



# **CESAM:**

# **CESAMES Systems**

# **Architecting Method**

# **A Pocket Guide**

January 2017

© C.E.S.A.M.E.S. 2017

This work is subject to copyright. All rights are reserved by C.E.S.A.M.E.S., whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and thus free for general use.

Permissions may be sought directly from CESAM Community – email: [contact@cesam.community](mailto:contact@cesam.community).

C.E.S.A.M.E.S. and the author are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither C.E.S.A.M.E.S., nor the author give a warranty, express or implied, with respect to the material contained herein, or for any errors or omissions that may have been made, but also for the use of this material on any kind of system design or development project.

## **Publisher**

CESAM Community

CESAM Community is managed by C.E.S.A.M.E.S, non-profit organization created under the French law of July 1, 1901.

15, rue La Fayette – 75009 Paris – France – email: [contact@cesam.community](mailto:contact@cesam.community)

Website: <http://cesam.community/>

SIRET: 518 815 741 00039

# Table of contents

Table of contents.....	3
Preface.....	7
CESAM Community and CESAMES Association .....	7
CESAMES Systems Architecting Method (CESAM) .....	8
How to Read this Pocket Guide? .....	8
CESAM Pocket Guide Organization .....	9
Chapter 0 – Introduction to CESAM .....	11
0.1    CESAM: a Mathematically Sound System Modeling Framework.....	11
0.2    CESAM: a Framework focused on Complex Integrated Systems .....	15
0.3    CESAM: a Collaboration-oriented Architecting Framework.....	18
0.4    CESAM: a Business-oriented Framework .....	21
Chapter 1 – Why Architecting Systems? .....	23
1.1    Product and Project Systems.....	23
1.2    The Complexity Threshold.....	25
1.3    Addressing Systems Architecting becomes Key .....	27
1.4    The Value of Systems Architecting .....	31
1.5    The key Role of Systems Architects.....	33
1.7    How to Analyze a Systems Architect Profile ? .....	35
Chapter 2 – CESAM Framework .....	37
2.1    Elements of Systemics .....	37
2.1.1    Interface .....	37
2.1.2    Environnement of a System .....	38
2.2    The three Architectural Visions .....	39
2.2.1    Architectural Visions Definition.....	39
2.2.2    Architectural Visions Overview .....	42
2.2.2.1    Operational Vision .....	42
2.2.2.2    Functional Vision .....	43
2.2.2.3    Constructional Vision.....	45
2.2.3    Relationships between the three Architectural Visions.....	46
2.2.4    Organization of a System Model .....	48
2.3    CESAM Systems Architecture Pyramid.....	49

2.3.1	The three Key Questions to Ask .....	49
2.3.2	The Last Question that shall not be Forgotten.....	51
2.4	More Systems Architecture Dimensions .....	52
2.4.1	Descriptions versus Expected Properties .....	52
2.4.2	Descriptions.....	53
2.4.2.1	States .....	53
2.4.2.2	Static Elements .....	55
2.4.2.3	Dynamics .....	58
2.4.2.4	Flows.....	60
2.4.3	Expected Properties .....	61
2.5	CESAM Systems Architecture Matrix .....	65
Chapter 3 – Identifying Stakeholders: Environment Architecture .....		69
3.1	Why Identifying Stakeholders? .....	69
3.2	The key Deliverables of Environment Architecture .....	70
3.2.1	Stakeholder Hierarchy Diagram .....	71
3.2.2	Environment Diagram .....	72
Chapter 4 – Understanding Interactions with Stakeholders: Operational Architecture.....		75
4.1	Why Understanding Interactions with Stakeholders? .....	75
4.2	The key Deliverables of Operational Architecture .....	77
4.2.1	Need Architecture Diagram.....	77
4.2.2	Lifecycle Diagram.....	78
4.2.3	Use Case Diagrams .....	79
4.2.4	Operational Scenario Diagrams.....	81
4.2.5	Operational Flow Diagram.....	81
Chapter 5 – Defining What shall Do the System: Functional Architecture .....		85
5.1	Why Understanding What Does the System? .....	85
5.2	The key Deliverables of Functional Architecture .....	87
5.2.1	Functional Requirement Architecture Diagram .....	87
5.2.2	Functional Mode Diagram.....	88
5.2.3	Functional Decomposition & Interaction Diagrams .....	89
5.2.4	Functional Scenario Diagrams .....	90
5.2.5	Functional Flow Diagram.....	91
Chapter 6 – Deciding How shall be Formed the System: Constructional Architecture.....		93
6.1	How to Understand How is Formed the System? .....	93

6.2	The key Deliverables of Constructional Architecture .....	95
6.2.1	Constructional Requirement Architecture Diagram.....	95
6.2.2	Configuration Diagram .....	96
6.2.3	Constructional Decomposition & Interaction Diagram .....	97
6.2.4	Constructional Scenario Diagram .....	98
6.2.5	Constructional Flow Diagram .....	99
Chapter 7 – Choosing the best Architecture: Trade-off Techniques.....		101
7.1	Systems Architecting does usually not Lead to a Unique Solution .....	101
7.2	Trade-off Techniques.....	103
7.2.1	General Structure of a Trade-Off Process .....	103
7.2.2	Managing Trade-Offs in Practice .....	104
Chapter 8 – Conclusion.....		107
8.1	A first Journey in Systems Architecting .....	107
8.2	The other Key Systems Architecting Topics .....	107
8.3	How to Develop a Systems Architecting Leadership? .....	108
8.4	Towards a New Systems Architecture Modelling Language .....	109
Appendix A – System Temporal Logic .....		111
Appendix B – Classical Engineering Issues .....		115
B.1	Product problem 1 – The product system model does not capture reality .....	115
B.2	Product problem 2 – The product system has undesirable emergent properties.....	118
B.3	Project problem 1 – The project system has integration issues .....	120
B.4	Project problem 2 – The project system diverts the product mission .....	122
Appendix C – Some Systems Modeling Good Practices .....		125
Acknowledgements .....		127
References.....		128



# Preface

## CESAM Community and CESAMES Association

CESAM Community, which allowed the origin of the CESAM Method and its Pocket Guide, is managed by CESAMES Association.

CESAMES Association emerged in 2009 as a spin-off of the Ecole Polytechnique “Engineering of Complex Systems” chair with the aim of disseminating systems architecting in academia & industry. To do so, the Association organizes awareness events all year long for the Scientifics and Professionals to meet and share about complex systems. As an example, CESAMES Association organizes on a yearly basis the “Complex Systems Design & Management” (CSD&M) international conference series. Since 2010 in Paris and 2014 in Singapore, this event gathers each year more than 200 academic & professional participants, coming from all parts of the world. The Association also manages thematic evenings and working groups, always with the same goal: increasing awareness about systems architecting methods and tools.

Thanks to these events, CESAMES Association has federated a significant international community of system engineers and architects. They all share the same vision: architecture and system engineering represent a key factor of competitiveness for the companies.

In order to reinforce its visibility and get more influence at a worldwide level, CESAMES Association has created in 2017 the official “CESAM Community”. Its mission remains the same: organizing the sharing of good practices in Enterprise Architecture and System Architecture and attesting their capacity to be implemented through the CESAM certification worldwide.

More precisely, CESAM Community works to:

- **Make architecture a key tool for business competitiveness** by disseminating its use in companies and by communicating the results of its implementation through the visibility and actions of CESAM Community.
- **Propose and develop the best practices of systems architecture in the industry and the services:** through the creation of publications and guides and the sharing of lessons of experience between architects, systems engineers and urban planners during the events of the community.
- **Propose a generic architecture framework but also offer high-level systems architecture frameworks,** specific to some industrial applications (the first will be aeronautics and automotive). This is to facilitate the work of system architects within these activities.
- **Facilitate access to the CESAM method and develop its use in France and worldwide.**

## CESAMES Systems Architecting Method (CESAM)

CESAM Community and its members act as the *initial developers & contributors of the CESAMES Systems Architecting Method (CESAM)* which is presented more in details in this pocket guide.

CESAM is a systems architecting & modelling framework, developed since 2003 in close interaction with many industrial leading companies. It is dedicated to the working systems architects, engineers or modelers to help them to better master the *complex integrated systems* they are dealing with.

CESAM framework indeed has a number of unique features:

1. First of all, CESAM has sound mathematical fundamentals which are providing a *rigorous and unambiguous semantics to all introduced architectural concepts*. This first property is clearly key for ensuring an efficient and real understanding<sup>1</sup> between the stakeholders of a system design project (which is often key for ensuring the success of such projects).
2. These bases do ensure that CESAM is a *logically complete lean system modelling framework*: in other terms, the architectural views proposed by CESAM are just necessary and sufficient to model any integrated system. This second property guarantees both the completeness of a CESAM system model and that no useless modelling work will be done when using CESAM.
3. Finally CESAM is *practically robust and easy-to-use* both by systems architects and systems modelers. This was indeed pragmatically observed among the very large amount of various concrete systems within many different industrial areas (aeronautics, automotive, defense & security, energy, train, etc.) that were already modelled and architected using CESAM.

Note also that CESAM framework – due to the right level of abstraction to which it positions – can be implemented and used with both quite all existing system modelling frameworks and software tools of the market. However it is worth noticing that Dassault Systèmes decided to choose CESAM as the core framework of its system modeling tool, Catia Systems Engineering®<sup>2</sup>.

Last but not least, one shall finally notice that CESAM intends to propose both a generic architecting framework (which is introduced in this pocket guide), but also to progressively offer specific concrete high-level systems architectures for a number of industrial application domains (the first ones will be aircrafts and cars) in order to facilitate the work of the systems architects within these areas.

## How to Read this Pocket Guide?

The CESAM pocket guide is organized in order to be read in many different ways. Typical reading modes are presented below depending on the reader's objectives.

- **Getting an Overview of CESAM Framework:** you shall then focus on Chapter 2 where all the core CESAM systems architecture concepts are presented.
- **Being Aware of Systems Architecting Benefits:** you may only read Chapter 1 where the main motivations of systems architecting are described.

---

<sup>1</sup> This feature is in particular fundamental for managing convergence between the stakeholders of a system design project. It explains thus why collaboration is also at the core of CESAM framework (see sections 0.3 and 1.4 for more details).

<sup>2</sup> Web Address: <http://www.3ds.com/products-services/catia/products/v6/portfolio/d/digital-product-experience/s/catia-systems-engineering/>



- **Modelling Systems:** read first Chapter 2 to get an overview of CESAM systems architecture views and follow then from Chapter 3 up to Chapter 6 in order to learn one-by-one what are all the views requested to model completely an integrated system.
- **Practicing Systems architecting:** a systems architect shall know how to model a system, but also, much more deeply, what are the needs the system shall satisfy, which will lead him to regularly take design decisions. We do recommend systems architects to read first Chapter 1 to be aware of the main motivations of systems architecting. You may then pass to Chapter 2 up to Chapter 7 in order to learn both the main systemic views, but also how to use them in an architectural decision process (which is discussed in Chapter 7). Chapter 8 will also provide you some indications on how to progress in systems architecting.
- **Understanding Systems Architecting Fundamentals:** Chapter 0 is dedicated to the reader who wants to discover the sound logical basis on which relies CESAM framework.

## CESAM Pocket Guide Organization

The CESAM pocket guide is organized in eight chapters, plus some appendices specifically dedicated to more specialized material, as described below.

- *Chapter 0 – Introduction to CESAM* that may be skipped for a first approach.
- *Chapter 1 – Why Architecting Systems?* which presents the main motivations of the systems architecting approach and thus of CESAM framework.
- *Chapter 2 – CESAM Framework* that provides an overview of all CESAM concepts.
- *Chapters 3 to 6*, that do present in details, one after the other, each architectural vision of the CESAM systems modelling framework:
  - *Chapter 3 – Identifying Stakeholders: Environment Architecture,*
  - *Chapter 4 – Understanding Interactions with Stakeholders: Operational Architecture,*
  - *Chapter 5 – Defining What shall Do the System: Functional Architecture,*
  - *Chapter 6 – Deciding How shall be Formed the System: Constructional Architecture.*
- *Chapter 7 – Choosing the best Architecture: Trade-off*, that introduces to systems architecture prioritization, a key tool for the systems architect.
- *Chapter 8 – Conclusion* which gives some hints to reader on how continuing the systems architecting journey initiated by this pocket guide.



# Chapter 0 – Introduction to CESAM

## 0.1 CESAM: a Mathematically Sound System Modeling Framework

CESAMES Systems Architecting Method (CESAM) is the result of 12 years of research & development (cf. [1], [3], [4], [16], [17], [18], [22], [27], [48], [49], [51], [52], [53]) including permanent interactions and operational experimentations with industry. The CESAM framework was indeed used in practice within several leading international industries in many independent areas with a success that has never wavered (see [14], [23], [26], [31], [32], [33], [35] or [36]) for some application examples). This huge theoretical and experimental effort resulted in *a both mathematically sound & practical system modeling framework*, easy to use by working systems engineers and architects.

We shall only present in this general introduction the more important fundamentals<sup>3</sup> of the CESAM framework that may help to better understand its philosophy. At this level, the key point is the logical consistency of the CESAM framework, naturally provided by its mathematical basis. Due to this strong consistency, anybody who agrees with logical reasoning<sup>4</sup> (which shall normally be the case with all engineers) will indeed be able to use and work with CESAM framework.

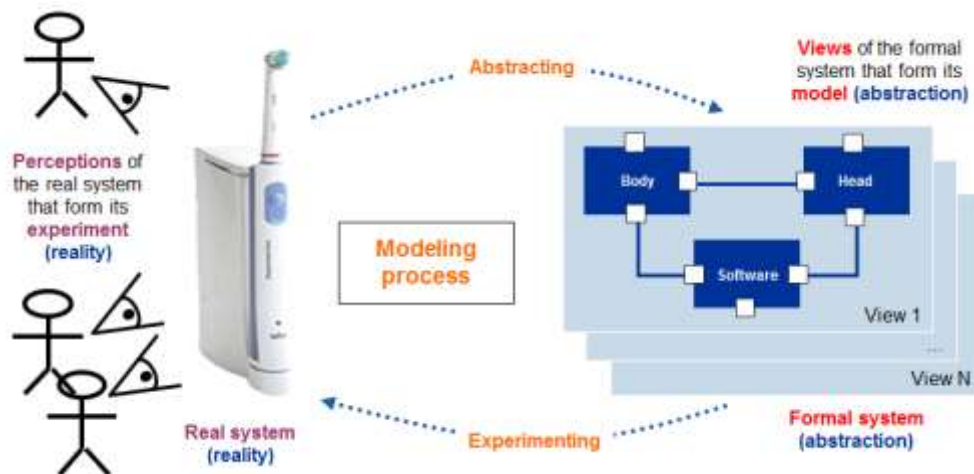


Figure 1 – The two abstracting/ experimenting sides of a system modeling process

Note now first that systems are considered in a modelling perspective within CESAM framework. This point of view immediately leads to distinguish the two core concepts of “formal system” and “real system”. It is indeed quite important to avoid making any confusion<sup>5</sup> between the result of a system

<sup>3</sup> CESAM framework is based on formal semantics, the mathematical theory – based on mathematical logics (cf. [12]) – which provides the fundamentals of computer science (see [89]) CESAM framework may be understood as an extension of usual formal semantics – which only addresses software systems – to more general types of systems.

<sup>4</sup> Logical reasoning is usually mixed with common sense. However one shall not forget that logics is a body of mathematical knowledge (see [12]), sadly largely unknown to engineers, that can be traced back to Aristoteles (see [84]).

<sup>5</sup> Unfortunately this confusion is quite common in systems engineering where such a distinction is usually not made both in practice and in most textbooks, handbooks and standards (such as [15], [42], [44], [47], [57], [61], [62], [71], [74] or [78]). Aslaksen appears in particular to be one of the rare systems engineering authors who proposed a formal definition for a system (see [9] or [10]). Sound systems modelling formalisms thus seem to be only found in the mathematical simulation literature (cf. [23], [73] or [100]) or in the few theoretical computer science system literature (cf. [21], [55] or [59]).

modelling process (the formal system) and the concrete object under modelling (the real system). Many design mistakes – with often high costs of non-quality – are indeed made by engineers when they forget that their system specification documentation is just an abstraction of reality, but not the “real” reality, and that their job is to ensure permanent alignment – through feedback loops with reality – between their system models and the real system, as it is and not as they think it is.

A *system modeling process* can thus be seen as a permanent back-and-forth between, on a first side, an *abstraction activity* that constructs a formal model of a real world object – using the tools of systemic analysis – and on a second side, an *experimentation activity* where the structure or behavior of the real object, as given or predicted by the model, are checked against those really observed in reality. The real object under modelling will thus be considered as a (real) system due to the fact that a system modeling process analyzes it as a (formal) system within a systemic modelling framework. By abuse of language, we often identify these two concepts of “system”, but it is important never to forget that “the map is not the territory” ([46]) in order to know at any time on what we reason! Note also that this approach points out that being a system is absolutely not an intrinsic property of an object<sup>6</sup>, but results from a modelling decision of a system designer. This being recalled, we can now provide the definition of a formal system<sup>7</sup> on which relies the CESAM modelling framework.

**Definition 0.1 – Formal system** – A *formal system*  $S$  is characterized on one hand by a input set  $X$ , an output set  $Y$  and an internal variables set  $Q$  and on the other hand by the following two kinds of behaviors that link these systemic variables among a given time scale  $T$ <sup>8</sup>:

- a *functional behavior* that produces an output  $y(t+) \in Y$  at each moment of time  $t \in T$ , depending on the current input  $x(t) \in X$  and internal state  $q(t) \in Q$  of the system ;
- an *internal behavior* that results in the evolution  $q(t+) \in Q$  of the internal state at each moment of time  $t \in T$ , under the action<sup>9</sup> of a system input  $x(t) \in X$ .

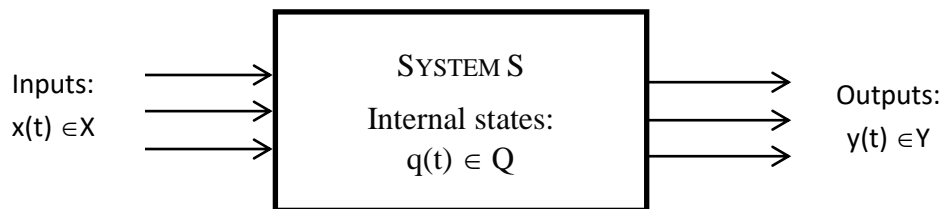


Figure 2 – Standard representation of a formal system

<sup>6</sup> This last consideration shows that it is merely impossible to give a sound definition of a real system since any object in the real world can be considered as a system as soon as a system designer decides it.

<sup>7</sup> The CESAM definition of a system provided by Definition 0.1 unifies two classical system modelling traditions: it indeed mixes the usual functional definition coming from the mathematical system simulation literature (cf. [23], [73] or [100]) and the state-machine system definition that emerged in theoretical computer science (cf. [21], [55] or [59]).

<sup>8</sup> A time scale  $T$  is a totally ordered set with a unique minimal element – usually denoted  $t_0$  – and where each element  $t \in T$  has a (unique) greatest upper bound within the time scale, called its successor and denoted  $t+$  within  $T$ . Up to rescaling, two types of time scales are key in practice: discrete time scales where  $t+ = t + 1$  and continuous time scales where  $t+ = t + dt$  where  $dt$  stands for an infinitesimal quantity. Discrete time scales model event-oriented systems (such as software systems which are regulated by a discrete clock) when continuous time scales are used to model physical continuous phenomena.

<sup>9</sup> When this action is not permanent, one may identify the involved input to a discrete *event* which occurs only at a certain moment of time  $t \in T$ , considered as instantaneous within  $T$ . Note however that an event is always relative to a given time scale: it may indeed not be instantaneous when analyzed from another, more refined, time scale.

This definition is deliberately very weak following Occam’s razor strategy<sup>10</sup> that prevails throughout CESAM framework. It simply means that a system is just defined by an input/output and an internal behavior<sup>11</sup>. This is of course not abnormal since we do want to capture commonalities between ALL real systems. These common points are therefore mechanically quite limited due to the tremendous diversity of the real objects to take into account. However we now have a unified system modelling framework, provided by the below Definition 0.2, in which we can uniformly reason on any type of systems and hence think system integration, which was one of our first goals (see section 0.2).

**Definition 0.2 – Real system** – An object of the real world will be called a *real system* as soon as its structure and its behavior can be described by a formal system (in the meaning of Definition 0.1) that will be then called a *model* of the considered real system.

According to this definition, quite all human artefacts – independently of their physical, informational or organizational nature – can thus be considered and analyzed as systems. Only changes the nature of the laws – given by physics, logics or sociology – that allow defining the functional and internal behaviors which are making them systems. The two previous definitions also help to understand that it is neither the nature, nor the size, nor the hierarchical position<sup>12</sup> that makes something a system, since quite all real objects can be seen as systems. As already pointed out, considering that an object is a system remains first of all a modelling (human) choice: it just means that this object will be abstracted through a systemic point of view, i.e. in the framework given by Definition 0.1.

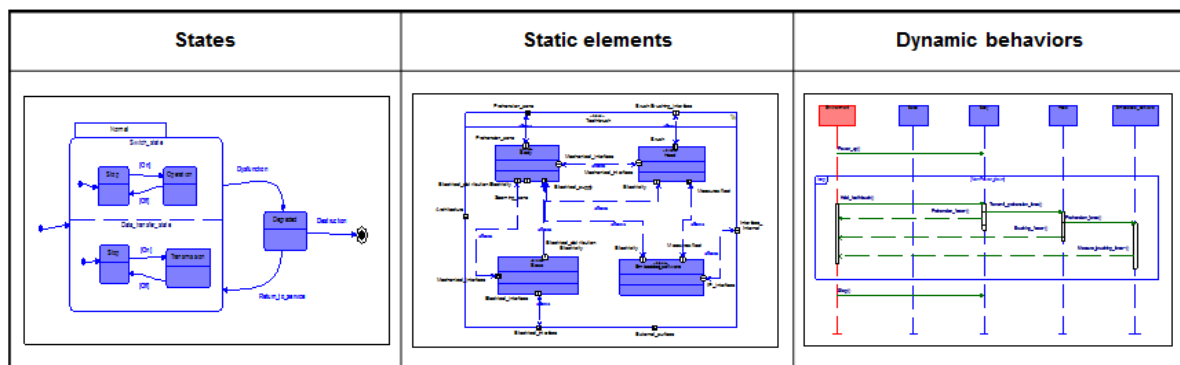


Figure 3 – Structure of a standard complete system specification

Note finally that the definitions we proposed above imply that a system  $S$  is completely specified (for a given time scale) if and only one is able to provide (see Figure 3):

1. *the states of  $S$* , i.e. a description of the evolution law of its internal variables, which can be typically achieved through a state-machine (see for instance [20] or [21]),

<sup>10</sup> Occam’s razor (see [84]) is a key scientific parsimony principle which expresses that there is no need to introduce a new concept when it can be explained by already existing concepts. The CESAM system modeling framework is constructed in that way: we always only introduced the minimal number of new architectural concepts that are necessary for system modelling purposes. This typically explains why we proposed such a simple system definition: it is indeed minimal for understanding the structure of CESAM framework (see end of section 0.1).

<sup>11</sup> The internal behaviour of a system is modelled by an evolution law of its internal variables, called “states”.

<sup>12</sup> A common mistake is for instance to consider that a “sub-system” of a system (see next section 0.2 for more details) is not a system, which is of course absolutely not the case according to our definitions (this remark allows in particular to apply recursively the principles of system modelling within a given system hierarchy).

2. a description of its functional dynamics, which can be naturally obtained through defining:
  - *the static elements* of  $S$ , in other terms the signatures of the functions required to express its functional behavior, which can be given by a static block-diagram (see [34] or [73]),
  - *the dynamic behavior* of  $S$ , that is to say the temporal dynamics<sup>13</sup> of these functions, which can be for instance defined by a sequence diagram (see [20] or [34]).

These last considerations are at the core of the CESAM systems architecting matrix (see section 2.5) and do explain the generic nature of the key columns (states, static elements, dynamic behaviors as described in Figure 3) involved in that modelling matrix.

Note that it may be useful to consolidate in a specific view all the exchanged input/output flows that do appear when describing the functional dynamics of a system. Such a view – which strictly speaking is already contained in the two other functional views introduced above – is indeed important for constructing the glossary of all flows exchanged within a system. This will lead us to the last column (objects) of the CESAM systems architecting matrix (see section 2.5).

Up to now, we however only dealt with an *extensional formalism* for systems. When using Definition 0.1, we are indeed obliged by construction to explicitly describe extensionally the behavioral and structural dimensions of a system (definition in extension). But there exists another different, but totally equivalent from a mathematical perspective, system specification formalism which is of *intentional nature*. This means that we do not need here to describe the behavior or structure of a system as before, but rather to express its expected/intended properties (definition in intension).

That new formalism is based on a formal logic, called *system temporal logic*, which is presented in full details in Appendix A. The only point to stress here is that we will now need to work with systems whose input, output and internal states sets  $X$ ,  $Y$ ,  $Q$  and timescale  $T$  are fixed. The associated system temporal logic allows then to syntactically define well-formed logical formulae<sup>14</sup>, called *temporal formulae*, which specify the sequences  $O$  of inputs, outputs and internal states values of such a system that can be observed among all moments of time  $t$  within the timescale  $T$ , as stated below:

$$O = (O(t)) \text{ for all } t \in T, \text{ where we set } O(t) = (x(t), y(t), q(t))^{15}.$$

We are now in position to introduce the notion of formal system requirement which refers to logical predicates within system temporal logic (see Appendix A for several examples).

**Definition 0.3 – Formal requirement** – Let  $S$  be a formal system with  $X$ ,  $Y$  and  $Q$  as input, output and internal states sets and  $T$  as timescale. A *formal requirement* on  $S$  is a temporal formula expressed in system temporal logic, based on  $X$ ,  $Y$ ,  $Q$ , and  $T$  as described in Appendix A.

Most of engineers would however be afraid working with system temporal logic. One is hence quite rarely<sup>16</sup> using formal requirements to specify real systems. The only utility of the previous definition will then just be to remember that one works with logical formalisms when one manages systems

---

<sup>13</sup> We here mean the underlying “algorithm” that expresses how the functional behaviour of  $S$  is obtained as a result of the interactions of a certain number of functions.

<sup>14</sup> Also called predicates in formal logic.

<sup>15</sup> We use here the formalism of Definition 0.1.

<sup>16</sup> Temporal logic is only classically used for the verification of real-time critical software systems.

requirements, which is usually forgotten. This explains why we now propose the following informal definition that intends to reflect what should be a good requirements engineering practice<sup>17</sup>.

**Definition 0.4 – Requirement** – A *requirement* on a real system S is then any sentence that intends to express a formal requirement on a formal system that models S.

Note that each formal requirement R specifies a set of systems, which is the set of all formal systems that satisfy to R. A set SR of formal requirements specifies then also a set of formal systems, that is to say the set of formal systems that satisfy to all elementary requirements of SR. This simple property does establish a connection between sets of (formal) requirements and sets of (formal) systems. One can then prove that this connection is bijective, which means in other terms that it is mathematically strictly equivalent to specify sets of systems using either Definition 0.1 or sets of requirements. We thus have introduced two different equivalent ways of specifying systems.

A wrong conclusion would now be to use either one, or the other formalism, for defining systems. Our two formalisms are indeed absolutely not equivalent in terms of engineering effort: some properties are indeed much easier to state using the formalism of Definition 0.1 when some others are much simpler to state using requirements<sup>18</sup>. The working systems engineer will thus always have to mix extensional and intentional formalisms, according to their relative “human” costs. A good system specification is therefore a specification that shall mix on one hand behavioral or structural descriptions in the line of Definition 0.1 and Definition 0.2 and on the other hand requirements in the line of Definition 0.3 and Definition 0.4.

Note finally that requirements will also provide us the first column (expected features) of the CESAM systems architecting matrix (see section 2.5)<sup>19</sup>.

## 0.2 CESAM: a Framework focused on Complex Integrated Systems

A second key point to stress is that CESAMES Systems Architecting Method (CESAM) is fundamentally a *complex integrated systems-oriented framework*. This means that CESAM is especially dedicated and adapted to the architecting and modelling of complex systems, that is to say – in equivalent terms – of non-homogeneous systems which result from an integration process.

To be more specific, let us first recall that *system integration* is the fundamental mechanism<sup>20</sup> that allows in practice to build a new system from other smaller systems (typically of hardware, software

---

<sup>17</sup> A good system requirement shall indeed always be unambiguous which will never be achieved when trying to express a formal requirement in natural language.

<sup>18</sup> The fact that a system behaves according to a given state machine is for instance easier to state by explicitly describing the concerned state machine, which gives rise to a single systemic view. Using requirements would contrarily require to define one logical formula per transition – stating typically that when the system is in state q and event e occurs, it shall pass in state q’ – which would give rise to N requirements where N is the number of involved transitions, which is clearly much more complicated than the first formalism. Conversely a safety property can usually be stated using just one unique requirement, but would give rise to many behavioral descriptions if one would like to express it in that other way.

<sup>19</sup> As a matter of fact, the systems architecture paradigm can also be expressed using the formalism of requirements. Any systems architecture problem can indeed be stated as the research of a system S that satisfy a set of requirements associated with the environment of S. As an immediate consequence of this genericity, the generic structure of a system provided by CESAM framework naturally reflects in the generic structure of the systems architecture process, which is the core property on which the CESAM systems architecting method relies.

or “humanware” nature), by organizing them so that the integrated system can accomplish – within a given environment – its missions (see for instance Figure 5). For the sake of completeness, a more formal definition of this key mechanism is provided below.

**Definition 0.5 – Integration** – Let  $S_1, \dots, S_N$  be a set of  $N$  (formal) systems. One says then that a (formal) system  $S$  is the result of the *integration* of these systems if there exists on one side a (formal) system  $C$  obtained by composition of  $S_1, \dots, S_N$  and on the other side dual abstraction and concretization operators<sup>21</sup> that allow to express:

- the system  $S$  as an abstraction of the system  $C$ ,
- the system  $C$  as a concretization of the system  $S$ .

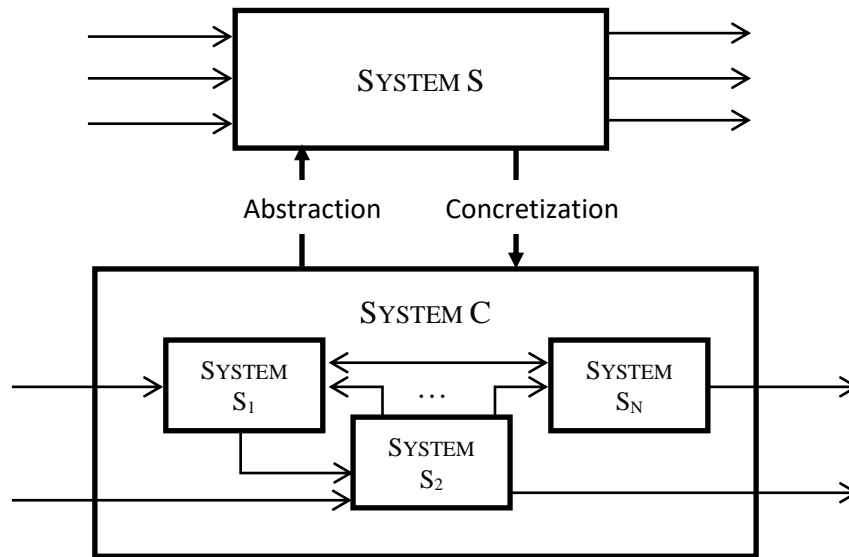


Figure 4 – Formal integration of formal systems

This somewhat technical definition has for fundamental purpose to position integration as a different mechanism from a simple composition of models in order to try to capture structurally emergence in our approach. The previous Definition 0.5 indeed clearly shows that the description of an integrated high level system, obtained through interconnection with other systems, requires having:

- models of each system contributing to the integrated high level system,
- a new high level model, specific to that integrated system.

In other words, *the mere knowledge of the models of the components of a system is never sufficient to model this system!* This modelling postulate can be seen as the direct translation of the universal phenomenon of *emergence* which is always a mechanical consequence of integration: an integrated

<sup>20</sup> Integration is thus an operator between systems that maps a series of systems into another new system. It is interesting to point out that most of the existing system “definitions” which can be found in the systems engineering literature (see for instance [30], [42], [44], [57], [61], [62], [71] or [74]) are defining systems as the result of an application of the integration operator. From a mathematical perspective, this gives unfortunately rise to inconsistent definitions since the set on which acts this operator is never defined. However all these “definitions” are clearly expressing a real pragmatic dimension of all systems, which is rigorously captured in our framework by the notion of integration as provided in Definition 0.

<sup>21</sup> In order to avoid mathematical technicity, we will not define here the notions of abstraction and concretization that shall be considered in the meaning of the theory of abstract interpretation (see for instance [25] for more details on this topic).



system will indeed always have emerging properties, that is to say specific properties that cannot be neither found in its components, nor deduced from the properties of its components.

This emergence postulate can be observed on all simple integrated systems of day-to-day life. Let us consider for instance a wall formed solely of bricks (without mortar to bind them) for the sake of simplicity. A simple systemic model of a brick can then be characterized by:

- a functional behavior consisting on one hand in providing reaction forces when mechanical action forces are acting on the brick and on the other hand in absorbing the light rays,
- an internal behavior provided by three invariant states “length, width, height”.

By composing such brick models, a wall will just be a network of bricks connected by mechanical action/reaction forces. But it will be difficult to get anything else from such a wall than a resulting functional behavior consisting in absorbing the rays of light. This is however clearly not the usual behavior of a wall, since we want to make account the existing holes in the walls (for the windows) that should just let the light pass! Note also that it is difficult – and probably vain – to try to express the form of a wall (which is one of its typical internal state) depending on the lengths, widths and heights of the bricks that compose it. All these facts call for the obvious need of creating a dedicated systemic model, more abstract, to specifically model a wall.

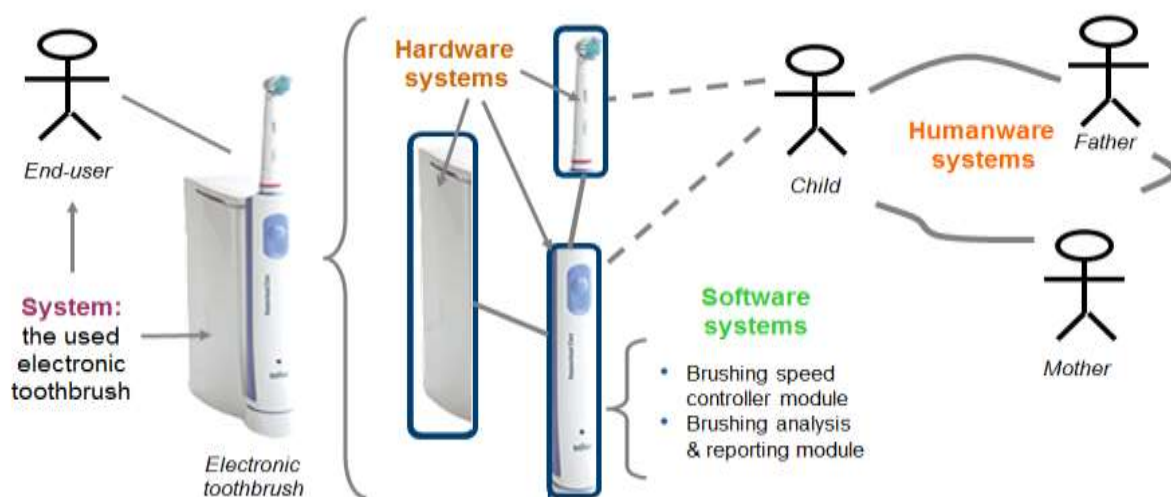


Figure 5 – Example of an integrated system: the used electronic toothbrush

These considerations may appear to be common sense, if not naive, but we unfortunately noticed that their consequences are usually not understood, which directly causes lots of non-quality issues as observed on many modern complex engineered systems. Most engineers & managers indeed still continue thinking that mastering the components of an integrated system is sufficient to master the system as a whole. However the very nature of the integration mechanism imposes not to have only components responsible when one has to design an integrated system: it is key to also have somebody specifically in charge of the integrative high-level model of the considered system, that is to say a systems architect, who does unfortunately often not exist<sup>22</sup> in most industrial organizations, as strange it may sounds when one is aware of the underlying systemic fundamentals!

<sup>22</sup> As a consequence, high level models of integrated systems are also often non-existent in practice ...

Note also that emergence obliges to privilege a top-down design strategy – rather than a bottom-up approach<sup>23</sup> – when one deals with integrated systems. As a matter of fact, it is indeed not possible to predict the emergent properties that result from integration: any bottom-up design strategy will thus statistically create numerous undesirable emergent properties that will be difficult to master. In such a context, the only possible efficient design strategy is to start by imposing the expected properties of the integrated system, and then deriving from them the properties required from its components, which naturally lead to a top-down design strategy.

Finally, note that the integration mechanism naturally hierarchizes systems, while giving a recursive dimension to systemic analysis. Engineered systems are indeed obtained in practice by a number of successive systems integrations: each integrated system thus generates in this way an integration hierarchy, which is also an abstraction hierarchy due to the nature of integration, called the *systemic hierarchy of the initial system*. This allows in particular to speak about the system, sub-systems, sub-systems, etc. levels (that are also called sometimes “layers” or “tiers”) of any integrated system.

### 0.3 CESAM: a Collaboration-oriented Architecting Framework

CESAM is however not only a mathematically sound system modeling framework, specifically focused on complex integrated systems. It is also an *architecting framework* which means that it is intended to efficiently support all the design decisions that a systems architect needs to regularly take during a complex system prototyping or development project. We must recall in this matter that modeling is not an end in itself<sup>24</sup>, but just a tool for architecting which is the key design process addressed by the CESAM framework. Architecting here means finding a solution that fulfills a series of external needs and constraints. It can be seen as an optimization process which has to construct and select the “best” system among a series of possibilities. Choice is thus intrinsic to that activity. Being able to make the “good” choices in a rational way is thus always key in any system design project. This is exactly the purpose of the CESAM framework which provides to the working systems architect a number of systemic views as a support to *collaborative architectural decisions*.

