998 (Guide for Developing System Requirements) defines a well-formed requirement as a statement that has the following characteristics:

- It relates to a system functionality (business need or capability)
- It can be validated
- It must be met or possessed by a system
- It solves a customer problem
- It achieves a customer objective
- It is qualified by measurable conditions and bound by constraints

In order to check all these characteristics, a template should be devised that incorporates each of these. The concepts to consider per requirement are the following:

- **ID:** A unique identifier based on the naming convention decided for the requirements.
- **Status:** The current status in which the requirement is situated
- **Description:** A prosaic description of the requirement, and what it entails.
- **Rationale:** A specification of the need for the requirement.
- **Category:** The category and subcategory in which the requirement is situated, based on the ISO standards (see the chapter on the requirements view for these)
- **Importance:** The importance of the requirement to the stakeholders, either quantified or by using a MoSCoW rating, as suggested in the Business Analysis Body of Knowledge (BABOK).
- **Stakeholders:** The stakeholder(s) with a vested interest in the requirement
- **Business Capabilities/Needs:** The business capability(ies) which are partly or completely covered by the requirement
- **Parent Criteria:** The criteria which necessitated this requirement, if any.
- **Acceptance Criteria:** The criteria with which we verify whether the requirement has been successfully implemented.

Requirements must be correct if the rest of the development effort is to succeed. The later in the project lifecycle a scope change is introduced, the more costly it is. So, any problems avoided in the requirements phase are worth their weight in gold. The reason for this is the ease with which a new requirement can be added or an existing one can be adapted, and what impact it has on the changes needed on the system we are trying to develop. This is shown in the illustration below. The later the change is introduced, the greater the ripple effect that is caused by the changing requirements.

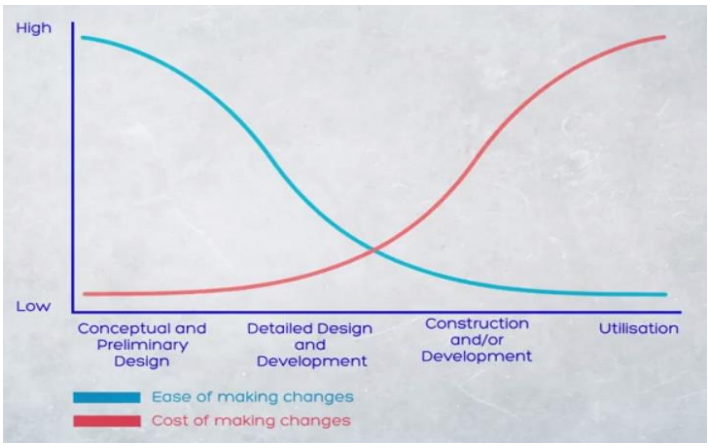


Illustration 6.2 – Requirement Change Impact

Every project finds its genesis in a set of business needs. An organization has a strategy (or mission) with which it approaches to market to claim its stake. This strategy translates in goals it needs to attain to make that strategy successful, and they in turn are the drivers for the business needs. And business needs generate the drive for a project. The process of eliciting these needs and translating them into requirements so that they become measurable and can be validated, is called requirements analysis.

Where Illustration 8.2 gives an elaborated overview of requirements analysis (using the Volere methodology), it usually comes down to an iteration cycle of Elicitation, Specification and Validation. The result of which we record into 2 documents. The first document is the Requirements Specification document in which we detail the following topics: the Management Overview, the Project Case, the Stakeholders, the Constraints, the Processes and the Requirements List. A second document is an Excel file containing all the requirements as well as all relevant information about them.

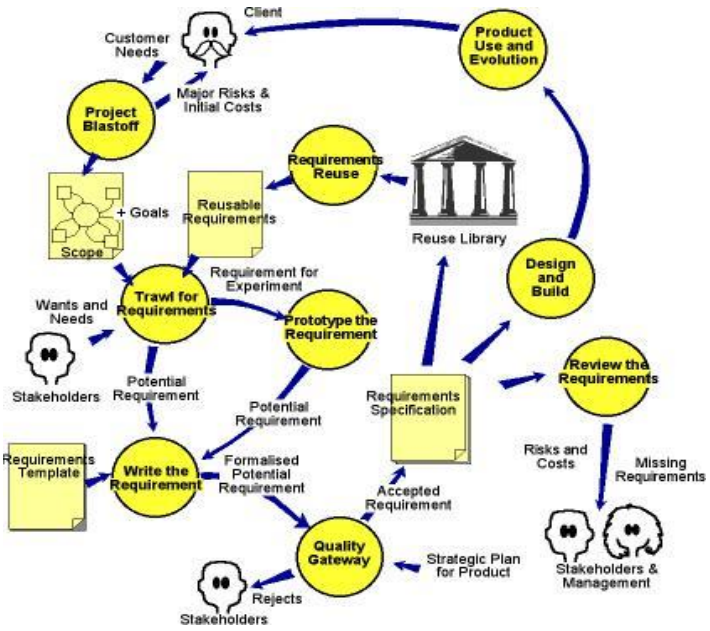


Illustration 6.3 – Volere Requirements Analysis

The Elicitation phase has two flavors, elicitation and elaboration. Where elicitation points to those requirements explicitly given by the stakeholders, elaboration requires analysis of the needs stated by those stakeholders to discover the requirements. This happens either through decomposition, splitting the stated need into the requirements that are needed to address that need, or by derivation where we have a requirement that infers one or several other requirements to be made possible. Typical techniques, employed during structured workshops, are simply asking the stakeholders for requirements, storyboarding, task analysis, the analyst's own insight, document analysis, reverse engineering, ethnography, prototyping, brainstorming and problem-solving thinking, etc. We list these in a Requirements Breakdown Structure to be included in the specification document, and which in turn is an input for the Work Breakdown Structure drafter by the project manager. An example of an RBS of a domestic security alarm can be seen in the illustration below.

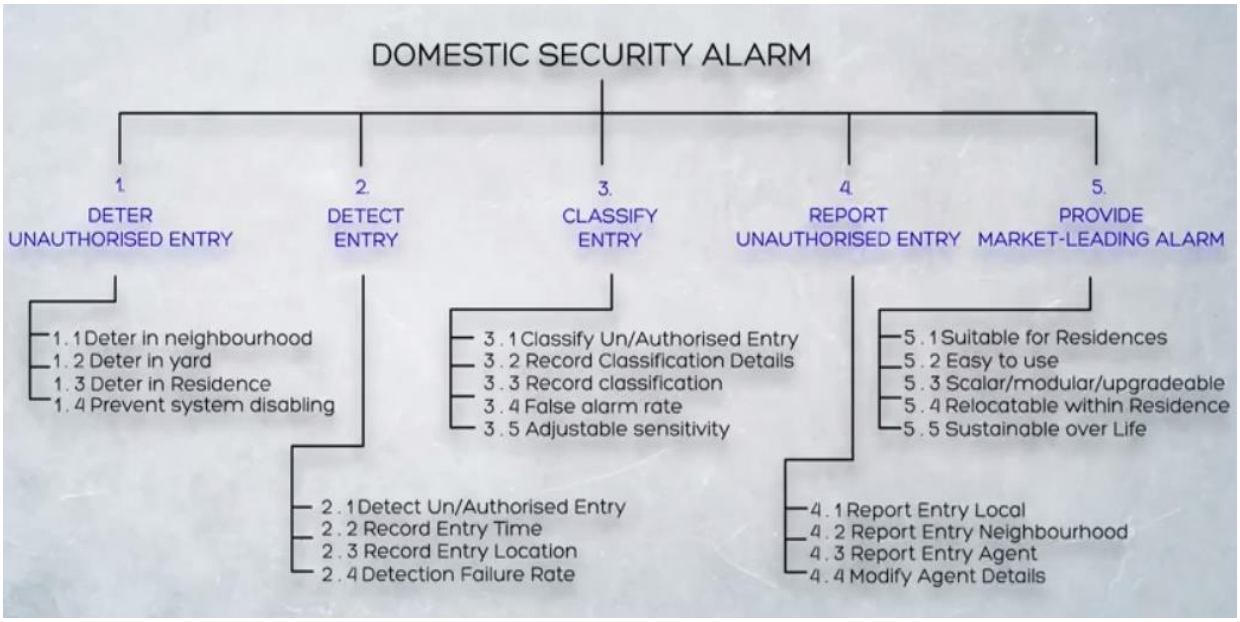


Illustration 6.4 – Example Requirements Breakdown Structure

As with any other analysis exercise where there is an interaction with people, the architect comes into contact with a wide range of opinions and perspectives of people who think they know what is required, and with a wide range of confusion and vagueness from those who don't know what they want. It is imperative to cut through the assumptions and attain accurate and relevant information. One of the methods for achieving this is 'Critical Thinking', pioneered by John Dewey in the early 1900's.

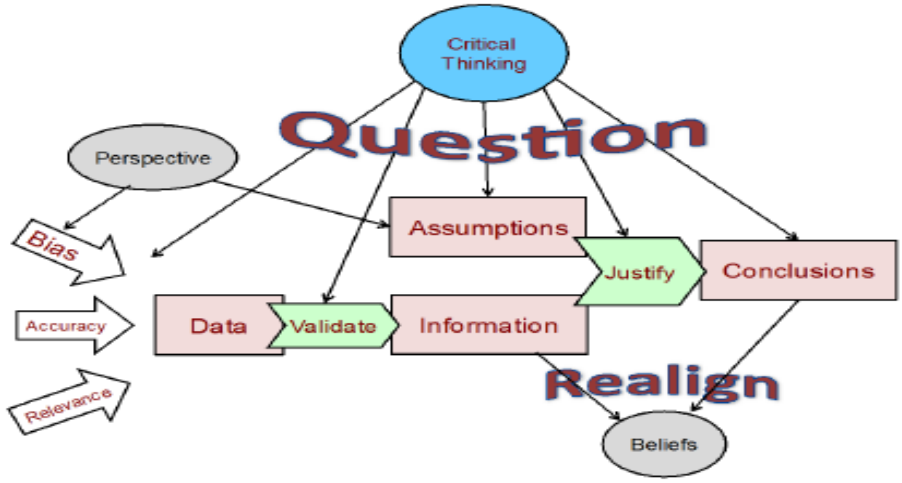


Illustration 6.5 – Critical Thinking Method

As shown in the illustration, the Critical Thinking Method tries to validate all data presented to the architect to determine its accuracy, bias and relevance to put the information we extract into context. However, in order to form the proper conclusions, we need to weigh the information against those volumes of data not presented (in the form of assumptions), and either justify them or discount them. The skills needed to aptly perform this thinking exercise are the following:

1. Recognize problems
2. Understand the importance of prioritization and order of precedence in problem solving
3. Gather relevant information
4. Recognize and question unstated assumptions and values
5. Comprehend and use language with accuracy and clarity
6. Interpret data to appraise evidence and evaluate arguments
7. Recognize the existence (or non-existence) of logical relationships between propositions
8. Draw warranted conclusions and generalizations
9. Put to test the conclusions
10. Adjust one's beliefs on the basis of wider experience

Next, the Specification phase is where we formally structure the list of requirements with different modelling techniques into an apprehensive, clear, precise, unambiguous, atomic, complete, testable, maintainable requirements, or creating SMART requirements. These requirements will be elaborated upon in the Build Phase of the project. Techniques employed here are the Unified Modeling Language (UML), Business Process Modeling Notation (BPMN), Semantics of Business Vocabulary and Rules (SBVR), Domain Specific Language (DSL), etc.

Finally we validate the list of requirements with the stakeholders through such techniques as reviews, prototyping, simulations, etc. in order to receive a formal signoff on the scope of the project. This is best done through a formalized entity (for example a Change Advisory Board) that meets at regular intervals to assess the status of all open or changed requirements, and then formalizes the decisions made. If an issue arises with the requirements, and the periodicity of the board is not sufficient to tackle this issue, an emergency convocation should be possible.

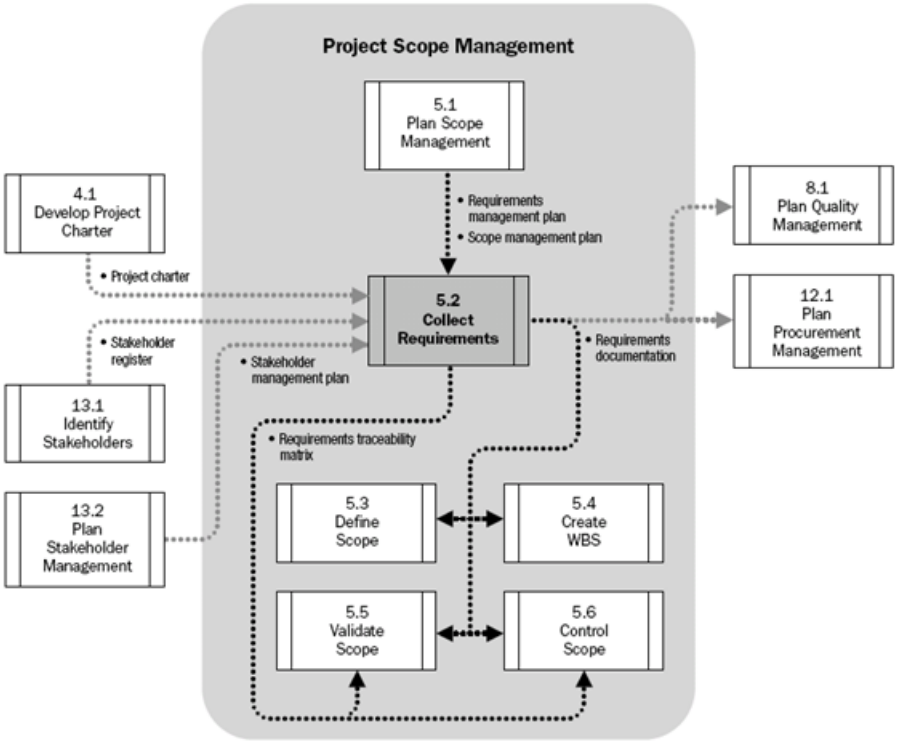


Illustration 6.6 – Requirements Management Process (PMBOK)

The tasks of this board are to evaluate requirements and their associated assumptions and risks when they arise during the course of the project. While the individual evaluations are largely an organic activity, no real procedural steps can be given. However, there are some guidelines that can be specified:

- When a requirement has no stakeholders that demand it, it is not needed, and should be set to that status until a stakeholder re-emerges that demands it.
- When a requirement has no acceptance criteria, it cannot be verified to have been fulfilled. It should be put on the 'Not Needed' status as long as it is not verifiable.
- When a new requirement is added to the list, a determination of its category (and if possible subcategory) should be made to simplify the task of verifying whether or not this is a duplicate

6.1.2 Risk Management

Risk was for a long time defined as "the chance of possibility of loss" by the ISO guide 73. Since then, influenced by the ISO 31000:2009 standard release, this definition has been altered to "the effect of uncertainty on objectives", indicating that the risk might yield positive outcomes as well as negative ones. The main idea of structuring risks is to divide them up in categories, of which one them is usually technical risks, where the link with solution architecture is the strongest. Typically, in any software project, the most significant risks come from connections made with parties outside of the immediate control of the project. For example, a web service exchange with a third party.