It is indeed important to remember that a technical system does never exist alone and that it cannot be designed independently of the people who are engineering it. A “good” systems architecture is in particular always an architecture that all stakeholders do share. The first job of any systems architect (by applying the core principles of systems architecting to the engineering system to whom he/she belongs) shall thus always be to understand and to identify the organizational architecture in which a given system development takes place: the technical architecture of the system under design is indeed highly correlated to that organizational architecture (we refer to section 1.1 for more details on this important topic). This explains why a systems architect must manage the two following types of activities which are of very different nature:

- On one hand, *technical activities* fundamentally centered on the definition of high level global system models, making explicit interfaces between all components of a system,
- On the second hand, *facilitation activities* centered on the construction of convergence on these models, creating a common vision between all stakeholders of the system.

---

<sup>23</sup> This is unfortunately still the most common design strategy in the industry for integrated systems ...

<sup>24</sup> Which most systems modelers do unfortunately forget!

Figure 6 illustrates these two kinds of systems architecting activities which all rely on system models. The initial version (initial version 0) of the system architecture presented in that figure is the result of a technical activity, when the second version (shared version 1) is the result of a facilitation activity. It is thus quite interesting to see the differences between these two system views: this illustrates the value brought by facilitation which is crucial for collaboratively constructing robust systems.

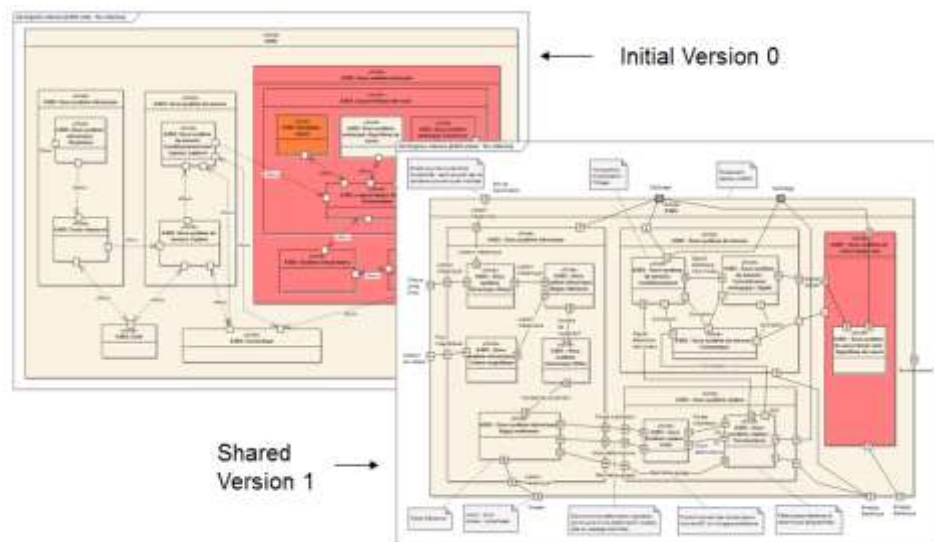


Figure 6 – Using models to converge on the same vision of a system

System models are thus key for ensuring a collaboration of quality which is mandatory in the context of complex integrated systems development (as we will see more in details in section 1.5). In these matters, the basic tool for managing collaboration and creating architectural convergence is *the collaborative systems architecture workshop*. We will now quickly sketch its mode of operation in order to better understand the “human” dimension of systems architecting.

The principle of such a workshop is quite simple since it consists “merely” in putting in the same place all stakeholders<sup>25</sup> that must convergence on a given architectural solution and submitting then them a first version of the intended system architecture. This is indeed key to obtain stable systems architectures. The groundwork for a collaborative systems architecture workshop is then to discuss and collaboratively modify the proposed system architecture, so that the final architecture, as resulting from such a process<sup>26</sup>, becomes a collective asset.

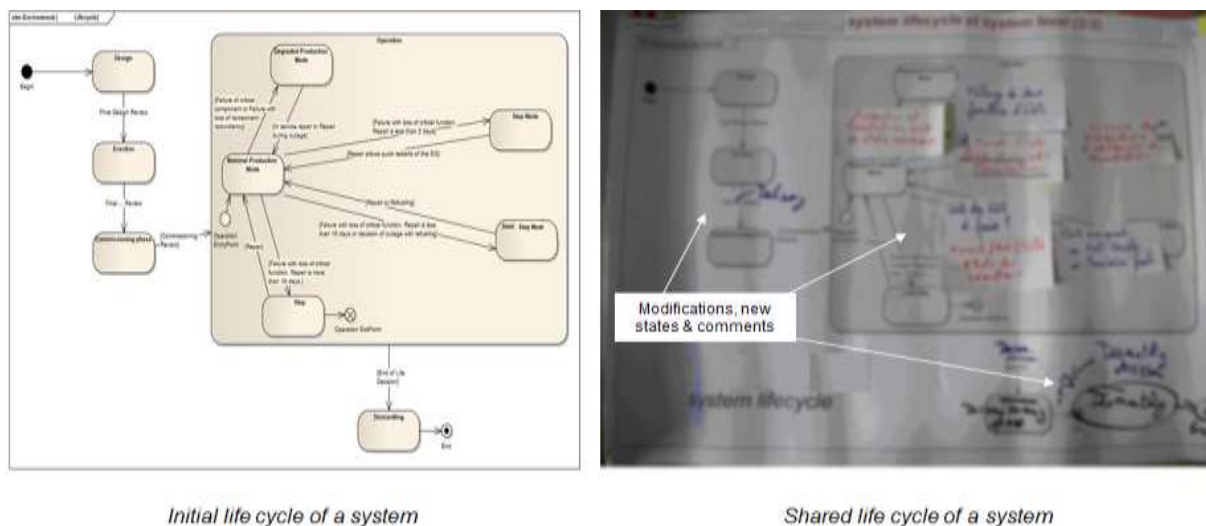
<sup>25</sup> This means that these stakeholders were comprehensively and correctly identified, which in itself is a difficult exercise with multiple traps. Moreover, it can also easily be understood that it is difficult to carry out convergence with 150 people! The effective implementation of a collaborative workshop also presupposes that we previously achieved an organizational architecture analysis, which sufficiently abstracted the “field” of a given system architecture by only identifying a limited number (typically no more than 15) of top level players, truly representative of the entire organizational scope of the target system, with whom we shall manage the required convergence work.

<sup>26</sup> To carry out such a work in practice, a simple way to proceed is typically to make visible the system model to share by printing the various architectural views on large A0 posters. One can then easily collectively discuss and change an initial system architecture by directly annotating these architectural diagrams (cf. Figure 7 for an illustration of this method). Note that the systems architect must manage that “live” modification process in order to permanently guarantee that the proposed changes have the consent of all participants and do not lead to sub-optimality at system level.

This method naturally leads to shared visions which are usually deeply engaging all involved actors. Note that the systems architect has a key role in this process since he/she must always ensure that the system architecture, on which all stakeholders converge, remains a satisfactory response with respect to all needs it shall take into account.

However this *modus operandi* assumes a key prerequisite, that is to say that all stakeholders of the workshop have a common systemic representation language. In other terms, the meaning of all system descriptions shall be the same for all participants, which is usually not the case. It is therefore highly recommended to start a collaborative systems architecture workshop by sharing with the participants the semantics of each of systemic representation that will be used<sup>27</sup>.

Once established these core bases, the work of convergence towards a shared architecture can be attacked, starting with a collective analysis of the proposed initial architecture. We then often see in practice that the first problems which occur are again syntactical problems, due to the fact that the vocabulary that is used to describe the elements of an architecture is not necessarily shared among its stakeholders! To solve this other recurrent issue, it can for instance be helpful to also share with all participants a glossary of key technical terms, in order to be sure that these terms have exactly the same meaning for everybody.



**Figure 7 – Initial and final models as managed during a collaborative systems architecture workshop**

Once these barriers are removed, we can consider that we have a solid base to attack the technical activities, strictly speaking. The work consists then in discussing the proposed system architecture and modifying it collectively, if necessary (see Figure 7), so that the final architecture resulting from the exchanges is shared by all at the workshop's end. Note that it is also mandatory to have an arbitration mechanism which will handle all the disagreement points, if any, for decision-making (in which case, a new loop with stakeholders may be needed to explain them the definitive choices).

As one can see here, the systems architecting technical process shall always be deeply integrated into a “human” engineering process without which no shared and no “humanely” robust systems

<sup>27</sup> This is again one of the important use and purpose of CESAM framework.

architectures would emerge (which by the way is often a sine qua non condition for them to also be technically robust).

## 0.4 CESAM: a Business-oriented Framework

Let us finally also stress on the fact that CESAM is also a business-oriented framework. As already pointed out, the CESAM framework was indeed used in many applications. In order to contribute to the quality increasing of systems projects, which is a key issue for our modern societies, we thus do want to share with the system community a part of that quite important practical experience.

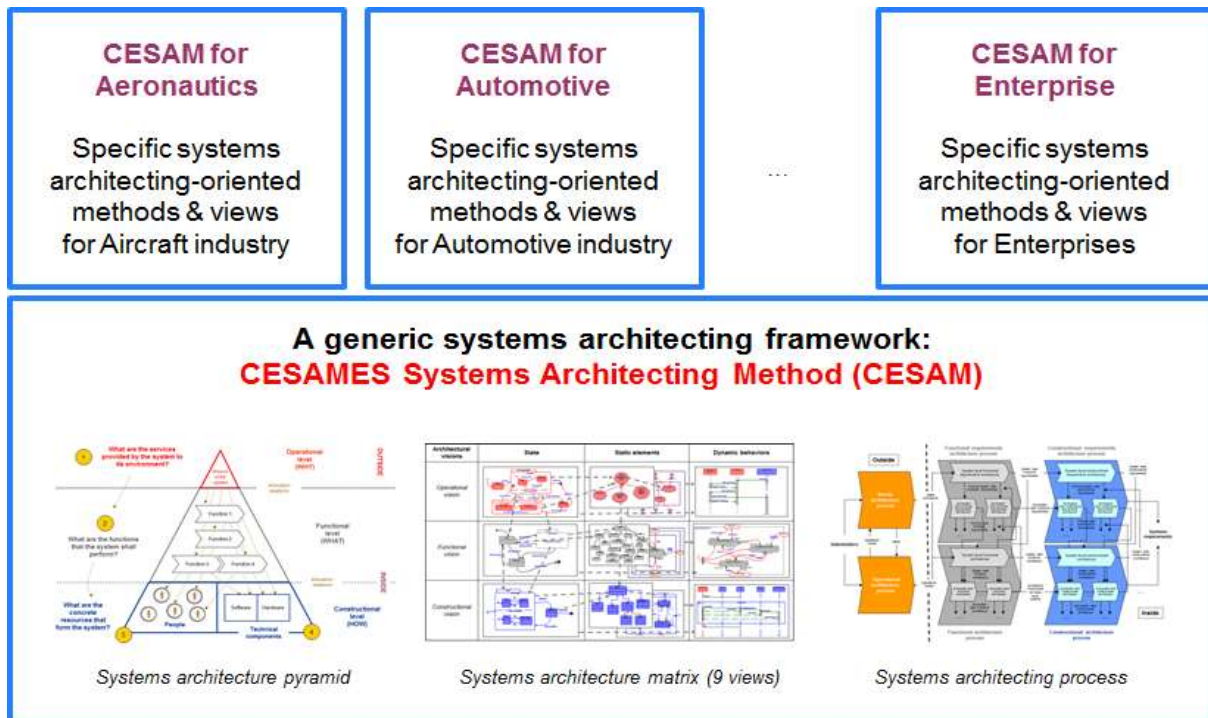


Figure 8 – Tentative structure of the CESAM frameworks

The CESAM framework is therefore intended to be organized in two layers:

- **a generic layer**, consisting of a *generic systems architecting framework* whose first part is presented in the current pocket guide,
- **a specific layer**, consisting of a *number of specific systems architecting methods & systems architecture views* per main application area (aeronautics, automotive, enterprise, etc.).

We do plan to progressively publish in the following decade these different parts of the CESAMES systems architecting body of knowledge.

This strategy plans to offer to all systems architects & engineers both a sound, complete and robust method for architecting complex integrated systems, but also to provide them a series of domain-specific systems architecting methods & systems architecture views. These elements, which will be already expressed and/or formalized within the CESAM framework, will be starting points to ease their systems architecting activities (see Figure 9 for an illustration of what could be such a starting



point in the aircraft industry<sup>28</sup>). It is indeed much easier to deform an existing system architecture in order to adapt it to a series of new needs, rather than constructing it from scratch.

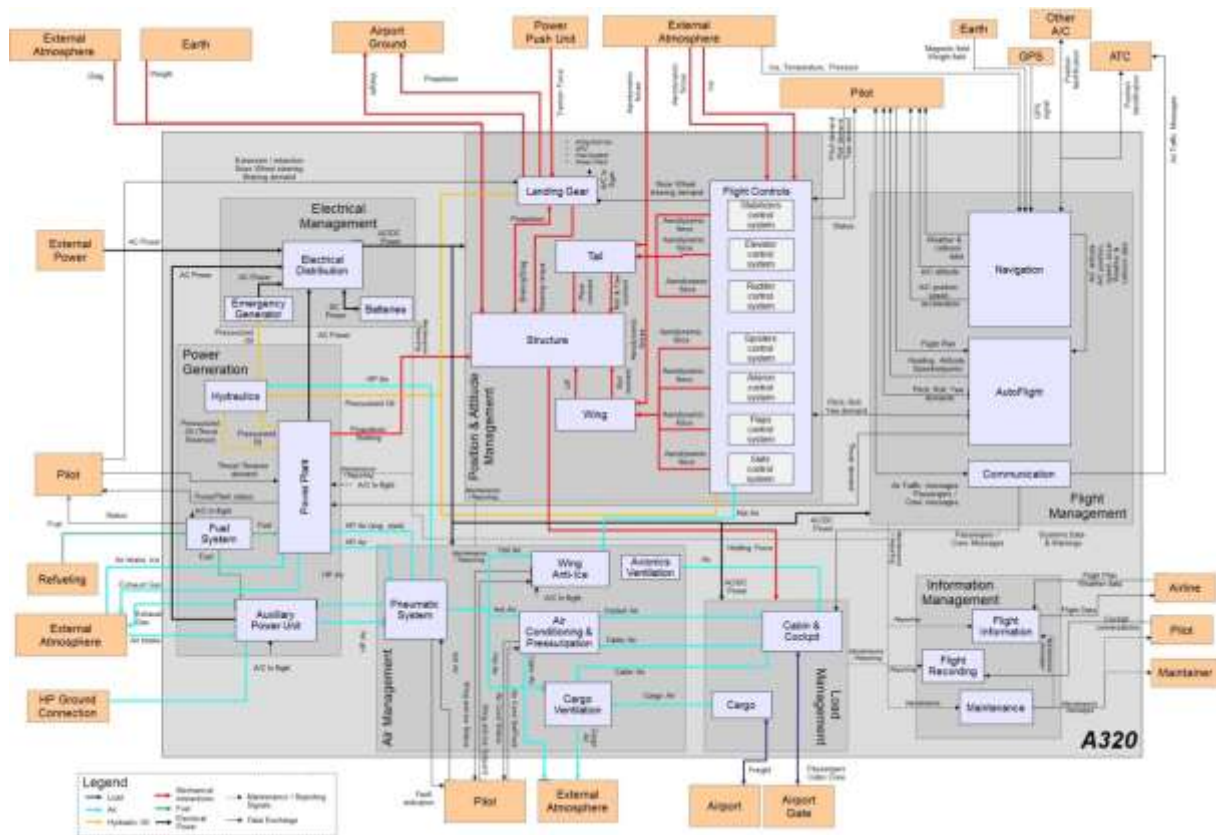


Figure 9 – Example of a standard functional / constructional architecture for an aircraft

<sup>28</sup> The architecture presented in Figure 9 is a constructional high level architecture (blue boxes are modeling components) of an aircraft – an Airbus A320 here – aligned with the underlying functional architecture (grey zones are modeling functional domains). It was constructed on the basis of operational pilot documentation, freely available on the Internet.

# Chapter 1 – Why Architecting Systems?

## 1.1 Product and Project Systems

Before going further, we need first to introduce a distinction that will be fundamental to understand better both what is systems architecting and how to read many classical engineering issues. It indeed appears that all engineered systems are always involving two kinds of systems (see Figure 10):

- the first one is clearly the *product system*, i.e. the integrated hardware and software<sup>29</sup> object which is under engineering in order to be finally constructed and put in service,
- the second one is the *project system*, that is to say the engineering organization (or in other terms the engineers) who is designing and developing the product system.

These two types of systems are of quite different nature: the product system is usually a technical-dominant system when the project system is clearly a human-dominant system. However as shown in Figure 10, they are highly and permanently coupled during all the design & development phases of the product system: the project system typically monitors the implementation status of the product system through adapted implementation actions that are changing this implementation status.

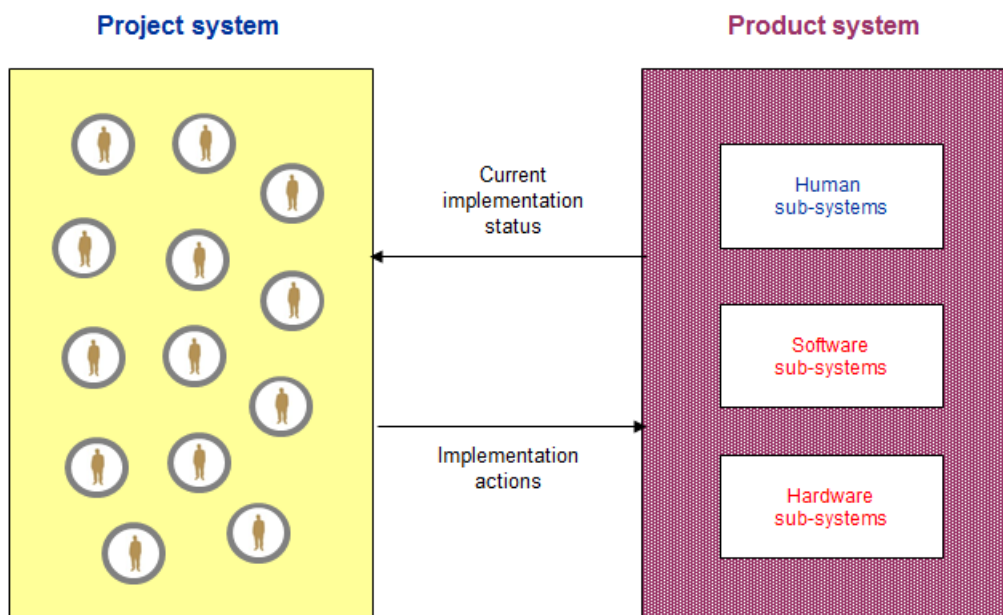


Figure 10 – Product versus project systems

The product / project distinction does seem very simple. However it appears in practice that most of engineers are thinking in terms of project activities realization and not of product characteristics achievement. Many engineering issues are thus coming from the fact that system development

<sup>29</sup> One may also possibly include a “humanware” dimension into a product if one must put a person, a group of persons or even an organization within the scope of the system under engineering.

project are often too project system-oriented and not enough product system-focused. One must indeed understand that there are two totally different ways of managing a system development:

- *Management mode 1 – project system management*: this first management mode – also the most common – groups all classical project management activities where a system development project is followed by means of a project agenda and a task achievement monitoring.
- *Management mode 2 – product system management*: this other management mode intends to monitor the progressive achievement of the desired product system, which is followed by means of systems architectural views.

These two management modes shall of course not be opposed, since they are fully complimentary. A key good practice, on which systems architecting relies, claims in particular that these two modes of management – respectively based on a project agenda and on systems architectural views – are both mandatory in the context of *complex systems development* (see sections 1.2 and 1.3 below).

The project system management mode is indeed not sufficient for ensuring the good achievement of the product system quality & performance when such a product became complex. One unfortunately observes in practice (see next section) too many situations where complexity is so high that project teams are not able anymore to master their product. These teams are thus discovering too late that they will never deliver the expected product within its cost, delay, quality and/or performance limits. The key motivation of systems architecting is just to provide to these engineering teams new product system-oriented tools to better master their complex integrated system development<sup>30</sup>.

- **Product system problems**
  - **Product problem 1 – The product system model does not capture reality**
    - *Typical issue*: the system design is based on a model which does not match with reality
  - **Product problem 2 – The product system has undesirable emergent properties**
    - *Typical issue*: a complex integrated system has unexpected and/or undesired emerging properties, coming from a local problem that has global consequences
- **Project system problems**
  - **Project problem 1 – The project system has integration issues**
    - *Typical issue*: the engineering of the system is not done in a collaborative way
  - **Project problem 2 – The project system diverts the product mission**
    - *Typical issue*: the project forgets the mission of the product

**Table 1 – Typical examples of product and project issues**

Note also finally that most of engineering issues occurring with complex systems can be classified into two categories, according to the product/project distinction, that is to say on one hand, *product problems*, referring to purely architectural flaws leading to a bad design of the product, and on the

---

<sup>30</sup> As a good practice, each systems architect shall always try to quickly understand whether a given system development project is only project-oriented and not project/product-balanced. This can be done by analyzing the words used within the project meetings. If the project team speaks only of agenda, milestones, activities, contractual relationships, deliverables, etc., without any reference to the underlying product, one can easily deduce a project-orientation for the project. This is statistically an important risk for the project since it does not fully master the product it is developing.



other hand, *project problems*, referring to organizational issues leading to a bad functioning of the project. Table 13 above provides an overview of typical such problems. All details on examples and analyzes illustrating these different complex systems issues can be found in Appendix B.

## 1.2 The Complexity Threshold

At this stage, it is now time to explain more precisely what “systems complexity” is and how it is connected with systems architecting. An element of answer is provided by the Constructive Systems Engineering Cost Model (COSYSMO; see for instance [79] or [84]). This *cost model* – which extends to general industrial systems a classical model well known for information systems since the 1980’s<sup>31</sup> – is based on an integration complexity measure of a product system<sup>32</sup> that recursively measures<sup>33</sup> the number of external & internal product interfaces within a given industrial system.

The COSYSMO cost model connects then the effort of engineering, expressed in men-months, required developing a given product system (denoted Effort in the below equation) to this intrinsic measure of complexity (denoted Complexity in the below equation), by the following relationship<sup>34</sup>:

$$\text{Effort} = A \times \text{Complexity}^{1+B},$$

### Equation 1 – Relationship between engineering effort and systems complexity according to COSYSMO

where, on one hand, A is a constant, depending on the size and performance<sup>35</sup> of the project system, and where, on the other hand, B is a statistic scale factor only related to the product system, than can move from 0.05 for simple systems up to 0.5 for large systems.

The key issue to point out here is the *non-linear nature* of Equation 1 which induces a largely non-intuitive relation between integration complexity and engineering effort, and thus system delivery delay which is directly connected to such an effort<sup>36</sup>. It is indeed important to see that integration complexity is proportional to the square  $N^2$  in the number N of components of a given system<sup>37</sup>. As a direct consequence, the engineering effort is then proportional to  $N^3$ , where N stands again for the number of component of the underlying system, when a large size system is involved, due to the 1.5 exponent in Equation 1 in this situation. In more familiar terms, this means that the engineering effort

<sup>31</sup> That is to say the Construction Cost Model (COCOMO) developed by Barry Boehm in 1981 (see [1], [19] or [83]). This other model expresses the effort for constructing an information system (in men-months) in terms of function points, which was aimed to be an intrinsic complexity measure of an information system, independent of the programming language.

<sup>32</sup> In this matter, note also that the notion of “complex system” cannot be formally defined. One can however define the notion of “complexity” for a system, as introduced in this section. This complexity definition will then allow discussing whether a given system is of low, medium or high complexity – but not whether it is complex or not, which is a typical false debate for us, even if many people like to discuss on the difference between complicated and complex, which is irrelevant from a purely scientific perspective which can only deal with measured complexity – or eliciting the connection between engineering effort and complexity as provided by the COSYSMO model that we presented here.

<sup>33</sup> The exact complexity measure provided by COSYSMO is more complicated than that. However the approximation that we are making here, for the sake of simplicity, is totally valid.

<sup>34</sup> Such a model is a statistical model, constructed on the basis of the analysis of several real engineering projects.

<sup>35</sup> A is 1.0 by default, but multiplicatively grows depending on product or project parameters such as number of critical algorithms, requirements and operational scenarios to manage, requirements and architecture understanding, expected level of service, migration and technological risks, project team experience, stakeholders cohesion, etc.

<sup>36</sup> Typical relation is  $\text{Delay} = 2.5 \times \text{Effort}^{1/3}$ , where Delay and Effort and respectively expressed in months and men-months. We refer to the appendix of [70] for a rational explanation of that apparently empirical law discovered by Boehm [19].

<sup>37</sup> When a system has N components, the number of its internal interfaces is indeed  $N^2/2$  on the average.

is multiplied by 8 (resp. 1,000) if the number of components of a system is multiplied by 2 (resp. 10), with a project team who would be able to easily absorb the increasing of complexity<sup>38</sup>.

As a result of that complexity laws, the engineering effort in a system development project will be quickly too important, when complexity grows, for being anymore handled by a single person. In other terms, there will be always a *cognitive rupture moment* where complexity is too high to be any longer efficiently individually mastered by a systems architect or engineer. This rupture point is of course difficult to formally define and it depends on the system maturity of a given industry<sup>39</sup>, but as far as we could regularly see in practice, it can always pragmatically be observed when complex systems designers began to express strong cognitive difficulties<sup>40</sup>.

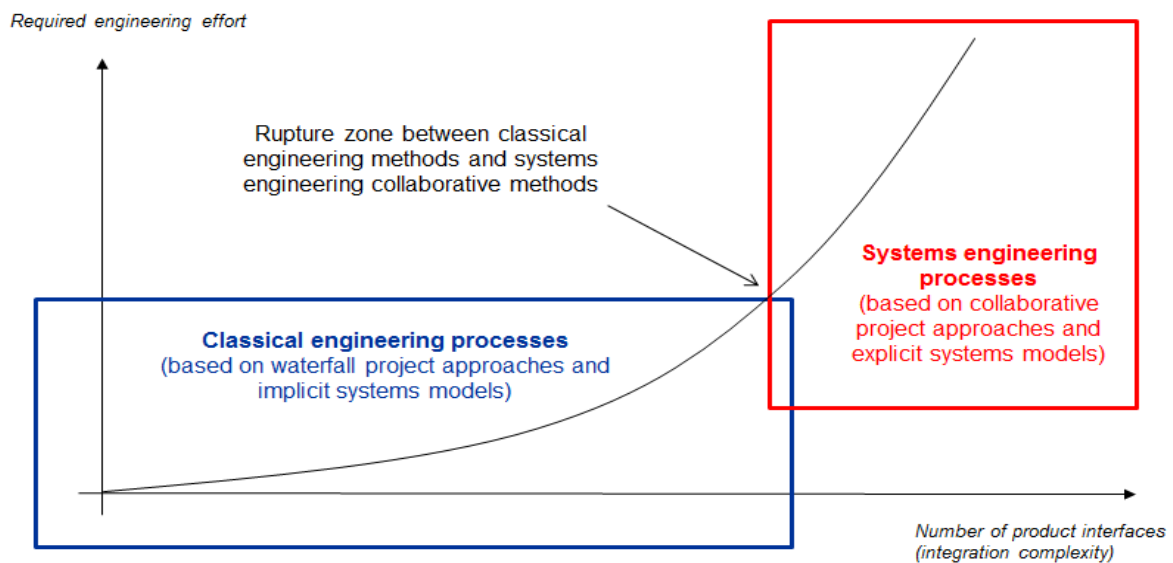


Figure 11 – Project effort and integration complexity relationship

The consequence of this situation is that we can now distinguish two types of engineering, depending whether the complexity of a given system lies before or after the complexity rupture zone that we pointed out (see Figure 11). The first type of engineering, working perfectly well for low complexity systems, is just what we will here call “*classical engineering*”: it is usually based, on one hand, on waterfall design project approaches, induced by a separation of engineering organizations by domain specialties, and, on the other hand, on implicit systems models where key technical knowledge only lies in the brains of a limited number of technical experts.

Such engineering unfortunately do not work anymore when the systems complexity threshold that we pointed out, is crossed. One arrives then in a domain where transversal collaboration becomes

<sup>38</sup> This probably never occurs, which means that the engineering effort increasing will be probably much more that the simple impact of the single complexity parameter, due to the A factor in Equation 1 that will probably also heavily evolve in such complex situations.

<sup>39</sup> The system maturity of an industry may evolve among time due to its industrial cycles (see Case study 1).

<sup>40</sup> Typical verbatims in such complexity situations are: “it is impossible to take a reasoned decision”, “we have the feeling of not mastering anymore anything”, “we are spending all our time to understand our problems and then there is not more time to solve them”, “we are fighting against a more and more increasing pressure of our environment”, “cost and delays are exploding and we cannot do anything to avoid it”...

key in order to complete individual expertise (which is of course still mandatory) and where explicit systems models are now mandatory due the fact that it will be basically impossible to handle implicitly the complexity one is facing. In other words, one enters in “*systems engineering*” which is the engineering approach dedicated to integrative complexity mastering, that we will discuss more in details in the two next sections.

### **System Maturity Cycles: the European Space Industry Case**

The European space industry is a sector where systems complexity is well mastered, due to a very early introduction of systems engineering that goes back to the 1960's when Europe decided to construct its own spatial capability. Systems engineering was then chosen as a key tool to reach this objective. It took afterwards two to three decades to totally master this practice which is today completely integrated within all engineering processes.

In the 1990's, the failure of the very first Ariane 5 launch (cf. Appendix B for more details on that case) led however to a deep evolution of the systems engineering processes. The issue was to better integrate and monitor the emergence of critical real-time embedded software with its difficult associated safety issues. A new industrial cycle began when these difficulties were mastered, which quite interestingly is finishing nowadays due to the growing pressure of the new low-cost competitors like SpaceX or Blue Origin.

The European space industry indeed totally reshaped with the creation – announced in June 2016 – of a new company – Airbus Safran Launchers – that integrates vertically two key players. It is too early to tell how will probably evolve again systems engineering in that context, but there is no doubt that this practice will play a key role in the success of the new challenges that the European space industry will have to face in the near future.

#### **Case study 1 – System maturity cycles: the European space industry case**

As a matter of fact, one shall finally also notice that integration complexity is increasing in most areas due to the impact of many new technological paradigms. The replacement of analogical control by numerical control or the new electrical architecture are for instance deep trends, which are regularly creating new software, electronical or electrical interfaces between components within numerous industrial systems, hence by the same way, mechanically increasing the integration complexity of the corresponding systems. Most of industries will thus cross the complexity threshold along the lifecycle of their products and thus be obliged to deal with systems engineering and architecting, if they want to be able to efficiently face this challenge.

### **1.3 Addressing Systems Architecting becomes Key**

Due to that complexity threshold as presented in section 1.2, which is – in our understanding – the root cause of most engineering issues observed in complex systems development (we here refer to Appendix B for more details), addressing systems architecting becomes therefore key for engineering organizations that are dealing with complex systems. These organizations are indeed confronted to the need of working efficiently in transversal mode, due to the core integrative nature of the systems that they are designing and developing.

This is exactly the context where systems architecting will bring all its value. This discipline allows reasoning about hardware & software components (technical dimension) and human factors (human dimension) related to a given system, both in a unified framework and in subsidiary with all existing disciplines & engineering fields (see Figure 12). To achieve this goal, the fundamental principle on which systems architecting relies, is a *logical vision of engineering* brought by systemics. As one can easily understand, it is indeed merely impossible to strongly consolidate all formalisms that are used to model and work with the different components of a heterogeneous system<sup>41</sup>. Systems architecting thus proposes a purely logical approach to federate all these “local” formalisms, rather than trying to consolidate them in a unique global formalism.

The main idea here is to simply use logic<sup>42</sup> as a pivot language in order to be able to work in the same way with all hardware, software and “humanware”-oriented disciplines, involved in a given complex system design & development context. Each discipline can indeed easily formulate its requirements, constraints, findings or models using the universal language provided by *logical predicates*, which are formally Boolean functions telling us whether a given property is satisfied or not depending on the values of its entries (see [91]). In a system context, a logical predicate is thus nothing more than a statement that may true or false for a given system. A typical reasoning in systems architecting will thus be based on logical predicates associated, on one hand with the whole considered system and on the other hand with the different involved components and disciplines<sup>43</sup>.

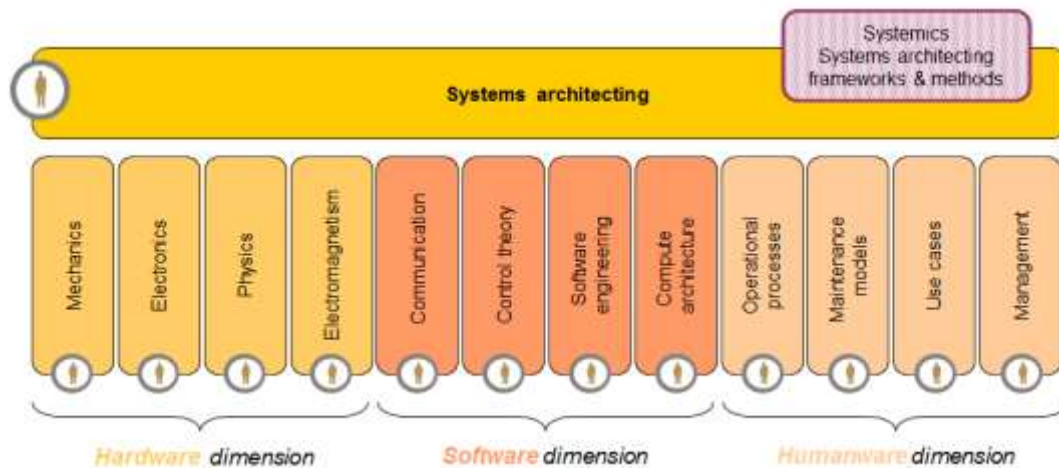


Figure 12 – The integrative & collaborative dimension of systems architecting

<sup>41</sup> Electromagnetism or fluid mechanics are for instance based on partial differential formalisms such as Maxwell or Navier-Stokes equations, when signal processing relies on a distinction between time and frequency domains, leading to Fourier or Z transforms formalisms, which have all typically nothing to do with the logical and discrete formalisms used in information technology. Human factors are moreover of a totally different nature, without any strong mathematical background, but they shall also be included in the global picture. We thus do not believe to the existence of a “universal theory of systems” that could federate all these different formalisms since they are analyzing the world in lots of mutually incompatible ways. In the best of our knowledge, systems architecting remains thus the only tool to consolidate all these points of view.

<sup>42</sup> In the mathematical meaning of that term (see for instance [12]).

<sup>43</sup> Imagine for instance that thermic experts tell us that predicate P1 = “the engine works well only when refrigerating fluid temperature is less than 30°C” is true, when we know through a discussion with fluid mechanics division the that predicate P2 = “the refrigerating fluid temperature can be over 30°C when pressure is low”. Somebody who has the global vision on that two domains can then easily deduce that P3 = “the engine may not work correctly when pressure is low” is true, since it is a logical consequence from P1 and P2, formally  $P1 \wedge P2 \rightarrow P3$  (where the  $\wedge$  symbol means AND). This is typically a (here quite simple) systems architecting type of (logical) reasoning.

As one can see, there is therefore absolutely nothing magic in the promise of systems architecting of proposing a unified framework for working with all systems dimensions, as a simple consequence of the universality of logic<sup>44</sup>! One may also notice that systems architecting can thus be fundamentally seen as an observational modeling approach<sup>45</sup>: the construction of a system model – in the meaning of systems architecting – is indeed the result of the observation of all components and engineering domains involved in the considered system and of the federation of all these observations in a unique logical model of system level.

Last, but not least, it is important to also point out that systems architecting always integrates a strong collaborative dimension: it is indeed simply not possible to construct a system model without implying all people who are representing the different involved system components and engineering domains, as soon as one wants to get a realistic model<sup>46</sup>. Moreover sharing a system model is also a key good practice for ensuring its robustness. Collaboration is thus at the very center of any systems architecting approach (cf. Figure 12, but also section 1.5 for more details).

All these elements allow easily understanding that systems architecting is the discipline by excellence which is required to efficiently address systems integration issues. At this point, it may thus be useful to position precisely systems architecting within systems engineering (see Figure 13).

To this purpose, let us first recall that systems engineering is nothing else than systemics applied to engineering. Systemics<sup>47</sup> here refers to the discipline that deals with – formal and real – systems. It provides holistic vision and holistic analysis methods (see Chapter 0 and [96]), integrating crosswise all dimensions of a given topic. Systemics applies to many application domains such as archaeology<sup>48</sup>, biology<sup>49</sup>, city planning<sup>50</sup>, enterprise<sup>51</sup> or psychology<sup>52</sup> to provide few non-exhaustive examples. From a systemics perspective, systems engineering is thus just another application domain.

---

<sup>44</sup> This probably also explains why systems architecting is so difficult to penetrate in practice in engineering organizations. Most of engineers are indeed very well trained in analysis and control theory, but absolutely not in logic which happens to be the core fundamental on which relies systems architecting. It is really a pity if one remembers that logic is probably the oldest scientific discipline in the world, which can be traced back to Aristoteles in the 4<sup>th</sup> century before Christ (see [84]).

<sup>45</sup> A typical example of observational model is Ptolemaeus epicyclic system which was a purely geometric model, used from the 3<sup>rd</sup> century before Christ up to Copernic in the 16<sup>th</sup> century, to explain the movements and variations in speed and direction of the apparent motion of the Moon, Sun and visible planets (cf. [65] or [85]). This model fitted perfectly with the observations since it was exactly constructed on that basis. It was also predictive, in particular with respect to phenomena such as the Moon and Sun eclipses. But it is nowadays considered as completely false from a physical perspective. This example illustrates what is expected from an observational model: precision and prediction, but not necessarily truth...

<sup>46</sup> The best way to construct an unrealistic system model is indeed probably to construct it alone in its corner...

<sup>47</sup> Systemics can be traced back to the 50's with the seminal work of H. Simon published in 1962 (cf. [75]). One usually also cite von Bertalanffy, who tried – but without being terribly convincing – to construct a “general systems theory” (cf. [80]).

<sup>48</sup> We refer in this matter to Case study 2.

<sup>49</sup> Systems biology defines itself as the computational and mathematical modeling of complex biological systems and as an emerging engineering approach applied to biological scientific research (see [93]). Systems biology is a biology-based interdisciplinary field of study that focuses on complex interactions within biological systems, using especially a holistic approach applied to biological research (holism instead of the more traditional reductionism).

<sup>50</sup> Systemics applied in city planning means considering all the many different dimensions of a city (economy, energy supply, entertainment, people welfare, traffic, water distribution, waste management, etc.) when taking an urbanistic decision. Due to the numerous interactions between the sub-systems of an urban system, it is indeed quite easy to take an optimal local decision which is globally under-optimal (e.g. developing public transportation in an area to resorb local traffic congestion, but that as a side-effect is increasing the value of the considered area, thus bringing middle class people in that area and moving population, and at the very end, creating new traffic jams since the new inhabitants had two cars in average, one for each person in a husband-wife couple). One thus understands that systemic models are of interest for urban design.

This fact must of course not make us forget that *systems engineering* (cf. [94] for more details) also has its own tradition that goes back to the 1950's, with first textbooks in the 1960's (see [96]) and its first industrial processes formalization with the seminal NASA systems engineering handbook in the 1970's (see [66] for current edition). Many other textbooks (such as [9], [10], [15], [30], [47], [57], [61], [62], [71], [74], [78]) and industrial standards (such as [5], [42] or [44]) were then constructed in the line of that initial works. More recently, the International Council on Systems Engineering (INCOSE) emerged in the 1990's. It is currently federating and developing the domain at international level (see for instance [42] for the INCOSE systems engineering handbook).

### **Systemics applied to Archaeology: the Cretan Case**

A puzzling example of systemics applied to archaeology is the recent understanding of how the Cretan civilization disappeared suddenly in the 15<sup>th</sup> century BC (cf. [7] or [67]).

The story started when the study of the Thera site on Crete revealed enigmatic geological strata: they were indeed identified by an expert in hydrology like a riverbed, but no river could be located there for both geological (due to the geology of the site) and archaeological reasons (due to the presence of a human habitat at the same place). Moreover the biological analysis of these strata highlighted then several marine fossils of high sea origin, although sea was quite far away from the site where the strata were.

By putting together their archaeological, geological, hydrological and biological data and by crossing them with those from other Cretan sites, archaeologists gradually understood that they discovered the last traces of a tsunami which destroyed the city they were studying.

This hypothesis was subsequently validated by checking with Carbon 14 dating techniques that the similar observations made on other archaeological sites took place in the same time period! Last but not least, volcanologists were involved in order to identify the possible origin of that tsunami, which was localized in Santorini. Oceanographers constructed then an oceanographic model to demonstrate – successfully – the propagation of a tsunami between Santorini and Crete, which are far away from around 200 kilometers.

As one can see, the combination of many disciplines was necessary to globally understand an apparently local phenomenon.

### **Case study 2 – Systemics applied to archaeology: the Cretan case**

Following INCOSE (see again [42]), systems engineering defines in particular as “an interdisciplinary approach and means to enable the realization of successful systems. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation while considering the complete problem

---

<sup>51</sup> Systems approach applied to enterprise typically gives rise to enterprise architecture frameworks (see [86]) such as for instance TOGAF® (cf. [77]) or our own (see Section 0.4).

<sup>52</sup> Systems psychology is a branch of both theoretical psychology and applied psychology that studies human behavior and experience in complex systems. Individuals are considered within groups as systems (see for instance [95]). A consequence of such an approach is that one cannot cure an individual who has psychological or sociological problems without trying to understand the groups to whom he/she belongs. Root causes of his/her problems may indeed be found at group level and shall thus be addressed at that level and not at the individual level. For instance, if an alcoholic father beats his child, one must typically understand the cause of his alcoholism and tackle it, rather than taking the child out of his/her family.



(operations, performance, test, manufacturing, cost & schedule, training & support, disposal)”. When one analyses more precisely the systems engineering processes, one can however decompose them naturally according to a product/project distinction as discussed in section 1.1, that is to say in the following two types of activities of quite different nature:

- Project-oriented activities, such as systems projects planning, follow up and monitoring, engineering referential and configuration management, reporting and quality, which may be seen as systems project management,
- Product-oriented activities, such as requirements engineering, operational, functional and constructional architecting, trade-off and safety analyses, verification & validation, which do cover exactly the scope of systems architecting.

In that perspective, systems engineering can be seen as the union of systems project management and systems architecting, which can thus also be analyzed as the core part of systems engineering, dedicated to the design & construction of robust systems models. System architecting shall thus be seen as the discipline synthesizing the methods and the tools which allow an exhaustive & coherent modeling of a system (in its triple operational, functional & constructional dimensions) in order to manage it efficiently during its lifecycle (design, test, deployment, maintenance, ...) <sup>53</sup>.

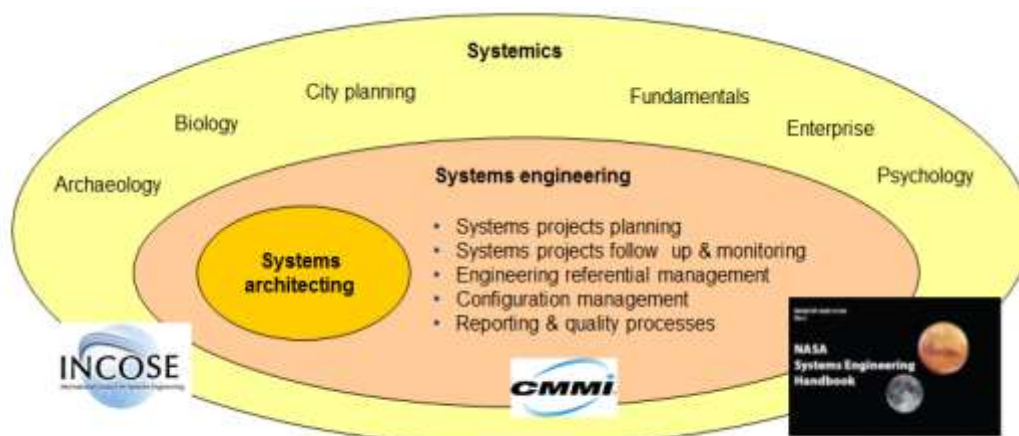


Figure 13 – Relative position of systems engineering and systems architecture within systemics

## 1.4 The Value of Systems Architecting

To complete our discussion on systems architecting, let us now focus on the value brought by this discipline, which was in particular quite well analyzed by Honour in [41]. The key point to understand is that the systems engineering approach in which systems architecting takes place, mainly consists in redistributing the engineering effort towards upstream phases of the project, in a “definition” phase, in order anticipating design risks as early as possible within a systems development project.

<sup>53</sup> Other alternatives definitions are proposed by 1) ANSI / IEEE 1471-2000 [6]: systems architecting is the process of describing the structure of a system, given by its components and its internal interfaces, and the relationships of the components of the system with its environment among time (i.e. starting from the design and including all possible evolutions of the system); 2) Wikipedia [92]: systems architecting is the process of defining a set of representations of an existing (or to be created) system, i.e. of its components (hard-ware, software, “humanware”, functions, roles, procedures, etc.), of the relationships that exist between these components and of the rules governing these relationships.

Figure 14 below, extracted from [41], perfectly illustrates this paradigm, where more time is initially spent better understanding the system to develop and thus reducing the project risks in the future, compared to a traditional design. Such an approach strongly relies on systems architecting since the “definition” phase will typically contain a strong part dedicated to the construction of a systems architecture file that will allow to analyze the system under design in all its dimensions<sup>54</sup>, the other part being devoted to the project planning. In that perspective, systems architecting can thus be seen as a risk management good practice in complex systems development contexts.

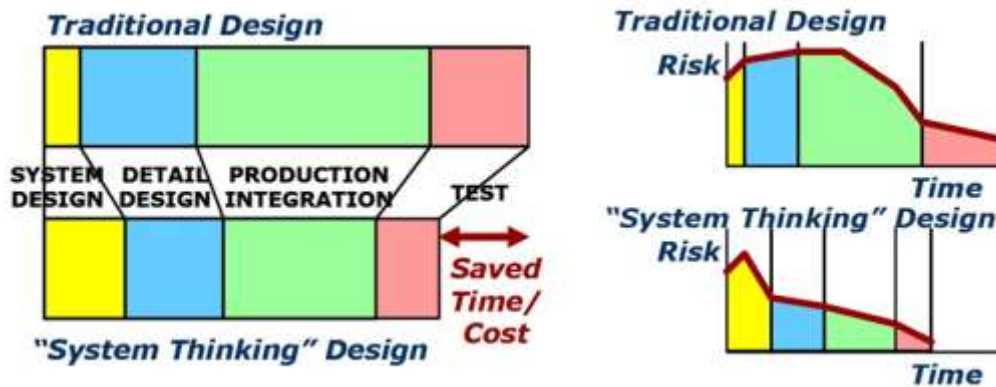


Figure 14 – Systems Architecting as a risk management practice<sup>55</sup>

Many evidences support this point of view (see for instance again [41] for many concrete examples). Among them, we will consider a quite interesting one which comes from NASA. In 1992, the financial controlling office of NASA indeed analyzed 32 major spatial programs conducted during between 1970 and 1990, comparing the final budget overrun with the budget ratio which was allocated to the initial “definition” phase, upstream the Preliminary Design Review (PDR), which does especially include the initial systems architecting analyses, as discussed above. The finding of that financial analysis was quite clear: budget overrun was indeed statistically inversely proportional to the budget dedicated to the definition phase. More precisely, it could be seen that budgets more or less always at least double when the definition phase is weak and that the optimal ratio to dedicate to the definition phase seems to be around 15 % of the global budget (see Figure 15 below). Consequently, this probably means that, to cover optimally product & project risks, a complex systems manager should allocate around 5 % of its global budget to systems architecting, strictly speaking, with another 10 % being reserved for project-oriented initiation and preparation activities.

One can thus understand that systems architecting is a key tool for mastering systems design and development projects in the respect of their quality, cost, delay and performance constraints (QCDP), as soon as one deals with complex systems produced by the integration of many technical systems (hardware & software) and human systems (people & organizations). We shall finally recall some key principles<sup>56</sup> provided by systems architecting that allow achieving this objective:

<sup>54</sup> Typically such as environment, lifecycle, use cases, operational scenarios needs, functional modes, functions, functional dynamics, functional requirements, configurations, components, constructional requirements, constructional dynamics, critical events, dysfunctional modes and behaviors, verification & validation.

<sup>55</sup> This figure was reproduced from [41].

<sup>56</sup> That one shall always have in mind during a systems design & development project.



- **Provide simple, global, integrated and shared visions of a system:** “to be simple” and capture completely all dimensions of a given problem in complex environments, which does not mean “to be simplistic”, is always very difficult...
- **First, think about needs and not about solutions:** this last point being unfortunately the common rule in most of systems development projects. One shall thus remember that the “customers” of a system project shall fundamentally feed the systems architecting process.
- **Sort out all “spaghettis” in a system design:** in order to avoid mixing everything (objectives, functions, technical constraints, etc.) and confusing ourselves while confusing others which are also again very common bad practices in too many complex systems contexts.

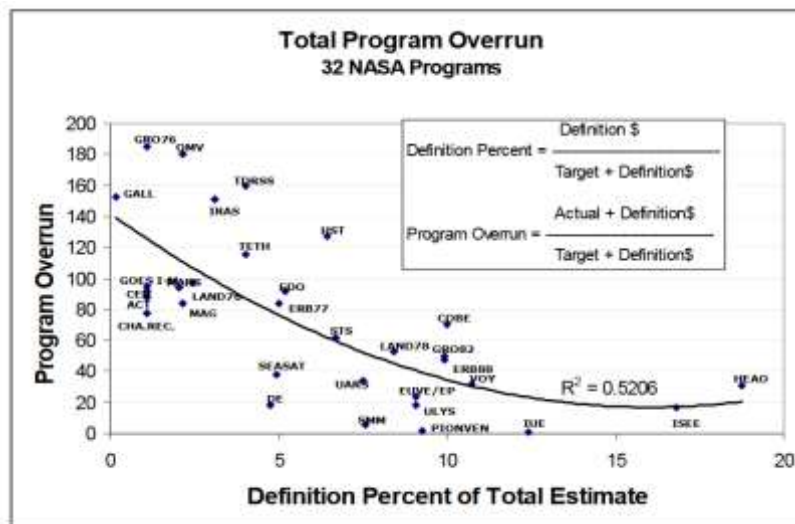


Figure 15 – NASA statistics supporting the importance of the definition phase<sup>57</sup>

## 1.5 The key Role of Systems Architects

Systems architecting would of course not exist without the right key people, i.e. systems architects, to support the architectural process! The systems architect shall indeed fundamentally be the core responsible of the systems integration issues. He/she shall thus ensure that the interfaces of the subsystems of the target system under design are reasonably robust for the purpose of the system development project<sup>58</sup>, or in other terms that they will not be questioned (or as little as possible) by the technical managers in charge of the different subsystems.

To achieve such objectives, it is necessary to play on a different dimension than the purely technical one. A key problem within system design is indeed that it is usually not enough to have the "best" possible system architecture for that it is automatically picked up and used by everybody. The global optimum, typically with respect to quality, cost, delay and performance criteria, for an integrated system is indeed never<sup>59</sup> the union of the local optima of each of subsystems that compose it: the

<sup>57</sup> This figure was reproduced from [41] where it was traced back to [37].

<sup>58</sup> And possibly beyond, but this is another issue, namely that of the reusability of reference systems architectures and of product lines design, we will not discuss here.

<sup>59</sup> This is true as soon as the system has not a linear behavior with respect to its entries, which is never the case for complex systems. This result can be traced back to Richard Bellman in the 50s (see [13]).

consequence is that each subsystem shall necessary individually be sub-optimal with respect to these criteria, if one wants to reach global optimality at system level.

This fact is unfortunately not easy to accept for the subsystems responsible who will not design their sub-systems with a global vision, as can have the architect of the whole system<sup>60</sup>. To solve this "human" problem, which is intrinsic to the design of integrated systems, one must put stakeholder alignment mechanisms at the heart of any system architecting process. These alignment activities shall in particular ensure that the structuring systems architectural choices will always be shared by their stakeholders. Systemically speaking, such mechanisms will indeed guarantee that the technical interfaces of the product system will always be discussed and accepted at the "human" interfaces to which they are allocated within the project system (see Figure 16).

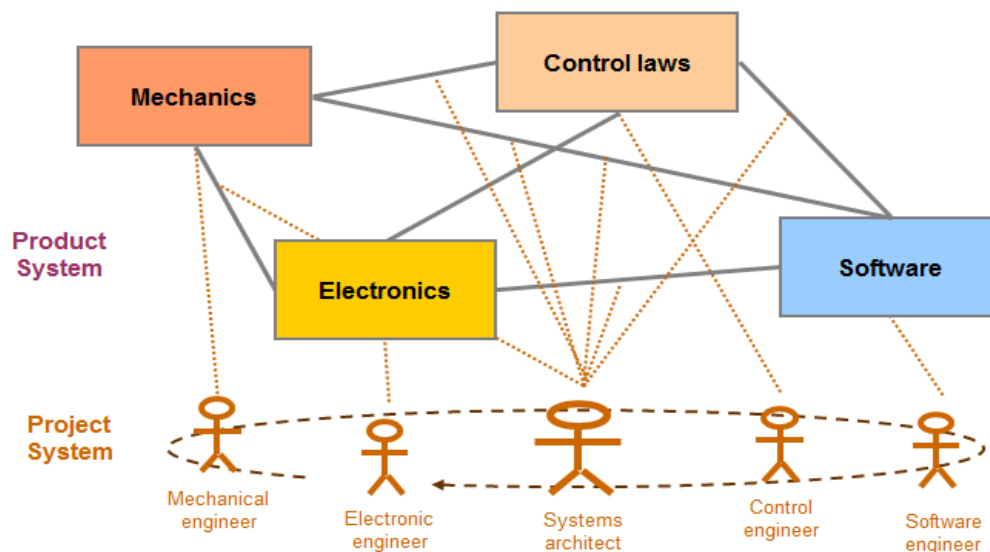


Figure 16 – The key role of the systems architect

This quick analysis therefore shows that a systems architect must always be fundamentally capable of converging actors on the architectural choices which he/she is responsible. This convergence work – which is a substantial and essential part of the systems architect job<sup>61</sup> – is however not simple at all. It indeed requires mastering, among purely technical competency, a set of completely different soft skills that could be qualified of “human” architecting and engineering since they are highly involving the “human” dimension of the project system. At that level, the systems architect shall for instance typically have the following key capability<sup>62</sup>.

- *Identifying all stakeholders of a given system architecture*: it looks as an apparently simple activity, but it is often terribly difficult to achieve in practice since it requires confronting the

<sup>60</sup> The difficulty of understanding the importance of system architecting comes from this situation. Engineers usually deal with technically homogeneous subsystems of a given system (i.e. the “boxes” of Figure 16) which are clearly visible to all. On the other side, the systems architect takes care of the system interfaces (that is to say the “arrows” in Figure 16) that nobody sees because they are strictly speaking not material. The architecture work is thus done somewhere in the invisible. It is thus difficult to detect for the uninitiated, while fundamental since it fixes the framework in which to do engineering (a bad architectural framework can typically only lead to a “bad” system).

<sup>61</sup> And that makes a good “systems” architect a kind of 6 feet sheep...

<sup>62</sup> Which are typically pre-requisites to achieve the success of any actors’ convergence in a systems architecting context.

complex and changing reality of engineering organizations. Ensuring the completeness and validity of a particular organizational analysis is indeed never easy. It is notably more than classical to forget key players within a given engineering scope and to identify erroneously others<sup>63</sup>. Moreover, organizations are often changing rapidly. Making a stakeholders mapping should thus always be constantly update if one wants to keep pace with reality.

- *Aligning these stakeholders on a same architectural solution*: this is a real “facilitation” skill in the best sense of the term since such a work requires a real difficult technical and human know-how. A systems architect will only succeed to achieve stakeholders’ convergence on given architectural choices if he plays consistently on both the technical side, that he/she should of course perfectly control to be credible, and the human level, in order to secure strong consensus that will withstand the test of time.

## 1.7 How to Analyze a Systems Architect Profile ?

To conclude this chapter, we will briefly present the key characteristics of an (ideal) systems architect profile. Such profiles are indeed quite rare and always difficult to find as a matter of fact. Having some clues on that still poorly explored topic may thus be helpful for any engineering organization that would like to increase its systems architecting capability.

With CESAM framework, systems architecting skills are indeed analyzed from the following three completely different perspectives, as described in Figure 17:

- *Dimension 1 – business & technical skills*, referring here first to standard capability such as scientific education, analysis capability and business orientation, but also to a multi-area knowledge, both from a product and engineering domain perspective;
- *Dimension 2 – soft skills*, that is say leadership, communication and facilitation ability (key for being able to create consensus within stakeholders), curiosity & open-mindedness, together with a strong customer orientation (key for analyzing correctly a systemic environment);
- *Dimension 3 – architectural skills*, with first of all abstraction and synthesis ability, modeling & needs capture capability, systems architecting main processes mastering and integration, verification, validation, qualification knowledge.

A good systems architect must indeed have a *well-balanced profile* with respect to all these three dimensions. This gives the key clue on how to identify systems architects. One shall indeed measure the maturity of a future systems architect on the previous three axes and check that this maturity is high in all of these axes, typically by analyzing concrete realizations done by the candidate, in order to ensure good systems architecting capability within somebody.

One may also pay attention to the following three unfortunately quite common anti-profiles that should not be confused with systems architect profiles. The first anti-profile is the *technical expert profile*, which refer to somebody who is excellent with respect to dimension 1, but much more poorly the other ones. A good systems architect was probably very good in several technical domains, but is clearly not anymore a technical expert, in the usual meaning of that term. The second anti-profile is the *manager* one who mainly developed in dimension 2. Good technical managers are usually very

---

<sup>63</sup> This can be for instance achieved by badly analysing the role of somebody in a given organization.

bad systems architects due to the fact that they have a project-oriented, and not a product-oriented vision. Thus do never take a managerial profile for a systems architecting mission, it will not work! The last anti-profile is a bit vicious since it corresponds to somebody who would have a good level in dimension 3, but not in the others, that is to say typically with poor soft skills. We are here referring to a *methodologist profile*, in other terms to somebody who knows very well all systems architecting and engineering methods, but without having the ability of creating consensus among them<sup>64</sup>. A good systems architect has clearly strong methodological capability, but he/she shall never be confused with a methodologist<sup>65</sup> (and vice-versa).

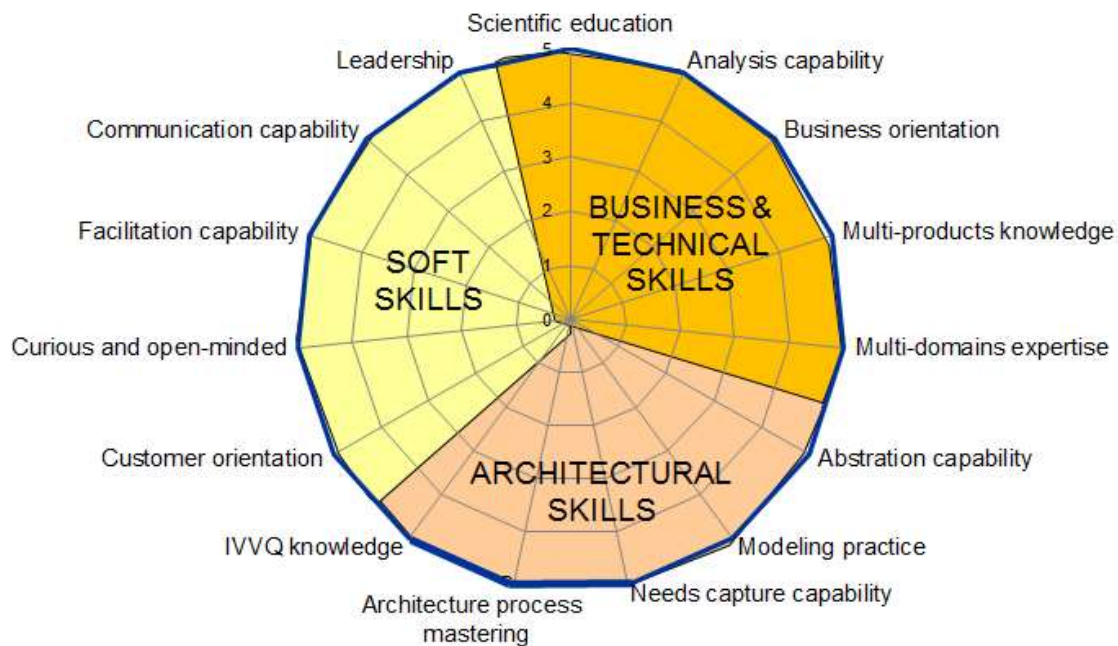


Figure 17 – Ideal profile of an ideal systems architect

Last but not least, do also not confuse systems architecting with systems modeling. Modeling is a tool for the systems architect, but never an end in itself. A systems architect shall master systems modeling, but more importantly he shall know why and how, and thus when, to model something. We refer to Appendix C for some good practices in that matter.

<sup>64</sup> Systems architecting is never a “behind the door” process. This does not mean that solitary design work is not necessary... An architectural work must always *identify the needs of the stakeholders and the constraints of the involved engineering domains*. This requires a huge presence on the field! A good architecture is indeed always an architecture shared by all stakeholders, both external and internal: thus a permanent alignment of all concerned contributors shall be ensured.

<sup>65</sup> Systems architecting is indeed not a quality process, taken here in a purely normative meaning. This does not mean that quality is not fundamental in Systems Architecture, on the contrary... Its effectiveness *is thus not measured by the syntactic realization of deliverables*, but by the intrinsic quality of the proposed architectural choices (which can often only be validated through peer reviews). *The production of documentation is also not its main objective*: an architectural work must remain smart and produce the « minimal effective » quantity of technical documentation which is required.

# Chapter 2 – CESAM Framework

## 2.1 Elements of Systemics

Before going further, we first need to introduce the notions of interface and of system environment. These elements of systemics will indeed be useful for presenting further the CESAM framework. We refer to Definition 0.1 and Definition 0.2 of Section 0.1 for the fundamentals of systemics, that is to say the core definition of a system, on which CESAM framework is constructed.

### 2.1.1 Interface

The concept of interface is the first key systemic concept<sup>66</sup> that we need to present. In this matter, let us thus now recall that an *interface* models an interaction, an exchange, an influence or a mutual dependence between at least two systems (some interfaces may indeed be complex when involving several systems<sup>67</sup>). Beware that an interface may not necessarily have a concrete implementation: it is just a way of expressing the relative impacts that different systems have one on the others<sup>68</sup>.

With respect to a given system *S*, interfaces may then be either *external* when they are involving the considered system and some other external systems, or *internal* when they are only relative to sub-systems of *S*. Figure 18 illustrates this notion on the electronic toothbrush example by showing two external interfaces between the toothbrush and the end-user (here one with his/her mouth, the other with his/her hand) and several internal toothbrush interfaces (that is to say two mechanical interfaces and one inductive interface between the base and the body of the toothbrush).

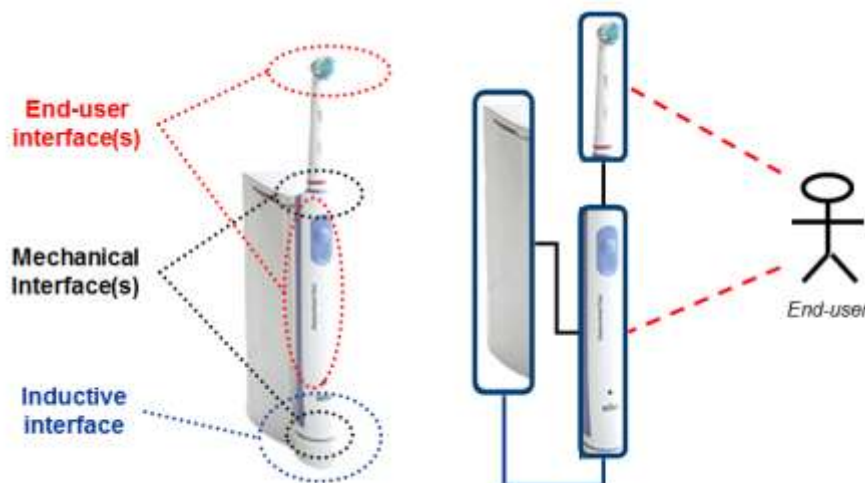


Figure 18 – Examples of interfaces for an electronic toothbrush

<sup>66</sup> Remember indeed that a systems architect deals fundamentally only with interfaces... (cf. section 1.5).

<sup>67</sup> The toothbrush has for instance a complex interface with the user during the brushing phase since both toothpaste, water and user mouth, teeth & hand are then involved at the same time.

<sup>68</sup> Most of people are making for instance the confusion between networks and interfaces. A network is indeed strictly speaking not an interface, but another system with which the system of interest has also a specific interface. In first stages of a design, one may of course abstract it and only consider the logical interface between the systems it connects, but the abstract interface involved in such a mechanism shall not be mixed with the network system in itself.

Note also that the technical interfaces, corresponding to a concrete interaction or exchange between several systems, are usually the easier to identify since they refer to a visible relationship between the involved systems. However the invisible interfaces, relative to an influence or interdependence between different systems that may typically be of strategic, political, societal or regulatory nature<sup>69</sup>, are also crucial. They are indeed much more difficult to find. They can thus be easily initially skipped and discovered too late, only at the moment where the designer will “see” their impact...<sup>70</sup>

### 2.1.2 Environnement of a System

The recursive nature of systemic analysis naturally leads us to introduce the notion of (systemic) environment of a system. We here mean a closed<sup>71</sup> super-system of a given system  $S$ , which will thus be a natural basis to begin a recursive analysis of  $S$ . To be more specific, we will say that a system  $Env(S)$  is an *environment* for  $S$  if it is a closed system that results from the integration of  $S$  and of another system  $Ext(S)$  that will be called the outside of  $S$  within its systemic environment  $Env(S)$ .

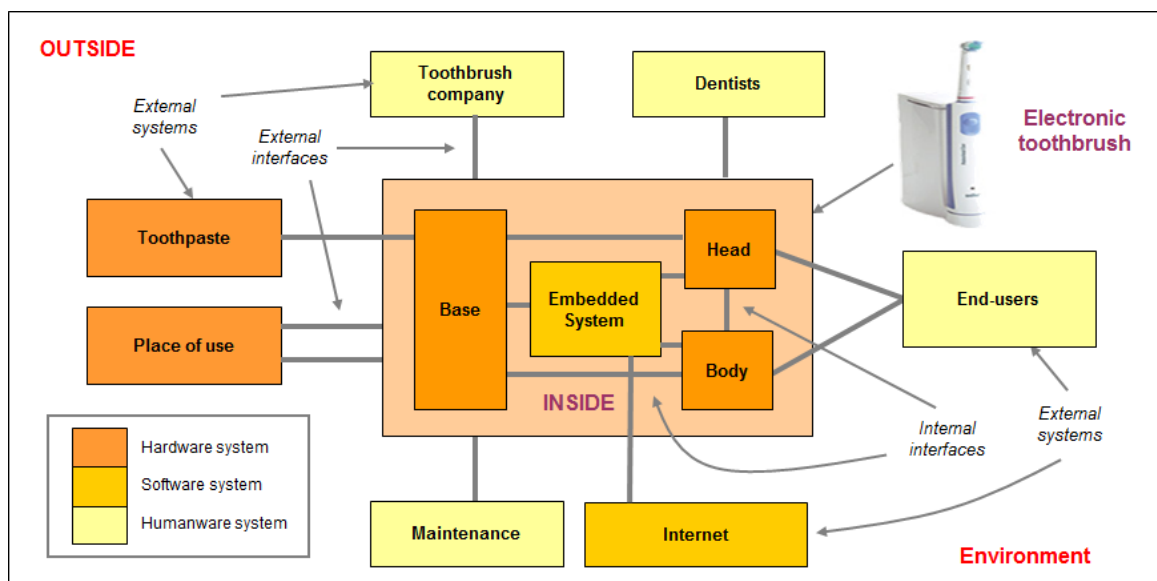


Figure 19 – Environment of an electronic toothbrush

A real system has of course many real systemic “environments” in the meaning of our definition, the physical universe in its whole being typically a common environment for each real system... However the pragmatic constraints of any system design & development project lead us to define *a reference environment* for any concrete system  $S$ , which will be called the environment of  $S$  by a slight abuse of language. This reference environment is the smallest “useful” systemic environment of the system of interest  $S$ . It is just the system that results from integration of  $S$  and all other real systems, external to  $S$ , that have an influence on its design. We will thus neglect in this way all other real external systems when one considers that they have no strong interdependence or no strong interaction with  $S$ <sup>72</sup>.

<sup>69</sup> Think also on the possible impacts of competitors, new technology, industrialization or maintenance on your system.

<sup>70</sup> Such issue typically occurs when existing stakeholders who were forgotten during the initial analysis, remembers to the project team that they exist! We refer to the first section of Appendix B for an illustrative case study of this situation.

<sup>71</sup> A closed system is a system which is considered to have no external interfaces.

<sup>72</sup> One can for instance typically consider that the Proxima Centauri star has simply no influence on most of the engineered systems on Earth, even if it is probably sending them lots of neutrinos each day...



Defining the environment of a system means defining its border, that is to say defining what is inside the system and what is outside the system. Figure 19 typically illustrates this key distinction on the electronic toothbrush example. We described there the main external and internal systems relatively to the considered case study, also specifying their hardware, software or “humanware” nature.

Note also that defining this border is typically the first key systems architecting decision that one must take in practice, since it allows precisely specifying the system of interest which is under design. This decision is usually quite difficult<sup>73</sup> to manage in reality, especially when a system is a small part of another system in which it should precisely be delimited.

One shall also beware of the fact that one can reason on many different systems, all of them being naturally connected to the system of interest, such as for instance the used system (where one put the user within the system of interest) or the maintained system (where one puts the maintenance system within the system of interest), that shall never be mixed.

The inside/outside distinction is also at the heart of the separation between the different visions that are used in systems architecting (see next section for more details). One can typically not speak of needs and requirements with respect to a system without having a clear border between a system and its outside, as we will see in the sequel of this pocket guide.

## 2.2 The three Architectural Visions

We are now in position to introduce the three systems architectural visions which will be our first key systems architecting tool for analyzing any system.

### 2.2.1 Architectural Visions Definition

The heterogeneity of the environment of a system requires to address it by means of different axes of architectural analysis in order to be able to integrate the whole set of various perceptions of the different system stakeholders<sup>74</sup>. Such a consideration naturally leads us to organize these points of views according to different architectural visions that are both necessary due to the variety of any systemic environment, but also useful since they allow decoupling the representations of a given system in different “properly” interrelated separated views<sup>75</sup> which always leads to better clearness and flexibility in terms of system design and development management.<sup>76</sup>

As a matter of fact, each integrated system *S* can always be completely analyzed from three different and complementary perspectives that give rise to three generic *architectural visions*, that is

---

<sup>73</sup> In the worse cases, it may typically take several months to identify precisely the scope of the system of interest.

<sup>74</sup> For an electronic toothbrush, these perceptions can be typically the one of the mother who wants good dental hygiene for her children, the one of the stressed business person who wants to clean his/her teeth as quickly as possible, the one of the dentist who will understand whether the toothbrush is efficient or not with respect to teeth cleaning and the one of the engineer who knows how to construct and how works the electronic toothbrush.

<sup>75</sup> This way of managing different views on the same system is in fact quite common in usual life. Think for instance of a tourist visiting a city. He/she will probably use many different views, typically provided by a touristic guide, a metro map and a city map. To find his/her way, he/she may for instance first chose the monument to visit in the touristic guide, then move there using the metro map and finally manage the local approach using a city map. In architectural terms, the tourist is thus taking information in different coupled views and integrating them in order to take the “good” decision!

<sup>76</sup> A classical difficulty is that such views can correspond with totally – and even sometimes opposite – different perceptions on the system, depending on the involved stakeholder.

to say operational, functional and constructional visions, each of them grouping different types of systemic models, as defined below:

- **Architectural vision 1 – Operational vision:** strictly speaking, the operational vision groups only the models of the environment of S – and not of S itself – which are involving S. Such operational models are thus describing the interactions of S with its environment.
- **Architectural vision 2 – Functional vision:** the functional vision groups all system level models describing the input/output dynamics of S, without making reference to its concrete components<sup>77</sup>. Such functional models are thus abstractly modeling the behaviors of S.
- **Architectural vision 3 – Constructional vision<sup>78</sup>:** the constructional vision groups the natural system level models of S constructed by composition of the lower level models associated with its components. Such constructional models are thus describing the structure of S.

An illustration of these three different architectural visions is provided by Figure 20 on the electronic toothbrush example. We sketched there our three different types of models, with their connections (see the last subsection of the current section for more details), each of them illustrating a different architectural vision. One sees that the operational vision is not interested by the toothbrush behavior or structure, but just by describing its interactions with (in this example only some) external systems, which are here Power supply, End-users and Internet. On the other hand, the functional vision gives the main toothbrush behaviors – i.e. Provide electrical power, Generate brushing power, Provide brushing capability – that allow producing these external interactions as captured by the operational vision, when the constructional vision shows how to implement concretely these internal behaviors through suitable components, here a base, a body, an head and an embedded software.

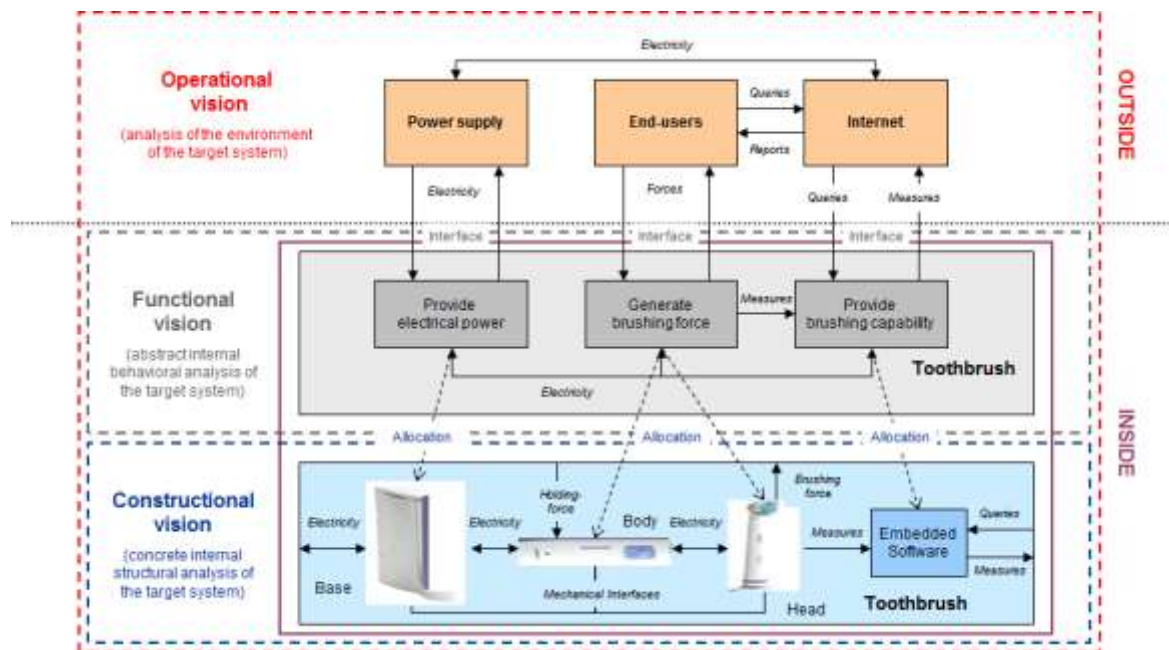


Figure 20 – Illustration of the three architectural visions on an electronic toothbrush

<sup>77</sup> That is to say without referring to any technological choice or to any chosen solution.

<sup>78</sup> Other names do classically exist for that vision. One may for instance also speak of *structural vision*. Some frameworks are also speaking of *logical vision* to denote the constructional vision in the CESAM meaning.