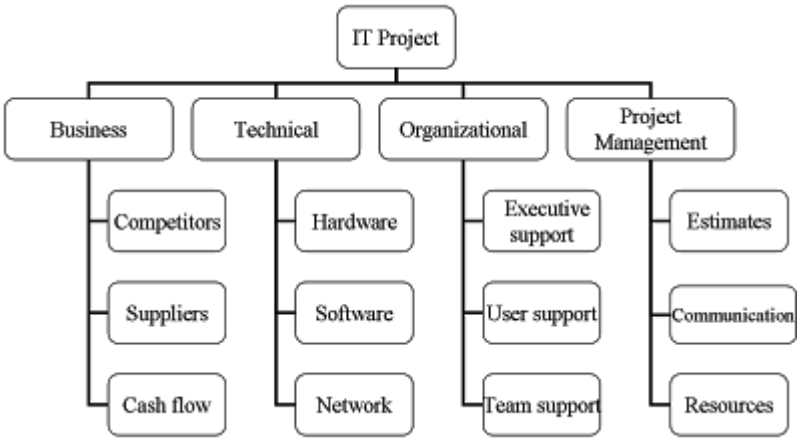


Illustration 6.7 – Example Risk Breakdown Structure

Beware, there is a difference between project risks (on which we are focusing with this document) and enterprise risks. Those risks play in categories more closely linked to the strategy and goals of the organization, as shown in the illustration. These risks are governed by other processes, for example using COBIT.

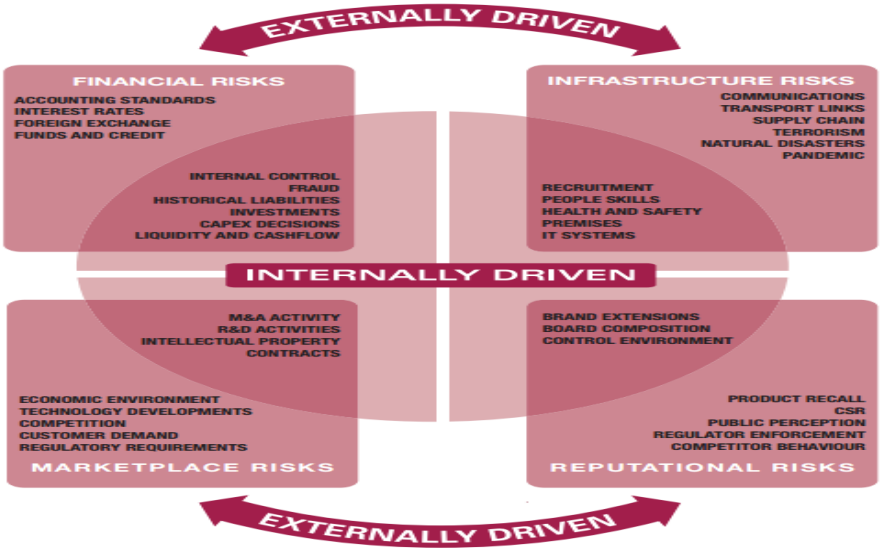


Illustration 6.8 – Requirements Categories (COBIT)

6.2 Service Management

FitSM, a lightweight standards family aimed at facilitating IT Service management (ITSM) co-funded by the European Union, defines it as a discipline formed by the entirety of activities – directed by policies, organized and structured in processes and supporting procedures – that are performed by an organization or part of an organization to plan, deliver, operate and control IT services offered to customers. There exist several frameworks and methodologies for this type of management, such as for example ITIL, which covers a wide range of these activities. The connection made with solution architecture resides primarily in the phase that ITIL names “Service Operations”, the day-to-day care of IT services.

It is these services that should be taken into account when devising how the solution will be managed and monitored when it is deployed in a production environment. Provisions to support these processes, whether they be technical or procedural, are to be detailed, and taken into account.

7 How it is Used

The main structure for the architect is the Solution Architecture document. As stated earlier, the need for a document is dictated by the need for communication with the stakeholders. Therefore, it is beneficial to all projects within a same organization to have a uniform way for this communication, enhancing reuse and recognisability.

To this end, a template, formed by project experience and best practices, is utilized. Bear in mind, this template is designed for completeness as a sort of checklist to guide the Architect in listing everything that he could need in order to produce a qualitative solution. This means that for any project this document is prone to tailoring, with the architect deleting sections not applicable to the project at hand, or adding a descriptive as to why these sections are not needed. The latter is the preferred approach.

The document, apart from the 5 views mentioned earlier, will also contain an introductory chapter containing such information as the management summary, summarizing the spirit of the document, the position of the project in a roadmap or program, a reference list of other relevant documents, a glossary with important definitions to be used throughout the project, relevant project information (such as for example stakeholders), and a TODO list to be resolved in later versions of the document. Since at least some of this information is also available in other documents (such as a Project Initiation Document), it is preferable to just write up a summary or mention a reference rather than just copying the information into the Solution Architecture document.

As said earlier, the Solution Architecture does not exist in a vacuum, and has several sources it uses as input to form the solution presented to a client. These sources are threefold and feed the different architectural design views described in this chapter, namely the presales information, the functional requirements and the non-functional requirements. An overview of these is seen in Illustration 6.0.

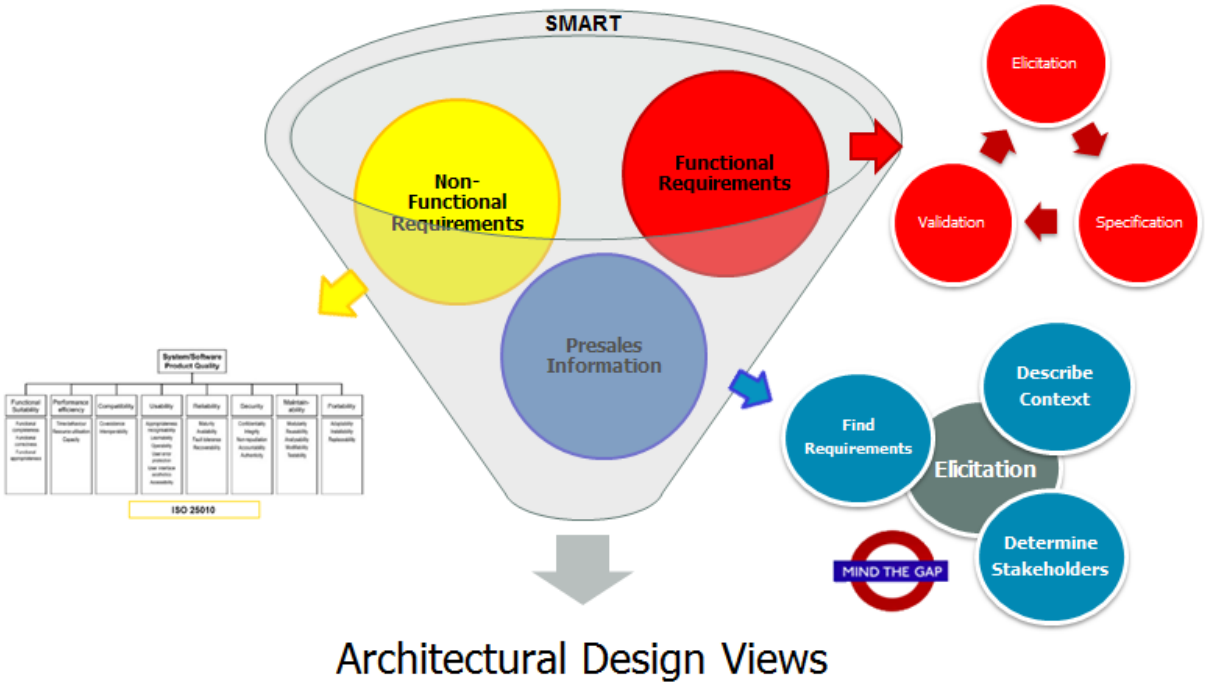


Illustration 7.0 – Input Channels for the Solution Architecture

Keep in mind that the success of presales inquiry can be mapped directly onto the possibility to work out the entire business context. Not only are the requirements (both functional and non-functional) of great import, but a good understanding of business and project context, as well as the interaction between stakeholders, contribute to the overall success of the Architectural Phase and ultimately the success of the project itself.

7.1 Requirements View

In this view, we detail the architectural significant requirements, be they functional requirements, non-functional or operational in nature. We explain how these requirements are realized by the architecture. Each of these requirements should have a reference to the chapter and view where their concerns are handled, as well as conditions for acceptance of the solution covering these requirements (sometimes called the verification strategy). They should also indicate which stakeholder group(s) have required them.

A cautionary note dictates that the architect should not attempt to capture a full-blown detailing of all requirements. Not only can this lead to analysis paralysis, but as former US Secretary of Defense, Donald Rumsfeld said it: "There are known unknowns; that is to say, there are things that we now know we don't know." It is a wise practice to guard your scope, and to adopt a pragmatic, time-boxed approach in your effort. The as-of-yet still unknowns should be categorized as risks (with a severity dependent on their relevance to the project and the enterprise as a whole).

7.1.1 Stakeholder Analysis

We start by identifying all stakeholders relevant to the architecture effort. Although stakeholder information should be found in detail in the Project Initiation Document, the Solution Architecture can categorize these stakeholders more precisely in their relationship to the system being built. Too many projects focus their attention too closely on what needs to be built, to the exclusion of those people who will be interacting with the system in any capacity. A great number of development project problems seem to be caused not so much by a failure to write requirements, as by a failure to perceive which specific stakeholders' viewpoints and goals were relevant. This causes whole groups of requirements, typically those related to the missing stakeholders, to be missed. We can safely say that the composition of the stakeholders is a good predictor for project risks.

To model the stakeholders and their interaction with the project, one can write large texts of prose as to how they would impact the project. However, a visual representation says so much more than thousand words, so we need to come up with a taxonomy to describe these groups of people and their goals for the project, be they either beneficial or hindering. A way to describe this, is to make use of the Onion Model. Stakeholders are attributed to a circle in the model of which these are the default circles:

1. **'The Product'** or **'The Kit'**: the item under development, e.g. a software program, a consumer electronics device, an aircraft, a communications network.
2. **'Our System'**: 'The Product' plus its human users and the standard operating procedures or rules governing its operation.
3. **'The Containing System'**: 'Our System' plus any human Beneficiaries of Our System (whether they are involved in operations or not).
4. **'The Wider Environment'**: 'The Containing System' plus any other Stakeholders.

Other circles may be introduced as necessary. The different influences that the stakeholders exert on the project, can also be mapped onto the onion model, as shown on the model to the right, which is an example for the development of a computer game. This way, the lifespan of stakeholder involvement can also be taken into account, giving an indication to when and why a certain set of requirements (linked to an emerging or leaving stakeholder group) will diminish or heighten in importance.

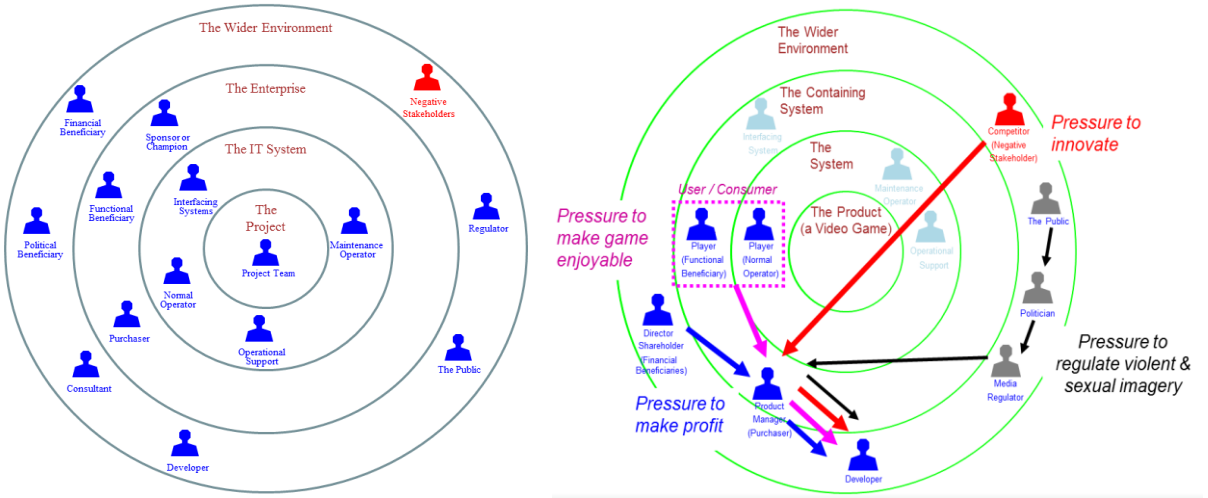


Illustration 7.1.1 – The Onion Model

7.1.2 Functional Requirements

Functional requirements are not use cases or business processes. They are simply put, a list of functions that the resulting solution needs to support, so that the business users are supported in their daily operations. In a strict waterfall project approach, use cases/business processes can be used, if they are mature enough *BEFORE* the solution architecture exercise is started, but this is seldom the case. In all other projects, a list of requirements needs to be compiled, that limits the wide span of human existence to the features we expect in the finalised product, or more simply put on which scope management can be performed. We need such a list to determine not only what the solution should be able to do, but also get a grasp of the importance of these functionalities in relation to each other (ranging from critical to nice-to-have). As stated before, we only need to list those requirements that have an impact on how the architecture and design of the solution will turn out. This listing is not to be trivialised. For example, the CHAOS report[2] states that 15% of all software projects fail because of a lack of a clear statement of requirements.

These requirements should already exist in a structured manner in various other project documents. Either in a requirements specification document, or as part of the business architecture. In either case, each functional requirement should come with a reference to its description, as well as a reference to a test scenario indicating which measuring technique will be used for the user acceptance of this functionality.

7.1.3 Non-Functional Requirements

The functional requirements are described both on a Use Case level, as well as a Business Process level, whereas the non-functional requirements are categorized by the ISO 25010 standard (see Illustration 6.1.2).

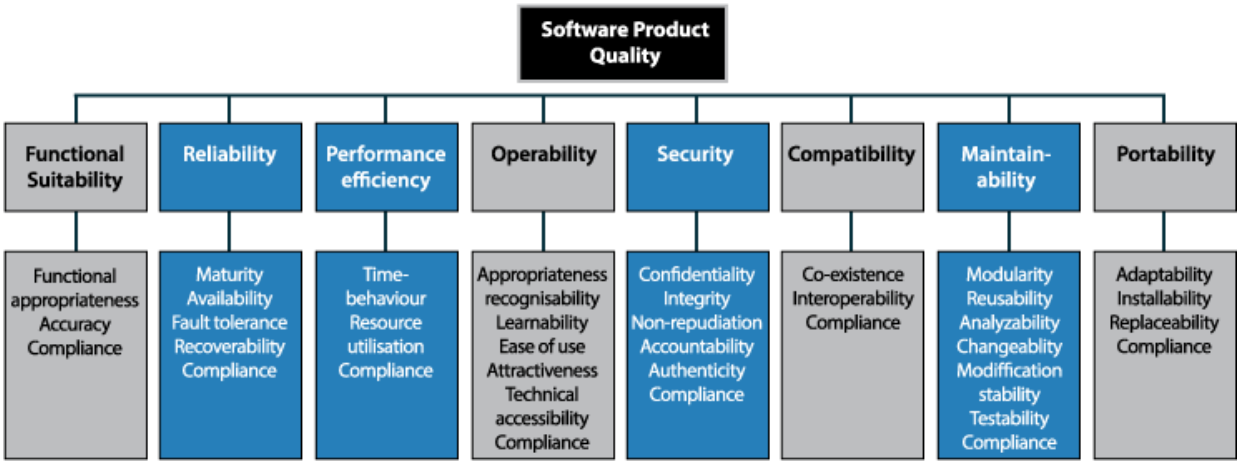


Illustration 7.1.2 –ISO 25010 Standard

Since the ISO standard focuses on quality and acceptance of the delivered product, the same model should be the basis to compose requirements as well. As a result the quality attributes can be considered as the non-functional requirements of the product focusing on expectations about functionality, usability, reliability, efficiency, maintainability and portability. However, in order to make the factors more complete and representative, each of the characteristics has been augmented with its proper set of characteristics.

Note however that not all of the characteristics are as important to every project. Some of them can be neglected or be given a very low priority. Hence, the priority indication is very important in the specification for every requirement. Some older architecture documents make a reference to the predecessor of this standard, namely ISO 9126. For an overview of what has changed between the two standards, see the appendix.

[2] The Standish CHAOS Report: a 2009 study on the success rate of projects in the IT industry

7.1.4 Operational Requirements

When a solution finally goes live, not only are its users a stakeholder, but there is also the team that will support the solution to consider as well. Therefore a set of requirements is usually determined by analysing how the managed services team will keep the solution up and running, as well as how they will cope with bug fixing and evolutionary maintenance. The listing of these requirements are to be found in the chapter of the Requirements View named "Operational Requirements".

As with non-functional requirements, so can these requirements be categorized by using an ISO standard, in this case the ISO 25022 standard, or the "Quality in Use" Model. A previous version of this was called the 9126-4 standard, but this one was replaced by the current standard in 2015. A visual representation of these requirements can be seen in the illustration below.

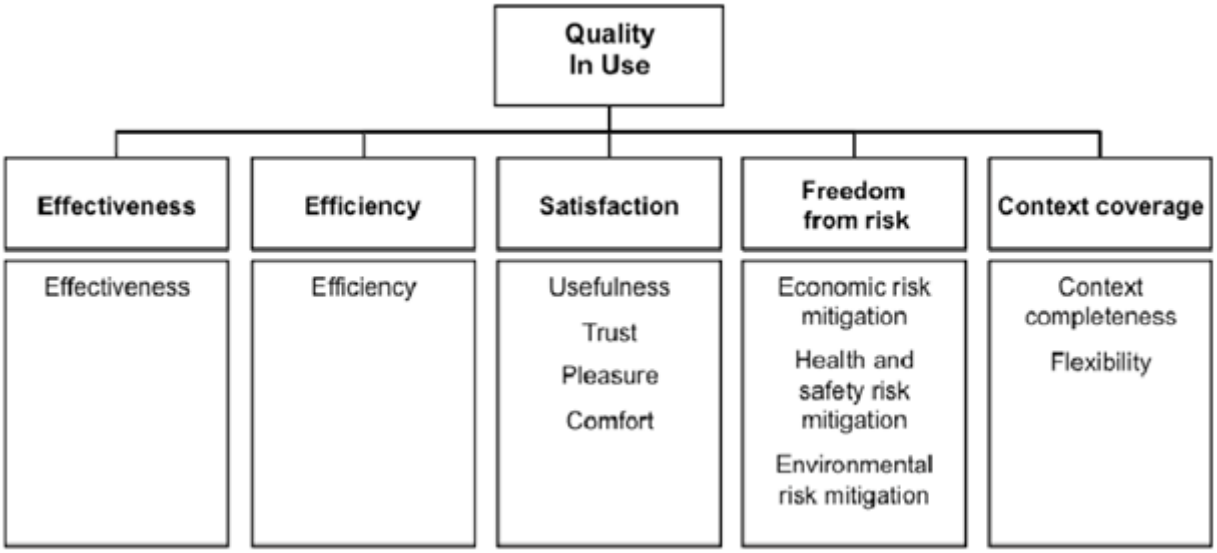


Illustration 7.1.3 – ISO 25022 Standard

The Quality in Use Model characterizes the impact that the product (system or software product) has on stakeholders. It is determined by the quality of the software, hardware and operating environment, and the characteristics of the users, tasks and social environment. All these factors contribute to the quality in use of the system. Most of these requirements are however only applicable in very specific cases.

7.1.5 Exit Requirements

Every system comes into being with a life expectancy. The transition from this system to the next solution takes place in the dispose phase of the project, and describes what requirements are derived from the need for such a transition. These requirements deal with the assessment of the assets in the current project which still have value for the organization after the system ceases to be. At the very least this is the relevant data stored in the project that must be imported into its replacement. Another example would be the hardware used in the project that can be repurposed for other needs. This repurposing always comes with the question of data security, and how thoroughly the hardware needs to be wiped before it leaves the current project. Each of these requirements might necessitate a hook into the current architecture to be accomplished, and as with all requirements, the earlier we are aware of them, the cheaper it is to implement.

7.1.6 Design Constraints

After gathering all necessary information about the requirements, we assess this collection in order to make the proper choices about architectural models. Seldom will the single choice of a model cover all requirements. For example the architect may decide upon a SOA model in order to optimise reuse of services throughout an enterprise, but decides to add a secondary CRM architecture model, because of the nature of the business model of the enterprise. The resulting architecture will incorporate the best practices and pitfalls of both into one solution.

We should take heed however, to always designate a lead architectural model. Are we designing a service oriented architecture with CRM flavours, or are we designing a CRM architecture with SOA flavours? This might not seem relevant at first, but its purpose is to create a hierarchy. That way we have a clear picture of which best practices have

precedence when the best practices of different models contradict each other. For example, a CRM model dictates all data (and therefore its services) should be defined in its relationship to the Client, while SOA dictates all services should envelop self-contained data models.

Once we determine the hierarchy of models, we describe these architectural and technological choices made in the following views. However, the choice of architectural models does not only affect the development team, but also has a strong impact on how the functional analysis of the project will need to happen. A classic N-tier model will still require the standard approach of use cases and boundary case, however an Event Driven Architecture requires for a functional analysis to centre around business events, event channels, and event consumers/producers. The choices made in the following views also come inherent with their own capabilities as well as constraints. For example, choosing a web application over a desktop application gives you the benefit of not needing deployments to every single station running your software, however it does restrict you on how the application is presented, by means of the limitations of a web browser. In this chapter we list all constraints that come with these choices, as in turn they become new requirements. Much in the style of TOGAF's Requirement Management, this is a living list that will continue to be updated well into any project.

To determine capabilities and constraints, we generally take the same approach for each architecture, based on the maturity model associated with it. The steps are easy and self-evident, as shown in illustration 6.1.4. We first draw out the current situation in relationship to the maturity model we are employing. Next, based on the requirements we elaborate on how the future solution should be, and mark it down in the Logical View. These two situations are compared with each other by way of Gap Analysis to provide an additional set of requirements and constraints to be entered into the Requirements view. Next we determine a roadmap on what activities should be performed to get to the solution. This last will however not be included in the solution architecture, but be included in the project planning.

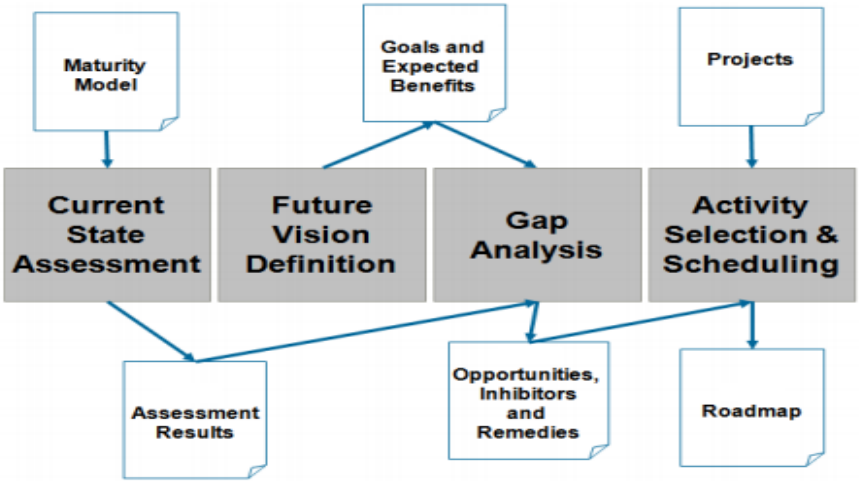


Illustration 7.1.4 – Oracle Roadmap Creation Process

7.2 Logical View

This view provides the high level overview of the internal and external logical elements (tiers, layers, components…) and the relations between them. It also serves to liaise with the business architecture of the project. The Object Management Group (OMG) defines a business architecture as follows:

> *A blueprint of the enterprise that provides a common understanding of the organisation and is used to align strategic objectives and tactical demands.*

For all intents and purposes we will consider the business architecture as a collection of descriptive core models defining the following information blocks:

- A canonical data model (and its links to industry standard ontologies)
- A description of the business processes (using BPMN, XPDL, or similar models)
- An Organizational Structure (as well as enterprise-wide security role structure)
- A description of the business rules and policies (using for example OMG's SBVR)

7.2.1 Decomposition Aspect

The Decomposition Aspect describes the logical decouple, which is a listing of the different logical elements that make up the structure of the project. We provide a high-level illustration of the project where these elements figure in their proper relevance and already indicate any reciprocal relationships there might be. We will elaborate the interaction between these elements later on (see chapter on the Integration Aspect). Illustration 6.2.1 shows a decomposition of a solution architecture will we use as example throughout the training document.

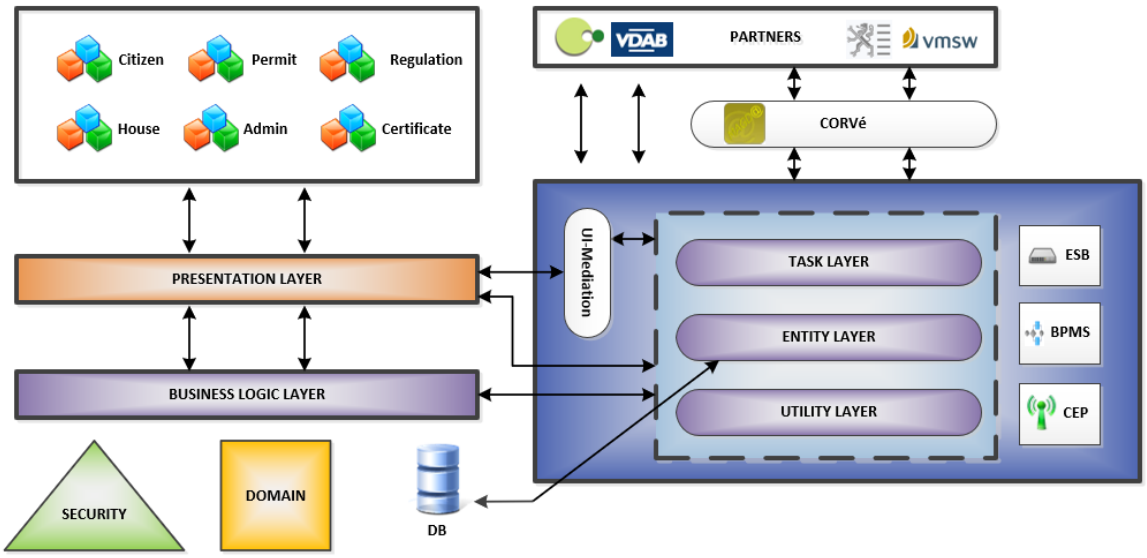


Illustration 7.2.1 – Logical Decomposition Example

We accompany this diagram with a description of the solution, or in the case of complex systems with a description of each of the individual components. This description will contain the purpose of the component, as well as any technology agnostic particularities, that might impact our choice of technological component later on. As complex systems warrant descriptions for each component, so will they cause technical designs to be drawn up for each of these components, as well as possible governance approaches, should these components be subject to change behavior different from the overall solution. For example, when versioning of web services in the task layer can happen without the need for a redeployment of the other components.

7.2.2 Integration Aspect

Where the decomposition aspect describes of what each of the components in your architecture does, this aspect describes how each component relates to other internal components (Internal Synergies), as well as external elements (External Synergies). A justification of each of the logical components, as to why they were separated from the others is also described. Examples of these justifications are motivations such as Logical Grouping, Reusability, Loose Coupling, and Separation of Concerns... It need not be pointed out that software and hardware should be treated equally as subsystem or component, whenever it is significant system-wise.

Another concern which find its place here (especially in the case of external dependencies) is the impact a certain element might have on such topics as load, performance, transactions, availability, concurrency, scalability, security, system robustness... Each flow is indicated with a direction as well as transport, and if needed security protocols.

The diagram from the Decomposition is to be elaborated to encompass the additional information brought by this aspect, as it can be seen in Illustration 6.2.2. Each of the Arrows represents a connection, and should have a clear indication of how the communication is set up. Within the green border, the internal communication is laid out, and the arrows outside of the green border are those with external partner systems.

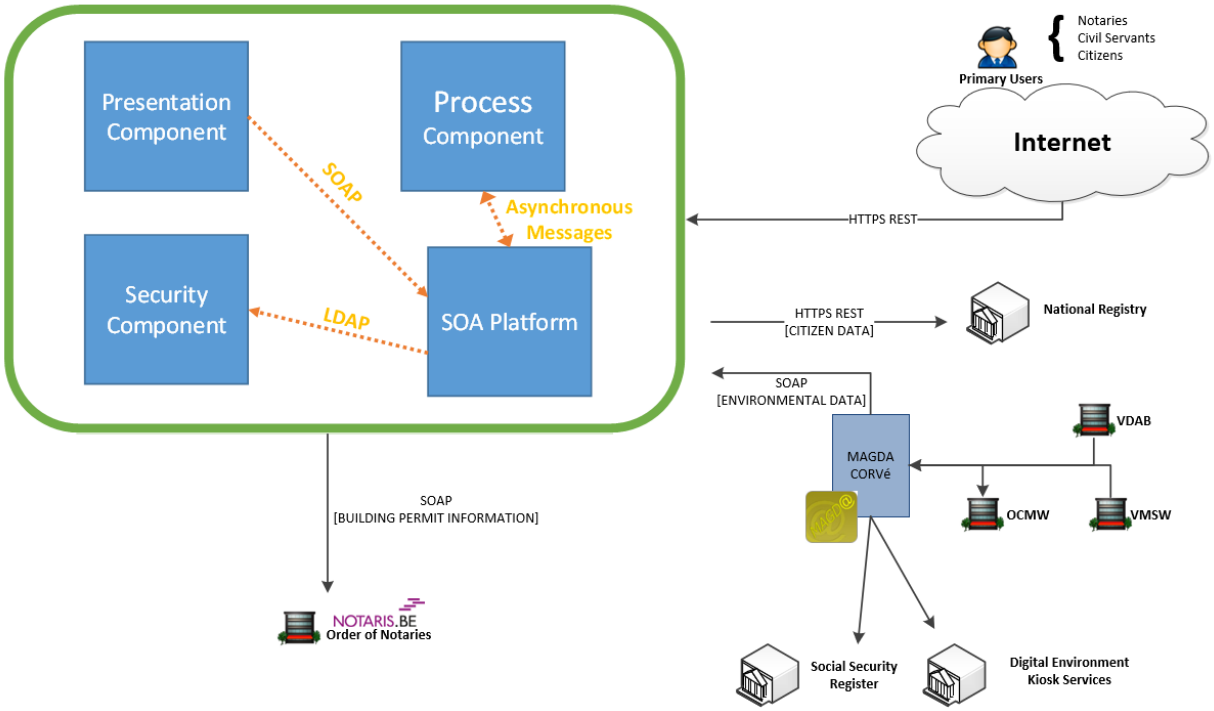


Illustration 7.2.2 – Example Integration Aspect

7.2.3 Data-Centric Concerns

The Information view of the system defines the structure of the system's stored and transient information (e.g. databases and message schemas) and how related aspects such as information ownership, flow, currency, latency and retention will be addressed. This Aspect is considered optional, and should therefore only be added if significant to the architecture and its stakeholders.

But first, it is important to understand the difference between data and information, as explained by Robert Logan and illustrated below.