It is also important to point out that the previous architectural visions definitions are consistent. In this matter, the key point is here only to be sure of the existence of functional models as defined above. This is however directly connected to the emergence postulate (see Section 0.2) that claims that the mere knowledge of the models of the components of a system and of their interaction laws is never sufficient to model the system that results from their integration. This fact explains why any system always has purely functional models, whose core fundamental role is to express the emerging behaviors<sup>79</sup> that one will never be able to capture and read within constructional models<sup>80</sup>.

### **Various Perceptions on a System: the Concorde Case**

The Concorde supersonic aircraft is a typical example of various – and often contradictory – perceptions on the same system.

From an engineering perspective, Concorde was indeed an outstanding success. Most British and French engineers are usually very proud of this great technological achievement.

But from a business and societal perspective, it was a total disaster. The supersonic aircraft was typically not able offering a real service to the end-customer. Concorde was indeed the fastest, but also the most expensive aircraft, with very few destinations offered (only Paris-New York & Paris-Rio de Janeiro) and at the end, a quality/price ratio which was strongly non-optimal. Due to chemical and noise pollution, it was also not an environmental-friendly aircraft, which blocked it during a long time to get the landing authorization in New York city as a consequence of the opposition of many neighbors' protection organizations.

When possible, which is not always the case as taught by the final failure of the Concorde story, the role of the systems architect is to find the best architectural balance between all these different competing points of views

### **Case study 3 – Various perceptions on a system: the Concorde case**

We can thus now understand why it is necessary to have three different types of models in order to model in practice a real system: the operational vision indeed captures the external viewpoint while functional and constructional views do capture the internal perspective, by modeling respectively firstly the emergent behaviors and secondly the concrete constitution of the considered system.

As we will see more in details in the sequel, architectural visions are of course key for in a systems architecting perspective: the first job of any systems architect will indeed always be to classify the

---

<sup>79</sup> Unfortunately, this is not a common understanding of the functional vision. When doing “functional analysis”, most of people are indeed just modeling the functions of the components of a given system, which is not functional analysis in our meaning since this activity shall focus on describing functions at system level, and not at component level.

<sup>80</sup> To understand this phenomenon, consider the example of a car whose constituent (high-level) systemic components are the car body, powertrain, binnacle, chassis and embedded electronics. The interaction of these components typically allows for features like “obstacle detection” which requires the cooperation of a radar (placed in the car body), an embedded software (within embedded electronics), a LED (positioned in the passenger binnacle), and possibly chassis or powertrain if one wants to act on the brakes and / or to reduce engine torque when an obstacle is too close to a car. Such a “transverse” feature is clearly difficult to catch in a purely constructional car model when one will see the flows exchanged between the various involved components of the vehicle without being able to account for their overall logic. Only a functional model at car level will allow capturing the semantics of such a transverse function.

modeling information according to our three architectural visions<sup>81</sup>, so as to obtain homogeneous system models each of them capturing a well-defined view.

## 2.2.2 Architectural Visions Overview

Let us thus now present more in details the three different – operational, functional & constructional – architectural visions that we introduced in the last section.

### 2.2.2.1 Operational Vision

The *operational vision* provides “black box” models of a given system where one does not describe the system of interest, but rather its interactions and its interfaces with its environment. Its core motivation is to understand in that way the motivation – that is to say the “Why” – of the system.

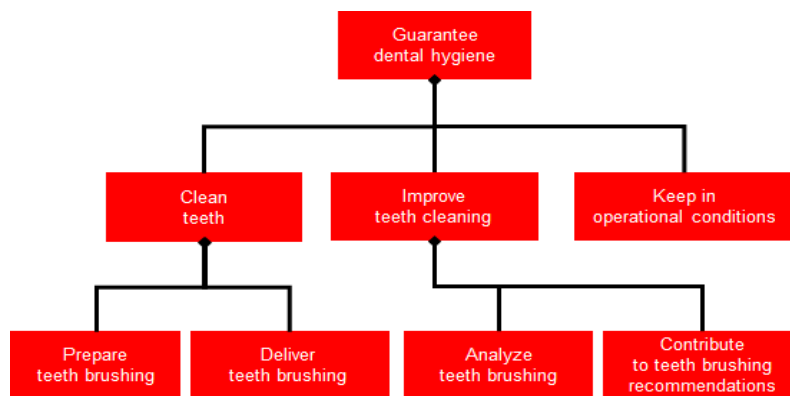


Figure 21 – Operational vision – Mission Breakdown Structure (MBS) of an electronic toothbrush

In this matter, the key point to understand is that an operational analysis manipulates concepts at environment level, which are mixing – by definition – both the system of interest and its external systems. The operational concept of “mission of a system” is a typical example of such a situation. Formally speaking, a *mission* for a given system  $S$  can indeed be defined as a function of the environment of reference  $\text{Env}(S)$  of that system<sup>82</sup>. When one analyzes for instance the function “To guarantee dental hygiene”<sup>83</sup>, whose functional behavior consists in transforming dirty teeth into healthy teeth and/or maintaining teeth in an healthy state, one can see that a toothbrush can clearly not achieve alone this feature, which also requires at least an end-user, toothpaste and water plus may be a dentist. Such function can thus only be allocated to the environment of the toothbrush, considered as a system in the meaning of Section 2.1, and not to the toothbrush alone. In other words, “To guarantee dental hygiene” is hence not a function of the electronic toothbrush, but a function of its mission, which means that it shall be interpreted as a mission – and again not a function<sup>84</sup> – of the toothbrush according to our above definition.

<sup>81</sup> That will be segregated more precisely according to a systemic analysis grid and organized in different abstraction levels, as we will see further in this pocket guide.

<sup>82</sup> In other terms,  $\text{Mission}(S) \equiv \text{Function}(\text{Env}(S))$  for every system  $S$ .

<sup>83</sup> Due to the functional nature of a mission, we do recommend to name it as a verb in infinitive form (cf. next subsection).

<sup>84</sup> The role of functional analysis is in particular to extract, strictly speaking, the functions of a system of interest which are hidden within its missions, that is to say the internal behaviors of the considered system that are only involving the system and nothing else around it (and thus also only partially contributing to the missions). For an electronic toothbrush, we may for instance analyze that the toothbrush is only achieving the function “To brush teeth”, which basically only provide brushing forces, as a partial contribution to the mission “To guarantee dental hygiene”. To illustrate this subtle distinction,

The operational vision relies on other operational concepts such as life cycle, operational contexts, operational scenarios or operational objects (cf. Section 2.4 for more details). All these concepts may also be managed at different levels of abstraction / grain. Figure 21 shows for instance the *Mission Breakdown Structure* (MBS) of an electronic toothbrush where its core missions are put in a hierarchy according to the fact that a high level mission needs the lower levels missions to be achieved.

The operational vision can also be seen as a natural interface between engineering and non-technical people. Typical examples of operational models are indeed for instance development, assembling or maintenance models (that specify how a product system will be managed by the associated design, manufacturing or support systems), but also marketing or usage models (that describe how a product system will be seen by the market or used the end-users) and business models (which explain how the constructing company will earn money with a product system). In this matter, the role of the operational vision is to express the information contained in these different business models within a language that can be understood by the system designers and used in the development process.

### 2.2.2.2 Functional Vision

The *functional vision* provides “grey box” models of a given system of interest where one begins to apprehend the inside of the system, but only in terms of input/output abstract<sup>85</sup> behaviors and not of concrete implementation choices, in order to begin understanding more deeply what does the system, without however knowing at this point how it is concretely structured. Its core motivation is to elicit in that way the behavior – that is to say the “What” – of the system.

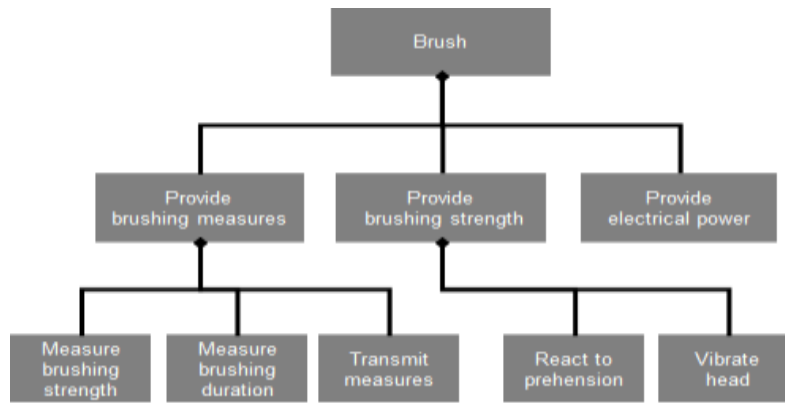
The core notion of the functional vision is of course the notion of “function of a system”, which refers to an input/output behavior of the considered system. In other terms, a *function* associated with a given system models a transformation process – which can be achieved by physical, software or even “humanware” resources – that transforms a given series of inputs into a given series of outputs. This explains why a common pattern to name a function is a verb followed by a complement, the generic patterns being typically “To do something” or “To transform inputs into outputs”. In any case, one shall always check when defining a function whether it expresses such a transformational behavior.

Contrarily to the operational vision, all functional concepts – such as functional modes, functional scenarios or functional objects (see Section 2.4 for more details) – are now uniquely referring to the system of interest, without involving any external system. All these concepts can again be managed at different levels of abstraction / grain. Figure 22 shows for instance the *Functional Breakdown Structure* (FBS) of an electronic toothbrush, where its main functions are put in a hierarchy according to the fact that a high level function needs the lower levels functions to be achieved (i.e. the “algorithm” of the high level function involves the lower functions as sub-routines).

---

we may take the other classical example of the cigarette system. Most of people will probably say that the core function of a cigarette is “To smoke”, but again it is easy to see that one cannot smoke without at least a smoker, a source of fire and air, plus probably also an ashtray. “To smoke” can hence be only allocated to the environment of a cigarette and it shall be interpreted as a mission – and not a function – of a cigarette. One may indeed understand that “To smoke” is a complex protocol requiring first a smoker, a cigarette and a source of fire to provide its own function “To deliver fire”, passing then through a loop where smoker “inspires pure air”, cigarette “propagates fire” & “delivers tar” and smoker “expires dirty air” up to arriving to the cigarette consumption, with a final request by the smoker of the ashtray function “To keep ashes” applied to the burned cigarette. This analysis shows that the underlying cigarette functions – or in other terms the intrinsic behaviors of a cigarette – are here “To propagate fire” and “To deliver tar”.

<sup>85</sup> That is to say independent of any technological implementation.



**Figure 22 – Functional vision – Functional Breakdown Structure (FBS) of an electronic toothbrush**

We may now point out that a key difficulty of functional analysis is the identification of *transverse functions* that is to say of functions that cannot be directly allocated to a single component of a given system. Such functions are indeed capturing the emergent behaviors resulting from the cooperation between the different components of a system, which by definition cannot be easily observed at constructional level. It is therefore always important to identify these functions in order to master the integration process since these functions are also telling us where different teams in charge of different components shall work collaboratively<sup>86</sup>. Within a functional breakdown structure, one may thus normally always find both component functions and transverse functions. Unfortunately most of engineers are often forgetting the transverse functions in their analyses, which leads them to lose the most important value of a complete functional analysis in a systems architecting perspective.

Another key point is that the functional vision is fundamental in systems architecting since it provides the deep invariants of any system. Any communication network will achieve for instance always the same basic functions such as “To receive messages”, “To route messages” or “To deliver messages”, independently of its implementation technology that may either purely manual (think to your snail mail operator) or based on many different techniques (Hertzian waves, twisted cables, copper wires, optical fiber, etc.). In a totally different direction, consider a State as an organizational system: one may observe that it always relies on the core function “To collect taxes”, consisting in taking money from the citizen pockets and bringing it in the State ones, which is basically invariant among time even if the tax collecting mechanisms evolve a lot from Roman antiquity up to our modern societies. In other words, *technology changes but functional architecture remains*. As a natural consequence, functional architecture always provides a robust basis for architecting a system. It indeed allows the systems architect to reason on a system independently of technology and thus to define, analyze and evaluate different implementation options for a given functional architecture. Such an approach is key to choose the best solution, which cannot be done if one directly works at constructional level where one will be glued in a given technical choice, without possibility of making another.

<sup>86</sup> We already provided an illustration on that situation in footnote 80 to which the reader may first refer. Another similar example is the thrust reversing function on an aircraft: this function, which reverts the air flow passing in an aircraft engine to decrease the speed of the aircraft when on ground, is provided by the cooperation of a cylinder that pushes a trap both located in the nacelle, the engine itself and a critical embedded software that coordinates the involved nacelle and engine components when thrust reversing operates. Such a function is typically transverse since distributed on several hardware & software components which are located moreover provided by different suppliers (typically one for the nacelle, one for the engine, one for the embedded system): identifying the function and putting it under control in the aircraft development project is thus totally key to ensure the success of its integration.

Good systems architectures are also based on *functional segregation principles*. This simply means that some key functional interfaces must be strictly respected at constructional level<sup>87</sup>. This gives rise to layered architectures where components are clustered in different independent layers connected by functional interfaces. Typical classical examples of such architectures are computer, mobile phone or communication network architectures that are organized in different independent layers, starting from the physical layer up to arriving to service layers (see for instance [90] for more details). One must thus for instance be able on one hand to change a signal processing protocol within the physical layer without any impact on the service layer and on the other hand to implement a new service or a service evolution in the service layer without any impact on the physical layer. Such a result is typically achieved by means of robust functional interfaces that shall also be stable among time, in order to absorb the technological evolutions that will naturally arrive in the life of any system<sup>88</sup>.

It is finally interesting to observe that the standard vocabulary used to discuss of the functional vision is traditionally different, depending whether the considered system is a technical or an organizational system. The term “function” is for instance usually reserved for technical systems (that may be of either hardware or software nature), when one rather uses the terms “process”, “activity” or “task” to express the same behavioral concept when dealing organizational systems. It shall however be clearly understood that processes, activities or tasks are in fact nothing else than functions of a given organizational system, considered at different levels of abstraction.

### 2.2.2.3 Constructional Vision

The *constructional vision* provides white box models of the system where one describes all concrete hardware, software and “humanware”<sup>89</sup> components of a system with their interactions. Its core motivation is to elicit in that way the concrete structure – that is to say the “How” – of the system. It is thus probably the most intuitive part of a systems architecture.

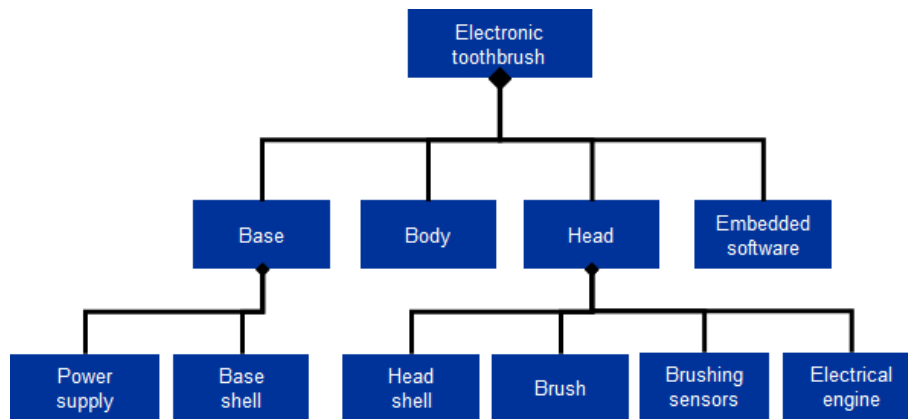


Figure 23 – Constructional vision – Product Breakdown Structure (PBS) of an electronic toothbrush

<sup>87</sup> In this matter, the role of the systems architect is to guarantee that such interfaces will never be violated in the design.

<sup>88</sup> Another motivation for such functional segregation is abstraction. It would indeed be basically impossible to develop a service if one would access directly to the physical layer of a computer system since the physical world is here usually highly non deterministic with many probabilistic phenomena that must be hidden to a service developer.

<sup>89</sup> Remember that men can be part of systems with either strong organizational dimensions such for instance as information systems, or when a human stakeholder plays such a key role (e.g. pilot, driver, operator, etc.) that it may be important to include him/her in the design, considering then an operated system rather than the underlying technical system alone.

The core notion of the constructional vision is of course the notion of “component of a system”, that refers to a concrete part of the considered system. In other terms, a *component* associated with a given system models a physical, software or even “humanware” resource that belongs to the system. In other word, each atom of a system shall belong to one and only one of its components. A common pattern to denote a component is thus just to use its usual technical or business name.

Exactly as in the functional vision, all constructional concepts – such as configuration, constructional scenarios or constructional objects (see Section 2.4 for more details) – are again uniquely referring to the system of interest, without involving any external system. All these concepts can be managed at different levels of abstraction / grain. Figure 22 shows for instance the *Product Breakdown Structure* (PBS) of an electronic toothbrush, where its components are put in a hierarchy according to the fact that a high level component results from the integration of the lower level components.

Note finally that the term “architecture” usually only refers to the constructional architecture of a system. One shall thus be aware of the fact that we will use this term in a much broader acceptance through our entire pocket guide, especially when speaking of systems architecture which does refer to the union of all architectural visions for a given system as introduced above.

### 2.2.3 Relationships between the three Architectural Visions

Last but not least, it is also important to point out the network of relationships existing between the three architectural visions, since they are at the heart of the systems architecting process. It is in particular especially important to maintain these relationships during the different design phases, which is difficult due to the “highly iterative & recursive nature” of systems architecting [66].

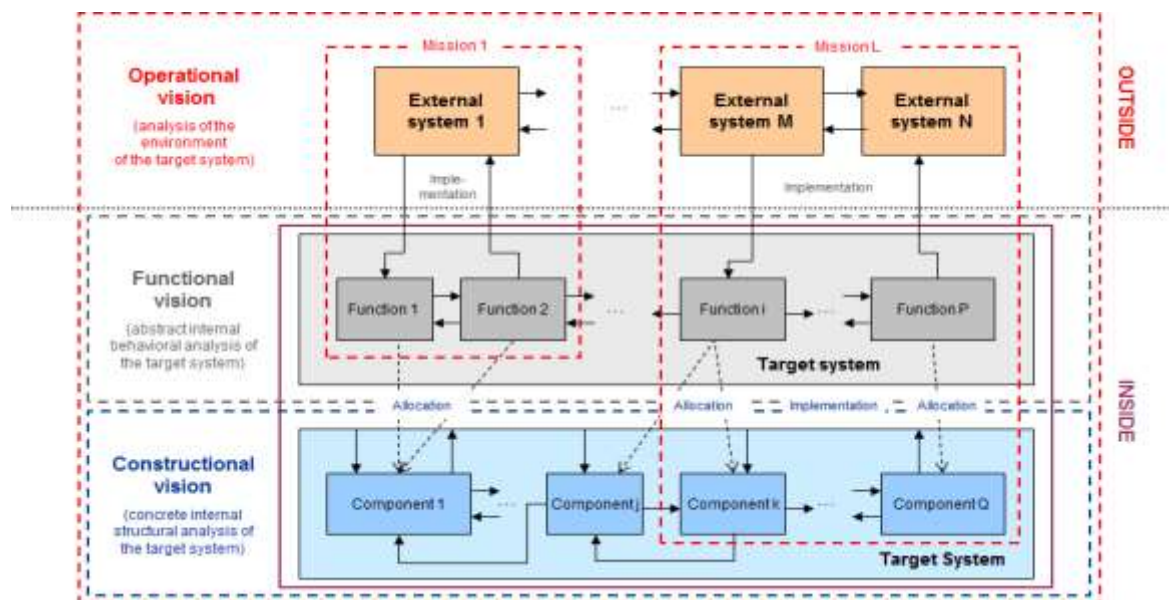


Figure 24 – Relationships between the three architectural visions

Figure 24 shows the generic relationships between the architectural visions as explained below.

- The operational vision connects first with the two other visions due to the fact that missions are naturally implemented by functions, but also components. Another way to make this

connection is to observe that all external flows between the different external systems and the system of interest, as provided by the operational vision, must be internally captured or produced (depending the external flows are input or output flows from the internal point of view of the considered system) by functions of the system of interest<sup>90</sup>.

- The functional vision connects back in the same way with the operational vision and forth with the constructional vision due to the fact that each abstract function must be concretely allocated to / implemented by some set of constructional components.
- The constructional vision connects then back to the two other visions according to the implementation and allocation relationships that we just pointed out.

Note in particular that **one should not think**<sup>91</sup> that operational artefacts do only allocate to functional artefacts, which on their side do only allocate to constructional artefacts. Such a vision would indeed be dramatically false. Operational, functional and constructional visions shall indeed be analyzed as a circle of three interdependent visions. Figure 25 illustrates this situation.

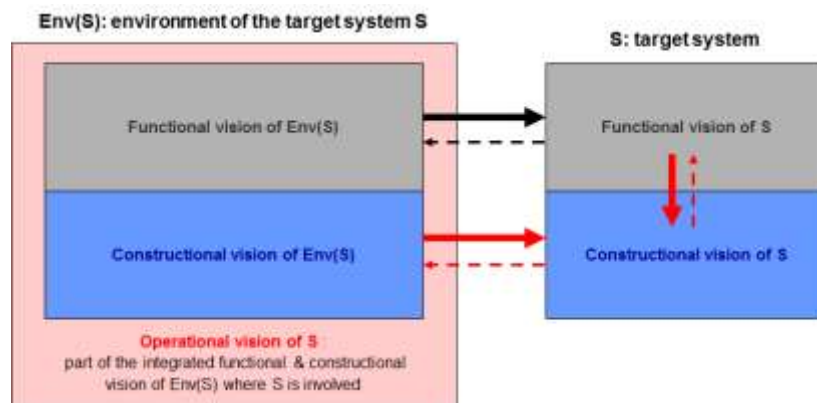


Figure 25 – Relationships existing between the three architectural visions

One must first understand that the operational vision is nothing else than a mixed functional and constructional description of the part of the environment of the system of interest which involves this last system. As an immediate consequence, the functional (resp. constructional) dimension of the environment  $Env(S)$  of a given system  $S$  naturally maps with the functional (resp. constructional) dimension of  $S$ . Such a property implies therefore that operational architecture is connected both to functional and constructional architectures of a given system. As a matter of fact, the geometry of a given system's environment – which is one of its typical constructional properties – maps for instance directly with the geometry of the considered system, without any connection with functions<sup>92</sup>. The same situation also holds for most of the physical properties of the environment.

On the other hand, one must also notice that there may be feedbacks from the constructional vision onto the functional vision and/or the operational vision and/or from the functional vision onto the

<sup>90</sup> In other terms, it is sufficient to continue each external flow, as identified in the operational vision, within the system to get the first functions of the system of interest. Functional analysis will then continue internally the same kind of analysis up to identifying exhaustively all functions, which can be checked by a functional synthesis proving that all identified functions are forming a coherent functional network.

<sup>91</sup> Which is unfortunately a common mistake ...

<sup>92</sup> The shape of my body typically implies the shape of a chair, without requiring any functional analysis...



operational vision. The choice of a specific technology at constructional level may indeed typically induce functions that were not directly requested. Deciding to implement a given service through an automated device creates for instance immediately the “To distribute electricity” function. In the same way, the choice of a specific function at functional level may allow new services that were initially not designed. As another example, just remember that nobody could imagine the creation of an entire new world of new services thanks to the apparently simple Internet functionality<sup>93</sup>!

## 2.2.4 Organization of a System Model

We are now in position to derive the first consequences of the CESAM framework on the structure of a system model. We are indeed now aware of two dimensions of any system, the first one being provided by the three architectural visions used to model a system, the second one being simply given by the abstraction / grain level on which a given system may be analyzed. On this last point, we shall just recall that any integrated system can be analyzed on the different levels of its integration hierarchy that is to say at system, sub-system, sub-sub-system, etc. levels.

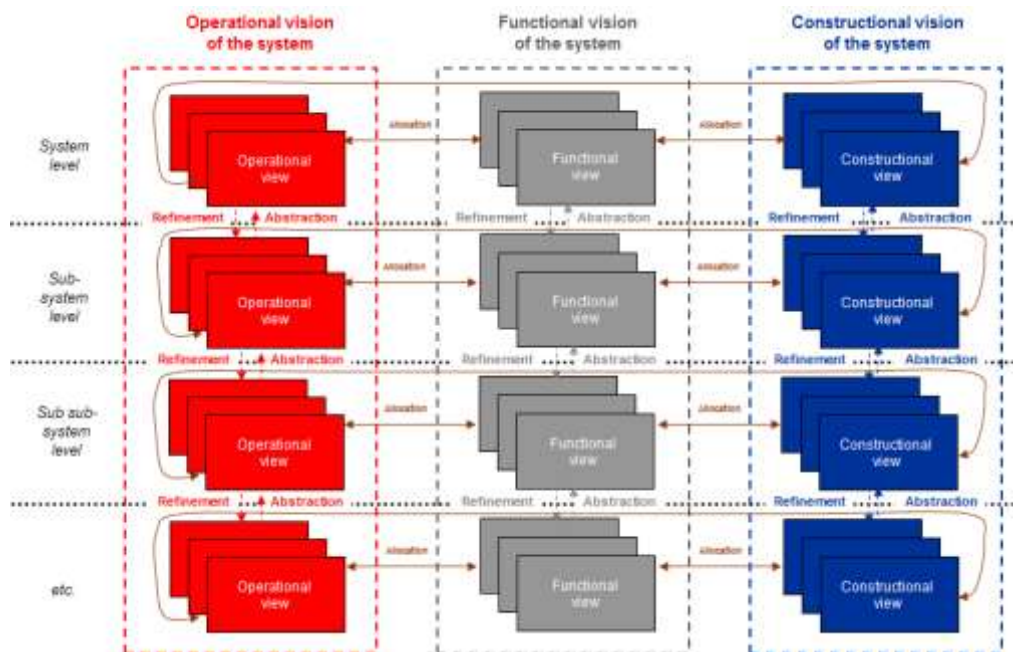


Figure 26 – Organization of a system model

Hence any system model can be naturally organized in a matrix way where the different system views are classified according to their architectural vision and the level of analysis within the system integration hierarchy, as depicted in Figure 26. Note that the horizontal relation between these views is allocation or implementation as explained in the previous sub-section, when the vertical relation is refinement when going from a high level view to a lower level view and abstraction when doing the converse. Refinement means here providing more details with respect to a given architectural view, when abstraction stands for a not (too much) destructive idealization<sup>94</sup> of a series of views, where

<sup>93</sup> Which functionally speaking is nothing else that allowing the exchange between different computers.

<sup>94</sup> An abstraction/refinement mechanism is formally provided [25] by a pair  $(\alpha, \rho)$  of applications between sets of so-called concrete objects and sets of so-called abstract objects, where *abstraction application*  $\alpha$  maps each set of concrete objects into a set of abstract objects and *refinement application*  $\rho$  maps each set of abstract objects into a set of concrete objects.



one will reason in terms of clusters from a lower level perspective, thus losing the details for getting the big picture on a given architectural topic<sup>95</sup>. On one hand, refinement is clearly the right tool when one wants to precisely analyze a problem. On the other hand, abstraction<sup>96</sup> is crucial for being able to define an architectural strategy without being lost in an ocean of details.

## 2.3 CESAM Systems Architecture Pyramid

### 2.3.1 The three Key Questions to Ask

As discussed in the previous section, any system can be analyzed from an operational, functional and constructional perspective. In order to achieve such analyzes in practice, one must simply remember that one shall just ask three simple questions<sup>97</sup> to cover these different architectural visions:

- *Key operational question: WHY does the system exist?*<sup>98</sup>
- *Key functional question: WHAT is doing the system?*<sup>99</sup>
- *Key constructional question: HOW is formed the system?*<sup>100</sup>

One usually summarizes these different questions in the *CESAM Systems Architecture Pyramid*, which is a simple graphical way to represent a system, presented in Figure 27 below. Such a pyramidal representation intends just to recall that details – and thus clarification of the system model – will permanently increase when moving from the operational to the constructional vision.

Note that the key point is of course the order in which these three core questions are asked during a system design process. The systems architecting consists indeed in following the previous order, i.e. passing from the first question (“Why?”), to the second one (“What?”) up to the third one (“How?”), in that exact order. Be however careful not to manage successively, that is to say one after the other, these three types of analyses which should largely overlap in practice. At some point, it is indeed just impossible to reason operationally without any vision of the concrete solution that will answer to the

---

These applications are then called an abstraction/refinement pair if and only if  $\alpha(\rho(A)) \subseteq A$  for each abstract set  $A$  (refining an abstract set and then re-abstrating the result cannot enlarge the initial abstraction) and  $C \subseteq \rho(\alpha(C))$  for each concrete set  $C$  (abstracting a concrete set and then refining the result cannot reduce the initial concrete scope).

<sup>95</sup> Abstraction & refinement are core mechanisms for systems architects, especially when creating architectural hierarchies. A quite frequent problem in architecture is indeed the excessive number of objects generated by a step of an architectural analysis. In order to handle them effectively and to achieve their real global understanding, we typically have to cluster and to synthesize them into abstract objects. This abstraction activity can be achieved by partitioning the objects in clusters of “similar” weakly inter-dependent objects, then clarifying systematically the key characteristics (goal, function, feature, etc.) of each group and naming each group consequently. Such a process will naturally lead to architectural hierarchies such as Mission, Functional or Product Breakdown Structures as introduced in the previous subsections.

<sup>96</sup> As a matter of fact, one observes in practice that abstraction is not at all an easy activity. Most of people are in particular not able to manipulate efficiently this mechanism. The key difficulty is indeed to find the “good” abstractions of a given problem, that is to say a good balance between too abstract and too detailed views. The key point is here to be able to manipulate coarse grain views, with which one can reason more easily and thus take the good strategic decisions, but that can also be refined in fine grain views. This requires the abstract views to be holistic in order to capture all dimensions of a given problem. What happens unfortunately often in real life is that one creates too simple abstractions to be useful!

<sup>97</sup> These questions shall just be seen as a mnemonic trick to remember the scope of each architectural view. Modeling of each view is indeed much more complicated than that.

<sup>98</sup> Or more precisely what are the services provided by the system to its environment?

<sup>99</sup> Or equivalently what are the behaviors/functions of the system?

<sup>100</sup> Or in other terms, what are the concrete resources that form the system?

operational architecture<sup>101</sup>. This typically leads to manage coarse grain functional and constructional analyses during the operational analysis. In the same way, it is not possible to reason functionally without any idea of the components that may implement the functional architecture. This obliges to manage middle grain constructional analyses during the functional analysis. As a consequence, good systems architecting practice is clearly to manage in parallel the three architectural analyses at the same time, but not at the same grain of analysis.

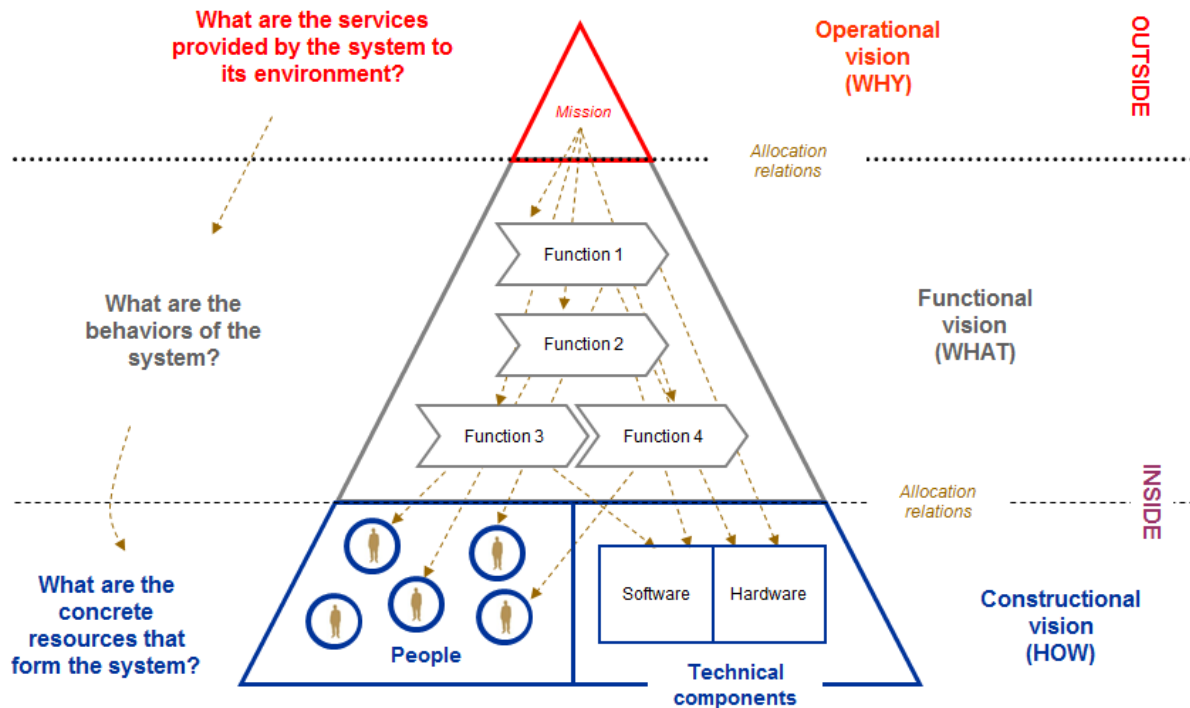


Figure 27 – The CESAM systems architecture pyramid

Organizing the systems architecting process in that way will allow passing from technical-oriented to value-oriented system design strategies. In most of classical system design strategies, the technical components are indeed usually the starting point and it is only at the end of the development phase that one begins to look how the developed system fits to its stakeholders needs. Such an approach is a *product-push* strategy from a marketing perspective and it may work well as soon as one is making incremental improvements on existing products in stable markets.

Unfortunately industry must nowadays more & more manage many technological ruptures within unstable environments. In that case, just pushing new products will have a high probability to fail. To increase success, one must thus invert the design logic in order to put stakeholders and their needs as a starting point to the product development. Systems architecting shall thus just be seen as the key methodological tool to implement such a *need-pull* strategy.

<sup>101</sup> An operational architecture that cannot be implemented in a concrete solution has a name: a science fiction movie. Such movies are indeed typically showing us use cases of technology that are just not concretely feasible. Think to Start Trek's hyper-propulsion or teleporter. The movie can be seen as an operational proof of concept of such technology, from which it is probably possible to make a coarse grain functional analysis. Unfortunately, we will never be able to achieve a detailed functional analysis due to the fact that no constructional architecture does exist in response to the operational architecture.

### 2.3.2 The Last Question that shall not be Forgotten

The three previous questions are however not the only ones that one should ask in the context of complex systems design & development. The last fourth key question, unfortunately often forgotten in real systems contexts, refers to the product/project duality as introduced in section 1.1. It simply consists in asking which person – in other terms “Who?” – is the project counterpart of the different product elements described in the three architectural visions of the product, that is to say:

- WHO owns each architectural element of the system?

This question can then be declined according to the three architectural visions as follows:

- *Operational perspective*: who are the stakeholders of the system?
- *Functional perspective*: who is in charge of the functions of the system?
- *Constructional perspective*: who is responsible of the components of the system?

Asking these different questions is clearly fundamental from a systems architecting perspective. First it is just impossible to capture the right needs without deeply interacting with the stakeholders of the system of interest, which requires identifying them as early as possible, thus leading to question 4.1. Secondly, we may recall that the robustness of a system design is directly correlated to the fact that all transverse functions are managed, i.e. under the responsibility of somebody within the project, which immediately motivates question 4.2. Third one cannot imagine influencing the design of a system without involving all functional & constructional responsables, which again obliges knowing them well, which can be achieved through questions 4.2 and 4.3. In this matter, we shall also recall that the role of a systems architect is usually to manage a complex socio-dynamics implying all these different actors, which cannot be done without perfectly understanding their personal motivations, their synergies and their antagonisms with respect to a given system design & development project. We are here again in the “who” sphere and not in a technical issue.

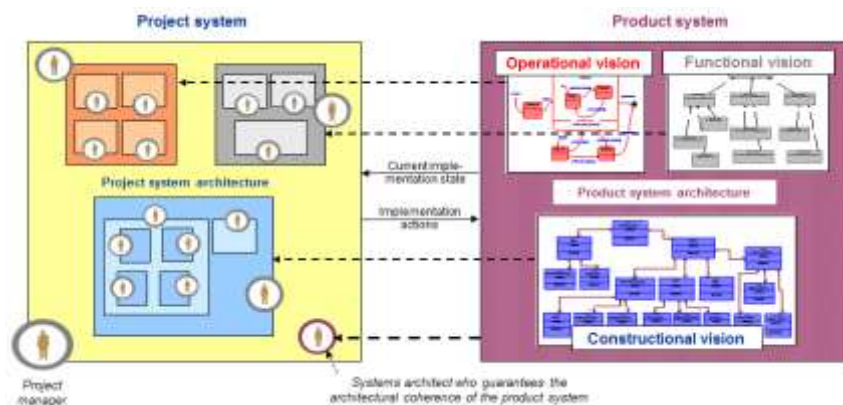


Figure 28 – Alignment of the project system architecture with the product system architecture

Note finally that the “WHO” question is also crucial in the construction of the project organization. A good project architecture indeed results from the mapping of all architectural elements of a given product system into the project system, that is to say onto people, where they shall be put under a single responsibility. This project/product alignment principle is indeed crucial to be sure that all operational, functional and constructional elements of a product system are taken in charge by

somebody within the project system, which obviously a sine qua non condition for the completeness of any engineering analysis, but also to guarantee that no product architectural perimeter has two project responsables which would mechanically lead to many engineering conflicts<sup>102</sup>. Figure 28 illustrates this alignment principle on the electronic toothbrush example: the boxes appearing on the project system side are for instance modeling project teams that correspond there to the first levels of decompositions of the different views which are provided on the product system side.

## 2.4 More Systems Architecture Dimensions

Architectural visions are however not the only architectural dimensions of a system. We shall now introduce a number of new dimensions that can be used to refine each architectural vision.

### 2.4.1 Descriptions versus Expected Properties

As already discussed in section 0.1, one must now recall that there exists two complementary ways of specifying a system. The first one refers to *descriptions*: in this specification mode, one explicitly<sup>103</sup> describes the behavior and structure, either of the system of interest (if one is reasoning functionally or constructionally) or of its environment (if one reasons operationally).