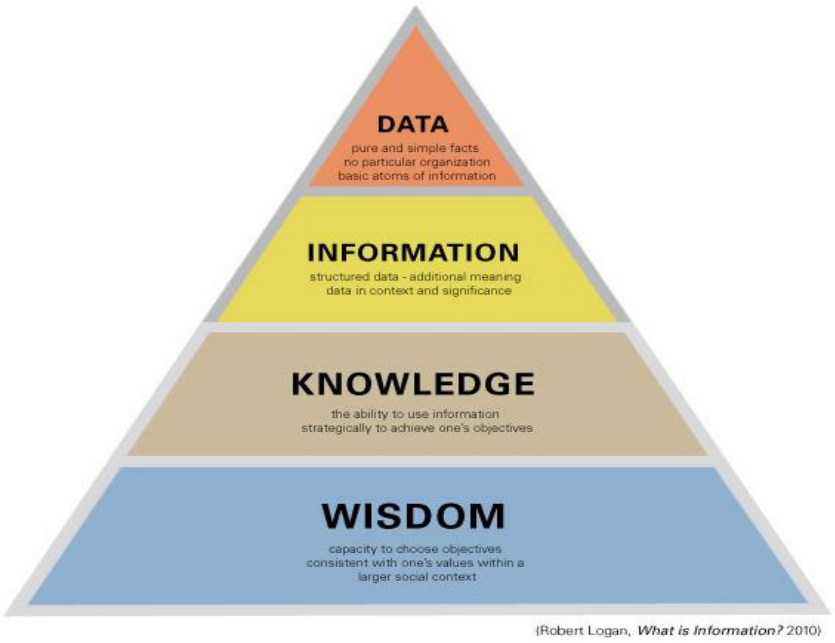


Illustration 7.2.3 – Master Data Management Architectures

Data is the representation of facts (e.g., text, numbers, images, sound, and video) for a particular subject and can be used as a basis for discussion, reasoning, measuring, and calculation. When it comes to IT, data tends to have its context and relationships held within a single IT system, and therefore can be seen as raw or unorganized. Data can also be seen as the building blocks for further analysis and investigation beyond a particular subject.

Information is quality data that has been organized within a context of increased understanding and timely access. Peter Drucker stated that "Information is data endowed with relevance and purpose." This relevance and purpose is achieved by subjecting data from one or more sources to processes such as acquisition, enrichment, aggregation, filtering, and analysis, to add value and meaning within a certain context. Therefore information must have meaning to the business in a language and format that is understood, easily consumed, and is accessible in a timely manner.

Knowledge is the insight and intelligence achieved from the formal and informal experience such as intuitive exploration and consumption of information. Knowledge can be seen as actionable information as it enables stakeholders to act in an informed manner when they fully understand its context (e.g., via interpretation of patterns, trends, and comparisons). *Wisdom* enters the realm of predictive actions. It becomes a tool and source for strategic considerations and performs predictions towards the successfulness of a course taken. This is typically accompanied by Big Data initiatives.

7.2.3.1 Information Structure

The idea of this chapter is to elaborate on the types of information that reside within the solution. We take the canonic data model provided by the business architecture, and divide it up into information classifications. An example of such a data classification, the ISO 27000 standard (Information Security Management), brings structure to data based on accessibility and confidentially: Unclassified Public, Proprietary, Client Confidential Data and Company Confidential Data. However, we do not only divide the information up in classifications based on security concerns, but also based on technical divisions, such as live data and reporting data. The individual relationships between the different elements of the canonic data model are not yet tackled unless they have an impact on the architectural choices. Translation into an Entity Relationship Diagram (ERD) for database design purposes will be done in the technical design effort.

This chapter can for example also contain the project strategies and best practices for later database design, such as for example the tactic for the structure of Primary Keys (Are they generated? Are they Composite Keys? Are there business entities that have a particular PK with for example functional meaning?). Another example is the strategy for deleting business entities from the database. Which entities require a logical delete and which can be physically deleted?

Another consideration is the use of standardized business ontologies to be used for the solution. Such ontologies usually impose requirements upon the architecture that realizes them. Such ontologies can be industry specific such as the Financial Industry Business Ontology (FIBO) or eTom for the Telecom industry, or can be linked to a specific business framework such as the SID Information Archicture from TM Forum's Frameworx methodology. It can even be scientific in nature, or related to any other semantic field.

Compliancy rules are also to be considered, as they usually have a reflection upon the technical data model. For example, compliancy to the Sarbanes-Oxley Act will force an extensive use of technical fields within the ERD for the purposes of auditing the different actions taken on the data within the solution.

7.2.3.2 Repository Structure

Some projects require the description of the different physical data repositories as this could have an impact on the Architecture. These can either be necessitated by the requirement for separation of data based on their significance of information classification, or by the requirements for a specific master data management architecture.

There are three distinct architectural flavors for realizing a master data management approach: Centralization, Registry and Coexistence. These distinctions are based on the data distribution and degree of centralization, as shown in illustration:

- *Centralization* – MDM with a central database or central business applications. This approach is widespread in practice, and allows for centralized control which standardizes the master data company-wide. The data is consistently changed through one or more business applications, and underlies the identical master data quality management or methods for guaranteeing quality.
- *Registry* – MDM with distributed data and a central directory. This approach assumes a high level of autonomy for the different system worlds, and decentralized implementation. As with database systems that are based on the "inverted lists" principle, there is a central list with references ("registry") which are required for the unique identification of a master data object specification. The actual pieces of information (master data values) are distributed between the different systems. In this concept, reading master data is connected to a high network load per retrieval instance, and is therefore recommended more for MDM approaches that provide a low degree of overall master data views.
- *Coexistence* – MDM with a hybrid approach with the coexistence of different data sources. In this approach, local copies in the distributed MDM systems are also provided and reconciled using master data distribution logistics. The copies and replications can thereby be "subtly" adjusted to the necessary degree. As with the registry approach, DQM takes place locally but with global starting points, as all relevant data are kept in the master in the "middle" (similar to the consolidated approach).

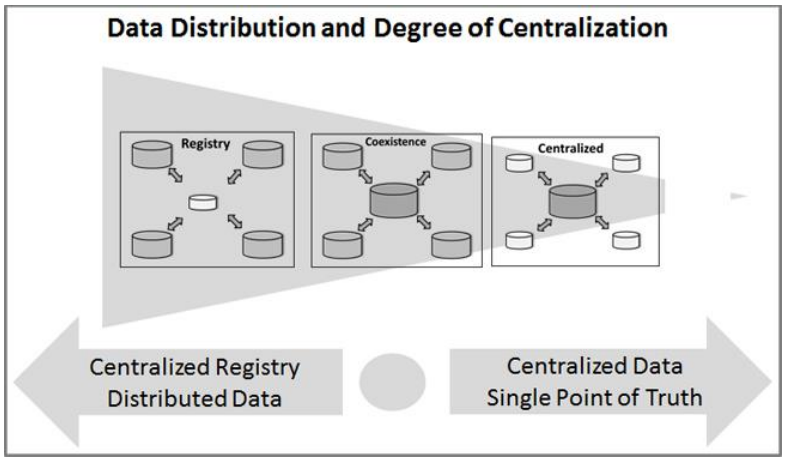


Illustration 7.2.4 – Master Data Management Architectures

Repository structures can also occur when a system has need for a storage of data based on its classification or use. For example, this will occur when we work with BPMS setups that keep data on stateful processes (or process data), which becomes irrelevant after the process has ended. This type of data follows different rules for archiving, privacy, security, etc. and therefore could be placed in a different repository to make these requirements more easily realized.

The repository descriptions should state their purpose, and should elaborate on how their information is structured. Examples of this structure are normalized diagrams, ORM-optimized, star diagrams, big data constructs such as James Kinsley's Lambda architecture, and which information classifications they contain…

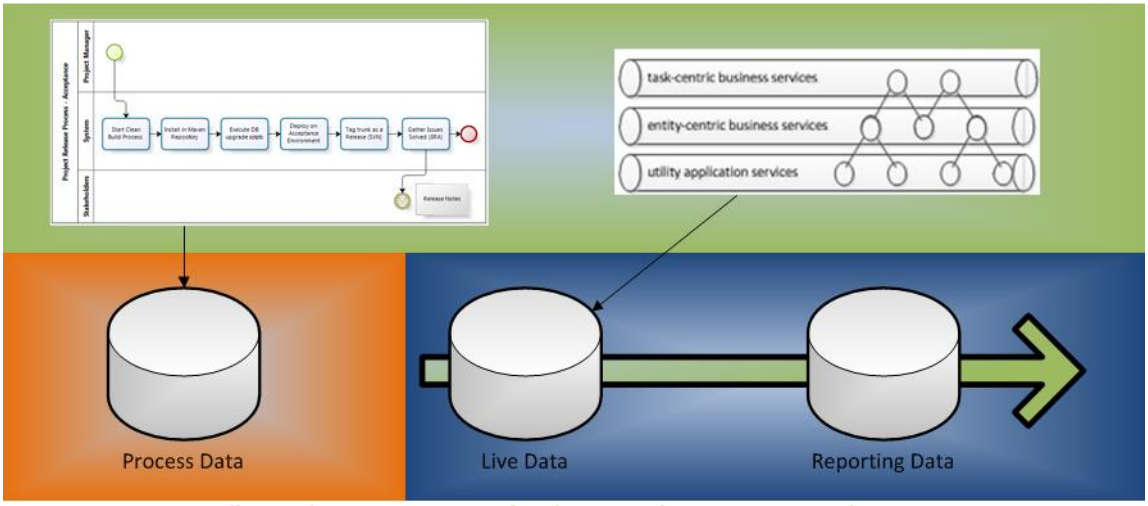


Illustration 7.2.5 – Example of a Repository Structure Diagram

7.2.3.3 Information Ownership

Information ownership refers to both the possession of and responsibility for information. Ownership implies power as well as control. The control of information includes not just the ability to access, create, modify, package, derive benefit from, sell or remove data, but also the right to assign these access privileges to others (Loshin, 2002). We define the relationships of the different elements within our Architecture to the data available in our system by using a table such as the one seen here:

Entity	System A	Subsystem B	Component C	Partner D
Entity 1	Master	r/o	Reader	Deleter
Entity 2	Delete	Master	None	Updater
Entity 3	None	Reader	Master	Reader
Entity 4	Master	None	None	Updater

In cases where the different component take ownership of a specific part of the data model, encapsulating it from the rest of the solution, and exposing it through an interface, this matrix can become quite extensive. The Thomas Erl (of SoaSchools fame) design pattern of an Entity Service Layer in a Service Oriented Architecture is an obvious example of such an approach.

As indicated in the table above, this ownership need not necessarily reside within the solution, but can be opened up to the solution by means of communication with a partner system, perhaps not even under the control of the organization the solution will be deployed in. An example of this can be found in the Belgian public sector where such information as organizational data can be fetched from the authentic source for this type of data, in this case being the Crossroads Bank for Enterprises (CBE).

7.2.3.4 Timeliness and Latency

When filled out, this chapter will outline the strategy for data archival and retention. If information needs to be copied around the system or is updated regularly, a description of how timeliness and latency requirements will be addressed is needed. This chapter will contain all relevant information towards that end.

This will entail that a plan is set for how long data will remain in the system from the moment of its creation. Typically, this will be determined by legal requirements. We usually set forth 3 levels of latency for the data, as seen in the illustration below, being Operational, Reference (or Reporting) and Archived Data. This strategy is usually applied to all data representing business entities. However, the strategy can vary from entity to entity. For example, technical logging data in the database could only have a place in the operational data, and after certain period be immediately discarded without ever having been passed into the Reference or Archived Dataset.

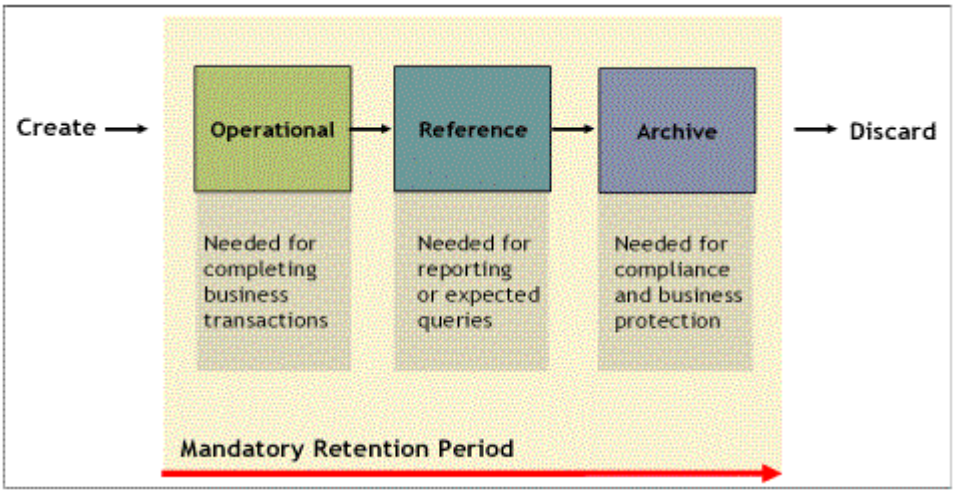


Illustration 7.2.9 – The Lifecycle of Data

7.2.3.5 Data Migration

Most projects aren't green field implementations and replace an existing application, or at least parts of it. In these cases, the data collected in the previous application can be considered valuable enough to warrant a strategy for retention in the new system. However, the old data structure is seldom directly transferable into the new system, and requires a migration to become of use in the new system. In this aspect, we plot out this strategy and how it affects our application. This aspect has a lot in common with Master Data Management projects. We will not go further into details into this realm than scratch the surface, where we stipulate that the proper approach is not a big bang data migration as specified in the Traditional approach (what we call ETL or Extract, Transform, Load) in Illustration 7.2.10, but rather an iterative process of analysis, extraction and validation.

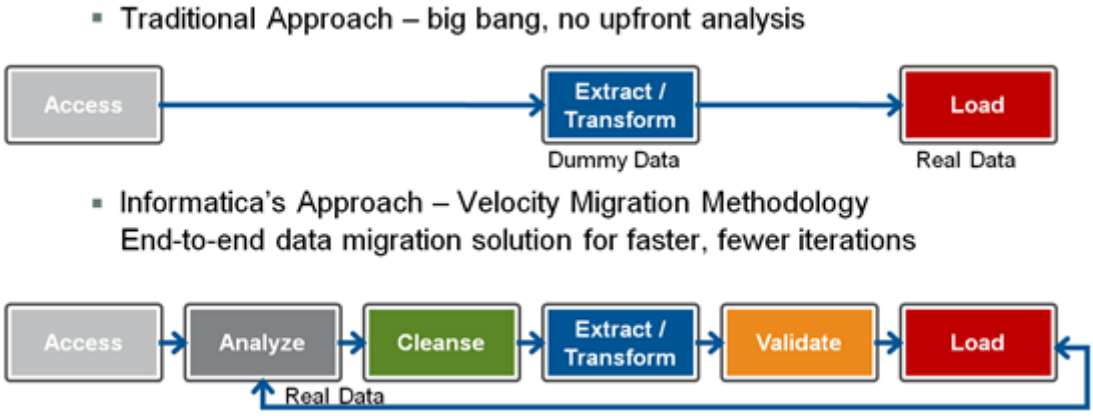


Illustration 7.2.10 – MDM Approach

7.3 Implementation View

The implementation view of the architecture defines guidelines and constraints on the software development process. It outlines the technologies and standardization of design, coding and testing, the decomposition of the system and the important architectural choices.

7.3.1 Technological Standardization

This Aspect elaborates the different technologies and standardizations to be used in the realization of the project.

7.3.1.1 Technology Stack

The technology stack can be represented graphically by extending the design we started using in the Logical View. As an example, Illustration 7.3.1 shows this using a stack, combining Spring MVC with EJBs.