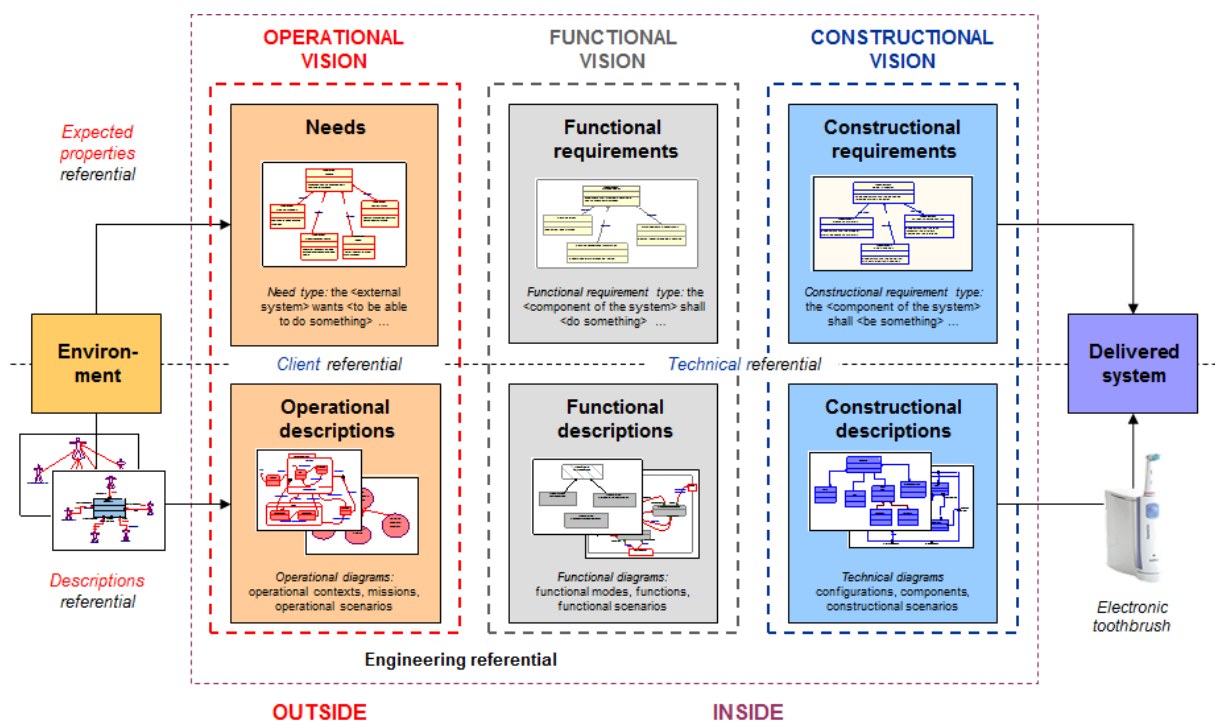


Figure 29 – Descriptions versus expected properties

The second way deals with *expected properties*: one is now not explicitly describing a system, but rather stating the (operational, functional and constructional) properties, expected/intended<sup>104</sup> to be satisfied by the system. Note these expected properties are usually called *requirements* in systems

<sup>102</sup> Such situations unfortunately often exist in practice, with sometimes with up to 10 responsables for the same product system architectural perimeter!

<sup>103</sup> This is why descriptions are considered as specifications *in extension*.

<sup>104</sup> This is why expected properties are considered as specifications *in intention*.

engineering (see section 0.1 for more details). This gives rise to the six different – but altogether exhaustive<sup>105</sup> – specification modes that are presented in the below Figure 29, that is to say:

- For descriptions: operational descriptions, functional descriptions, constructional description,
- For expected properties: needs<sup>106</sup>, functional requirements, constructional requirements.

As explained in section 0.1, one can equivalently (from a purely theoretical point of view) completely specify any system by using either descriptions, or expected properties. However these two modes of specification are absolutely not equivalent from an engineering effort perspective (see again section 0.1). On one hand, descriptions are indeed well adapted to define and to synthesize the behavioral & structural dimensions of a system. On the other hand, performances of a system are typical expected properties. But the converse is totally false. As a good practice, an efficient and optimal – in terms of engineering time spent – system specification shall mix descriptions (reserved for defining behavioral & structural elements and their dynamics) and expected properties (reserved for performance). This trick allows drastically reducing the requirements volume in a specification file, thus improving its readability, since descriptions are usually encoded by a huge amount of requirements.

## 2.4.2 Descriptions

Descriptions can be separated in four different views, each of them modeling a different dimension of a system. States are first modeling time. Static elements are then depicting the core objects of each architectural vision when dynamics are describing their temporal behavior. Flows are finally consolidating the exchanges involved in these dynamics. One should also note that all these system views are both exhaustive – they allow modeling completely a system – and non-redundant – each view provides a specific perspective which is not covered by the other views – due to the foundations of our system architecting framework (see section 0.1).

### 2.4.2.1 States

A *state*  $T$  associated with a given system  $S$  is modeling a period of time, that is to say a set consisting of one or more intervals of time, where the system  $S$  can be analyzed in a homogeneous way from the perspective of a given architectural vision. A state can be usually specified by its initiation and termination events<sup>107</sup>, which are both modeling phenomenon occurring instantaneously, i.e. at a specific moment – not interval – of time. As one may imagine, the initiation (resp. termination) events do correspond to moments of time – the same type of event can indeed occur at different moments of time – where the period of time modeled by  $T$  begins (resp. ends).

States are used to model time. In each architectural vision, a key temporal analysis consists indeed in decomposing in different states the time line of a system from birth to death. In such analyses, one can then model the usual temporal behavior of a system as a succession of states in which lies the system, one after the other. Think for instance to a normal day of a human person which begins in the “Sleeping” state, passing then to the “Morning dress” and “Breakfast” states, before arriving to

---

<sup>105</sup> This key property is ensured by the mathematical foundations of the CESAM framework (see Chapter 0).

<sup>106</sup> We will use here the term “need” instead of “operational requirement”, even if they are equivalent. We indeed prefer to reserve the term “requirement” for functional & constructional uses in order to separate strictly the domain of the question (expressed with needs) and the domain of the solution (stated with requirements).

<sup>107</sup>  $T = \cup [t1, t2]$  for all moments of time  $t1$  and  $t2$  such that an initiation (resp. termination) event occurs at  $t1$  (resp.  $t2$ ).

the “Transportation” and “Working” states, passing a new time in the “Transportation” state, then in the “Relaxing”, “Dining” and “TV listening” states, before going back again to the “Sleeping” state. We will discuss more precisely in Chapter 4 that kind of analysis for systems, based on states.

There are therefore logically three different types of states for any given system  $S$ , depending on the considered architectural vision, which are defined as follows:

- Operational states are called *operational contexts*: an operational context for  $S$  is a period of time  $OC(S)$  characterized by the fact that external interactions of  $S$  during  $OC(S)$  do only involve a certain fixed set of stakeholders or external systems of its environment.
- Functional states are called *functional modes*: a functional mode for  $S$  is a period of time  $FM(S)$  which is characterized by the fact that the behavior of  $S$  during  $FM(S)$  can be modeled by only using a certain fixed set of system functions.
- Constructional states are called (technical) *configurations*: a configuration for  $S$  is a period of time  $TC(S)$  – usually identified with the involved components – characterized by the fact that the structure of  $S$  during  $TC(S)$  does only consist of a certain fixed set of system components.

In other terms, one changes of operational context, functional mode or configuration if and only if a new stakeholder, function or component appears within the life of a given system. Passing from the “Rainy” to the “Sunny” operational context typically means that the Rain stakeholder disappeared and was replaced by the Sun stakeholder. Replacing stakeholder by function or component would then lead to similar examples for the two other types of states that we introduced.

States types	Toothbrush states	Initiation event	Termination event	Characteristic set of architectural elements
<i>Operational contexts</i>	Bathroom	Entered in bathroom	Taken out of bathroom	Bathroom, electrical distribution
	Teeth brushing	Start brushing	End brushing	Bathroom, end-user toothpaste, water
	Reparation	Failure detection	Back in bathroom	Communication device, end-user, repairer
<i>Functional modes</i>	Idle	End Working	Start working	Provide (only) mechanical reaction
	Active	Start working	End working	Provide electrical power, brushing forces & measures
	Passive	Failure	Failure fixed	None (all functions are typically broken)
<i>Configurations</i>	Children	Child-head on	Child-head off	Children-dedicated brushing head
	Adult	Adult-head on	Adult-head off	Adult-dedicated brushing head
	Broken	Component crash	Component replacement	Some non-core components disappeared

**Table 2 – Examples of states for an electronic toothbrush**

Table 2 illustrates the notion of states with some examples for an electronic toothbrush, where we explicated the set of architectural (static) elements characterizing each state.

One can see on these examples that there is no one-to-one correspondence between these different states. The toothbrush can indeed typically be in the “Bathroom” operational context and in either an “Active” or a “Passive” functional mode (in that last case, it would mean that the toothbrush is broken, but that the user did not noticed it) and in either a “Children” or an “Adult” configuration. In the same way, the toothbrush can be in the “Active” functional mode, but in either a “Bathroom” or a “Reparation” operational context (this last case corresponds to the situation where the toothbrush was repaired in the reparation workshop) and in lots of different configurations. The toothbrush may finally also be in a given configuration, but in various operational contexts or functional modes as one can easily check on the previous Table 2. These examples do thus show that each type of state may be allocated with many other types of states without any simple relationship in this matter.

Let us end by providing the standard representation of states – in most modeling languages – which are usually modeled by means of oval shapes, as one can see in the below Figure 30.

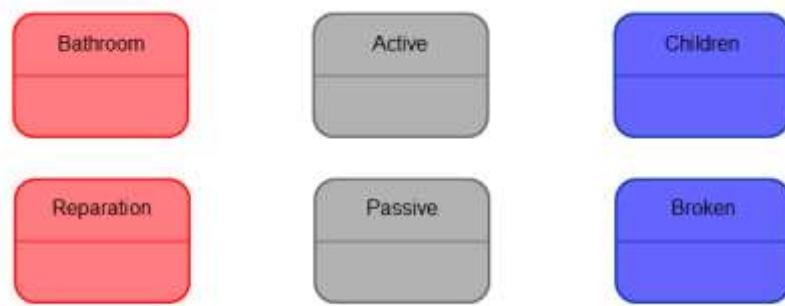


Figure 30 – Standard representations of states

#### 2.4.2.2 Static Elements

A *static element* with respect to a given system  $S$  refers to an input/output mechanism associated with  $S$  from the perspective of a certain architectural vision. We are using the term “static element” to emphasize that this new system description does not focus on the temporal dynamic (see next sub-section for that other point) of the involved input/output mechanism, but provides just a non-temporized definition of such a mechanism without explicating its “algorithm”.

There are therefore logically three different types of static elements for a given system  $S$ , depending on the considered architectural vision, which are defined as follows:

- Operational static elements are called *missions*: a mission of  $S$  is an input/output behavior of the environment  $\text{Env}(S)$  of  $S$ , involving both  $S$  and other external systems.
- Functional static elements are called *functions*: a function of  $S$  is an abstract implementation-independent intrinsic input/output behavior of  $S$ , that is to say that only involves  $S$ .
- Constructional static elements are called *components*: a component of  $S$  formally refers to a concrete implementation-dependent intrinsic input/output behavior of  $S$ . A component of  $S$  is therefore naturally identified to a concrete part of  $S$ .



Missions shall never be mixed with functions or components, since they do not refer to the system of interest, but to its environment. Functions and components refer both to the system of interest, but in two different ways. Functions are indeed independent of any concrete implementation of the system of interest when components do always refer to its specific concrete implementation. There are thus two types of functions: on one hand, transverse functions that can only be implemented by using several components; on the other hand, unitary functions that can be implemented by using a single component (such functions are thus “simply” modeling the components behavior). Transverse functions are very important since they do model transverse system behaviors that, by definition, cannot be easily analyzed at constructional level. One may finally observe that the existence of such transversal (or equivalently emergent) behaviors is intrinsic to any system since it is directly the consequence of the emergence postulate (see Section 0.2).

Note also that the standard way for stating a mission or a function of a given system is to use the “To do something” pattern in both cases. The only difference lies in the subject associated with the verb that describes the mission or function. This subject shall consist in external systems or stakeholders in the case of a mission (“external systems cooperating with the system shall do something”) when it shall only be the system alone for a function (“the system shall do something”). On the other hand, components are usually stated using only names referring to concrete objects forming the system.

Table 3 illustrates the notion of static elements with some examples for an electronic toothbrush. We provided some inputs and outputs for each proposed static elements.

Static elements types	Toothbrush static elements	Inputs	Outputs
<i>Missions</i>	To clean teeth	Dirty Teeth	Cleaned teeth
	To improve teeth cleaning	Clean Teeth	Cleaner teeth
	To keep in operational conditions	Working toothbrush	Working toothbrush
<i>Functions</i>	To provide brushing strength	Grip forces LV electricity	Brushing forces
	To provide brushing measures	Raw measures LV electricity	Measurement data
	To provide electrical power	Medium voltage (MV) electricity	Low voltage (LV) electricity
<i>Components</i>	Body	Grip forces Structural forces	LV electricity
	Head	LV electricity	Brushing forces Structural forces
	Base	MV electricity Support forces	LV electricity Support forces

**Table 3 – Examples of static elements for an electronic toothbrush**

As already mentioned, static elements are related by *allocation relations*. Each function contributes for instance to one or more missions, which corresponds to the fact that a mission can be obtained by composing a function with some other input / output behavior (see the example given in note 84):

such a situation is then expressed by saying that this mission is allocated to the considered function. In the same way, a component can contribute to a mission and/or a function, which will be expressed by saying that such a mission or function is allocated to the considered component.

We may also provide the standard representations of the three different types of static elements – in most modeling languages – which are usually modeled by circles for missions, ovals for functions and boxes for components, as one can see in the below Figure 31. We also expressed on this last figure the different allocation relationships that may exist between these different static elements.

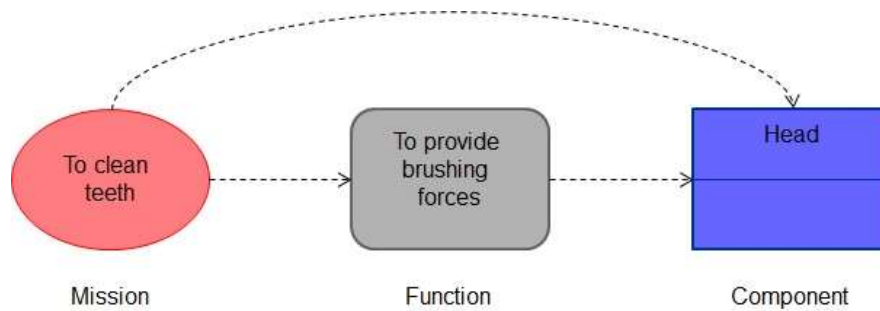


Figure 31 – Standard representations of static elements

Note finally that static elements can occur at different abstraction levels that do also correspond to different integration levels, resulting both in an abstraction and an integration hierarchy. Hence it is always crucial to be able to specify how different types of static elements are connected altogether by such relationships. The standard representations of these abstraction / integration relationships is provided by the below Figure 32, where we also put the associated allocation relations (beware that arrow ends, which express relationships, are squared when dealing with components).

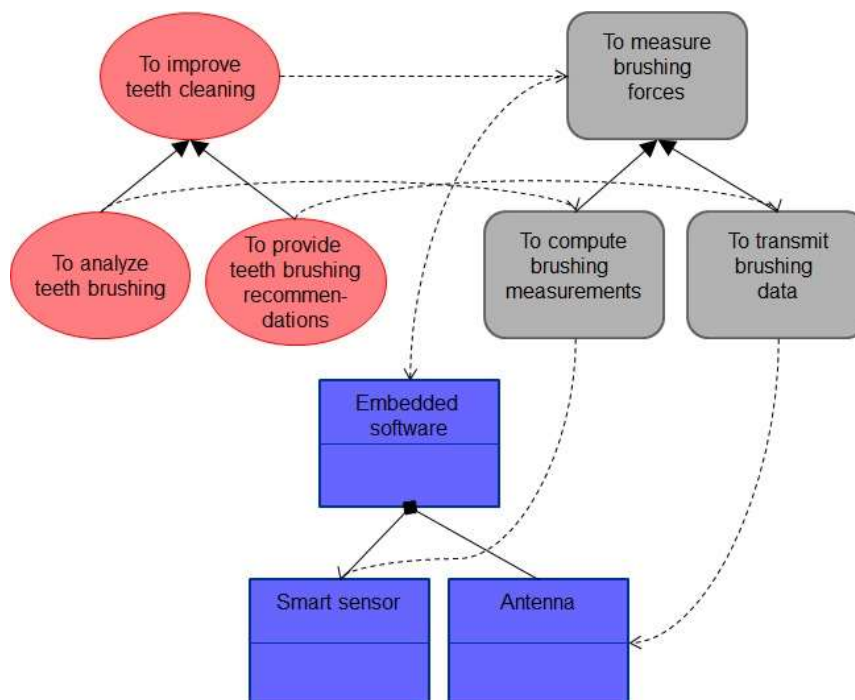


Figure 32 – Standard representations of integration relations between static elements

For the sake of completeness, one also needs to explicitly represent the full *integration mechanism* that relates the different static elements of lower level that are abstracted by a static element of higher level (see Definition 0.5). Figure 33 below shows an example – in the line of Definition 0.5 – of standard representation of such an integration mechanism – where oriented arrows labelled with an exchanged flow represent interfaces<sup>108</sup> – between different constructional components of the same level of abstraction (here the head, body and embedded system that are forming the “brush” part of an electronical toothbrush). Similar representations do also exist with functions or missions.

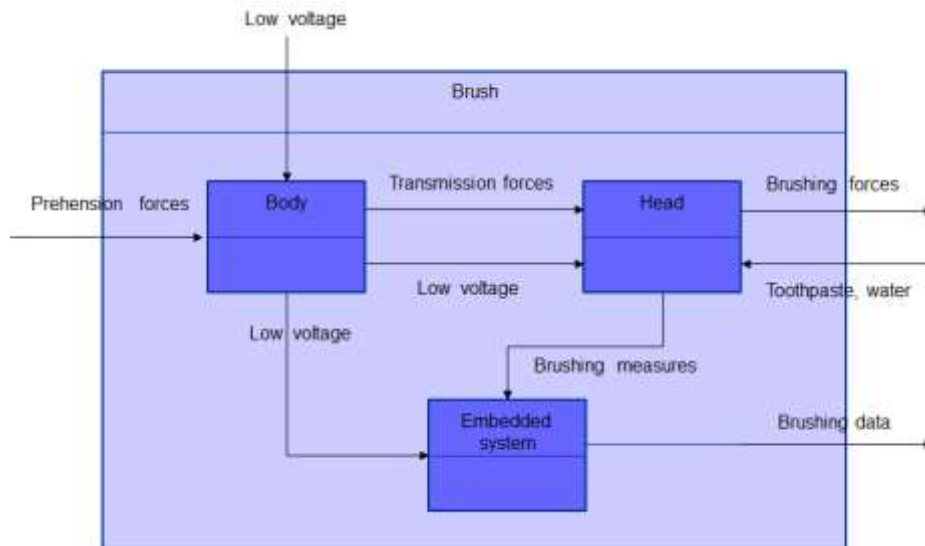


Figure 33 – Interfaces standard representation

### 2.4.2.3 Dynamics

The *dynamic* of a static element of a system  $S$  refers to its temporal behavior or equivalently to an algorithmic description of such a temporal behavior. Dynamics are completely crucial if one wants to precisely specify the behavior of any system or any system mission, function or component.

There are therefore logically three different types of dynamics for a given system  $S$ , depending on the considered architectural vision, which are defined as follows:

- Operational dynamics are called *operational scenarios*: an operational scenario of  $S$  is an algorithmic description of the interactions existing between the system (considered as a black box) and its environment,
- Functional static elements are called *functional scenarios*: a functional scenario of  $S$  is an algorithmic description of the interactions existing on one hand internally between the functions of  $S$  and on the other hand externally with the environment of the system,
- Constructional static elements are called *constructional scenarios*: a constructional scenario of  $S$  is an algorithmic description of the interactions existing on one hand internally between the components of  $S$  and on a second hand externally with the environment of the system.

<sup>108</sup> We recall that an interface between two static elements  $E$  and  $F$  is formally nothing else that the couple  $(E, F)$ . With such an interface, one may associate both flows exchanged between  $E$  and  $F$  and flows exchanged between  $F$  and  $E$  (we refer to the Flows subsection that follows for more detailed information on flows).

Note that these three types of scenarios have exactly the same nature since they are all describing an exchange algorithmic. The only difference comes here from the nature of the exchanges that are described by these different scenarios.

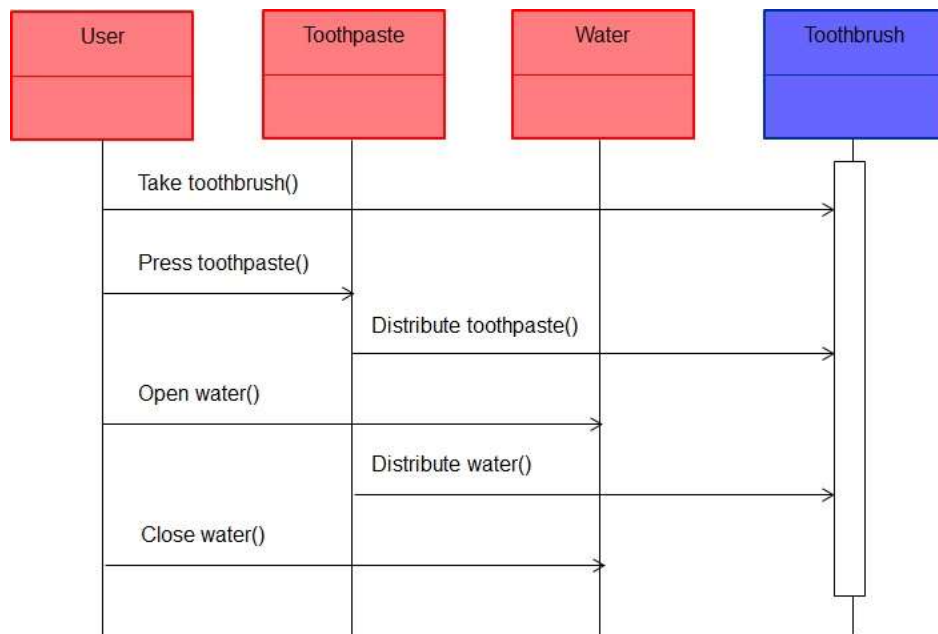


Figure 34 – Standard representations of an operational dynamic

Let us now end by addressing the question of the standard representations of the different types of dynamics which are typically given – in most modeling languages – by *sequence diagrams*. Figure 34 depicts this formalism with an example of operational dynamic description, here the initiation of a toothbrush use where one specifies the interactions existing between all involved stakeholders and the system. Sequence diagrams provide indeed an efficient and classical formalism for representing distributed algorithms (see [20] for more details). In this mode of representation, the different elements in interaction are positioned on the top of the diagram and each of them has a timeline going from top to bottom that represents time (each element having its own time). One models then an interaction by putting – one after the other – arrows, going from the initiator to the receiver of an interaction between two elements, with an indication either of the function, used by the interaction initiator to manage the interaction, or of the flow which is exchanged during the interaction (see our next subsection for more details). These arrows are thus following the sequential order of a given interaction, as depicted in Figure 34. Note finally that one indicates the interacting sequences that are highly coupled by a large rectangle at the level of the modeled system.

For the sake of completeness, we also provide an example of constructional scenario that can be found in Figure 35 below which represents the exact constructional counterpart of the previous operational scenario. Functional scenarios are represented exactly in the same way, components being just replaced by functions. Note that the difference between a functional or constructional scenario and an operational scenario is only that the environment is a black box in the first situation when it is the case of the system in the second situation<sup>109</sup>.

<sup>109</sup> There are of course also mixed scenarios where one may provide details on both the environment and the system.

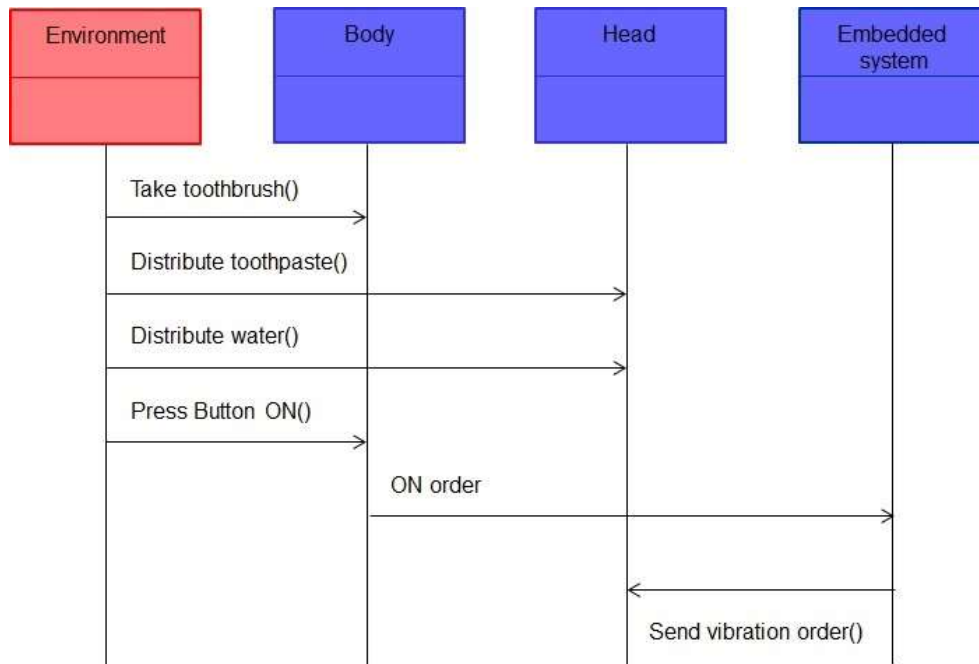


Figure 35 – Standard representations of a constructional dynamic

#### 2.4.2.4 Flows

A *flow* associated with a system *S* models an object – matter, energy, data, information, etc. – which is exchanged either externally between the system and its environment, or internally between two functional or constructional elements of the system.

Flows may be quite different depending on the vision. To illustrate that last point, let us take the example of a traffic light system. When the traffic light is red, it operationally sends a stop request to the drivers that are looking at it. The operational flow exchanged between the traffic light and the drivers can then be modeled by a “STOP ORDER” flow. On the functional level, one would however typically say that the traffic light is just sending red light (which is interpreted as a stop signal by the drivers) to its environment, which may be modeled by a “RED” functional flow. Finally it is amusing to observe that at constructional level, the red color is just produced by lighting the first (starting from the top) *white* traffic light, in connection with a red filter. The associated constructional flow can thus be modeled by “WHITE 3” to express that situation.

There are therefore logically three different types of flows for a given system *S*, depending on the considered architectural vision, which are defined as follows:

- Operational flows or objects: an *operational flow or object* of *S* is an object that is exchanged between *S* and its environment, i.e. between *S* and one of its external system,
- Functional flows or objects: a *functional flow or object* of *S* is an object which is an input or an output of one of the functions of *S*, i.e. which is exchanged between the functions of *S*,
- Constructional flows or objects: a *constructional flow or object* of *S* models a concrete object which is exchanged between the components of *S*.

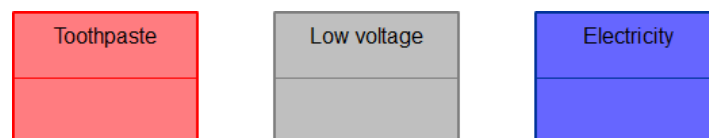
Flows are usually simply stated using only names that are referring to concrete objects which are exchanged between different systems. Table 4 below illustrates these various notion of flows with some examples for an electronic toothbrush.

Flow types	Flow	Nature
<i>Operational Flows</i>	Toothpaste	Matter
	Brushing data	Data
<i>Functional Flows</i>	Low voltage	energy
	Brushing measure	Data
<i>Constructional Flows</i>	Electricity	energy
	Brushing pressure signal	Data

**Table 4 – Examples of flows for an electronic toothbrush**

Note that flows are naturally related by allocation relations. As illustrated by the traffic light system example provided above, an operational flow OF may be allocated to either a functional flow or a constructional flow which may be operationally interpreted as OF. In the same way, a functional flow FF may also be allowed to a constructional flow that may be functionally interpreted as FF. Flows hierarchies are therefore naturally induced by these allocation relations.

Let us end by providing the standard representations of the three different types of flows – in most modeling languages – which are modeled by “objects” or “blocks” in the usual meaning given to this concept in object-oriented modeling (see [20] or [34]).



**Figure 36 – Standard representations of flows**

### 2.4.3 Expected Properties

As already stated above in section 0.3, expected properties are formally speaking *logical predicates* related with the system of interest (see Chapter 0 or Appendix A for more technical details on that core logical concept that goes back to Aristoteles). An expected property is thus nothing else than a Boolean function<sup>110</sup>, i.e. a function  $P$  that maps a system on TRUE or FALSE, depending whether the property that  $P$  models is satisfied or not by the system:

$$P: S \longrightarrow P(S) \in \{ \text{FALSE}, \text{TRUE} \} .$$

<sup>110</sup> This is thus also true for needs and requirements according to the definitions that follow.

This consideration allows avoiding confusing expected properties with their constitutive elements. An expected property indeed typically expresses that a given system shall behave or be structured with a certain external or internal performance<sup>111</sup>. The involved behavioral or structural elements & performances shall thus not be mixed with the property, since they are just not logical predicates.

It is also important to remember that one can analyze in practice a given system *S* from an external or from an internal perspective. This consideration leads to the key distinction between needs & requirements, which are the two first main types of expected properties on *S*:

- **External perspective**<sup>112</sup> – **Need**: a *need* with respect to *S* is a property that is expected or imposed by the environment *Env(S)* of *S*, expressed in the language of the environment<sup>113</sup> (i.e. only referring to operational descriptions & performance),
- **Internal perspective** – **Requirement**: a *requirement* on *S* is a functional or constructional property that shall be satisfied by *S*, expressed in the language of the system (i.e. only referring to functional or constructional descriptions & performances).

One shall of course understand that these definitions are fundamentally relative to a given system. A need with respect to a system *S* is indeed a requirement on the environment *Env(S)* of *S*. In the same way, a requirement on *S* is also a need with respect to a subsystem of *S*. One should hence always remember to point explicitly out the system of interest to which refers any need or requirement.

Note also that we used here above a voluntarily different terminology depending on the external or internal perspective we may take with respect to expected properties. A good system architecting practice indeed consists in strictly separating the domain of the *question* – expressed using needs – from the domain of the *solution* – expressed by means of requirements<sup>114</sup>. In other terms, needs shall be only reserved to model questions, when requirements shall model the corresponding answers. This point is crucial since many engineering problems are due to the fact that stakeholders are often expressing their “needs” in an intrusive way, i.e. in terms of requirements in our meaning. This bad practice both limit the ability of the designers to propose better alternative solutions and prevent them to know the real needs hidden behind requirements<sup>115</sup>, which may ultimately lead to bad solutions from an end-user perspective even if fitting perfectly to their requirements.

Note finally that requirements on a system *S* can be of course refined into two sub-types depending whether one is dealing with functional or constructional visions:

---

<sup>111</sup> In other terms, expected properties do express the performances that shall be satisfied by the system of interest or by its environment, depending whether one deals with the functional/constructional or operational visions.

<sup>112</sup> External perspective is here a synonym of operational vision.

<sup>113</sup> One cannot thus use the language of the system to express a need.

<sup>114</sup> This point is illustrated by Case study 4.

<sup>115</sup> In a recent customer specification file for military vehicles, one could for instance find the demand *C* ≡ “The vehicle shall be painted in green” which was just copied/pasted from previous files. This expected property is typically not a need, but a constructional requirement. A good systems architect shall then try to understand the functional & operational expected properties from which *C* derives. In this case, *C* can be traced back to a functional requirement *F* ≡ “The vehicle shall not be visible”, itself coming from a quite simple need *N* ≡ “Soldiers shall not die when in operations”. At this point, one now has the rationale of the green color which was just motivated by the fact that it allows to be invisible in European battlefields when green is dominant. But conflicts are not anymore taken place in Europe, but rather in Middle East or Afghanistan where ocher and sand colors are dominant. The analysis of the root need allows thus to understand that *C* is a very bad constructional choice to implement the functional requirement *F* which is still valid. One shall of course rather request *C'* ≡ “The vehicle shall be painted in ocher/sand colors”, which could not be suspected if only staying at constructional level.



- **Functional requirement:** a *functional requirement* on S is a property that shall be satisfied by the behavior of S, expressed in the functional language of the system (i.e. only referring to functional descriptions & performances),
- **Constructional requirement:** a *constructional requirement* on S is a property that shall be satisfied by the structure of S, expressed in the constructional language of the system (i.e. only referring to constructional descriptions & performances).

To write down properly these different types of expected properties, one shall use the standard statement patterns – referring to the notions introduced above – that are provided in Table 5.

<i>Need</i>	The “external system <sup>116</sup> ” shall “do something / be formed of something” with a certain “operational performance” in a given “operational context”.
<i>Functional requirement</i>	The “system” shall “do something <sup>117</sup> ” with a certain “functional performance” in a given “functional mode”.
<i>Constructional requirement</i>	The “system” shall “be formed of something <sup>118</sup> ” with a certain “constructional performance” in a given “configuration”.

**Table 5 – Standard statement patterns for needs and functional & constructional requirements**

Table 6 illustrates the three previous different types of expected properties – written in accordance with the above standard statement patterns – on an electronic toothbrush, with a picture of the toothbrush part which is (partially) specified by them. In this example, the proposed need was derived first into a functional requirement that was derived then into a constructional requirement. One can thus immediately trace back here the last constructional choice to the associated service provided to the end-users, which is useful – especially to analyze the stakeholder value brought by a given technical decision – since this service cannot be seen at constructional level.

<i>Need</i>	End users shall get a positive <sup>119</sup> feeling when efficiently cleaning their teeth.
<i>Functional requirement</i>	The electronical toothbrush shall display an encouraging message within 1 second when cleaning performance has been met.
<i>Constructional requirement</i>	The electronical toothbrush shall have a user interface of 2.5 cm x 1cm in each configuration.



**Table 6 – Examples of expected properties per architectural vision**

<sup>116</sup> Or equivalently a stakeholder of the system (see Chapter 3).

<sup>117</sup> “To do something” shall always refer here to an existing function of the considered system.

<sup>118</sup> “To be formed of something” shall always refer here to an existing component of the considered system.

<sup>119</sup> Positive refers here to the performance.

Note finally that the previous types of expected properties are *complete* by construction with respect to the CESAM framework. In other words, one can always specify any system in intension by *using only* needs, functional requirements and constructional requirements, without anything else.

### Needs or Requirements? The Tanker Case

In the 70's, important leaks occurred when pumping out liquid natural gas from tankers. The issue was that dangerous cracks appeared in the shell of the tankers due to the very cold temperature – closed to 0°K – of the gas.

To solve that problem, engineers began to think in terms of technical solutions which lead to a strong requirement on a metal for tankers shells that should not crack at liquid gas temperature. This was however a very bad idea since the resulting solution would probably have requested many years of R&T and cost hundreds of millions euros.

Another way of reasoning is to express the problem in terms of the question to solve, which is “how to avoid the gas to enter in contact with the shell metal?” Such a question expresses a simple constraint on the gas (which is a need in CESAM formalism) that has to be fulfilled.

Such a question has then a simple and quite cheap solution, consisting in putting pieces of folded cardboard under the leaking points. The gas is then retained by the cardboard before quickly evaporating, which avoids all troubles.

As one sees on that illustrative simple example, there may be a huge gap between thinking in terms of solutions (requirements) and in terms of questions (needs).

#### Case study 4 – Needs or requirements? The tanker case

There is in particular no need to introduce the concept of “non-functional requirements” that exists in many other architectural referential (cf. [42], [43] or [44]) and does often refer to “ity” properties of a system such as availability, maintainability, operability, safety, reliability, security, etc. All these properties can indeed easily be expressed in terms of needs. To be more specific, let us take the example of a maintainability property for a given system which would probably be expressed by most engineers by stating that  $M \equiv$  “the system shall be maintainable”. However “To be maintainable” can clearly not be considered as an internal function of a system since it does not refer to any input/output behavior, but rather to a permanent status of the system. Property  $M$  is therefore neither a functional, nor a constructional requirement<sup>120</sup>, nor a need<sup>121</sup> within our framework. It has thus absolutely no status at all, which shows that it is probably a bad specification! The good way of expressing the property  $M$  is then just to identify the hidden stakeholders behind – which are here just maintenance teams – and to understand what are expecting these stakeholders. In our example, this would lead us to reformulate  $M$  by stating instead  $M' \equiv$  “the maintenance teams shall maintain the system with a certain performance”<sup>122</sup> which is now obviously a need since it expresses an expectation of the environment of the system. The reader can do the same kind of exercise for the

<sup>120</sup> Since it also obviously does not directly refer to a property of the components of the system.

<sup>121</sup> Strictly speaking, it indeed only refers to the system and not to its environment.

<sup>122</sup> Which is now correctly written since “To maintain the system” is clearly a behavior of the maintenance teams.

other classical “non-functional properties” of a system to convince him/herself that all these properties are just bad formulations of needs.<sup>123</sup>

## 2.5 CESAM Systems Architecture Matrix

We are now in position to introduce CESAM Systems Architecture Matrix, which is presented in Table 7 below. This matrix is just a synthesis of the different architectural dimensions that we introduced within this chapter. It indeed presents all the types of views that allow to exhaustively describe any system, classified according to:

- a first axis of classification corresponding to the three *architectural visions*, that is to say the operational, functional and constructional visions,
- a second axis of classification corresponding to *behaviors*, that is to say the conjunction of:
  - *expected properties*,
  - all *descriptions*, i.e. states, static elements, dynamics and flows.

Crossing these two axis, one thus immediately gets the matrix of Table 7 where we listed the names of all different views that were introduced along the current section. As already stated above, the completeness of all these views in matter of system specification is an immediate consequence of all the material that we introduced along the previous pages.