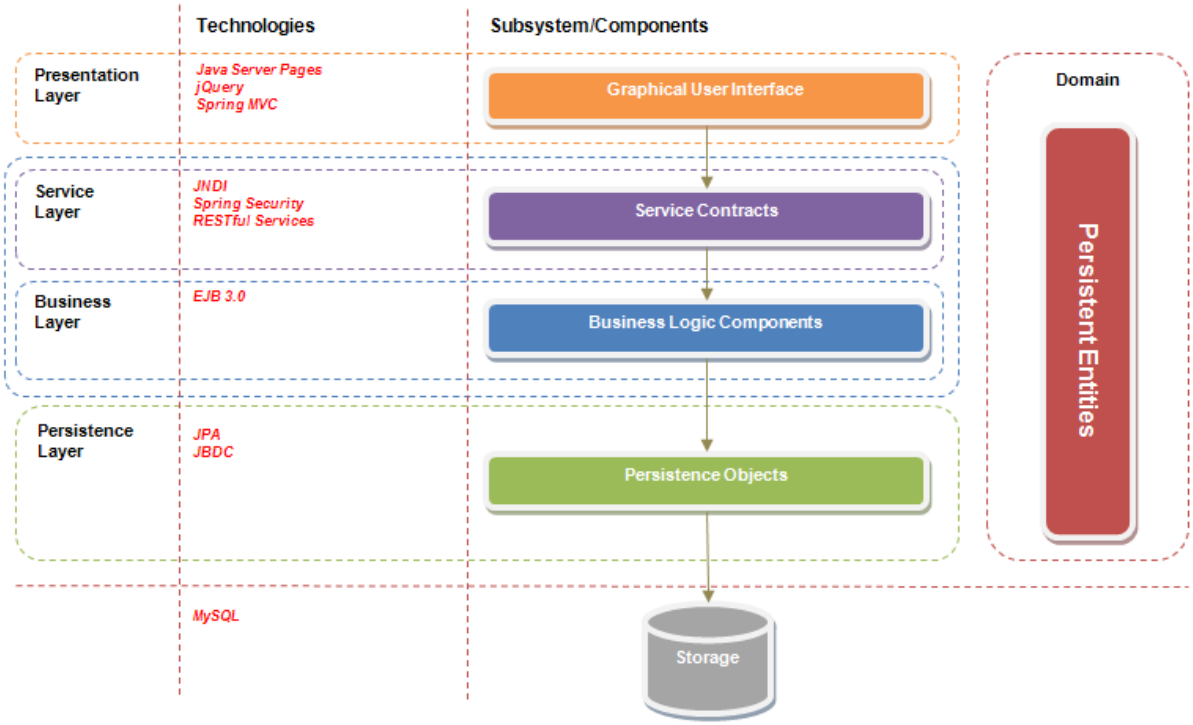


Illustration 7.3.1 – Technology Stack of a Spring Application

7.3.1.2 Standards to Follow

As for the standardizations, these are a listing of all existing standards to be applied to the project. Basically, we can divide these standards into 4 distinct groups: Industry Standards (such as for example Java Code Conventions), supplier-specific Standards, Compliancy Standards (such as for example Sarbanes-Oxly) and Custom Standards (such as Customer Demands, or project-specific requirements). An overview of this can be seen in Illustration 7.3.2. These standards can apply to technical cross-cutting concerns (Exception Handling, Logging, Transaction Management, Concurrency, Internationalization, Security…), but also structural decisions (such as package & class nomenclature, convention over coding decisions…)

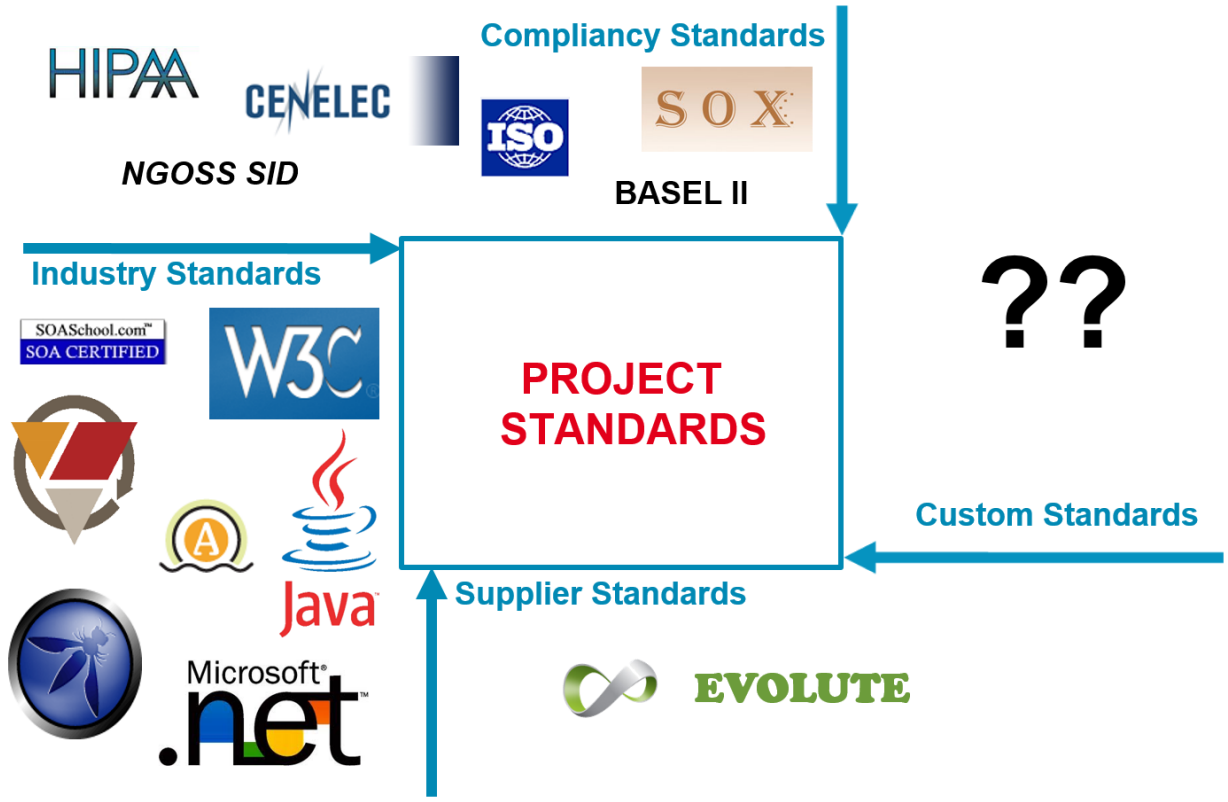


Illustration 7.3.2 – Standards applying to a Project

7.3.1.3 Application Lifecycle Management

The Project Platform off which we leverage our ALM approach is another standardization to be described in this Aspect. A common choice for Java development projects would be the Atlassian ALM strack (see Illustration 7.3.3). It also specifies any additional information on ALM processes that deviate from the standard approach.

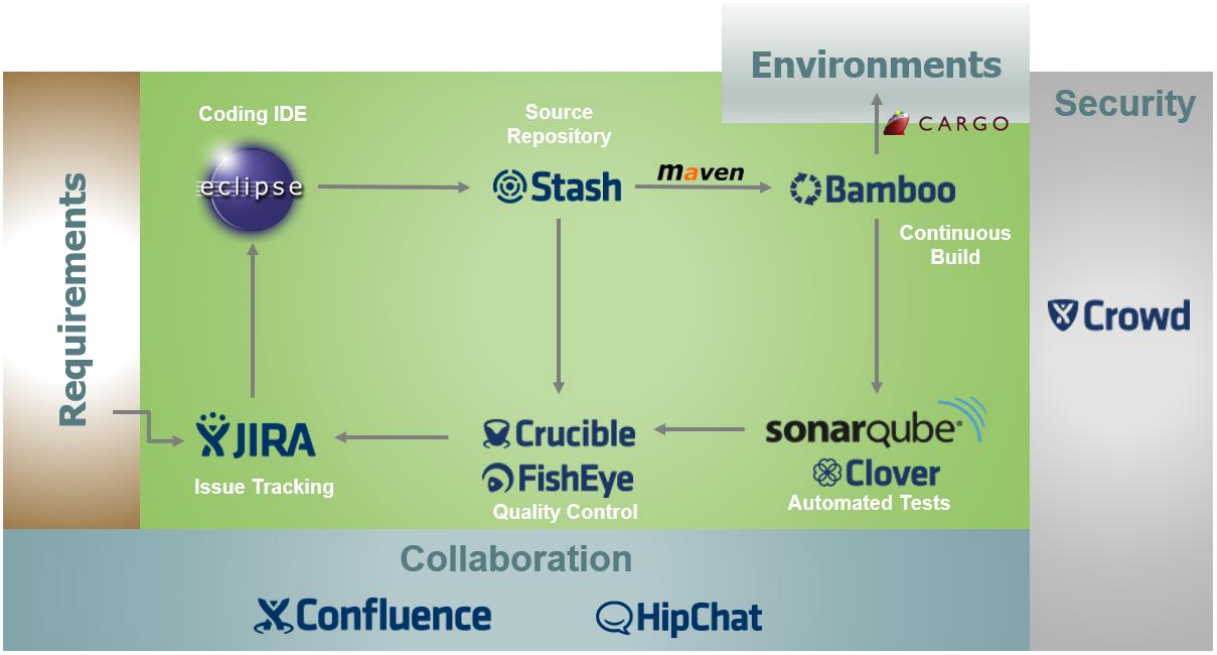


Illustration 7.3.3 – Example ALM Approach

7.3.1.4 Testing Approach

A third pillar of our standardization is the testing approach. Here we specify our testing tools and frameworks and strategies in accordance with the Master Test Plan of the project. In the document an overview such as Illustration 7.3.4 can be added. This indicates which tests should be done, when they are to be executed, and which tools or frameworks are used to perform them. The Aspect should also clarify specific targets the testing approach might have such as code coverage or performance thresholds.

Tests		Not Needed	Manual	Build	Continuous Build	Tools
Unit						
Integration						
Regression						
System	Performance					
	Load					
	Stress					
	Capacity					
Code Quality	Standards					
	Coverage					
	Reviews					
Security						
Functional						

Illustration 7.3.4 – Testing Approach

The purpose of writing **unit tests** is to ensure that the developer dry runs his code sufficiently for the desired behavior. Design patterns such as interfaces, dependency injection, inversion of control and the like make code more testable. Two different methods of unit testing can be applied. Black Box (BB) Testing is a method of unit testing where we access the code by its public interfaces without paying attention to its inner working. This can be considered as specification testing. This method is ideal for verifying the functional and technical analysis documentation of the component, but can result in the fact that not all branches of the codes within have been checked. White Box (WB) Testing is a method where we target pieces of a specific component. There is a direct link with the code within the component, yielding a larger coverage of the code, but necessitating more effort to keep automated tests in line with code evolution. Some metrics we can test for during unit testing are shown in the table below.

Type of Test	Cat	Purpose
Code Coverage	WB	Verify that all lines of code are called in at least one test in order to detect possible dead code.
Branch Coverage	WB	Verify that all permutations of a decision gateway are tested.
Error Guessing	WB	Insert possible values into the function in order to detect possible bugs and/or common mistakes, such as causing a division by zero.
Input Boundary Testing	BB	Enter into the input parameters the minimum value for each parameter, the maximum value for each parameter and a value somewhere between the two to test extreme cases/
Output Boundary Testing	BB	Devise tests that result in the return parameters to take the maximum, minimum and a medium value to determine whether they are correctly typed.

We could also consider more esoteric tests such as the Lack of Cohesion Metric (LCOM4) tests, code duplication tests, circular dependency tests and in extension package tangle tests to determine how many circular dependencies between different packages exist. This list is certainly not exhaustive. There could even be a need to introduce tests that check whether the architectural guidelines are followed. For example, tests that verify the naming convention of parameters or the existence of security coding for each function could be added to the test runs.

When we assert the proper interaction between two or more components, we use **integration tests**. These tests verify the interfaces between these components, or as tests of assumptions made during the implementation process when these interfaces are not yet known. This type of testing typically forgoes the use of stubs or mocks for any component within our proper solution.

Having a structured approach to **regression testing** is a factor in the successful delivery of any software. The regression test battery is a collection of tests which are used specifically to verify whether new releases of code still follow the requirements and therefore expected behavior of said code. Also, with all automated tests, one axiom holds

true. If a bug is detected in other testing types, and it can be reproduced automatically, such a test should be included in the regression test battery to verify that the bug has been resolved after applying the fix, as well as act as an indicator should the bug be reintroduced in later development.

For each development language there are sets of standards developers are wise to follow. The compliance to these best practices is ascertained using **code quality tests**. For example, when dealing with Java development, we use PMD, FindBugs, CheckStyle or similar tools to perform static code analysis of this compliance. These tools normally plug into the tools used in the ALM setup. They are performed as part of the continuous inspection effort, and reflect the choices made in the standardization aspect of the Implementation View.

System testing involves tests directed at the entire solution to verify the ability of the solution to deal with extraordinary circumstances that challenge the limits of the system. We verify the performance of the system, as well as perform tests geared towards stress of the system resources and its capacity.

Security testing for software is usually split up into two methods. We perform manual tests called penetration tests to verify the confidentiality, integrity and availability of the data in our solution, and we also verify the rules set out by OWASP. For each of these we either specify a mitigation or devise a manual or automated test.

Functional testing is done as part of the master test plan, and has little impact on the technical testing practices. It is included in the overview simply to specify when in the software delivery life cycle they take place.

There are also tests that lean more into the hardware aspect of the solution, such as **high availability tests**, such as the simulation of a server failure to verify failover mechanism, or an uncontrolled shutdown of a server in the cluster, or **disaster recovery testing**, to determine that the procedures for business continuity (such as the activation time for a passive site) are sufficient.

A good practice in tests that are automated is to use environment variables where possible, so that tests become independent of which environment they are executed against, and can as such be reused when environments change purpose (for example when an old acceptance environment is replaced, and it becomes the next test environment).

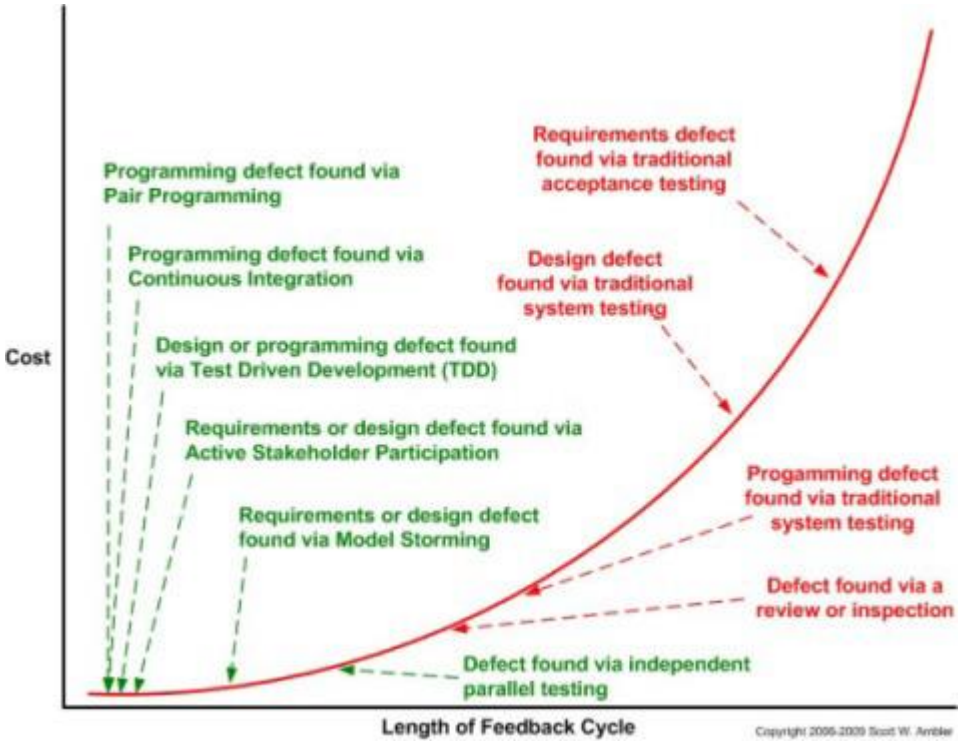


Illustration 7.3.5 – Cost Curve for Bug Fixing

As with requirements gathering, the later in the project lifecycle we detect bugs, the more expensive it becomes to fix them. As shown in the infographic above, this is also determined by the type of development the project team chose to adopt for the project.

Bear in mind that software testing eventually faces a point of diminishing returns. At some point, after the most obvious and easy to find bugs are located, the only bugs left are, by definition, those that are harder to find. The rate at which

these more complex bugs are found, decreases, and finding each bug requires more hours of testing. Architects should ask themselves whether the cost of finding these remaining bugs outweighs the loss of revenue or business productivity in not shipping the software to customers. In other words, at some point, you just have to ship the software, even knowing there are still bugs remaining, because continuing to hunt for bugs is no longer cost effective.

7.3.2 Artifact Overview Aspect

This Aspect of the system provides the view on how the logical architectural elements are decomposed and mapped on to the physical system artifacts. This can be done graphically as shown in Illustration 7.3.6.

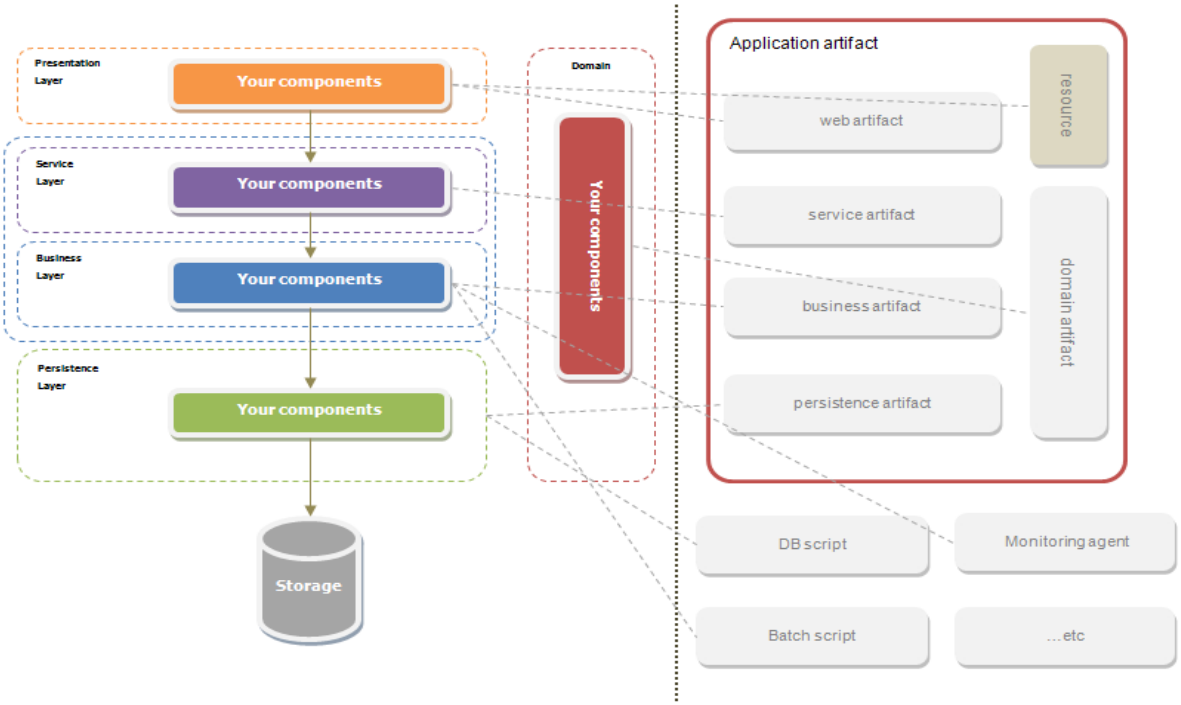


Illustration 7.3.6 – Artifact Overview

An important point to address, is which parts of your artifacts are to be environment agnostic, meaning they do not change based on which environment they are deployed in, and which artifacts will be influenced by this. A best practice is to make sure most of your artifacts fall in the former category. The latter is populated by configuration files, and data scripts with dependencies to live data.

7.3.3 Rationale Aspect

This Aspect outlines the important technological choices that have been made, the risks concerning these choices, and the impact they have on the implementation. A possible way to do this can be seen in Illustration 7.3.7. The rationale for the choices made in the logical view are evident from the coverage of the requirements that these choices list in the requirements view, as well as their comparison to how their alternatives rate.

	available resources	vendor locking	mobile support	Adappability (agility)	Version control	database abstraction	testability	testing tools	installability	learnability	supportability	technology life cycle	portability	SOA	lincensing
client															
Swing						na							na		
Jgoodies						na							na		
view															
jsf						na							na		
richfaces						na							na		
service and binding															
seam			na										na		
ejb 3.0			na			na							na		
infinispan						na									
drools			na			na							na		
guvnor			na			na	na						na		

LEGEND	
5	excellent
4	very good
3	good
2	more or less
1	low
0	non existant
nr	irrelevant

Illustration 7.3.7 – Rationale of the Architectural Choices

The first consideration to make is the decision between building the component and buying an existing solution. Or in other words, do solutions exist that cover all (or at least most) of the needs of the project, or are the workings of the organization so distinct that a custom fit is needed. When deciding on a software package, some principles should be uphold in order to secure a sound choice:

1. Buying a package means joining a network. Although the customers of such a package normally don't even know each other, this is nevertheless a common interest in keeping the product alive and well. The continued evolution of the product is in the best interest of its buyers. The training and education of the users of the product also adds to the investment an organization in the bought product. To a large extent, this investment represents a sunk cost, requiring some level of risk mitigation (such as the implications for the dispose phase).
2. Always take a long-term perspective. As the lifespan of most hardware and software decreases to only a few years due to the rapid developments in the IT landscape, the data in these systems is more durable than the tools handling it. An overview of the evolution of the product therefore gleans insight into the stability of its inner workings.
3. Safety in numbers is a saying that can also be applied to package vendors. How viable is the company that holds the intellectual property of the product? The more customers (thus the higher the market share), the more likely it is the vendor will go on in the future. Vendors losing market share (usually due to inadequate technology) are sometimes plagued with the need for rapid development in a last ditch effort to stave off the inevitable replacement.
4. Beware of vendors veering away from standards. The compatibility with other software depends on this, and every time a piece of proprietary software is added to the product, the switching cost for this product becomes higher, maybe even becoming a de facto standard, dethroning the standard, or diminishing it so that the community around it suffers the switching cost.
5. Choose a package with the right type of standardization for the organization. The types of standardization are the following:
   - *Standardization of the user interface:* A common strategy to reduce the need for user training.
   - *Standardization of input/output:* Greatly enhances the facility of connecting with other software or partners.

- *Standardization of data structure:* ensures backward compatibility with data stored in legacy systems, as well as ensure access to the data in other information systems in the future.
- *Standardization of skills:* Hiring of employees with specific skillsets or enabling a formalized approach to the training of new employees.

6. The accessibility of knowledge about the package is an important principle. When working with custom development, the organization carries the entire burden of training and retaining knowledge, whereas a package can distribute part of this burden across all its users (over the different organizations using the product). This shared knowledge should however be readily accessible, or its benefit over custom development is lost.

Another important aspect to consider is the collection of licensing constraints that govern our choices for technical components. It is imperative that these licenses do not conflict with each other nor with the purpose and requirements of the solution. In Java development, there are for example several open source licenses, each with their own rules. Some of these rules contradict each other, making it so certain frameworks cannot be used together in a single solution. An example of such conflicting licenses are the Common Development and Distribution License (CDDL) that was commissioned by Sun Microsystems and the well-known GNU General Public License. For clarity purposes, we also include a short description of each license in this chapter, as well as an overview of the period of use, should the license be constrained in time. This is information that will later be used in the service management of the solution.

7.3.4 Evolution Aspect

If the evolution is addressed in the non-functional requirements, this section should be removed. This Aspect addresses the evolution from the implementation view point and details how the architecture supports the organizational and technical evolutions.

7.4 Physical View

Although the Infrastructure Design document will make a complete 360° view of all hardware involved, in the physical view we will summarize the infrastructure information and filter it according to the impact it will have on the Architecture. This is generally preceded by a graphical overview of the “to be”-topology of the enterprise (as seen in Illustration 7.4.1). More often than not, this view will make references to topics handled in the Infrastructure Design.

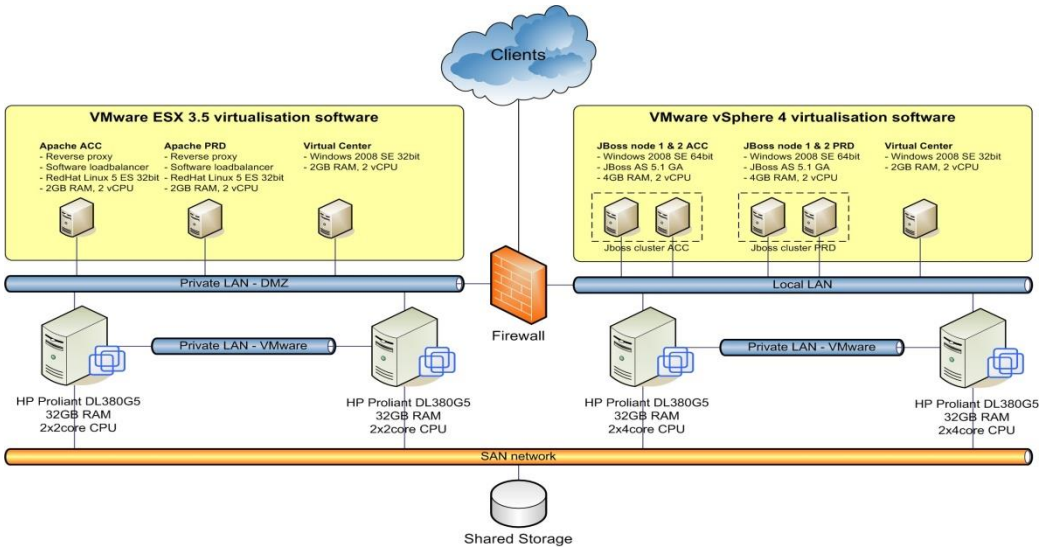


Illustration 7.4.1 – Graphical Overview of the Enterprise Infrastructure

7.4.1 Environment Topology

This view will also specify the different environments (development, test, acceptance, production...) with their rationale as to why these different environments are needed. As an example, this rationale will look something like this:

Staging Environment (AKA pre-production)
The staging site is used to assemble, test and review new versions of the solution before it goes into production. The staging phase of the software lifecycle has hardware that mirrors the hardware used in the production environment. The staging site provides a final QA zone that is separate from the Acceptance environment.

Accompanying this rationale, the physical characteristics of the environments are listed. These are comprised of three distinct parts. The first part of the physical characteristics are the hardware specifications, listing how many CPU’s the environment has, the size of the memory, physical storage media… A second part is the middleware setup. Here we specify how the setup looks for each of the environments (as in Illustration 7.4.2).

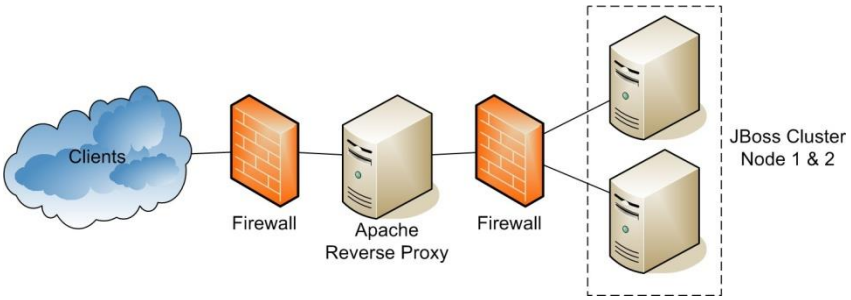


Illustration 7.4.2 – Middleware Setup Diagram

The description of the middleware should also cover a comprehensive overview of all ports that have a significance in the environment, in order to properly setup functionalities such as firewalls and load balancers. But this overview can also serve to verify no conflicts are generated by the different components within our solution.

Jboss Ports			WEB	SOA	SBM Portal	SBM EJB	GeefDossierKBI	SA-WEB	SA-SOA	AppDynamics
	Port	Description	F12	F13	F14	F15	F16	F17	F18	F19
		Port offset	**144**	**233**	**377**	**610**	**987**	**1597**	**3333**	
	1098	*Naming Service - RMI requests from client proxies*	*1242*	*1331*			2085	2695	4431	
	1099	*Naming Service - Main port*	*1243*	*1332*			2086	2696	4432	
	8009	*AJP Port*	*8153*	*8242*			8996	9606	11342	
	8080	*HTTP Port*	*8224*	*8313*			9067	9677	11413	8990
	8083	*Dynamic class and resource loading*	*8227*	*8316*			9070	9680	11416	
EJB-Server										
	16005	*Naming Service - RMI requests from client proxies*			*1475*					
	16003	*Naming Service - Main port*			*1709*					
	58009	*AJP Port*			*8996*					
	16001	*HTTP Port*			*9677*					
	16004	*Dynamic class and resource loading*			*10667*					
	16000				*4181*					
Portal-Server										
	18005	*Naming Service - RMI requests from client proxies*				*1708*				
	18003	*Naming Service - Main port*				*2086*				
	68009	*AJP Port*				*9606*				
	18793	*HTTP Port*				*10664*				
	18004	*Dynamic class and resource loading*				*12264*				
	18000					*6765*				

Illustration 7.4.3 – Port Usage Overview

Finally, we elaborate on the configuration of the database, such as for example the calculation of the initial size and the yearly extent, as demonstrated in Illustration 7.4.4. Other calculations to this effect include the load the database will be subjected to at different times, as well the number of concurrent transactions it need to be able to handle in order to guarantee demanded response times.

Entity	Size	Count	Total
ENCODING	3.5 KB	1	3.5 KB
HISTORY	1.5 KB	10	1.5 KB
INSPECTION	3 KB	15	45 KB
INFRACTION	7 KB	10	70 KB
INFRACTOR	4 KB	15	60 KB
PARTICIPANT	10 KB	11	110 KB
TASK	8 KB	10	80 KB
TOTAL			**370 KB**

This calculation gives us 370 KB per Encoding and all its related Entities. These Encodings are approximately once a day per Department, giving a total of 370 KB x 21 x 270 = 2 GB.

The DB will have to accommodate an initial size of 2 GB, and a possible extension by 2 GB every year.

Illustration 7.4.4 – Database Initial Size & Yearly Extent

7.4.2 Datacenter Topology

Much in the style of the chapter describing the repository structure for information, there is sometimes a need for the description of the specifications of the datacentres hosting the solution. Typically, we will describe the details of these datacentres when required for security purposes. This can go from detailing the physical security measures, such as the presence of the badge system, to business continuity needs, such as fire suppression systems or failover mechanisms, as stipulated by the ISO 2700 standard.