Visions	Expected properties	Descriptions			
		States	Static elements	Dynamics	Flows
<i>Operational vision</i>	Needs	Operational contexts	Missions <sup>124</sup>	Operational scenarios	Operational flows or objects
<i>Functional vision</i>	Functional requirements	Functional modes	Functions <sup>125</sup>	Functional scenarios	Functional flows or objects
<i>Constructional vision</i>	Constructional requirements	Configurations	Components <sup>126</sup>	Constructional scenarios	Constructional flows or objects

**Table 7 – CESAM System Architecture Matrix**

As explained in subsection 2.4.3, one will of course always have to find the good balance between expected properties and descriptions when specifying a system. CESAM System Architecture Matrix is thus only a help to be sure that all dimensions of a system were taken into account during its modelling, but it does in no way provide – neither CESAM System Architecting Method does – an automatic specification mechanism for systems. Systems architecture indeed remains an art where expertise, experience and competency of systems architects are clearly fundamental!

<sup>123</sup> Availability, operability, safety, reliability or security issues do for instance typically refer to customers, end-users and/or operators expectations.

<sup>124</sup> Including descriptions of all integration mechanisms involving missions.

<sup>125</sup> Including descriptions of all integration mechanisms involving functions.

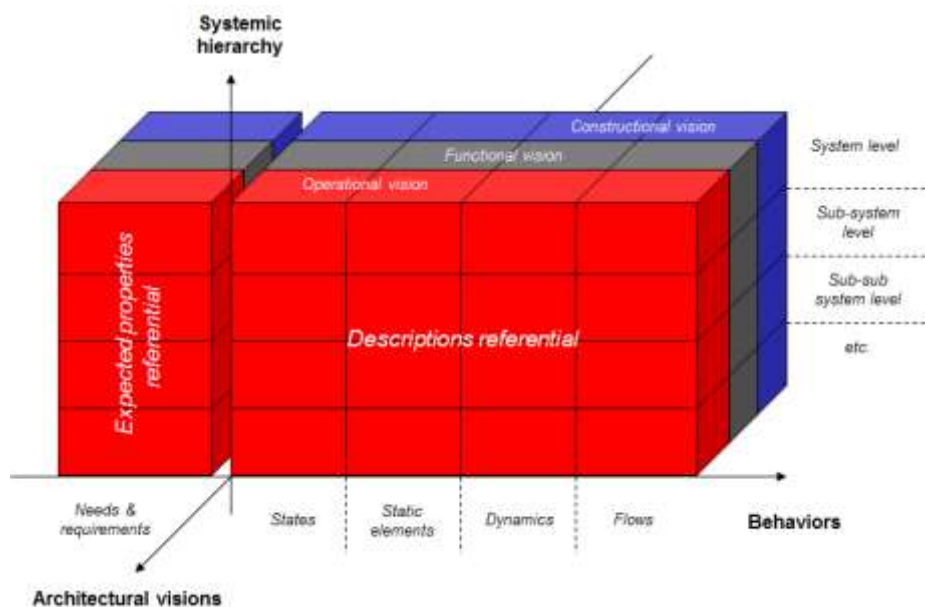
<sup>126</sup> Including descriptions of all integration mechanisms involving components.

To concretely illustrate this last notion, let us now provide an example of a partially completed CESAM System Architecture Matrix for the electronical toothbrush.

Visions	Expected properties	Descriptions			
		States	Static elements	Dynamics	Flows
<i>Operational vision</i>	End-users want to have less than one cavity in average per 5 years due to teeth brushing	Teeth brushing	Brush teeth	Teeth brushing scenario	Toothpaste
<i>Functional vision</i>	The electronic Toothbrush shall produce a brushing force of 0.5 N in automatic mode	Automatic mode	Produce a brushing force	Brushing force production scenario (functional)	Brushing force
<i>Constructional vision</i>	The electronic toothbrush shall have a removable head in children & adult configurations	Children configuration	Head	Brushing force transmission scenario (concrete)	Electricity

**Table 8 – Example of a CESAM System Architecture Matrix for the electronical toothbrush**

One can now understand why system modeling is so unintuitive. If one completes CESAM System Architecture Matrix by adding system abstraction / integration levels, one may indeed understand that a system model looks much more to a cube than to a matrix as depicted on Figure 37 below that represents CESAM System Architecture Cube, the 3D-version of the 2D CESAM System Architecture Matrix. One can thus understand that it is easy to be lost in such a multi-dimensional world!



**Figure 37 – CESAM System Architecture Cube**

Note also that the three first descriptions types – that is to say states, static elements and dynamics – are the most important since the last one – flows – is just a dedicated synthesis, focused on exchanges, which consolidates information that can already be found in the views corresponding to

dynamics. Restricting the CESAM System Architecture Matrix to these three first descriptions types leads us thus to a simpler matrix – the so-called CESAM 9-views matrix<sup>127</sup> – which provides the minimal number of descriptions to construct when “modeling” a system<sup>128</sup>. An example of such CESAM 9-views matrix is provided below on the electronical toothbrush case study.

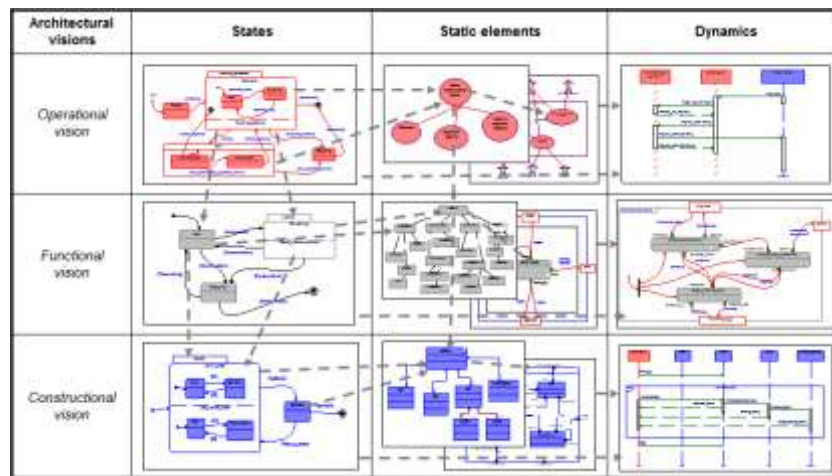


Figure 38- Example of a CESAM 9-views matrix for an electronical toothbrush<sup>129</sup>

At this point, note finally that CESAM Systems Architecting Method is nothing else than a certain way of moving in the CESAM System Architecture Matrix, starting from the knowledge of all use cases provided by the system lifecycle up to arriving to a quite precise vision on all constructional scenarios of the system. We will not develop this point here since the forthcoming chapters are dedicated to the presentation of the main deliverables of that process.

<sup>127</sup> This terminology was invented by Vincent Vion, chief systems architect of PSA Peugeot Citroën.

<sup>128</sup> Beware that modelling is considered in this pocket guide in a broader way with respect to the usual meaning of this concept which, for most authors, only refer to description – in the meaning of subsection 2.4.2 – construction.

<sup>129</sup> Our example of functional scenario is represented here using an activity diagram (see [20]) which an alternative to the representation mode introduced and discussed previously.



# Chapter 3 – Identifying Stakeholders: Environment Architecture

## 3.1 Why Identifying Stakeholders?

Stakeholder identification, or equivalently environment architecture, is a key systemic analysis: each mistake in this analysis may indeed result in flaws in the product under design. One must indeed remember that a system is nothing else than a concrete answer to a series of needs<sup>130</sup> and that these needs are coming from external stakeholders. As an immediate consequence, forgetting important stakeholders, misevaluating their role and/or considering erroneous ones will result in missing needs and/or working with wrong needs, and hence in missing requirements and/or working with wrong derived requirements, relatively to a given system. The resulting concrete system that might be developed on such a basis will therefore typically either miss the functions and components that are specifically addressing these missing needs, or have unnecessary functions and components that are associated with these wrong needs. One can therefore easily understand the crucial importance of correctly identifying stakeholders – that is to say the necessary and sufficient ones – since any system development process fundamentally relies on the quality of this identification.

To stress on this last point, one shall also have in mind the potential cost(s) of a wrong stakeholder identification which explain why one must put the necessary energy in this core initial analysis. In this matter, one commonly considers that such costs follow a geometric progression within the system lifecycle (see Figure 39): if the correction of an error during stakeholder identification has cost 1 (i.e. typically adding a missing stakeholder or replacing an erroneous stakeholder in this analysis), one usually considers that correcting the consequence of that error will have cost 10 when done during system design, cost 100 when corrected during detailed design, cost 1.000 when discovered during integration and even cost 10.000 if managed when the system is in service. One must thus spend enough time initially in order to achieve an as-sound-as-possible environment architecture.

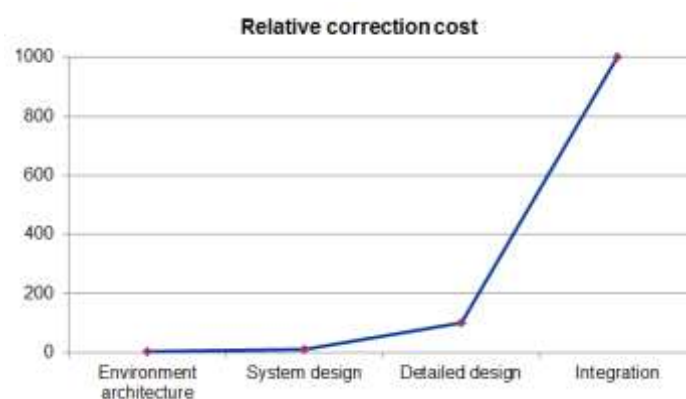


Figure 39 – Impact of an error in environment architecture

<sup>130</sup> We recall that we are using here the term “need” in a technical way. A need indeed refers to any property expected (which would correspond to a need in the common sense) or imposed (which would rather correspond to a constraint in the common sense) by the environment of a given system (see subsection 2.4.3 for more details).



Any system architect shall in particular always be highly anxious of being sure that stakeholders were correctly identified since all the development process rely on that initial analysis!

**Stakeholder Identification may Disrupt the Nature of a Design Problem:  
the Mobile Industrial Tool Case**

AVORE is an industrial company that produces heavy industrial electrical generators, each product having typically a weight of more than 100 tons. The average production duration of these generators was unfortunately around one year due to the fact that the assembly lines of AVORE's plants were never optimized.

An obvious reason for such a bad production delay was due to the fact that the electrical generators were produced in an industrial chain organized in three successive workshops where the generators were progressively assembled. Due to the important weight of the generators, the logistical time required to move the generators from one workshop into the other represented 40 % of the total construction time, which was not "lean" at all!

The method responsible of the Alsacian plant of AVORE had then a brilliant idea. Instead of moving the generators to the workshops, why not doing exactly the converse and moving the workshops to the generators, which would suppress a lot of stupidly lost time. Following this intuition, he launched a study for developing a mobile industrial tool – allowing to do the production activities of the second and third workshops of his plant – that would allow to manage the construction of the generators without moving them.

The initial step of this study was logically dedicated to a stakeholder identification. The first identified stakeholders were then naturally the plant, the industrial department of AVORE and the customers who would be the first beneficiaries of the new mobile industrial tool. In a second step, it was however understood that this tool had unfortunately a strong impact on another less important plant of AVORE, located in Normandy and dedicated to absorb the customer demands that the Alsacian plant could not manage. It indeed appeared that the efficiency increasing, brought by the mobile industrial tool, allowed the Alsacian plant to manage all customer requests, without any need of a supplementary plant.

The discovery of the Normand plant as a new stakeholder changed radically the problem which was not anymore a simple technical optimization question, but a deep political issue. Additional stakeholders emerged them immediately: trade unions, local Normand politicians and finally AVORE's general direction who canceled after two months of discussions the mobile industrial tool project in order to avoid any social trouble ...

**Case study 5 – The Mobile Industrial Tool Case**

### **3.2 The key Deliverables of Environment Architecture**

For any system S, environment architecture has two core deliverables:

1. the *stakeholder hierarchy diagram* that hierarchizes all stakeholders associated with S – or equivalently external systems – according to an abstraction hierarchy (see below),
2. the *environment diagram* that describes the exchanges existing between S and its first level stakeholders – or equivalently external systems – with respect to the above hierarchy.

These two deliverables are presented more in details below.

### 3.2.1 Stakeholder Hierarchy Diagram

Let  $S$  be a system and let  $Env(S)$  be its reference environment (see section 2.1). The *stakeholder hierarchy diagram* of  $S$  is then a hierarchical exhaustive representation of all stakeholders – or equivalently all external systems to  $S$  – that belong to  $Env(S)$ , a stakeholder  $H1$  being under another stakeholder  $H2$  in this stakeholder hierarchy if and only if  $H1$  is contained in  $H2$  (simply viewed here as sets), meaning that  $H1$  is a special case of  $H2$  or equivalently that  $H2$  is more abstract than  $H1$ .

The project system is for instance a typical stakeholder, associated with any engineered system that hierarchically abstract – in the above meaning – an engineering team, the supply chain, an industrialization team, a plant and the delivery logistics. The engineering team may then be recursively hierarchically decomposed into a project management team, a systems architecture team, different specialty engineering teams and a verification & validation team. The same type of recursive decomposition applies of course for all other first order stakeholders.

Figure 40 below also provides an illustrative partial example of stakeholder hierarchy diagram for an electronic toothbrush, a stakeholder or equivalently an external system being – classically – represented here by a graphic depicting a person<sup>131</sup>, when the inclusion or abstraction relationships on which this hierarchy relies are – also quite classically – represented by arrows.

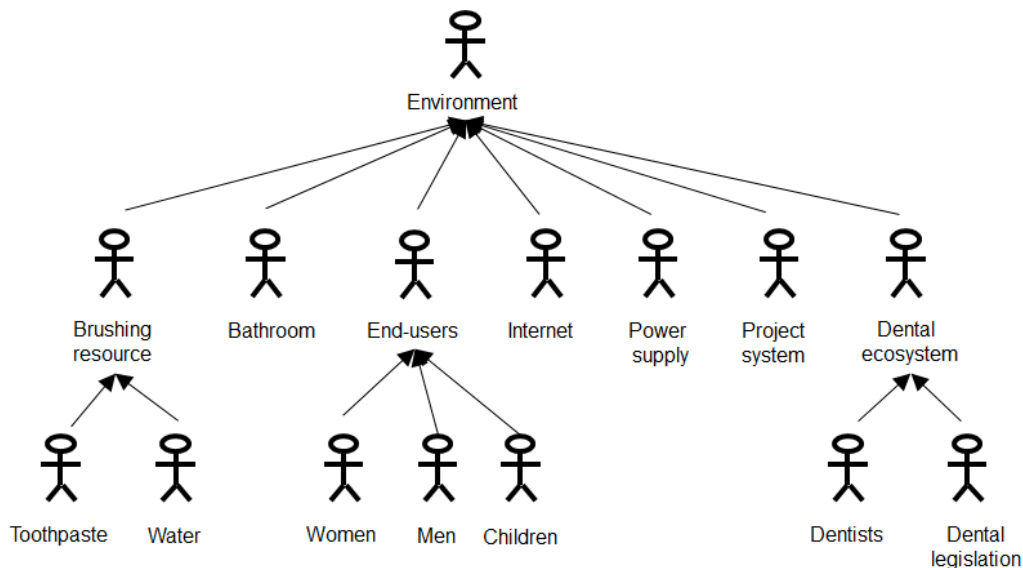


Figure 40 – Example of a stakeholder hierarchy diagram for an electronic toothbrush

When dealing with the stakeholder hierarchy diagram, the main standard difficulty is to find the “good” abstractions. One shall indeed avoid to have too much stakeholders of first level, but also too many levels of abstractions if one wants to be able to efficiently use such a view. The 7x7x7 rule provides a simple trick to use in order to organize optimally this diagram. This ergonomic principle indeed claims that a human being can only holistically understand a maximum number of more or

<sup>131</sup> This is just a representation which may typically model any hardware, software or “humanware” system as soon as they are part of the reference environment of the considered system.

less 350<sup>132</sup> data, as far as they were hierarchically clustered into 7 main groups of data, each of them again decomposed into 7 subgroups, each of them finally containing 7 elementary data. The 7x7x7 principle is therefore a precious tool for organizing a stakeholder hierarchy diagram: the limitations of the human brain indeed oblige to respect it for constructing such a diagram, as soon as one wants to easily read and communicate with these diagrams. As a consequence, a typical “good” stakeholder hierarchy diagram have no more than 7 high level stakeholders, each of them decomposed in around 7 medium level stakeholders, refining finally into 7 low level stakeholders. Note of course that the number 7 shall just be taken as an order of magnitude. Obtaining up to 10-12 high level stakeholders in a stakeholder hierarchy diagram would typically not be a heresy: however one must probably not go further without at least checking whether this number is justified. Finally one shall not hesitate to construct additional stakeholder hierarchy diagrams for refining such an analysis as soon as all relevant stakeholders are not captured.

### 3.2.2 Environment Diagram

Let again  $S$  be a system and  $Env(S)$  be its reference environment (see section 2.1). The *environment diagram* of  $S$  is then a representation of:

- the system  $S$  and all the high level stakeholders – or equivalently external systems to  $S$  – in the meaning of the stakeholder hierarchy introduced in the previous paragraph,
- all flows exchanged between the system and its stakeholders, that is to say between  $S$  and the external systems of its reference environment.

Figure 41 that follows gives an example of environment diagram, here associated with an electronic toothbrush, taking here the same representation for stakeholders of  $Env(S)$  than in the stakeholder hierarchy diagram, when the system  $S$  is modeled by a box.

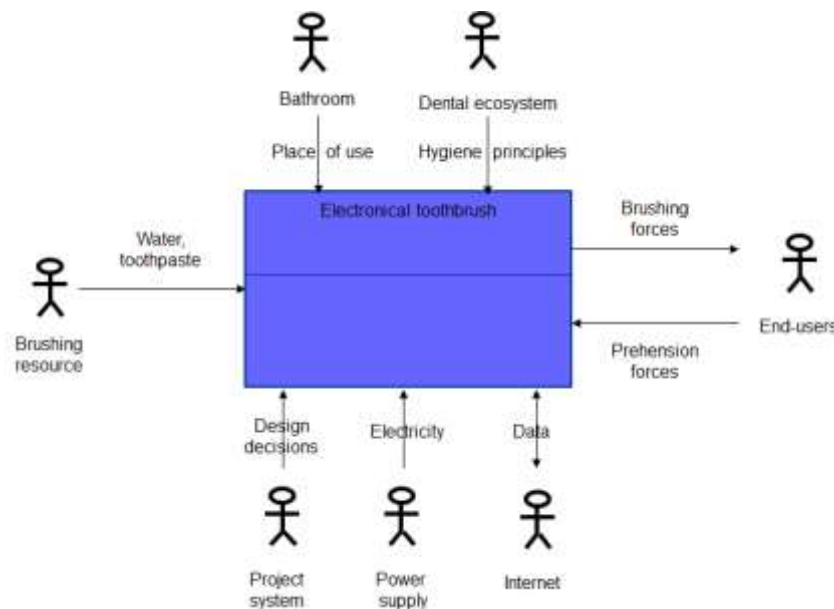


Figure 41 – Example of an environment diagram for an electronic toothbrush

<sup>132</sup> 7 multiplied by 7, multiplied again by 7 makes 343.

A classical way of organizing an environment diagram for a given system  $S$  consists in respectively positioning the following stakeholders, i.e. external systems, in the four quadrants of the diagram:

- in the left hand-side of the diagram: key input systems of  $S$ ,
- in the right-hand side of the diagram: key output systems of  $S$ ,
- in the top of the diagram: key constraining systems with respect to  $S$ ,
- in the bottom of the diagram: key resource systems with respect to  $S$ .

These key input, output, constraining and resource systems associated with  $S$  within the environment diagram will be respectively denoted below by  $I(S)$ ,  $O(S)$ ,  $C(S)$  and  $R(S)$ .

Note that the environment diagram of Figure 41 is for instance typically organized in such a way. This mode of representation is useful since it equips such a diagram with a natural semantics. One may indeed automatically read the mission of the considered system on an environment diagram organized in such a way using the following pattern:

<b>Pattern of mission statement of a system <math>S</math></b>
The system $S$ shall transform inputs of $I(S)$ into outputs of $O(S)$ under constraints $C(S)$ and with resources $R(S)$ .

**Table 9 – Pattern of mission statement of a system**

In a similar way, functional analysis is also naturally oriented by such a representation mode. The core functions of a system  $S$  are indeed then the functions that are connecting the input systems  $I(S)$  to the output systems  $O(S)$ , when the piloting and the support functions of  $S$  are the functions that are mainly exchanging with respectively the constraining systems  $C(S)$  and the resource systems  $R(S)$ .

The environment diagram is thus a very important diagram which induces structuring architectural orientations for a system. It can also be used for monitoring the first systems architecting activities. A good environment architecture shall indeed fundamentally always be balanced: the number of lower level stakeholders or of needs per high level stakeholder shall typically be more or less the same. Any difference of balance in these numbers shall therefore necessarily puzzle the good systems architect who must question it, even if there is a rational explanation to it. As we can see, the environment diagram is a very rich diagram and a precious tool for the systems architect!



# Chapter 4 – Understanding Interactions with Stakeholders: Operational Architecture

## 4.1 Why Understanding Interactions with Stakeholders?

Operational architecture, or equivalently operational analysis, intends to precisely understand the interactions among time between the system of interest and the external systems of its reference environment, or equivalently of its stakeholders. Motivations of environment architecture, as already discussed in subsection 3.1, also apply in the same terms – *mutatis mutandis* – to operational architecture. Exactly as for stakeholder identification, any forgetfulness, misunderstanding or error that could occur during the operational architecture process may indeed have disastrous and costly consequences on a system under development.

As previously pointed out in subsection 3.1, one must indeed understand that any function and any component of a system ultimately intends to provide an answer to a stakeholder and thus is always involved in the different interactions that are taking place between the system and its environment (of reference). One can therefore just not design – at least with a reasonable level of confidence – the different functions or components of a system without understanding the missions to which they contribute. Many design mistakes are typically done when designers do not have any precise idea of the various operational contexts in which the system they are developing will be used. We can thus only stress here the imperious necessity of always managing operational architecture in any systems architecture process, which will also allow to bring back “meaning” to the end-engineer who is working on a little part of a large system. Medieval cathedrals – whose construction took centuries – would probably never have been built if all involved workers had not a deep understanding of the global target to which they were contributing ...<sup>133</sup>

The decoupling between operational architecture and both functional & constructional architecture is also fundamental. This apparently simple principle is in fact much more subtle than it seems. It typically allows to develop systems that have very different operational architectures, but similar functional & constructional architectures. This is the principle of product lines where one constructs “flexible” systems which have a very large customer diversity – in order to fit as much as possible to the needs of the market – but with a very low technical diversity<sup>134</sup>. The idea is to develop a family of systems with a very large number of operational architectures, corresponding to different customer needs, on the basis of 1) standard functional and constructional elements (usually corresponding to everything that the customer does not see) and 2) a limited number of additional specific functional and constructional elements that are capturing the operational – or equivalently here the customer – diversity (and thus the value that is perceived by the customer). Such an approach allows to deliver highly customized products to the customer that are constructed using only standard modules. Many

---

<sup>133</sup> In “The announcement to Mary”, a play written by the French writer Claudel whose action takes place in the Middle-Age during the construction of the Saint-Rémi church in Reims – France, two sculptors are working on a little statue located in the front of the church. One asks them what they are doing and they are answering: “we are building a cathedral”.

<sup>134</sup> This last principle is the basis of *diversity management* whose purpose is to maintain in configuration among time such flexible architectures.

industries – automotive, consumer electronics and even food & beverage<sup>135</sup> – are currently using with success this type of architecture for their products.

Despite of its core importance as stressed above, operational architecture is unfortunately still an analysis which is not well known and often not practiced at all, probably since it may be considered as not enough technical and concrete (engineers usually like to quickly jump into technique ...). We thus must emphasize on the value of operational architecture which is – even if apparently not too technical<sup>136</sup> – a core technical analysis that can only be done by a systems architect. One shall indeed understand that operational architecture deliverables are structured in order to be easily mapped with functional and constructional deliverables. Thus it is just impossible to perform any operational architecting without understanding precisely its functional and constructional consequences, which can only be done by a technical-minded person, typically a systems architect.

### **A typical Lack of Operational Architecture: the Airbus A400M Atlas Case**

The Airbus A400M Atlas is a multi-national, 4-engine turboprop, military transport aircraft. It was designed by Airbus in order to replace older transport aircraft, such as the Transall C-160 and the Lockheed C-130 Hercules. The project began in the early 1982, as the Future International Military Airlifter (FIMA) group, set up jointly by Aérospatiale (France), British Aerospace (Great-Britain), Lockheed (USA) and Messerschmitt (Germany), when the first flight of the A400M took only place on December 2009 from Seville, Spain, as a result of a tremendous number of development program delays (that moreover also lead to huge cost overruns as one could have expected).

The root cause of these problems can probably be traced back to the requested operational architecture for this aircraft. The A400M was indeed requested to support both tactical (i.e. managing supplies and equipment transportation in a theater of military operations) and strategic (i.e. transporting material, weaponry or personnel over long distances) missions. But it happens that these two operational contexts are radically different: on one hand, the tactical context requires the aircraft to manage short landing & take-off distances and to have low-pressure tires allowing operations from small or poorly prepared airstrip; on the other hand, the strategic context is characterized by long landing & take-off distances and by high-pressure tires for moving heavy charges on (normal) military airports. Similar discrepencies could also of course be observed at the level of the aircraft control logics. Thus, as one can easily guess, implementing these two very different use cases in the same constructional architecture is not a “piece of case”.<sup>137</sup>

### **Case study 6 – The Airbus A400M Atlas Case**

---

<sup>135</sup> All types of Danone or Nestlé yoghurts are for instance made using the same white mass (standard module #1) and the same preforms, i.e. tubes from which different types of cups are produced through adapted blowing (standard module #2). When one makes a yoghurt, the industrial chain begins by a “generic” type of yoghurt, before adding the specific additives (the specific modules) that are given different flavors to the yoghurts. This principle is called *late differentiation*: it allows to react quickly when a competitor launches a new product or when the customer taste evolves, since one just need changing the last machines of the industrial chain to adapt without reorganizing the full chain (which takes time and money).

<sup>136</sup> Operational architecture can indeed be seen as an interface between the stakeholder and the engineer worlds, since it offers a language that can be understood both by stakeholders and engineers, which explains its apparent simplicity.

<sup>137</sup> This case study also illustrates that operational architecture cannot be done in isolation. One must always understand and validate its functional and constructional consequences before freezing an operational architecture.



## 4.2 The key Deliverables of Operational Architecture

For any system S, operational architecture has five core types of deliverables:

1. the *need architecture diagram* that hierarchically organizes all needs – with respect to S – according to an refinement hierarchy (see below),
2. the *lifecycle diagram* that describes how S passes – with indication of the associated events – from an operational context to another one, starting from its birth up to its death,
3. the *use case diagrams* that are describing – in a purely static way – the missions of S that are contributing to a given operational context,
4. the *operational scenario diagrams* that are describing – in a dynamic way – the interactions taking place between S and its stakeholders<sup>138</sup> in a given operational context,
5. the *operational flow diagrams* that synthetizes all flows – with their logical relationships – exchanged between S and its reference environment during the lifecycle of S.

These different types of deliverables are presented more in details below.

### 4.2.1 Need Architecture Diagram

Let S be a system. The *need architecture diagram* of S is then a hierarchical exhaustive representation of all needs with respect to S, a need N1 being under another need N2 in this hierarchy if and only if one can logically deduce N1 from N2<sup>139</sup>. In this last situation, one says then more precisely that N2 refines into N1, which explains why one speaks of a need refinement hierarchy.

Figure 42 below now shows a (partial) need architecture diagram for an electronic toothbrush, a need being – classically – represented here by a 2-part box, whose first top part is a short name summarizing the need scope and second bottom part is the need statement, when the refinement relationships on which the need hierarchy relies are – also classically – represented by arrows.

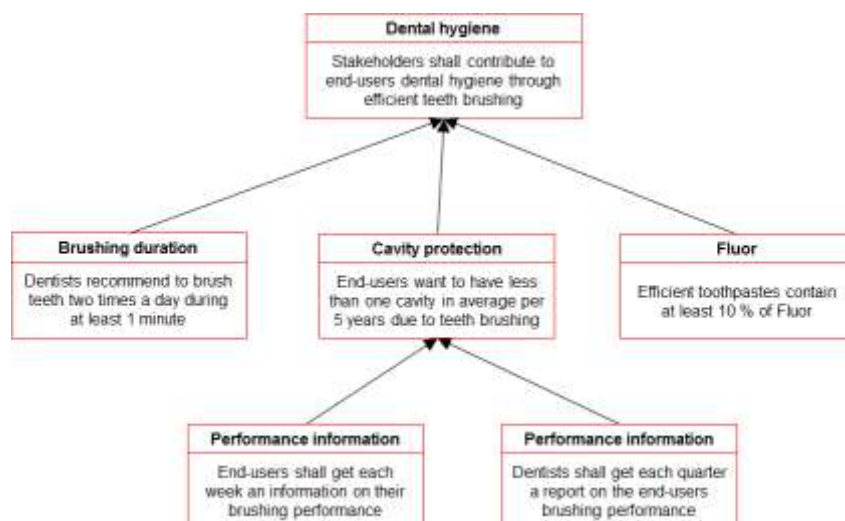


Figure 42 – Example of a need architecture diagram for an electronic toothbrush

<sup>138</sup> Or equivalently external systems.

<sup>139</sup> Remember that needs are logical predicates (see subsection 2.4.3).

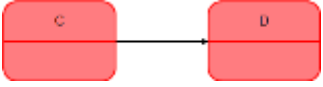
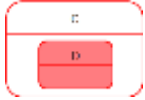
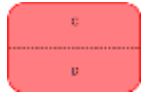
The same issue that was already pointed out in the first part of subsection 3.2 when dealing with the stakeholder hierarchy diagram can also be expressed – more or less in the same terms – with the need architecture diagram: organizing a need refinement hierarchy is indeed always difficult since one shall avoid to have too much needs of first level, but also too many levels of refinements if one wants to be able to efficiently use such a view. The 7x7x7 rule (see first part of subsection 3.2) is again precious to handle this real difficulty. As a consequence, a typical “good” need architecture diagram associated with a given system shall have no more than 7 high level needs, each of them refined in around 7 medium level needs, finally also refining in the same way into 7 low level needs. Note again that the number 7 shall of course be just taken as an order of magnitude. Obtaining up to 10-12 high level needs in a need architecture diagram could of course work: however one must probably not go further without analyzing whether this number is justified. Finally one shall not hesitate to construct additional need architecture diagrams for refining such an analysis as soon as all relevant needs are not captured.

#### 4.2.2 Lifecycle Diagram

Let  $S$  be again a system. The *lifecycle diagram* of  $S$  is then a representation of:

- the operational contexts of  $S$ , with their relative temporal relationships (consecutiveness, inclusion or simultaneity)<sup>140</sup>,
- the events that cause the different transitions between each operational context of  $S$  and the immediately consecutive ones.

To draw such a diagram, we shall give the standard representations of the three temporal relations between operational contexts that we introduced above, which are provided by Table 10, where  $C$  and  $D$  stand for generic operational contexts. Remember here that operational contexts are modeled by ovals, as introduced in the first paragraph of subsection 2.4.2.

Temporal relation	Semantics	Graphic representation
Consecutiveness	$D$ is consecutive to $C$ when an termination event of $C$ is exactly equal to an initiation event of $D$	
Inclusion	$D$ is included in $C$ when the period of time underlying to $D$ is contained in the period of time underlying to $C$	
Simultaneity	$D$ is simultaneous to $C$ when the periods of time underlying to $C$ and $D$ are exactly equal	

**Table 10 – Graphic representations of temporal relationships between operational contexts**

<sup>140</sup> These three temporal relations are necessary and sufficient to model any temporal relationships between operational contexts among a system lifecycle. To be convinced of that claim, let us analyze the (only embarrassing) situation of two intervals of time  $P$  and  $Q$  that overlap, i.e. such that  $P = [s, t]$  and  $Q = [u, v]$  with  $u < v$ . One can model such a situation with our temporal relations by first decomposing  $P$  into  $P1 = [s, u]$  and  $P2 = [u, t]$  and  $Q$  into  $Q1 = [u, t]$  and  $Q2 = [t, v]$  and observing then that  $P2$  is consecutive to  $P1$ ,  $Q1$  is simultaneous to  $P2$  and  $Q2$  is consecutive to  $Q1$ .

Figure 43 below provides an illustrative lifecycle diagram associated with an electronic toothbrush, taking here the standard representation of operational contexts and of their temporal relationships that we already introduced, when events – that induce operational context transitions – are modeled by arrows labelled with the name of the relevant event. Note also that the initial (resp. termination) events in each operational context do not respect this rule since they are conventionally modeled by small black circles (resp. by white circles containing a black circle).

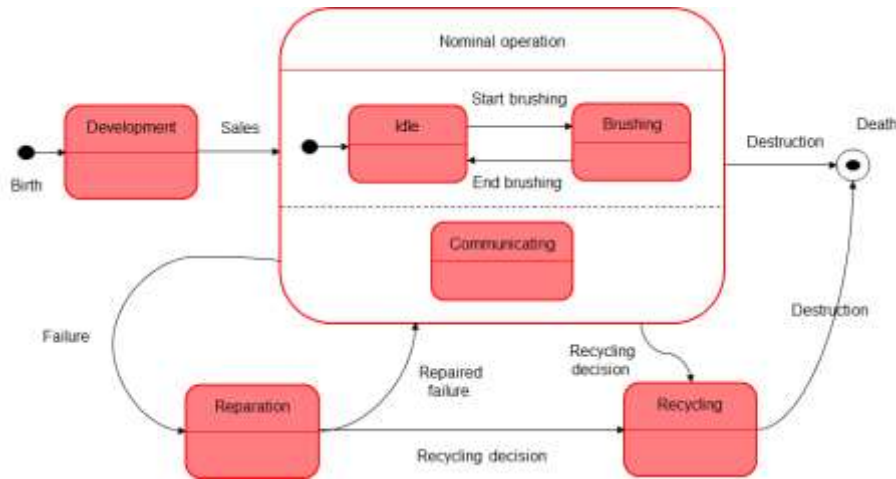


Figure 43 – Example of lifecycle diagram for an electronic toothbrush

Note finally that there is a perfect symmetry between the environment diagram, dedicated to model the space in which evolves a system, and the lifecycle diagram, dedicated to model the periods of time through which passes a system. Since space and time are always both required to specify any system behavior (that necessarily takes place somewhere at a certain time), one can easily see that these two diagrams are equally important to specify any system.

### 4.2.3 Use Case Diagrams

Let  $S$  be a system, let  $\text{Env}(S)$  be its reference environment and let  $q(S)$  be an operational context of  $S$ . A *use case diagram* associated with  $S$  and  $q(S)$  is then a static representation of the missions achieved through the collaboration of  $S$  and its external systems within  $\text{Env}(S)$ , during the period of time which is modeled by  $q(S)$ , which explicitly specifies:

1. the external systems of the environment of  $S$  that are contributing to each mission,
2. the missions that contribute to another mission.

Note that it is something necessary, when modelling a use case diagram, to also represent behaviors of  $\text{Env}(S)$  in which the system  $S$  is not contributing at all. This is done by just indicating that  $S$  is not contributing to such a function of  $\text{Env}(S)$ .

Figure 44 that follows provides an example of use case diagram, associated with the “Brushing data transmission” operational context of an electrical toothbrush. The square represents the system of interest when we used again the “person” representation to model its stakeholders. A mission is (resp. not) placed in the square when the system of interest contributes (resp. does not contribute) to it. In the same way, one puts a line between a stakeholder and a mission when the stakeholder

contributes to such a mission<sup>141</sup>. One also indicates by a rigid arrow when a mission contributes to another mission and by a dashed arrow when a given mission  $M_1$  is mandatory to manage another one  $M_2$ . Beware at that last level since the standard convention in this matter is highly counter-intuitive: one indeed models such a situation by putting a dashed arrow from  $M_2$  to  $M_1$  and not the converse. Note finally that it is interesting to observe on that example that the motivation of the “brushing data transmission” cannot be found in a mission of the electronic toothbrush, but rather in the “Follow brushing recommendations” which is an environment function involving only end-users and dentists without the electronic toothbrush. One can then easily understand the value of a use case diagram on such a situation which would clearly not be possible to describe without a specific environment-oriented diagram such as a use case diagram.

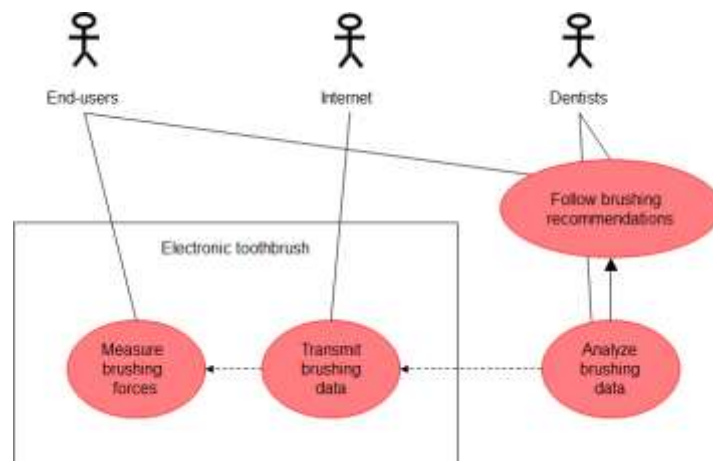


Figure 44 – Example of use case diagram for an electronic toothbrush

It is interesting to point out that if one considers the special – limit – case of the operational context equal to the complete lifecycle of a given system  $S$ , the associated use case diagram would especially provide a hierarchical representation of all missions of  $S$ , jointly with the indication of the different stakeholders that are contributing to each mission of the system. For the sake of readability, one can of course decompose that last use case diagram into the two following use case diagrams:

1. the first one providing just a hierarchical representation of all missions of  $S$ , which shall be naturally called the *Mission Breakdown Structure* (MBS) of  $S$ <sup>142</sup>,
2. the second one providing the indication of the stakeholders that contribute to each mission of the system, whose semantics is equivalent to a *mission / stakeholder allocation matrix*.