Another conceivable description is the availability and range of the scaling solutions in place for the different environments, although this could just as easily be included in the descriptions of the server topology, should this chapter prove not needed for the solution at hand.

7.4.3 Deployment Aspect

Accompanying all these environments are their related deployment diagrams. They specify how to deploy the different software artifacts, database scripts... of our solution onto each environment. Added to this diagram, a clarification should be added detailing the time schedules for each of these changes (can they occur daily, weekly, on notification,

impromptu...), as well as a list of stakeholders that are allowed to trigger a deployment. Usually this level of detail is only provided for the Acceptance and Production environments.

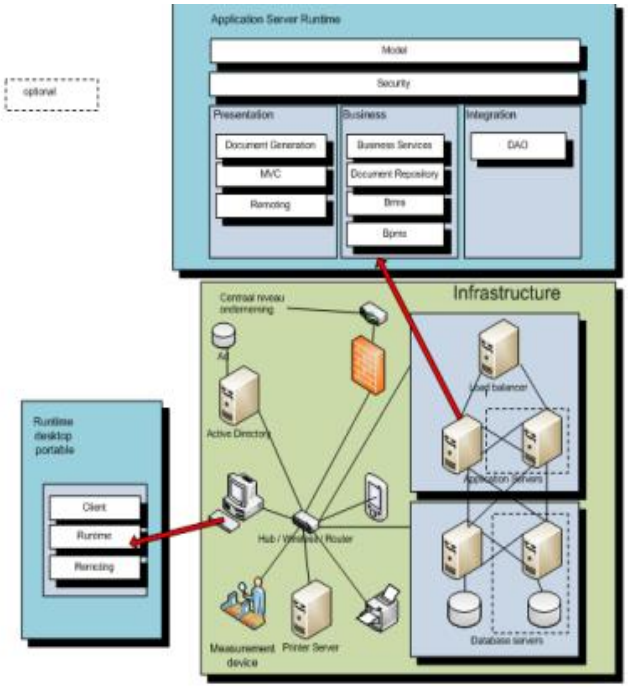


Illustration 7.4.5 – Deployment Diagram Example

7.4.4 Communication Channels

The communication channels between the different physical components of the architecture are also described in the Physical View. Not only should the technologies and protocols be listed, it is also important to describe the physical characteristics of these communication channels and which cross-cutting concern is handled how in the solution. These characteristics should be mapped onto the different communication channels we have identified in the Integration Aspect of the Logical View.

One of the most important cross-cutting concerns is **security**. For each of the communication channels, there should be some elaboration on how this channel is secured, and if not, why this is not needed. For example, this relates to whether the web interface is encrypted on the channel (HTTPS) or through message encrypting. A strategy for the exposure of web services could be stated here, or in the case of EJBs, which decisions have been made to decide whether a method should be accessible in a remote fashion.

Another of these concerns, namely **reliability**, could then be described in terms of SLAs enforced on the channels, which could be linked directly to the existence of measures for **failover** behavior and **efficiency**. The number of cross-cutting concerns to be described for each of the communication channels can be derived from the relevant non-functional requirements.

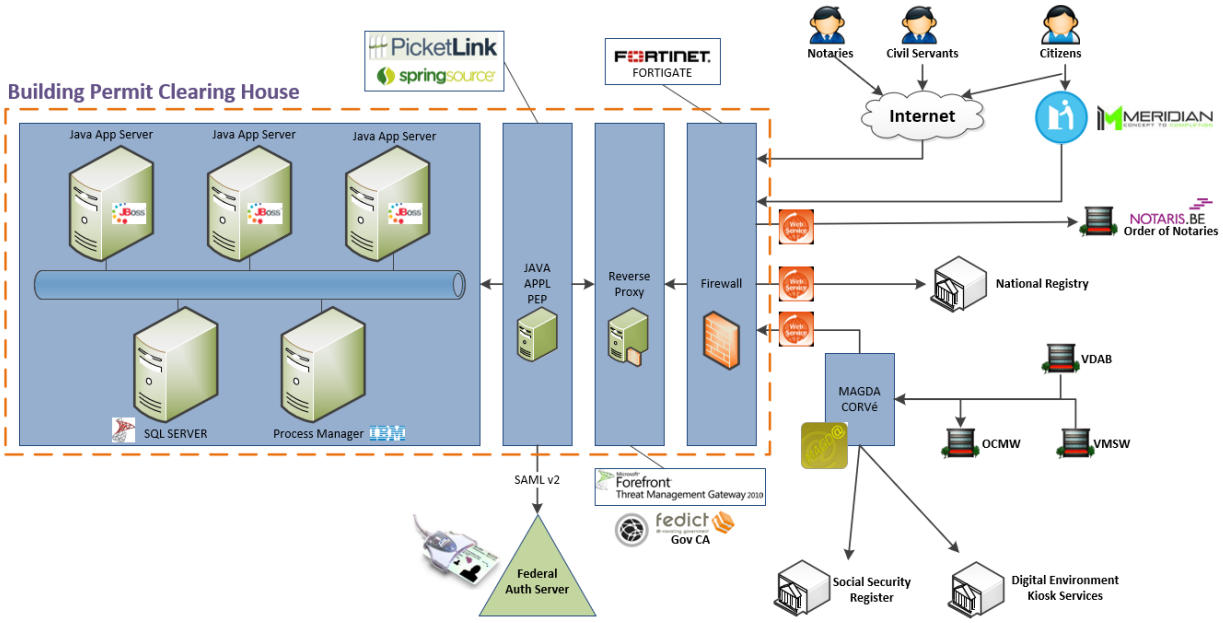


Illustration 7.4.6 – Communication Channels Diagram Example

The detail in which the different channels are described is dependent on the project, however, most of the time, a full-blown OSI layer description such as the one in the illustration below, is seldom needed. We mostly only specify the session to application layer, with the occasional mention of the transport layer for security reasons, as stipulated earlier.

Illustration 7.4.7 – OSI Layer Example

7.5 Operational View

The Operational View of the architecture presents the missing parts of the architectural answer to the operations-related requirements with a focus on the procedures to be followed by the service management team. Where previous viewpoints elaborate the tools the solution will provide to the service management team, here we list the different procedures that team must employ to effectively use the tools provided. These procedures can range over a vast number of topics, such as how to determine the sanity of the system, or how to write up proper release notes. Just as with the requirements view, it should serve as an overview and checklist of the different work procedures, rather than define them. These procedures are most likely already elaborated in existing documentation within the organization, or will be elaborated in the service management document for this project.

Some of the descriptions added to this part of the solution architecture document will be very project dependent (fueled by the requirements), such as for example the need for a disaster recovery procedure, or the implementation of a security breach protocol.

7.5.1 Operational Timeline

Each solution should have an operational timeline described in this chapter. This is an overview of all relevant happenings on the solution platform, which could influence its behaviour. For example, the execution of a scheduled batch might have significant impact on the available resources, and thus influence performance a great deal. An example of such a timeline can be seen in the illustration below. This will also become a checklist for empty slots in which to insert additional services that require a heavy load on resources such as CPU or I/O.

	MO	TU	WE	TH	FR	SA	SU
0:00							
1:00							
2:00	DB BACKUP	DB BACKUP	DB BACKUP	DB BACKUP	DB BACKUP	DB BACKUP	DB BACKUP
3:00	DB BACKUP	DB BACKUP	DB BACKUP	DB BACKUP	DB BACKUP	DB BACKUP	DB BACKUP
4:00	DB BACKUP	DB BACKUP	DB BACKUP	DB BACKUP	DB BACKUP	DB BACKUP	DB BACKUP
5:00	DWH population	DWH population	DWH population	DWH population	DWH population	DWH population	DWH population
6:00	Sync Service Calls	Sync Service Calls	Sync Service Calls	Sync Service Calls	Sync Service Calls	Sync Service Calls	
7:00							
8:00							Weekly Mgmt Report Generation
9:00							
10:00							
11:00							
12:00							
13:00							Patch & Upgrade Window
14:00						Release Window	
15:00							
16:00							
17:00							
18:00							
19:00							
20:00							OS Optimalization Processes
21:00							
22:00							
23:00	Full Virus Scan	Full Virus Scan	Full Virus Scan	Full Virus Scan	Full Virus Scan	Full Virus Scan	Full Virus Scan
Legend	Business Hours						
	System Offline						
	Heavy System Load						

Illustration 7.5.1 – Operational Timeline

The granularity of the timeline is dependent on the needs of the project, with the weekly view such as shown here usually being the standard, but one might imagine different behaviour for each day of the month or even making these timelines split per resource (CPU, Memory, I/O, bandwidth…). Another possible split is per component of the solution, if these should have dedicated resources.

7.5.2 System Installation & Upgrade Strategy

Although the proper processes for installation and upgrading are described in the installation instructions, and the locations of our different artifacts are to be found in the Physical View, this view allows us to describe any supporting elements of our application for these processes. This can for example be a database table tracking all database scripts already executed on its environment, or the creation of a sanity check within the application which can be accessed after a new deploy to verify whether the deploy did not create any instabilities. Or the rules for a communication plan covering a new release of the solution. These supporting elements can also be positioned at the middleware level, such as an application server cluster to support deployment without downtime.

7.5.3 Performance Monitoring

To measure is to know. In order to be able to tune our solution and increase its performance, we first need to determine which component is lacking. In some business models, we can even go so far as to say that a lack of attention for performance can be a very costly affair. A research paper by the Oracle Applications User Group from 2009 yields some interesting graphics on this subject. We can clearly see negative experiences are very badly tolerated by customers, who cite slow response times as 80% of such experiences.

Figure 2: Typical Reactions to Web Application Glitches (From a Customer's Point of View)

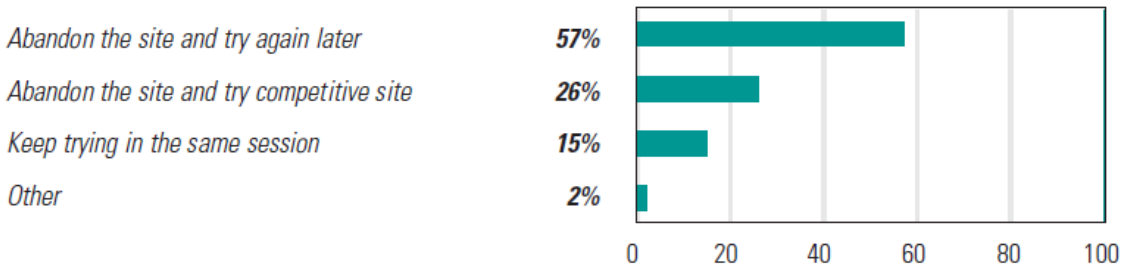


Figure 3: Impact of Negative Website Experiences on Future Purchases (From a Customer's Point of View)

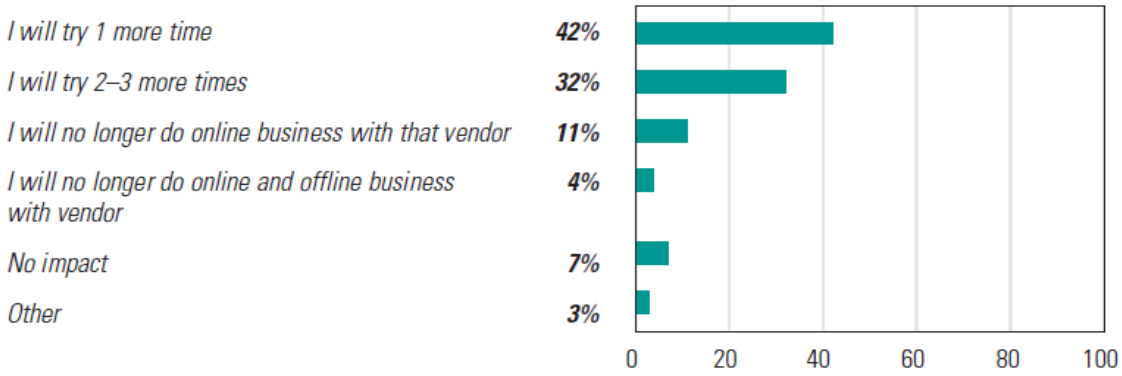


Illustration 7.5.2 – OAUG Statistics

So to be able to monitor the performance in Operate phase, we must be able to provide the necessary metrics. We specify in this aspect the steps necessary to expose these metrics to the proper monitoring instances. However, there are various degrees of granularity when it comes to performance monitoring. We categorize these much like the "Maslow Hierarchy of Needs" as proposed by Alois Reitbauer. These levels can be seen in Illustration 6.5.2. The level of granularity needed for the project is usually determined by the SLA's, but usually a proper mix of cost, time/process and quality concerns is reached.

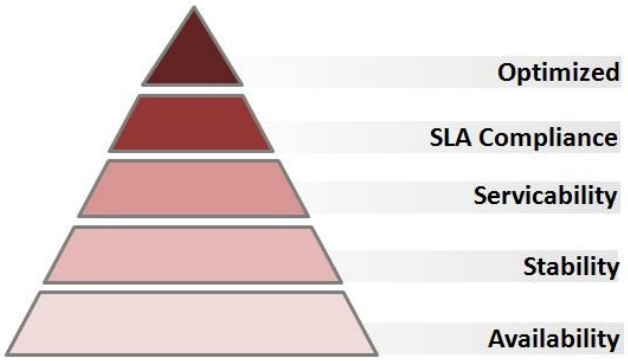


Illustration 7.5.3 – Maslow applied to Performance Management

The first level (availability) involves the monitoring of basic system resources such as CPU, memory or physical disk space, as well as availability checks of the system and the application level (pings, custom availability checks…). The next level (stability) is to ensure that our environment is working properly. These metrics are usually drawn using a management interface, such as for example JMX and JSR-77 in the Java world or PerfMon counters for .NET. The idea is to verify whether our environment is working as intended.

The first two levels don't concern themselves with the user perspective of the application. This is where level 3 (serviceability) comes into the picture. This level is addressed by application monitoring, where metrics are provided at the application level informing us of the individual transactions of users. We can learn for example the response times of the application from monitoring at this level.

The next level (SLA compliance) is used to ensure the level of performance needed by the users of the application. This involves end-user monitoring to determine the "feel" of the performance with our user base. End-user experience is the performance perceived by the actual user at a specific point in time. As we handle SLA's at this level, a methodology for resolving violations of any SLA can be envisioned in this phase. To solve such a violation, advanced metrics such as component-level information (sometimes down to the coding level) are required. This could include database calls, web service calls… At this level, we can also set up provisions for continuous monitoring of these metrics to enable proactive maintenance.

The final level (proactive performance management) is striving for the maximum performance with minimal resource usage. This is an iterative process that never ends, and is more of a strategic consideration than a technical one. We determine the next performance drain, and try to fix it.

7.5.4 Configuration Management

The traditional software configuration management (SCM) process is looked upon by practitioners as the best solution to handling changes in software projects. It identifies the functional and physical attributes of software at various points in time, and performs systematic control of changes to the identified attributes for the purpose of maintaining software integrity and traceability throughout the software development life cycle. The supporting tools for this on a design time level are typically handled by the tools determined in our ALM approach. However, some configurations can be delegated to run time level. A typical example for this is the management of the security aspect (or more specifically Access Control Management) which is usually handled by a component of the implemented solution.

8 Where we use it

No project should be attempted without thinking about the architecture and formalizing it by writing up a Solution Architecture document. It is a best practice. However, project size do vary, and in light of this, tailoring should be applied responsibly. Not all projects require all documentation. And even within documents such as the Solution Architecture Document. As it is with the chapters within the template for the Solution Architecture document. However, it is wise to keep these chapters as a form of checklist to verify that all angles have been covered, and when a chapter handles a topic that is not needed, to write a brief rationale in the chapter of why it is not needed.