Note finally that if no hierarchically related missions occur when modelling a given use case diagram, the semantics of such a diagram is completely contained within the associated operational scenario diagram (see next paragraph). One shall then decide the diagram to take since the use case diagram has no need with the associated operational scenario diagram (the converse being not true).

<sup>141</sup> Let  $UC(S)$  be a use case diagram associated with a given system  $S$ . A mission which is put in the square part of  $UC(S)$  and which is connected through lines to stakeholders  $S_1, \dots, S_n$  within  $UC(S)$  is then formally a function of the system resulting from the integration of  $S$  with  $S_1$  up to  $S_n$ .

<sup>142</sup> If one decides to model such a Mission Breakdown Structure (MBS), one must beware to the readability of such a view. All the recommendations based on the 7x7x7 rule that we previously gave for the stakeholder and the need architecture diagrams will then of course also apply – mutatis mutandis – in order to efficiently model the MBS.

#### 4.2.4 Operational Scenario Diagrams

Let again  $S$  be a system,  $\text{Env}(S)$  be its reference environment and  $q(S)$  be an operational context of  $S$ . An *operational scenario diagram* associated with  $S$  and  $q(S)$  is then a dynamic representation of the missions achieved through the collaboration of  $S$  and its external systems within  $\text{Env}(S)$ , during the period of time which is modeled by  $q(S)$ , which explicitly specifies all interactions occurring between  $S$  and the stakeholders – or equivalently the external systems – of its reference environment.

The following Figure 45 shows an example of operational scenario diagram, associated with the “Reparation” operational context of an electrical toothbrush. We refer to the suitable paragraph of subsection 2.4.2 for all details on the semantics of the below representation.

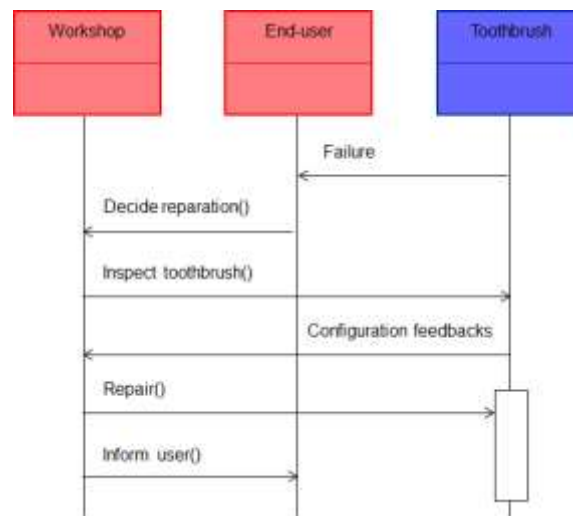


Figure 45 – Example of operational scenario diagram for an electrical toothbrush

An operational scenario diagram provides therefore an explicit algorithmic description which models the behavior of the environment of a given system during a given operational context. As already stated in the last paragraph, one must always choose whether using either a use case diagram, or an operational scenario diagram to specify an operational context, when no hierarchically related missions occur within the use case diagram, since this last diagram will not add any semantics to the corresponding operational scenario diagram.

#### 4.2.5 Operational Flow Diagram

Let  $S$  be a system. The *operational flow diagram* associated with  $S$  is a consolidated description of all operational flows associated with  $S$  and of respectively:

1. their logical relationships,
2. their abstraction relationships<sup>143</sup>.

It plays therefore the role of the operational “data model”<sup>144</sup> of the system. Note that one also may split this diagram into two diagrams, each of them covering the two above points.

---

<sup>143</sup> We recall that a flow  $A$  is abstracted by a flow  $B$  if and only if  $A$  is a special instance of  $B$ . In relativist mechanics, Matter is for instance abstracted by Energy.

Figure 46 below shows an example of (partial) operational flow diagram, associated with an electrical toothbrush. The different logical relationships, which exist between the various operational flows (or objects) represented in that diagram, are modeled by rigid lines (without any arrow) labelled with the denomination of the corresponding relationship. Note that one usually uses a verb – in the third form of singular – to name such a logical relation: in an operational flow diagram, the line connecting a flow of type A with a flow of type B that represents a logical relation between A and B is typically labeled by a verb such as “is related to” in order to express that “A is related to B” or that “B is related to A”. To avoid ambiguity, one places the relationship denomination closer to the first term of such a logical relationship.<sup>145</sup>

One may also put on the extremity of these lines an indication of the *arity* of these relationships: if  $n$  operational flows of type A can be associated with  $m$  operational flows of type B within a given logical relation, one just puts a label with “ $n$ ” (resp. “ $m$ ”) at the A-extremity (resp. B-extremity) of the line put between A and B that models the corresponding relation (we recall that  $(n,m)$  is called the arity of such a logical relationship). Note also that, by convention, one puts “\*” instead of a natural number when there are not known limits to the number of involved elements.

Finally, on another totally different hand, the abstraction relationships that are provided in the above diagram are modeled – according to a classical convention – by squared arrows.

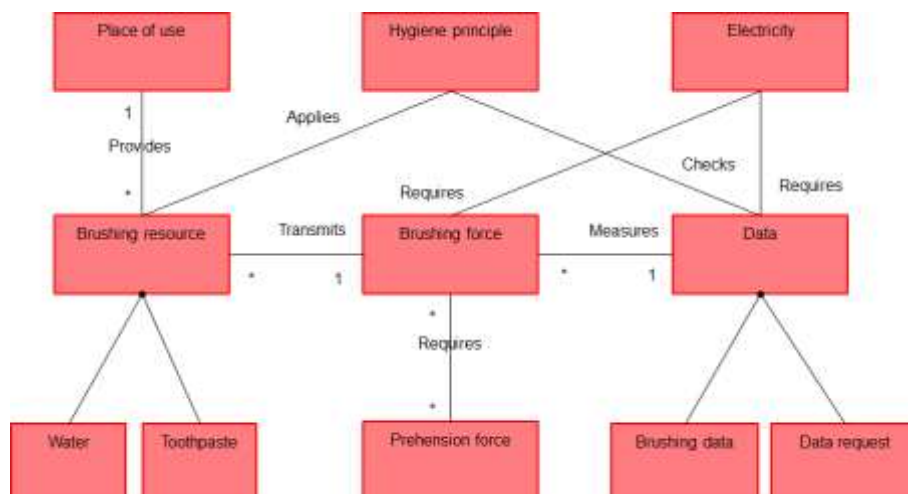


Figure 46 – Example of operational flow diagram for an electronic toothbrush

As already stated the operational flow diagram defines the operational flow or object model of a given system. It is completely “dual” to the environment, use case or operational scenario diagrams since it focuses on flows, and not on the different functions, either of the system or of its reference environment, that are producing these flows. Unfortunately most of engineers, who usually do not have any computer science or software engineering background, do not understand the importance and the value of this new type of flow-oriented diagram ...

<sup>144</sup> Beware that, even if we use the syntax of a data model for the operational flow diagram, this last diagram is not really a data model since it does not represent (only) data, but also physical objects, business objects or even informal information that may be exchanged by “humanware” stakeholders of a given system.

<sup>145</sup> Strictly speaking one should put two labels on each line between any flow A and any flow B in order to express both the logical relations between A & B and between B & A. However this would be too heavy which explains our convention.

We must therefore emphasize that such a diagram is of high importance since it rationally describes in a consolidated and organized way all inputs and all outputs of a given system. Hence it gives the operational “dictionary” of the system, that is to say the list of all objects that are operationally manipulated by the system. This dictionary is of high value for ensuring a common vision between all project actors involved in the operational architecting process: these actors shall normally – in an ideal world – only use the terms of that dictionary when discussing of an operational object. One may easily understand that such a principle allows to avoid any ambiguity between the system designers and the project system stakeholders, but also within the different specialty engineers. It is thus key for ensuring a good collaboration between all these actors.





# Chapter 5 – Defining What shall Do the System: Functional Architecture

## 5.1 Why Understanding What Does the System?

Functional architecture, or equivalently functional analysis, intends to describe precisely the different functions of a system and their relative interactions<sup>146</sup>. The core motivation of functional architecture is to start understanding and specifying in details the system, but only in terms of behaviors – i.e. in other terms, to understand and specify what does the system – and not of concrete structure, i.e. in its functional nature ... as one would have easily guessed!

It is indeed important to have first a functional description of a system, and not to dig immediately in the technique, if one wants to be able to rationally manage trade-offs between different options<sup>147</sup>. Functional architecture is indeed usually independent of the technological choices<sup>148</sup> – at least at some level of abstraction – which means that functionally reasoning – of course at the good level of abstraction<sup>149</sup> – on a system allows to reason at the same time on many different constructional options that we will be able to compare and to evaluate later (see Chapter 7 dedicated to these trade-off analyses). One must indeed understand that skipping functional analysis by directly going to technical design is a very bad practice, even if widespread in engineering organizations, since it just means that one makes a very strong design choice without even being conscious of that choice. As a consequence, one will just be glued in that choice, without being able to move to radically different options that may be more adapted.

We must also stress that functional architecture is absolutely fundamental to capture emergence and transversal behaviors that can only be modeled using its tools. By definition, all emergent behaviors can indeed not been captured by constructional architecture since they are functional properties of an integrated system (we refer to subsection 0.2 for more details). One must hence use functional architecture to describe and model such properties. As an immediate consequence, functional architecture is key tool to structure transversal collaboration in an engineering organization whose purpose is indeed just to manage efficiently the emergent transversal behaviors of a system.

---

<sup>146</sup> And also how these functions are connected to missions. This point – even if important – will however not addressed in this pocket guide since it can be easily addressed through a suitable allocation matrix.

<sup>147</sup> Functional architecture allows in particular to make early cost analysis as soon as one have an idea of the average cost of an elementary function (see also the Constructive Cost Model for Systems – COSYSMO – [79]). Such a feature is especially interesting for trade-offs that may also be done at functional level (in order to choose between two competing functional architectures for a given system).

<sup>148</sup> The car function “Produce torque” is for instance totally independent of the technology: it exists on a car either with a thermic, or an electrical engine.

<sup>149</sup> As a consequence of that simple remark, functional architecture is absolutely of NO USE if its analyses go too much in the detail. Detailed functional architecture indeed 1) either overlaps with constructional architecture as soon as detailed functions identify with the high level functions of the components of the considered system, 2) or be totally misaligned with its constructional architectural (which means that the identified functions do not naturally map with components). In the first case, functional architecture overlaps with constructional architecture since the two analyzes do provide exactly the same semantics. In the second case, functional architecture is dangerous since its results – which have here no concrete value – may mislead the system designers. In the two case, it is therefore a waste of time and money.

Last, but not least, functional architecture also allows to organize a system in functionally decoupled – as much as possible – sub-systems. This is very important if one wants to avoid too many impacts when there is a change in a system design<sup>150</sup>. This idea gives rise to layered functional architectures where each functional layer is strictly functionally segregated from the other ones by rigid standard functional interfaces that are managed in configuration. This architectural functional segregation / decoupling principle provides huge flexibility: in an ideal world, one will indeed be able to change a function in one layer without any impact on the other layers, as soon as the functional interfaces between layers are respected. We refer to the concrete examples of systems organized according to such a principle – that is to say computers, mobile phones or communication networks – that are provided in the functional architecture subsection of section 2.2.

### **Transversal Behaviors are Crucial to Master: the Airport Radar Case**

ALTHEIS is a leading airport radar company in the world. They developed a new airport radar on the basis of a modular generic functional architecture, each generic functional module being devoted to a certain part of a radar treatment chain and managed by a dedicated engineering team. The idea was to replace development, when dealing with a specific radar deployment, by parametrization. Each generic functional module had thus to be specifically parametrized when instantiated on a given concrete airport application.

However the overall amount of parameters to manage was quite high: around one million elementary parameters that could be organized in around 50.000 high level parameters, each of them with a specific business meaning, were indeed managed by the development engineers. This huge complexity lead initially to what could have become a real industrial disaster: when a radar was parametrized and put in service, it happened that the radar never stabilized since bugs were permanently appearing on the field which regularly obliged the radar to go back to the factory to be reparametrized, which could only be done by the development team due to the technicity of the parametrization. This situation was terribly uncomfortable and clearly economically not sustainable<sup>151</sup>.

CESAMES was appointed to analyze and to try to solve that issue. It appeared that its root cause was connected to the lack of a shared and explicit radar functional architecture. When the parametrization was initially done, each team in charge was indeed not conscious at all of its functional interdependence – through transversal functions – with the other teams. As a consequence, each parametrization done locally at the level of one team was in conflict with the other parametrizations, which lead to the observed problems.

The solution – provided by CESAMES – was to architecture all parameters in alignment with the radar Functional Breakdown Structure (FBS), by clustering the parameters according to the different functions of the FBS. A parameter architecture team – managed by a functional architect – was then created to manage, guarantee and maintain among time the functional coherence of each of all these parameter clusters.

### **Case study 7 – The Airport Radar Case**

---

<sup>150</sup> See the first case study of Chapter 6 to see what can unfortunately happen in case of a design evolution ...

<sup>151</sup> A radar business model is indeed based on first a fixed initial price that just covers the development costs and secondly yearly maintenance fees on which the constructor is making its benefit. One can thus easily understand that permanent bugs are just destroying the radar business model.

## 5.2 The key Deliverables of Functional Architecture

For any system S, functional architecture has five core types of deliverables:

1. the *functional requirement architecture diagram* that hierarchically organizes all functional requirements – with respect to S – according to a refinement hierarchy,
2. the *functional mode diagram* that describes how S passes – with indication of the associated events – from a functional mode to another one, starting from its birth up to its death,
3. the *functional decomposition & interaction diagrams* that are describing – in a purely static way – the functions of S with their interactions<sup>152</sup>,
4. the *functional scenario diagrams* that are describing – in a dynamic way – the interactions taking place between the functions of S, in a given functional mode,
5. the *functional flow diagrams* that synthetizes all flows – with their logical relationships – absorbed or produced by the functions of S during the “functional mode cycle”<sup>153</sup> of S.

These different types of deliverables are presented more in details below.

### 5.2.1 Functional Requirement Architecture Diagram

Let S be a system. The *functional requirement architecture diagram* of S is then a hierarchical exhaustive representation of all functional requirements of S, a functional requirement R1 being under another functional requirement R2 in this hierarchy if and only if one can logically deduce R1 from R2<sup>154</sup>. In this last situation, one says then more precisely that R2 refines into R1, which explains why one speaks of a functional requirement refinement hierarchy.

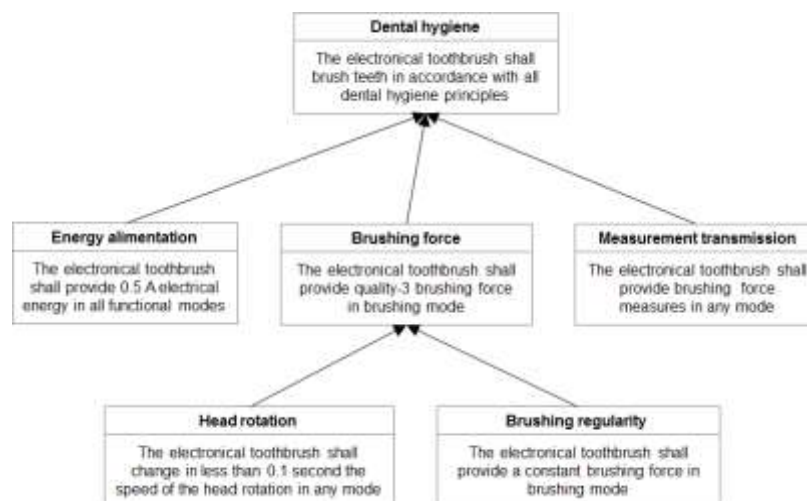


Figure 47 – Example of a functional requirement architecture diagram for an electronic toothbrush

Figure 47 illustrates here a (partial) functional requirement architecture diagram for an electronic toothbrush, a functional requirement being – classically and exactly similarly to a need – represented

<sup>152</sup> Usually only at global level, but also possibly in only a given functional mode.

<sup>153</sup> That is to say the period of time modeled by the functional mode diagram.

<sup>154</sup> Remember that functional requirements are logical predicates (see subsection 2.4.3).

here by a 2-part box, whose first top part is a short name summarizing the functional requirement scope and second bottom part is the functional requirement statement, the refinement relations, on which relies the functional requirement hierarchy, being – also classically – represented by arrows.

Note that exactly the same issue already pointed out for the need requirement architecture diagram takes also place with the functional requirement architecture diagram: organizing a functional requirement refinement hierarchy is indeed always difficult since one shall avoid to have too much functional requirements of first level, but of course also too many levels of refinements if one wants to be able to efficiently use such a view. The 7x7x7 rule (see first part of subsection 3.2) is again a precious tool to handle this real difficulty. As a consequence, a typical “good” functional requirement architecture diagram associated with a given system shall have no more than 7 high level functional requirements, each of them refined in around 7 medium level functional requirements, finally also refining in the same way into 7 low level functional requirements. Note again that the number 7 shall just be taken as an order of magnitude. Obtaining up to 10-12 high level functional requirements in a functional requirement architecture diagram could of course work: however one must probably not go further without analyzing whether this number is justified. Finally one shall not hesitate to construct as many additional functional requirement architecture diagrams as necessary, for refining such an analysis as soon as all relevant functional requirements are not derived and/or captured.

## 5.2.2 Functional Mode Diagram

Let S be again a system. The *functional mode diagram* of S is then a representation of:

- the functional modes of S, with their relative temporal relationships (consecutiveness, inclusion or simultaneity)<sup>155</sup>,
- the events that cause the different transitions between each functional mode of S and the immediately consecutive ones.

The standard representations of the temporal relations between functional modes introduced above are given – mutatis mutandis – by Table 10, if one now interprets C and D as functional modes.

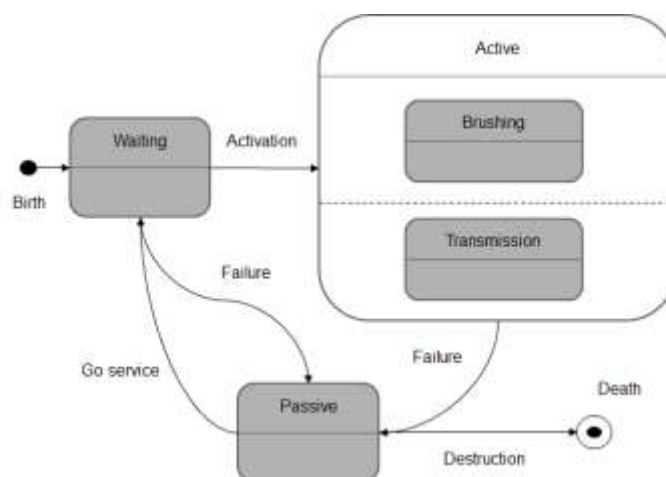


Figure 48 – Example of functional mode diagram for an electronic toothbrush

<sup>155</sup> We refer to the second paragraph of subsection 4.2 in this matter.

The above Figure 48 provides in particular an illustrative functional mode diagram associated with an electronic toothbrush, taking here the standard representation of functional modes and of their temporal relationships that we introduced, when events – that induce functional mode transitions – are modeled by arrows labelled with the name of the relevant event. Note also that the initial (resp. termination) events in each functional mode do not respect this rule since they are conventionally modeled by small black circles (resp. by white circles containing a black circle).

It is finally important to understand that the functional mode diagram is key since it models – from a functional perspective – time. Following the intuition that we developed at the end of the second paragraph of section 4.2, one could say that the next diagrams – i.e. the functional decomposition & interaction diagrams – are modeling the “functional space” in which functions are evolving. Since space and time are always required to specify any functional behavior (that takes place “functionally somewhere” at a certain time), these two diagrams are completely complementary.

### 5.2.3 Functional Decomposition & Interaction Diagrams

Let  $S$  be a system. The *functional decomposition diagram* associated with  $S$  is then a hierarchical representation of the functions of  $S$ , a set  $F_1, F_2, \dots, F_N$  of functions being under another function  $G$  in this hierarchy if  $G$  is the result of the integration – in the meaning of Definition 0.5<sup>156</sup> – of the functions  $F_1, \dots, F_N$ <sup>157</sup> ( $F_1, \dots, F_N$  are then classically called “sub-functions” of  $G$ ). The *functional interaction diagrams* associated with  $S$  are then just the different representations – there is one functional interaction diagram per integration relationship involved in the functional decomposition diagram – of each such integration relationship that does exist between the different functions appearing in the hierarchy modeled by the functional decomposition diagram.

Figure 49 below now provides an illustrative partial example of functional decomposition diagram for an electronic toothbrush, where the integration relationships on which this hierarchy relies are – quite classically – represented by squared arrows.

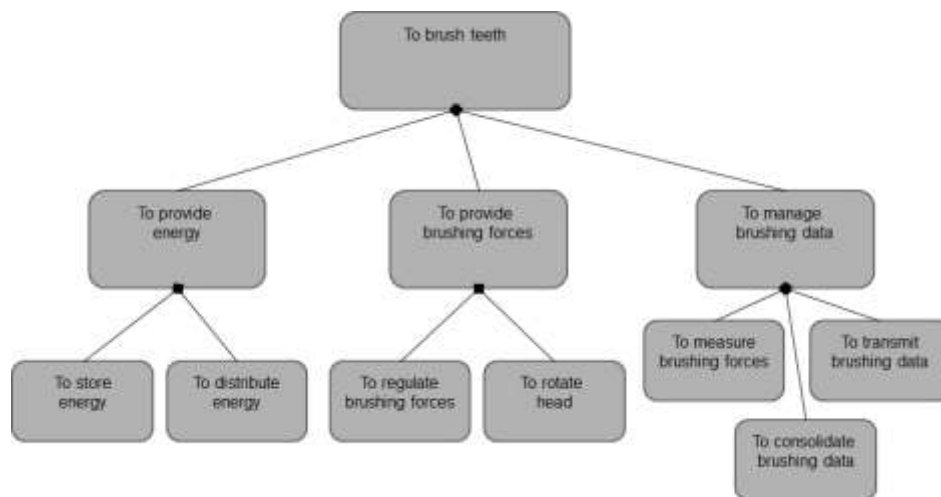


Figure 49 – Example of a functional decomposition diagram for an electronic toothbrush

<sup>156</sup> Considered here uniquely for its functional scope

<sup>157</sup> Due to our definition of the integration operator, this hierarchy is therefore an abstraction hierarchy.

We also give an example of a functional interaction diagram for an electronic toothbrush that can be found in Figure 50. In this example, we modeled the integration relationship existing between the global function of the toothbrush and its first “sub-functions” (using the formalism of a classical “activity diagram” in the UML or SysML meaning; see [20] or [34]). Note also that external interfaces are here – quite classically – represented by white squares at the border of the oval representing the integrated function (here the global function of an electronic toothbrush).

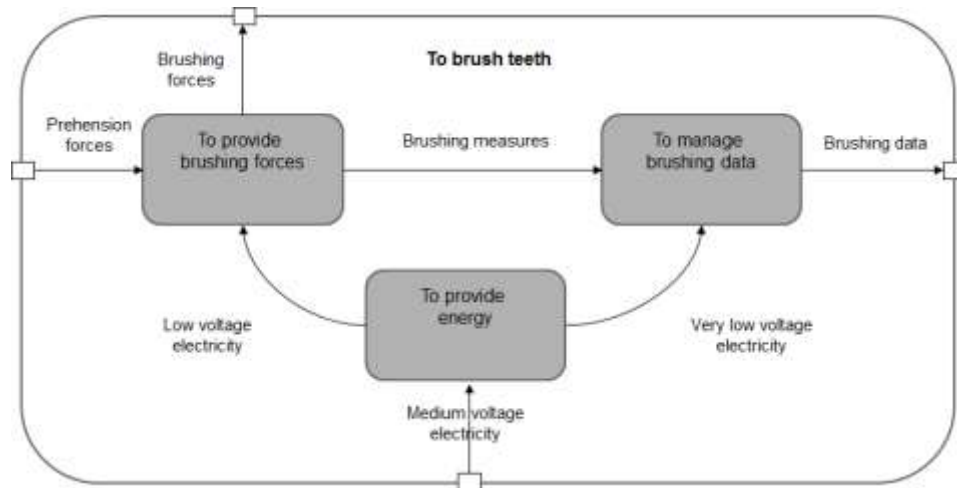


Figure 50 – Example of a functional interaction diagram for an electronic toothbrush

The functional decomposition of a system  $S$  modeled by the functional decomposition diagram is also classically called the Functional Breakdown Structure (FBS) of  $S$ . Similarly to the Mission Breakdown Structures that we introduced in the last chapter, it provides the exhaustive dictionary of functions of the system and thus has a key role in guaranteeing a common understanding on the functional scope of a system, which is mandatory for efficient transversal collaboration between the different actors and stakeholders of a system development project.

One must however beware to the readability of such a view. All the recommendations based on the 7x7x7 rule that we previously gave for the stakeholder and need architecture diagrams of course also apply – *mutatis mutandis* – to efficiently model the Functional Breakdown Structure of a system.

Last, but not less important, we refer to Figure 9 in Chapter 0 for a concrete functional interaction diagram associated with an aircraft: it represents how all first-level sub-functions may be integrated to obtain the high-level global function of an aircraft.

## 5.2.4 Functional Scenario Diagrams

Let again  $S$  be a system and  $q(S)$  a functional mode of  $S$ . A *functional scenario diagram* associated with  $S$  and  $q(S)$  is then a dynamic representation of the interactions that are taking place between the functions of  $S$  during the period of time which is modeled by  $q(S)$ .

The below Figure 51 shows an example of functional scenario diagram, associated with the “Active” functional mode of an electrical toothbrush. We refer to the suitable paragraph of subsection 2.4.2 for the fundamentals of the semantics of this representation. However we were obliged to introduce richer semantics with respect to the one that was introduced in subsection 2.4.2. The below diagram



indeed expresses that as far as the electronic toothbrush is in active mode (which was modeled by the big box with “loop” on its top left and the “[active mode]” indication<sup>158</sup> in its top middle), it shall do in parallel (which was modeled by the big box with “par” on its top left), that is to say at the same time, three types of operations (that are separated by dashed lines in the “alt” big box): the first one being brushing force management, the second one being brushing data management and the third one being energy management. For the sake of completeness, one shall also know that there exists an “alt” (for alternatives) box which allows to express “if then else” situations<sup>159</sup>.

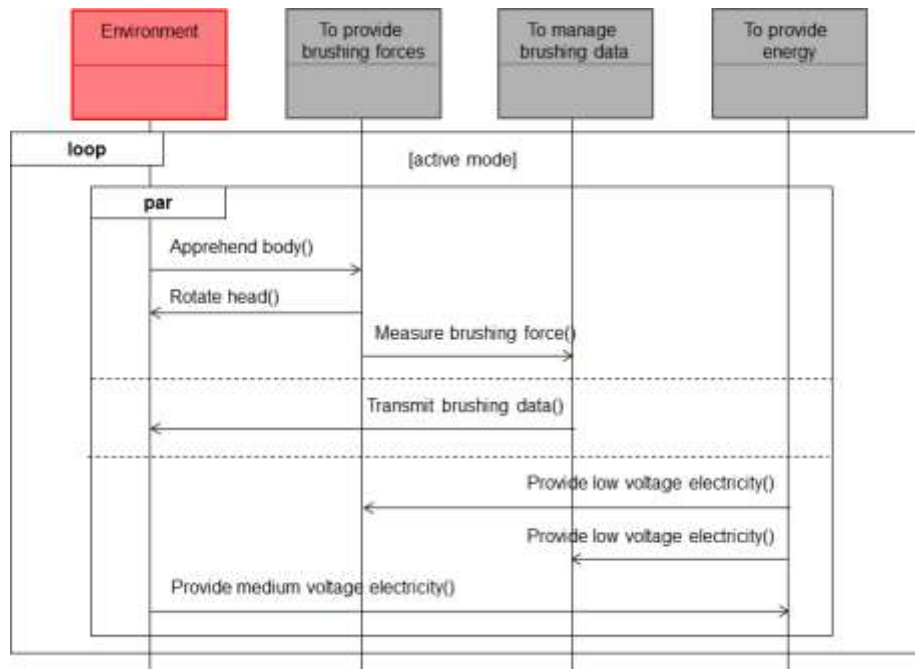


Figure 51 – Example of a functional scenario diagram for an electronic toothbrush

A functional scenario diagram provides the explicit “algorithm” which is underlying to the functional behavior of the system in a given functional mode. This is key to understand finely what – at least functionally – happens during a given functional mode. The enriched semantics that we introduced indeed allows to express any distributed algorithmic property of a system<sup>160</sup>.

### 5.2.5 Functional Flow Diagram

Let S be a system. The *functional flow diagram* associated with S is a consolidated description of all functional flows associated with S and of respectively:

1. their logical relationships,
2. their abstraction relationships (see the last paragraph of Chapter 4).

<sup>158</sup> This indication means “as soon as”. Hence we meant here “as soon as the system is in active mode”.

<sup>159</sup> The “if then else” situation is expressed by a big box labelled “alt” on its top right, split in two parts – Part 1 (top) and Part 2 (bottom) – separated by a dashed line, with a “condition” denomination at its top middle. Its semantics is that when the “condition” is satisfied (resp. not satisfied), the system shall do the instructions of Part1 (resp. Part2).

<sup>160</sup> This comes from the fact that usual algorithmic only requires the “while” (modeled by the “loop” box) and the “if then else” (modeled by the “alt” box) operators. When one passes to distributed – that is to say parallel – algorithmic, it is then sufficient to add the “parallel” operator (modeled by the “par” box).

Hence it plays the role of the functional “data model”<sup>161</sup> of the system. Note that one also may split this diagram into two diagrams, each of them covering the two above points.

Figure 52 below shows an example of (partial) functional flow diagram, associated with an electrical toothbrush. Its syntax follows exactly the same principles than for the operational flow diagram (see the last paragraph of the previous chapter).

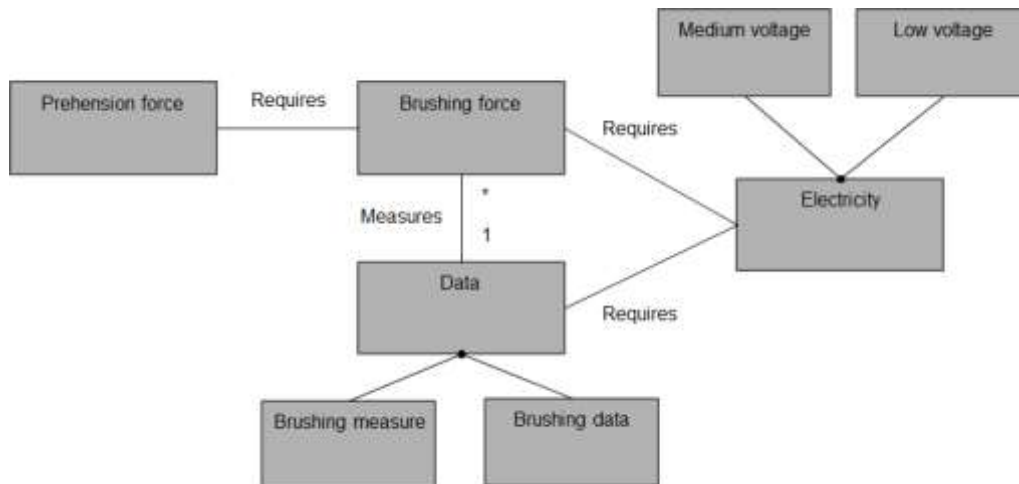


Figure 52 – Example of functional flow diagram for an electrical toothbrush

As already stated the functional flow diagram defines the functional flow or object model of a given system. It is completely “dual” to the functional decomposition, interaction or scenario diagrams since it focuses on flows and not on the functions of the system that are producing these flows.

We must thus again emphasize that such a diagram is of high importance since it rationally describes in a consolidated and organized way all inputs and all outputs of the functions of a given system. Hence it gives the functional “dictionary” of the system, that is to say the list of all objects that are functionally manipulated by the system. This dictionary is of high value for ensuring a common vision between all project actors involved in the architecting process: these actors shall normally – in an ideal world – only use the terms of that dictionary when discussing of a functional object. One may easily understand that such a principle allows to avoid any ambiguity between the different project actors. It is thus key for ensuring a good collaboration between these actors.

<sup>161</sup> Beware that, even if we use the syntax of a data model for the functional flow diagram, this last diagram is not really a data model since it does not represent (only) data, but also physical objects, business objects or even informal information that may be exchanged with “humanware” stakeholders of a given system.

# Chapter 6 – Deciding How shall be Formed the System: Constructional Architecture

## 6.1 How to Understand How is Formed the System?

Constructional architecture, or equivalently constructional analysis, intends to describe precisely the different components of a system, but also all their relative interactions<sup>162</sup>. The core motivation of constructional architecture is to concretely understand and specify in details the system, in terms of structure – i.e. in other terms, to understand how is formed the system – and not of behaviors, i.e. in its constructionally nature ... as one would have naturally guessed!

Constructional architecture is key since it consolidates all architectural analyses in a concrete vision of the considered system. It makes in particular the synthesis between a top-down design approach, as provided by the systems architecting process, and a bottom-up one, which is typically induced by the constraints due to the existing product architecture or by the new possibilities brought by the advances of technology. All the idea of constructional architecture is thus to find the best possible balance between these two apparently contradictory, but in reality completely complementary, approaches. As a consequence, constructional architecture intends to solve a “simple” – to state, but not to solve – multi-dimensional optimization problem: “what is the best concrete architecture – i.e. suitable components with their organization – which answers to the stakeholder needs (top-down approach) while integrating all industrial and technological constraints & opportunities (bottom-up approach), within classical cost, quality, delay and performance objectives?”<sup>163</sup> This problem is of course highly non-trivial and highly complex in practice due to its large number of parameters and variables. The motivation of constructional architecture is – quite modestly – to propose some key tools that may contribute to that objective.

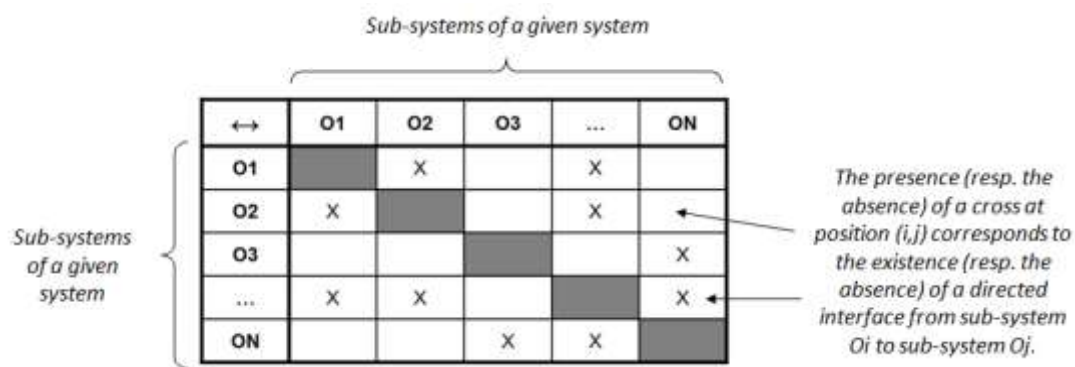


Figure 53 – Design Structure Matrix (DSM) of a system

Constructional architecture also allows to integrate “by design” some important key architectural principles within a system. The best way to prevent the propagation of a local problem throughout a

<sup>162</sup> And also how these components are connected both to missions and functions. This point – even if important – will however not be addressed in this pocket guide since it can be easily managed through suitable allocation matrices.

<sup>163</sup> It is always important to be able to state properly any constructional architecture problem using such a pattern.

system and its transformation into a global problem, is for instance to organize the system as an integration of decoupled sub-systems, i.e. with minimal mutual interfaces. This property can be easily depicted on the Design Structure Matrix (DSM) associated with a given system  $S$  – see Figure 53 – where one indicates a connection from a sub-system  $O_i$  of  $S$  to another sub-system  $O_j$  of  $S$  if and only if there is a directed interface from  $O_i$  to  $O_j$ : minimizing the interfaces of  $S$  therefore means finding a constructional decomposition minimizing the number of elements of such a matrix.

Last, but not least, one must point out that constructional architecture is the main input of a series of important engineering activities such as specialty engineering analyses, safety analyses, verification and validation, that we will however not develop here (see [42], [43], [44] or [66]).

### **What happens when a System has not a Decoupled Constructional Architecture: the Fighter Aircraft F/A-18 Case<sup>164</sup>**

The standard aircraft F/A-19 is a fighter & attack aircraft that was developed by McDonnell Douglas for the US Army in the late seventies. It was designed for being aircraft carrier based, supporting 3,000 flight hours, 90 minutes average sorties and 7.5 g positive maximal accelerations and having 15 years of useful life.

In the early nineties, the Swiss Army decided to acquire this aircraft. However due to its geographic and politic specificities, Switzerland had very different requirements. First due to their neutrality policy, they did not wanted a fighter, but an interceptor aircraft. Since Switzerland does not have any sea (Leman lake does not count for one ...), it was demanded to have a land based aircraft. Swiss people do also like that their belongings last a long time, so they requested their aircraft to support 5,000 flight hours with a 30 year useful life. Finally one shall remember that Switzerland is a very small country with mountains to avoid during each sortie, which lead to be able to support on one hand 40 minutes average sorties and on the other hand 9g positive maximal accelerations.

When McDonnell Douglas engineers analyzed these demands, it was quickly understood that the US version of the F/A-18 could easily respect the Swiss requirements as soon as its less resistant fatigue components were replaced. A deeper analysis showed them that it was even sufficient to replace a few – weighting  $\approx 500$  grams – structural aircraft parts made in Aluminum by equivalent components made in Titanium (which is much more robust).

Unfortunately when this – apparently quite small – change was done, the center of gravity changed which required stiffening the fuselage and increasing the gross takeoff weight to rebalance the aircraft. Due to that various changes, the weight distribution evolved within the aircraft, impacting the flight control software which was necessary to redesign. Various other changes occurred and at the very end, the industrial construction processes and the associated plant were even highly impacted, leading to a 10 million \$ overall cost for a little initial change of less than one kilogram ...