There will also be deviations in the standard architectural template, depending on the type of architecture we are describing. We can divide these different types into three different groups as shown in the illustration. Oracle uses a similar approach to architecture by calling these categories perspectives, although they only make the distinction between a technical and an industry perspective.

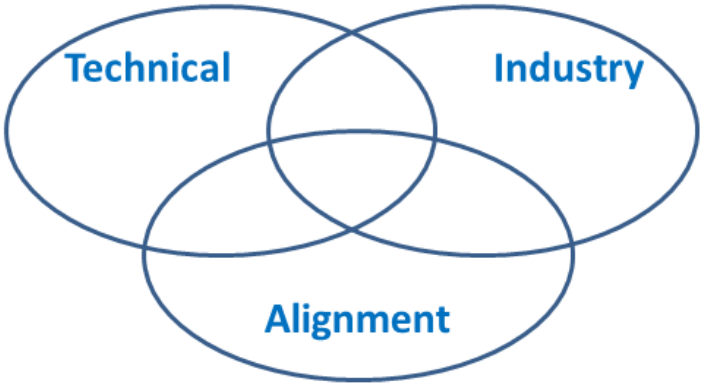


Illustration 8.1 – Architectural Categories

A technical architecture is an architecture where the concepts and best practices of the architecture are predominantly determined by the technology behind them. Examples of this category are Service Oriented Architectures, Event Driven Architectures, and the like. Industry architectures are the other side of the coin, and are almost completely dominated by how the industry to which they apply is structured. We have for example telecom architectures, financial architectures, public sector architectures, and so on. The final category are the alignment architectures, and these are the architectures that although they still have strong technological ties, they likewise incorporate industry concepts. These are the BPM architectures, the CRM architectures, the ERP architectures…

It is very unlikely that a single architecture type will be able to cover all requirements set out for any solution. Most solutions have several applicable architecture types. In these cases, it is imperative that a hierarchy be made clear between these choices, as indicated in the chapter about design constraints. This way it will become easier later on to decide which best practice to follow, should the best practices of the applicable architecture contradict each other.

9 What Comes After

9.1 Technical Design

The detailing of the components identified in the architecture document occurs in the accompanying technical design. Where the architecture stipulates the scope, guidelines and best practices for the technical design to work in, the technical design does the same for the development and configuration of the individual components. This design happens in two phases of differing granularity. First we devise a global technical analysis, drawing the broad strokes of the design, and then we proceed to go into detail for each of the identified components. As discussed previously, these components are those we identified in the logical view of the solution.

A global technical design consists of 4 major parts. The first description is the component overview. As opposed to the logical view, this diagram doesn't show these components in relation to the requirements they cover, but in relation to the archives and source repositories making up the development environment. An example of this can be seen in illustration 9.1. This diagram is accompanied by a brief description of each of the components, as well as their proper technical design, or in the case their documentation warrants a separate document (usually due to the size of the documentation), a reference to the proper document.

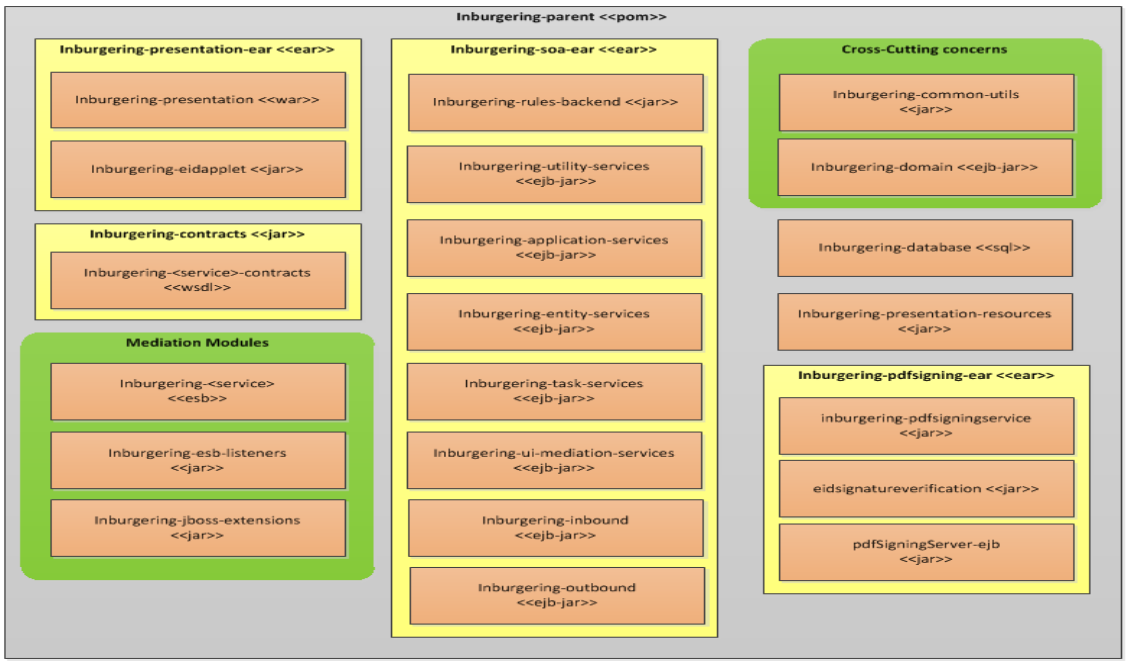


Illustration 9.1 – Example for a Component Overview

Another piece of the global technical design is the domain specification. In this chapter we go into detail about the Information Aspect of the Logical View. We translate the data model from the business architecture and functional analyses into an Entity Relationship Diagram, and complete this description with guidelines for data validation, possible data transformations due to technical processes, and the technical implementation of the auditing requirements. As with the context diagram descriptions, these parts can also be described using a reference to a document containing this information.

The third factor in a technical design is the overview of the non-functional and cross-cutting concerns. Typically, we discuss the guidelines that are ubiquitous in all components of the architecture, and how we implement these concerns. Think here of such concerns as internationalization (I18N), transaction management, exception handling, logging, security…

Finally, we describe the build process. How we construct the artifacts for deploy, as well as where to find the configurations of each of the environments. In build scenarios, always aim to make all artifact environment independent, so that artifacts should not be built specifically for a target environment. This will make you build more robust, and less error-prone. Do not forget to highlight how this build process interfaces with your application lifecycle management approach, as described in the Implementation View of the architecture.

The specific technical designs per component, either as part of the component overview or as a separate document, are harder to put into a generic structure, as each of these components has their proper concerns and best practices to guide them, based on the type of component or its function. For example, a visualization component might have chapters on how its data input validation works, which JavaScript constructs where made or used to render it Web2.0, and how it realized the user experience style guide of the solution, where an SOA entity layer component would describe the rules for the service contract and the transaction boundaries of the individual services. However, some recurring information can be considered to be present in any specific technical design. These generic structure items are the naming conventions within the component, an overview of internal structure and its essential parts and possible configurations.

A third type of technical design exist, namely the technical index. These documents serve mainly as a tool for impact analysis in complex orchestrations of components. For example, when multiple components are strung together in an automated business process, we describe this orchestration along with the connections with the individual components. Complex orchestrations taking place in the ESB (for example using BPEL), which are not described in a proper document, can also be the content of such an index document.

9.2 Software Estimation

At some point in any project, the implementation crew will be asked to give estimates of the solutions that will need to be constructed. This can happen as early as the presales effort, and will extend will into the technical design effort. It is by definition an effort that will not be exact. The only certainty you have about estimates is that they will be wrong. There are many techniques about that try to attack this problem, such as Function Point Analysis, Story Point Analysis or COCOMO. We will not elaborate on these in this document, but rather set out a few ground rules when attempting estimates.

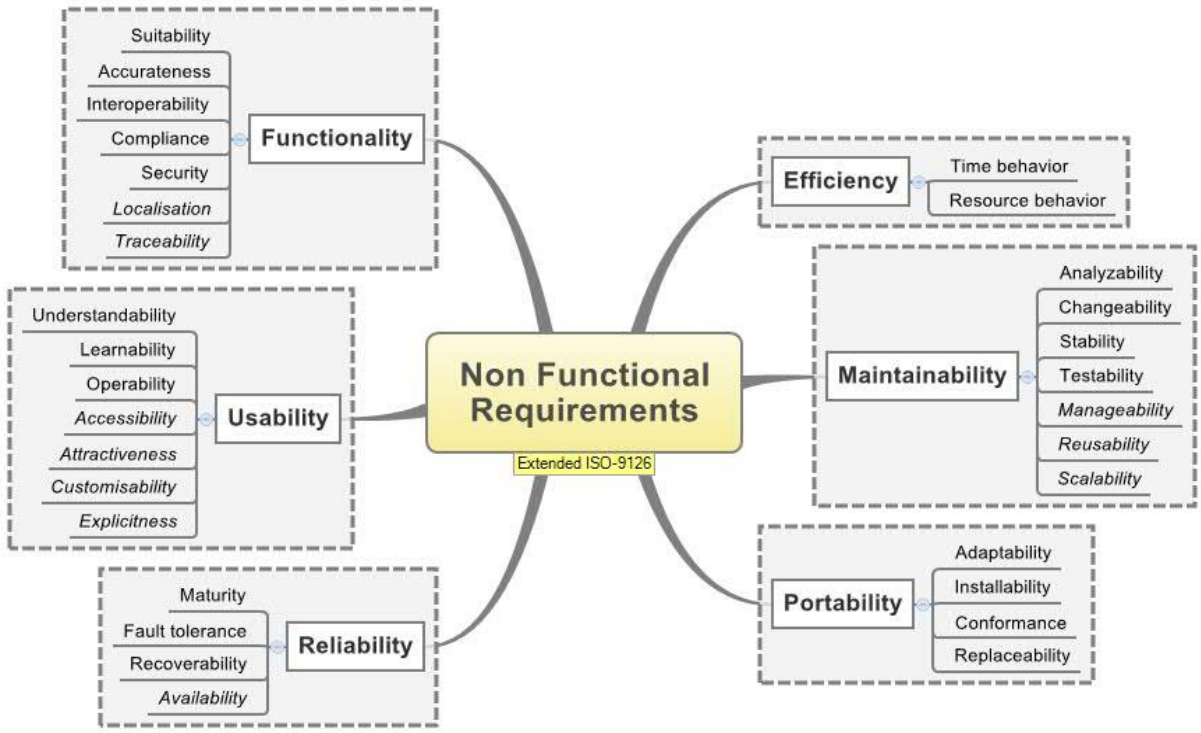
First off, there is a risk when giving out absolute values for estimates, that they will be taken by the other stakeholders as such, and that they will try to hold you to these estimates. Try to convey a level of uncertainty by giving estimates as a range with a low-high range. Return to these estimates when the particularities of the solution to be built becomes clearer, iteratively narrowing this range. It is a good practice to plan these moments of reevaluation into the project plan, so they are clear from the start, and can be taken into account.

Another common difficulty is being able to estimate a complex solution all at once. The key here is to break up the large, complex project into smaller, more manageable tasks (much like a work breakdown structure). Estimate these smaller tasks, and then combine these estimates into a single estimate for the entire project. Do not mistake this combination for a simple addition of the estimations, as additional complications might arise from combining smaller tasks into a coherent whole.

If pressed for an exact number, and not a range, a simple buffer can be provisioned in the estimates based on the standard deviation of the range. The following formula can be used, with a chosen risk aversion based on the software you are developing and its uncertainties:

```
Standard Deviation = (Worst Estimate – Average Estimate) /2
Buffer = Average Estimate + Standard Deviation * Risk Aversion
Single-Value Estimate = Standard Deviation + Buffer
```

Appendix: Evolution from ISO 9126 to 25010



The International Standard ISO/IEC 25010 revises ISO/IEC 9126-1:2001, and incorporates the same software quality characteristics with some amendments:

- Context coverage has been added as a quality in use characteristic, with subcharacteristics *context completeness* and *flexibility*.
- *Security* has been added as a characteristic, rather than a subcharacteristic of functionality, with subcharacteristics *confidentiality*, *integrity*, *non-repudiation*, *accountability* and *authenticity*.
- *Compatibility* (including *interoperability* and *co-existence)* has been added as a characteristic.
- The following subcharacteristics have been added to existing product quality characteristics: *functional completeness, capacity, user error protection, accessibility, availability, modularity* and *reusability*.
- Compliance with standards or regulations that were subcharacteristics in ISO/IEC 9126-1 are now outside the scope of the quality model as they can be identified as part of requirements for a system.
- When appropriate, generic definitions have been adopted to extend the scope to computer systems, rather than using software-specific definitions.
- To comply with ISO/IEC Directives, definitions have been based on existing ISO/IEC when possible, and terms defined in this International Standard have been worded to represent the general meaning of the term.
- Several characteristics and subcharacteristics have been given more accurate names.

The following table lists the differences between the characteristics and subcharacteristics in the new International Standard, and ISO/IEC 9126-1:2001:

ISO/IEC 25010	ISO/IEC 9126-1	Notes
Quality in use	**Quality in use**	Quality in use is now a system quality
Effectiveness	Effectiveness	
Efficiency	Productivity	Name aligned with efficiency in ISO/IEC 25062 and ISO 9241-11
Satisfaction	Satisfaction	
Usefulness		No previous subcharacteristics

Trust		No previous subcharacteristics
Pleasure		No previous subcharacteristics
Comfort		No previous subcharacteristics
Freedom from risk	Safety	
Economic risk mitigation		No previous subcharacteristics
Health and safety risk mitigation		No previous subcharacteristics
Environmental risk mitigation		No previous subcharacteristics
Context coverage		Implicit quality issue made explicit
Context completeness		New subcharacteristic (it is important that a product is usable in all required contexts of use)
Flexibility		New subcharacteristic (enables a product to be used in new contexts of use)
Product quality	**Internal and external quality**	Internal and external quality combined as product quality
Functional suitability	**Functionality**	New name is more accurate, and avoids confusion with other meanings of "functionality"
Functional completeness		Coverage of the stated needs
Functional correctness	Accuracy	More general than accuracy
Functional appropriateness	Suitability	Coverage of the implied needs
	Interoperability	Moved to Compatibility
	Security	Now a characteristic
Performance efficiency	**Efficiency**	Renamed to avoid conflicting with the definition of efficiency in ISO/IEC 25062
Time behavior	Time behavior	
Resource utilization	Resource utilization	
Capacity		New subcharacteristic (particularly relevant to computer systems)
Compatibility		New characteristic
Co-existence	Co-existence	Moved from Portability
Interoperability		Moved from Functionality
Usability		Implicit quality issue made explicit
Appropriateness recognizability	Understandability	New name is more accurate
Learnability	Learnability	
Operability	Operability	
User error protection		New subcharacteristic (particularly important to achieve freedom from risk)
User interface aesthetics	Attractiveness	New name is more accurate
Accessibility		New subcharacteristic
Reliability	**Reliability**	
Maturity	Maturity	
Availability		New subcharacteristic
Fault tolerance	Fault tolerance	
Recoverability	Recoverability	
Security	Security	No previous subcharacteristics
Confidentiality		No previous subcharacteristics
Integrity		No previous subcharacteristics
Non-repudiation		No previous subcharacteristics

Accountability		No previous subcharacteristics
Authenticity		No previous subcharacteristics
Maintainability	**Maintainability**	
Modularity		New subcharacteristic
Reusability		New subcharacteristic
Analysability	Analysability	
Modifiability	Stability	More accurate name combining changeability and stability
Testability	Testability	
Portability	**Portability**	
Adaptability	Adaptability	
Installability	Installability	
	Co-existence	Moved to Compatibility
Replaceability	Replaceability	