Well decoupling a system's constructional architecture is therefore crucial!

### **Case study 8 – The Fighter Aircraft F/A-18 Case**

---

<sup>164</sup> The author is grateful to Professor Olivier de Weck (MIT, USA) who told him this case study.

## 6.2 The key Deliverables of Constructional Architecture

For any system S, constructional architecture has five core types of deliverables:

1. the *constructional requirement architecture diagram* that hierarchically organizes all constructional requirements – with respect to S – according to an refinement hierarchy,
2. the *constructional mode diagram* that describes how S passes – with indication of the associated events – from a configuration to another one, starting from its birth up to its death,
3. the *constructional decomposition & interaction diagrams* that are describing – in a purely static way – the components of S with their interactions<sup>165</sup>,
4. the *constructional scenario diagrams* that are describing – in a dynamic way – the interactions taking place between the components of S, in a given functional mode,
5. the *constructional flow diagrams* that synthetizes all flows – with their logical relationships – absorbed or produced by the components of S during the “configuration cycle”<sup>166</sup> of S.

These different types of deliverables are presented more in details below.

### 6.2.1 Constructional Requirement Architecture Diagram

Let S be a system. The *constructional requirement architecture diagram* of S is then a hierarchical exhaustive representation of all constructional requirements of S, a constructional requirement R1 being under another constructional requirement R2 in this hierarchy if and only if one can logically deduce R1 from R2<sup>167</sup>. In this last situation, one says then more precisely that R2 refines into R1, which explains why one speaks of a constructional requirement refinement hierarchy.

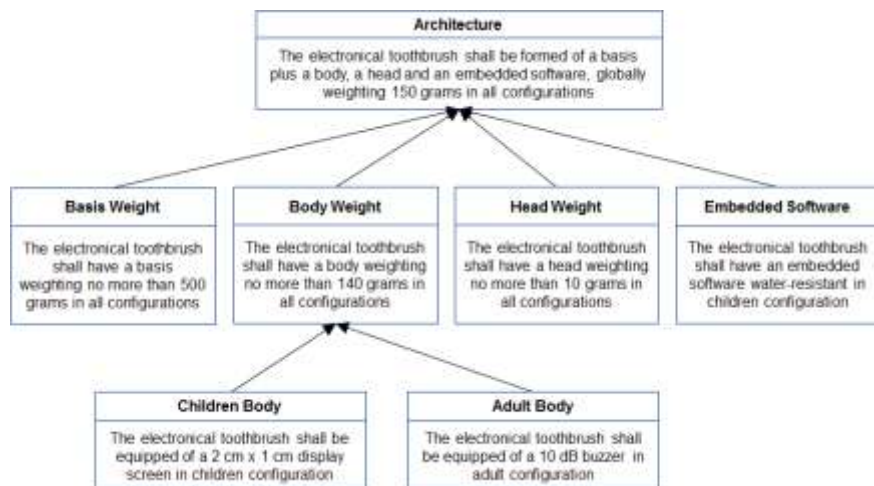


Figure 54 – Example of a constructional requirement architecture diagram for an electronic toothbrush

The above Figure 54 shows a (quite partial) constructional requirement architecture diagram for an electronic toothbrush, a constructional requirement being – classically and similarly both to a need and to a functional requirement – represented here by a 2-part box, whose first top part is a short

<sup>165</sup> Usually only at global level, but also possibly in only a given configuration.

<sup>166</sup> That is to say the period of time modeled by the configuration diagram.

<sup>167</sup> Remember that constructional requirements are logical predicates (see subsection 2.4.3).

name summarizing the constructional requirement scope and second bottom part is dedicated to the constructional requirement statement, when the different refinement relationships on which relies the constructional requirement hierarchy are – also classically – represented by arrows.

The same issue that was already pointed out, both for need and functional requirement architecture diagrams, also happens in the same terms for the constructional requirement architecture diagram: organizing a constructional requirement refinement hierarchy is indeed always difficult since one shall avoid to have too much constructional requirements of first level, but of course also too many levels of refinements, as soon as one wants to be able to efficiently use such a view. The 7x7x7 rule (see first part of subsection 3.2) is again precious to handle this real difficulty. As a consequence, a typical “good” constructional requirement architecture diagram associated with a given system shall have no more than 7 high level constructional requirements, each of them being refined in around 7 medium level constructional requirements, finally also refining in the same way into 7 low level constructional requirements. Note again that the number 7 is just an order of magnitude. Obtaining up to 10-12 high level constructional requirements in a constructional requirement architecture diagram could of course work: however one must probably not go further without analyzing whether this number is justified. Finally one shall not hesitate to construct as many additional constructional requirement architecture diagrams as necessary, for refining such an analysis as soon as all relevant constructional requirements are not derived and/or captured.

## 6.2.2 Configuration Diagram

Let S be again a system. The *configuration diagram* of S is then a representation of:

- the configurations of S, with their relative temporal relationships (consecutiveness, inclusion or simultaneity)<sup>168</sup>,
- the events that cause the different transitions between each configuration of S and the immediately consecutive ones.

The standard representations of the temporal relations between configurations introduced above are given – mutatis mutandis – by Table 10, if one now interprets there C and D as configurations.

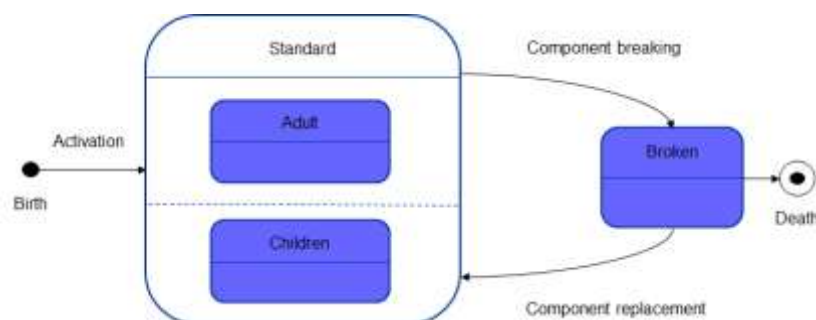


Figure 55 – Example of configuration diagram for an electrical toothbrush

The above Figure 55 provides an example of configuration diagram associated with an electrical toothbrush, which is here quite simple, taking the standard representation of configurations and of

<sup>168</sup> We refer to the second paragraph of subsection 4.2 in this matter.

their temporal relationships that we introduced, when events – that induce configuration transitions – are modeled by arrows labelled with the name of the relevant event. Note also that the initial (resp. termination) events in each configuration do not respect this rule since they are just conventionally modeled by small black circles (resp. by white circles containing a black circle).

The configuration diagram is key since it models – from a purely constructional perspective – time, even if it is not immediately obvious to see here. Following the intuition that we developed at the end of the second paragraphs of sections 4.2 and 5.2, one could consider that the next diagrams – i.e. the constructional decomposition & interaction diagrams – are modeling the “constructional space” in which components are evolving. Since space and time are always both required to specify any constructional reality (that takes place somewhere at a certain time), one can easily see that these two diagrams are completely complementary.

### 6.2.3 Constructional Decomposition & Interaction Diagram

Let  $S$  be a system. The *constructional decomposition diagram* associated with  $S$  is then a hierarchical representation of the components of  $S$ , a set  $C_1, C_2, \dots, C_N$  of components being under another component  $D$  in this hierarchy if  $D$  is the result of the integration – in the meaning of Definition 0.5 – of the components  $C_1, \dots, C_N$ <sup>169</sup> ( $C_1, \dots, C_N$  are then classically called “sub-components” of  $D$ ). The *constructional interaction diagrams* associated with  $S$  are then just the different representations – there is one constructional interaction diagram per integration relation involved in the constructional decomposition diagram – of each such integration relationship that does exist between the different components appearing in the hierarchy modeled by the constructional decomposition diagram.

The following Figure 56 now provides an illustrative partial example of constructional decomposition diagram for an electronic toothbrush, where the integration relationships on which such an hierarchy relies are – quite classically – represented by black-squared (resp. white-squared) arrows when the low-level component is mandatory (resp. optional – which allows to model product options using this last syntax –) with respect to the depicted integration relationship.

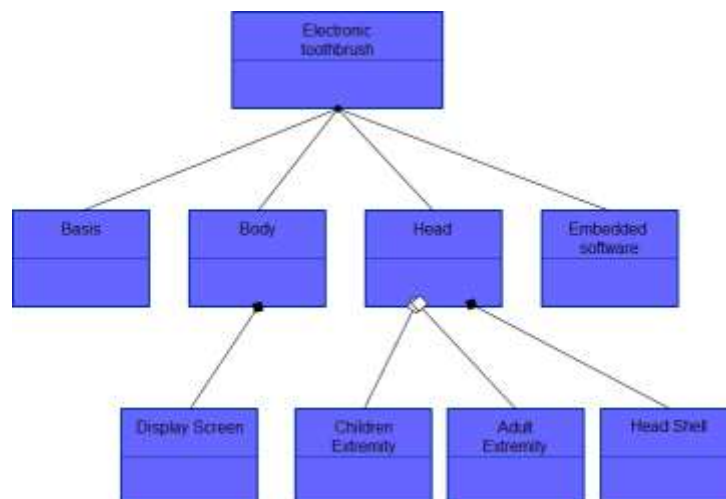
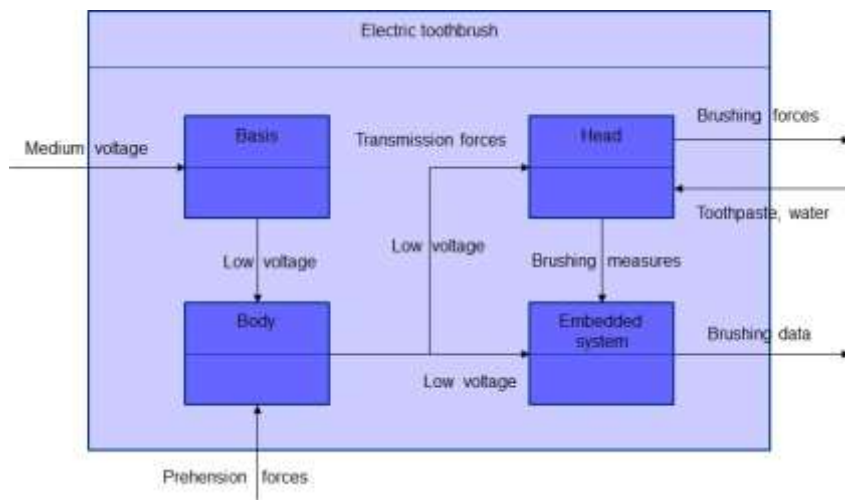


Figure 56 – Example of a constructional decomposition diagram for an electronic toothbrush

<sup>169</sup> Due to our definition of the integration operator, this hierarchy is therefore again an abstraction hierarchy.



We also give an example of a constructional interaction diagram for an electronic toothbrush that can be found in Figure 57. It provides the integration relationship existing between the electronic toothbrush in its whole and its first “sub-components” as they are appearing above in the previous constructional decomposition diagram.



**Figure 57 – Example of a constructional interaction diagram for an electronic toothbrush**

The constructional decomposition of a system  $S$  modeled by the constructional decomposition diagram is also classically called the Product Breakdown Structure (PBS) of  $S$ . Similarly to the Mission or the Functional Breakdown Structures that we introduced in the two last chapters, it provides the exhaustive dictionary of components of the system and has a key role in guaranteeing a common understanding on the constructional scope of a system, which is mandatory for efficient transversal collaboration between the different actors and stakeholders of a system development project.

One must however beware to the good readability of such a view. Just observe in this matter that all the recommendations based on the 7x7x7 rule that we previously gave for the stakeholder and the need architecture diagrams of course also apply – mutatis mutandis – for efficiently modeling the Product Breakdown Structure of a given system.

Note also that Figure 9 in Chapter 0, even if formally functionally oriented, can also be easily re-interpreted as a constructional interaction diagram, here associated with an aircraft: each “box” of this diagram, even if, strictly speaking, representing a first-level sub-function of an aircraft, may indeed also naturally be interpreted as a first-level sub-system of an aircraft.

#### 6.2.4 Constructional Scenario Diagram

Let again  $S$  be a system and  $q(S)$  a configuration of  $S$ . A *constructional scenario diagram* associated with  $S$  and  $q(S)$  is then a dynamic representation of the interactions that are taking place between the components of  $S$  during the period of time which is modeled by  $q(S)$ .

The following Figure 58 shows an example of constructional scenario diagram, associated with the “Children” configuration of an electrical toothbrush. It shows how the electronic toothbrush sends an encouraging message to the end-user when in a children configuration (see subsection 2.4.3 for the

corresponding illustration). We do refer to the suitable paragraph of subsection 2.4.2 for all the details on the semantics of the below representation.

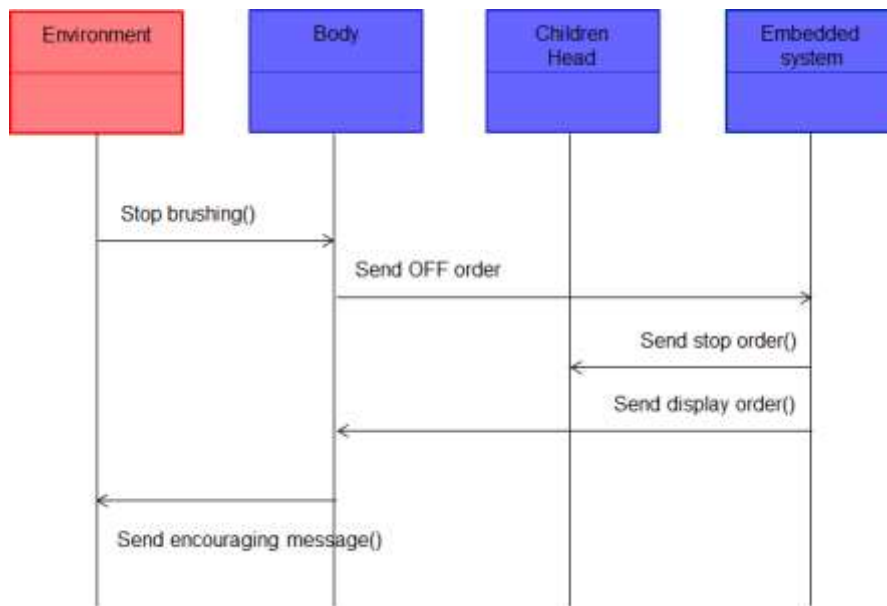


Figure 58 – Example of a constructional scenario diagram for an electronic toothbrush

A constructional scenario diagram provides therefore the explicit “algorithm” which is underlying to the constructional interactions of the system that occur in a given configuration. This is key to finely understand what concretely happens during a given configuration.

Observe finally that this last diagram has also a functional nature since it models in a certain way a “constructional behavior”: one shall thus always beware not to overlap at this level with functional architecture in order to avoid to make two times similar analyses.

### 6.2.5 Constructional Flow Diagram

Let  $S$  be a system. The *constructional flow diagram* associated with  $S$  is a consolidated description of all constructional flows associated with  $S$  and of respectively:

1. their logical relationships,
2. their abstraction relationships (see the last paragraph of Chapter 4).

Hence it plays the role of the constructional “data model”<sup>170</sup> of the system. Note that one also may split this diagram into two diagrams, each of them covering the two above points.

The Figure 59 that follows shows an example of (partial) constructional flow diagram, associated with an electrical toothbrush. It can be constructed by consolidating all the constructional flows that are appearing in the high-level constructional interaction diagram which is provided by Figure 57 for an

<sup>170</sup> Beware that, even if we use the syntax of a data model for the constructional flow diagram, this last diagram is again not really a data model since it does not represent (only) data, but also physical objects, business objects or even informal information that may be exchanged with “humanware” stakeholders of a given system.

electronical toothbrush. Its syntax follows exactly the same principles than for the operational & functional flow diagrams (see the last paragraph of the last two previous chapters).

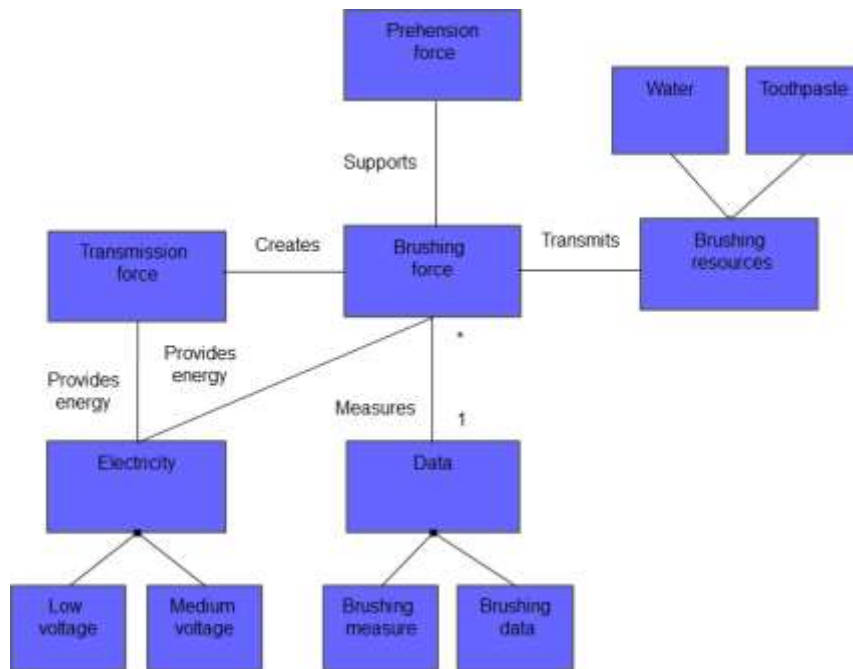


Figure 59 – Example of constructional flow diagram for an electronical toothbrush

As already stated, such a constructional flow diagram defines the constructional flow or object model of a given system. It is therefore completely “dual” to the constructional decomposition, interaction or scenario diagrams since it focuses only on flows and absolutely not on the different components of the system that are producing these flows.

We must therefore emphasize that such a diagram is of high importance since it rationally describes in a consolidated and organized way all inputs and all outputs of the components of a given system. Hence it gives the constructional “dictionary” of the system, that is to say the list of all objects that are constructionally – that is to say concretely – manipulated by the system. Hence this dictionary is of high value for ensuring a common vision between all project actors involved in the architecting process: these actors shall normally – in an ideal world – only use the terms of that dictionary when discussing of a constructional object. One may easily understand that such a principle allows to avoid any ambiguity between the different project actors. It is thus completely key for ensuring a good collaboration between these actors.

# Chapter 7 – Choosing the best Architecture: Trade-off Techniques

## 7.1 Systems Architecting does usually not Lead to a Unique Solution

Systems architecting is not like mathematics since systems architecting issues have commonly never only true or false answers, which may be disorientating for the beginner. A systems architecture process does indeed classically lead to many different possible and valuable solutions. So the core questions in systems architecting are always choice and decision among these various options. The key point is of course to be able to make these choices and decisions in the most rational possible way, which is the main purpose of systems architecting and of this pocket guide.

Note first that systems architecture options can occur in each architectural vision: there are usually lots of choices to do in terms both of needs or requirements prioritization and of missions, functions or components selection. One must in particular always arbitrate between performance and cost in any systems development context, under quality and delay constraints. These key indicators lead to make regularly arbitrations relatively either to the level of coverage both of stakeholder needs and of systems requirements, or on the scope covered by a system's architecture.

Trade-offs – that is to say the specific engineering activities that result in making a choice between various architectural options – are thus permanent within any systems architecting process where decision plays a key role. As a consequence, if one wants these decisions to be as rational as possible, one needs both to organize the trade-off processes in the most efficient way and to have rigorous methods for taking such decisions on the basis of explicit and shared decision criteria. The trade-off techniques presented in this current chapter try to propose a valuable answer to this reasonable and strategic objective of any system development process.

Trade-offs are however never easy. A trade-off is indeed a situation that involves losing one feature of a system in return for gaining another quality or aspect. More colloquially, if one thing increases, some other thing must decrease. Trade-offs can occur for many reasons, including simple geometry (into a given amount of space, one can fit either many small objects or fewer large objects). As already mentioned above, the idea of a trade-off always in a system development context implies a decision that has to be made with full understanding of both the upside and downside of a particular choice, such as when somebody decides whether to invest in stocks (more risky but with a greater potential return) versus bonds (generally safer, but lower potential returns).<sup>171</sup>

Note finally that “human engineering” is a key point when managing trade-offs. As noticed above, making a trade-off in any systems architecting context means eliminating an architectural option for choosing another one, which means privileging some part of the organization against another one, due to the fact that one shall never forget that there are always people behind systems (see the last paragraph of section 2.3). As can be imagined, these human and/or political issues are at the heart of the difficulties when managing trade-offs in practice.

---

<sup>171</sup> This paragraph was highly inspired of the Wikipedia article on “Trade-off” (see [97]).

### **Prioritizing a System Budget: the Data Warehouse Case**

TELBYTE is a leading communication company in Europe. In order to make offers – as well as possible – adapted to their customer needs, the general direction of TELBYTE decided to develop a data warehouse<sup>172</sup> where all existing customer information shall be stored in order to provide suitable data for a number of customer-oriented internal services.

A scoping study showed that around 1,500 data sources were to be connected with the data warehouse in order to be able delivering all the 300 internal services that were requested by around 10 marketing-focused teams within TELBYTE. A dedicated project – named APERO – was then launched on this basis in order to construct the corporate data warehouse for a budget of around 40 million euros.

Unfortunately 9 months after having started, APERO faced a severe budget restriction – due to bad market figures – of 20 %. The project team did not have any idea on how to deliver the expected services to the marketing department within a reduced budget of 32 million euros. A trade-off was clearly required and APERO asked CESAMES to manage it.

CESAMES analyzed the situation and understood quickly that the complexity of the project was too high and value was totally absent of the way the APERO project was working, which explained why the project was not able to manage alone the trade-off it was facing. Several actions were then managed in parallel by CESAMES, during around a month, for creating the conditions of a successful trade-off through a collaborative prioritization workshop. The first one was to reduce the data source complexity: by clustering the initial 1,500 data sources according to their origins, it was possible to only handle 250 data clusters. The second one was to give 100 tokens to each marketing team and to ask them distributing their tokens on the services they were requesting: as an immediate side-effect of putting value at the heart of the problem, their number diminishes from 85 % to arrive at only 50 services. Each team indeed understood that it was necessary to concentrate their tokens on the services with highest value if these services wanted to have a chance to be selected during a collective prioritization process<sup>173</sup>.

As a consequence, the complexity of the trade-off problem to solve passed from  $1,500 \times 300 = 450,000$  possible source / service to  $250 \times 50 = 12,500$  possible cluster / service choices to arbitrate, that is to say a 97 % reduction of complexity which allowed to make successfully a collective arbitration – both of the data clusters to interface with the data warehouse and on the precise service scope to offer – during a one-day collective prioritization workshop involving all concerned business actors. The result achieved during that workshop allowed covering 95 % of the marketing demands within the new 80 % restricted budget<sup>174</sup>, which made everybody happy since renouncements were quite light at the very end.

### **Case study 9 – The Data Warehouse Case**

---

<sup>172</sup> A data warehouse is a huge data basis, constructing with specific technology to guarantee transactional performance.

<sup>173</sup> One can easily understand that any team which would not have concentrated their token on a limited number of choices could not win in a collective prioritization as soon as another team decided to have such a strategy. Hence the best strategy for everybody is concentrating the tokens, which is a classical Nash equilibrium in the meaning of game theory (see [87]).

<sup>174</sup> This illustrates another point that one shall in mind when doing trade-offs. Costs are often distributed with respect to value according to Pareto laws: 20 % of the costs allow offering 80 % of the value when 80 % of the costs does only deliver 20 % of the value. This type of law does explain why the arbitration could work in the data warehouse case and why it was – in some sense (this case indeed required a lot of work) – so easy to achieve.

## 7.2 Trade-off Techniques

### 7.2.1 General Structure of a Trade-Off Process

The objective of any trade-off process is to help its involved stakeholders to make a rational choice among several possible architectural choices, based on shared decision criteria. As a consequence, any trade-off process shall consist in the following main steps:

1. identify the set of stakeholders involved in the trade-off decision process,
2. identify & share the architectures that shall be discussed,
3. define & share the decision criteria to be used,
4. evaluate each possible architecture according to the decision criteria,
5. prioritize the evaluated architectures.

Steps 1 to 3 can be typically prepared by the systems architect alone (or with a small team), without opening too much the circle of stakeholders. Steps 4 and 5 shall however necessarily be achieved through a collective decision process – typically managed in a collective prioritization workshop – as soon as one wants the involved stakeholders to be part of the decision in order them to be collectively engaged in the decision (and respect it). Each of these steps is quickly described below.

The first step – stakeholder identification – is not the easier. We however will here only to the last paragraph of section 2.3 were this question was already discussed. The second step – architecture identification – “just” consists in applying all the systems architecture techniques we presented from Chapter 3 up to Chapter 6 and tracing the valuable architectural options that occur during the systems architecting process. The third step – decision criteria identification – can then rely on need and requirements architecture as discussed in the previous chapters: the “good” decision are indeed typically structuring high level needs and/or requirements. One thus sees that these three first steps totally rely on already discussed systems architecture techniques.

The fourth step – architecture evaluation – is quite easy since the only complementary technical ingredient it requires is to provide an evaluation scale when defining the decision criteria. The main difficulty here lies however not in technique, but in the human dimension of that step, that is to say in workshop facilitation as soon as one is using a collaborative prioritization workshop to achieve that evaluation, as we highly recommend.

The fifth and last step – prioritization – is the more complex since one cannot apply any deterministic protocol to achieve it. Prioritization can only be obtained by a collective discussion, using the results of step 4 as an input, without however being prisoners of them, in order to make collectively the most rational choice, taking into account all the dimensions of the decision that will be brought by the different actors involved in a collective prioritization workshop.

One shall finally point out that such a technique is not making the decision, just helping to prepare it<sup>175</sup>. Ultimately the real decision can indeed only taken by a suitable system governance board. It is thus extremely important to integrate any trade-off technique within a shared governance process where roles are transparently distributed between the people involved in the trade-off protocol as

---

<sup>175</sup> This is why one also speaks of decision-aid techniques for such trade-off techniques.



presented above and the people involved in governance<sup>176</sup>, who are the only ones habilitated to take the decision as already mentioned above.

### 7.2.2 Managing Trade-Offs in Practice

As already discussed in the last subsection, it is a good practice – that we can only highly recommend – to manage trade-offs in practice through collaborative prioritization workshops, involving all key actors that may be impacted by the trade-off decision. We would therefore just like to conclude this short chapter dedicated to trade-off techniques by presenting some illustrations coming from a real collaborative prioritization workshop.

The first one is an example of vote that took place during such a workshop. Each sticker corresponds to a vote of one of the forty-five domain experts who participated to this workshop. These experts were asked to evaluate nine architectural options (represented through columns in Figure 60, the seven first ones being prepared by the systems architect before the workshop, while the two last emerged through the collective discussion during the workshop) in order to know :

1. their value, measured through the covering of five structuring needs (in green in Figure 60),
2. their level of risk, measured through four risk factors (in red in Figure 60).

Each of these value / risk criteria were then evaluated by the involved experts on a low-medium-high scale, the evaluation being concretely achieved by putting a sticker on the relevant location of the evaluation 12 meters x 1,5 meter panel depicted in the below Figure 60.



Figure 60 – Example of a collective vote during a prioritization workshop

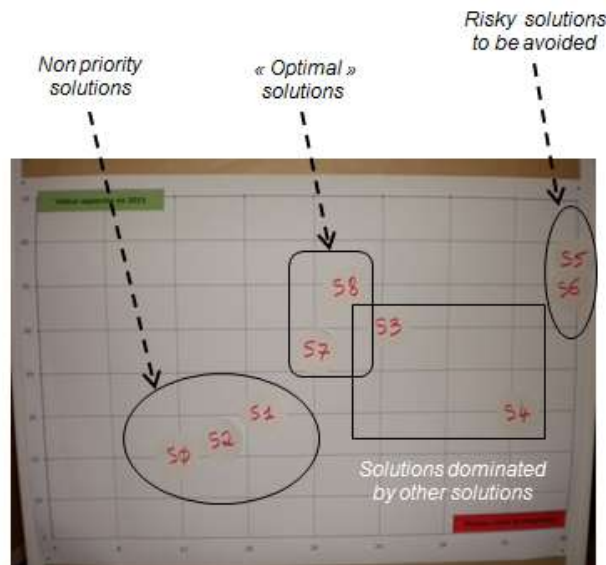
---

<sup>176</sup> It is in particular a key good practice to separate completely the roles and not to mix the people who are preparing the decision with the people who are taking – formally – the decision. Such a role separation indeed allows the governance body not to follow the recommendation brought by the trade-off process, which may happen – even if normally not if the systems architect made a good job – if some strategic decision criteria were not taken into account during the collective prioritization workshop.



It is also interesting to share the synthetic final evaluation that achieved during this collaborative prioritization workshop. The nine architectural options that were analyzed during the workshop are now put on a value / risk matrix – each option being now represented by a simple dot in that matrix – where one can easily compare them (see below Figure 61). One can then easily see that:

- architectures S5 and S6 would lead to a project with the maximal level of risk,
- it is better take S8 rather than S4 and S3, since S8 has more value and less risk than these two other variants, the same conclusion occurring for S7 with respect to S4,
- architectures S0, S1 and S2 have poor business value.



**Figure 61 – Example of a collective evaluation during a prioritization workshop**

On the basis of that analysis which was collectively shared during the prioritization workshop, the participants to the workshop proposed to their governance body to launch a project oriented toward architecture S8, with architecture S7 as a fallback option. Their choice was fully confirmed by the governance committee that took place in the week following the workshop. As a matter of fact, the collective prioritization technique allowed to successively achieve a complex arbitration on a 700 million euro budget dedicated to a strategic system development project!



# Chapter 8 – Conclusion

## 8.1 A first Journey in Systems Architecting

This pocket guide intends to offer to the reader a first journey through systems architecting and we hope that it was appreciated. We thus explored the following topics as provided in Table 11.

Chapters	Topics covered
Chapter 0	Systems Fundamentals
Chapter 1	Systems Architecting Motivations
Chapter 2	CESAM Framework Overview
Chapter 3	Environment Architecture
Chapter 4	Operational Architecture
Chapter 5	Functional Architecture
Chapter 6	Constructional Architecture
Chapter 7	Trade-offs Techniques

Table 11 – Systems architecting topics covered within the pocket guide

However one shall notice that our pocket guide does of course not cover the full domain of systems architecting. We only focus on the fundamentals that every systems architect – even junior – shall know. Thus we put the stress on system modelling, since one just cannot do any systems architecting activity in practice without models. Models do indeed form the language of systems architecting: they can be used to analyze a problem, to describe a system, to elaborate a common vision, to communicate on a solution ... The internal coherence of a system model – that leads to permanent crossed checking between the various diagrams we introduced – is also a very powerful tool for the systems architect in order to be confident in the robustness of an architecture.

## 8.2 The other Key Systems Architecting Topics

We did not discussed in this pocket guide, that was just intended to be an introduction to this huge domain<sup>177</sup>, of many other systems architecting topics – summarized in Table 12 below, that could be addressed by all people who wants to specialize in systems architecting. Among them, we may cite a first group of three important topics – even if usually not the ones to learn on a first step – that should also be part of the common knowledge of any senior systems architect, that is to say:

- *verification & validation* (how to guarantee that a real system satisfies to its needs & requirements and to its descriptions?),
- *project architecture* (how to optimally structure a development project, taking into account the constraints coming from systems architecture?),

---

<sup>177</sup> Systems architecting is a continent to explore, to rephrase an expression due to Marcel Paul Schützenberger, one of the seminal fathers of modern computer science (personal communication).

- *collaborative architecture* (how to organize efficiently the collaborative between the internal and the external actors of a development project?).

In a second group, we may also distinguish the following three quite specialized – and technically much more difficult – topics that may be typically reserved for expert systems architects:

- *safety* (how to integrate dysfunctional analyses within a system architecture?<sup>178</sup>),
- *system family architecture* (how to architecture a full family of systems?),
- *system of systems architecture* (how to architecture a system of systems?).

Note finally that there also some other classical topics such as needs capture techniques, advanced functional architecture, systems architecture simulation, interface architecture or design to X<sup>179</sup> techniques, that could also be placed in that second group.

Other core topics	Specialized topics
Verification & Validation	Safety
Project Architecture	System Family Architecture
Collaborative Architecture	Systems of Systems Architecture

Table 12 – Other systems architecting topics

### 8.3 How to Develop a Systems Architecting Leadership?

Mastering the techniques contained in this pocket guide would clearly be a first necessary important step for any systems architect. However systems architecting cannot and shall not be reduced to systems modeling. The core of systems architecting indeed consists in helping system development teams to make rational & shared optimal choices in complex environment. As a consequence, the main difficulty of systems architecting is not to master modelling in its own, which by the way is not easy. The main difficulty of systems architecting is indeed to be able to create a common vision – involving all project actors – around the system in development with the help of system models.

This last point can only be approached through concrete development projects with real experiences of consensus building. Understanding the “theory” is clearly not enough as soon as one is dealing with human relationships issues which are in fact at the core of systems architecting practice. One shall indeed understand that one cannot manage a convergence protocol, in order to create a shared vision among project actors on a given topic, in the same way than a technical study or a prototype development. When dealing with people, mistakes or bugs are typically forbidden<sup>180</sup>: contrarily to a purely technical problem, human issues shall always be managed with great care. All the complexity of the systems architect job is therefore to be able to manage convergence protocols – with their inherent socio-dynamic difficulties – in complex technical environments<sup>181</sup>.

<sup>178</sup> The question is absolutely not to become a safety specialist, which is a totally different type of expertise, but to be able to efficiently manage the interface between systems architecture and safety.

<sup>179</sup> X can mean cost (design to cost), value (design to value), maintainability (design to maintainability), etc.

<sup>180</sup> Telling to someone to do something, then sometimes after to do something else, then again after to do something again different, will probably never work ...

<sup>181</sup> The mix between socio-dynamics and technique is of course the core difficulty of systems architecting in practice.

To progress in systems architecting, the systems architect shall thus clearly develop its non-technical competency (consensus creation, workshop animation, meeting facilitation, etc.), but which has to be applied in technical complex contexts. In this matter, practice is absolutely fundamental. One can just not achieve developing leadership in systems architecting without confronting the complexity of real development situations. In this matter, one may note that CESAMES is offering dedicated 6 to 10 month on-the-job trainings in systems architecting. Such trainings are especially devoted to that leadership construction through the concrete application of a full systems architecting process on a real system and the achievement of a complete systems architecture specification file (we refer to the corresponding item in the training section on our website: [www.cesames.net](http://www.cesames.net)).

## 8.4 Towards a New Systems Architecture Modelling Language

Last, but not least, we would like to point out that the different diagrams we introduced in this pocket guide were all constructed on the basis of a SysML syntax (see for instance [34]) in order both to avoid disorienting the reader and to let him/her able to use standard modeling tools to practically apply the material he/she will found in this pocket guide.

However one can easily see by reading this pocket guide that such syntax is not perfectly adapted to the needs of systems architecting, due to the differences and/or discrepancies that exist between the different views. The diagrams used to statically specify missions, functions are components together with their interactions are for instance not homogeneous, which is clearly absurd<sup>182</sup>, thus leading to lots of easily avoidable – with a good modelling language – technical difficulties while modelling systems. We would thus like to launch a call to the systems engineering community for creating a architecture-oriented system modelling language with a sound mathematical-based semantics such as the one provided by the CESAM framework within this pocket guide.

---

<sup>182</sup> Other typical issues are also coming from the fact that, in most modelling languages, environment's objects do not have the same type that system's objects. This is just a terribly bad choice since it prevents to make use of the natural recursion brought by a system hierarchy.



# Appendix A – System Temporal Logic

Formal requirements are expressed in *system temporal logic*<sup>183</sup>, a mathematical formalism that we did not present below and that we will discuss in full details in this appendix.

System temporal logic is a formal logic that extends the same classical notion for computer programs (see for instance [8], [11], [45], [54], [58], [69] or [64]). Such a logic intends to specify the sequences of input/output and internal observations that can be made on a formal system whose input, output and internal states sets  $X$ ,  $Y$ ,  $Q$  and timescale  $T$  are fixed. In other terms, system temporal logic specifies the sequences  $O$  of inputs, outputs and internal states values that can be observed among all moments of time  $t$  within the timescale  $T$ , as stated below:

$$O = (O(t)) \text{ for all } t \in T, \text{ where we set } O(t) = (x(t), y(t), q(t))^{184}.$$

It is based on atomic formulae that may be either “TRUE” or equal to  $O(x, y, q)^{185}$  where  $x$  (resp.  $y$  or  $q$ ) is either an element of the input set  $X$  (resp. output set  $Y$  or internal states set  $Q$ ) or equal to some special symbol  $\emptyset$  (that models an arbitrary value), excepted that  $x$ ,  $y$  and  $q$  cannot be all equal to  $\emptyset$ .

$$\text{TRUE}, O(x, y, q) \text{ for all } x \in X \cup \{\emptyset\}, y \in Y \cup \{\emptyset\} \text{ and } q \in Q \cup \{\emptyset\} \text{ with } (x, y, z) \neq (\emptyset, \emptyset, \emptyset).$$

## Equation 2 – Atomic formulae within system temporal logic

System temporal logic will then manipulate logical formulae – i.e. well-formed predicates – that are expressing the expected properties of the sequences of inputs, outputs and state variables of a formal system among all moments of time  $t$  within a considered timescale. Such a logic involves the two following kinds of logical operators (see [3] and [4] for more details):

- Two classical truth-functional operators: AND and NOT,
- Two specific temporal operators  $X$  (neXt) and  $U$  (Until) whose syntax is provided below:
  - $X f$ , meaning that formula  $f$  is fulfilled at next state,
  - $f U g$ , meaning that formula  $f$  is fulfilled until  $g$  becomes fulfilled.

We will provide soon the system semantics of all these different operators. However we now are in position to syntactically define a *temporal formula* as any well-formed logical formula that may be obtained by applying recursively these different logical operators, starting with an atomic formula.

For the sake of simplicity, one may also introduce, on one hand, two other truth-functional operators, OR and  $\rightarrow$  (implies), and on the other hand, two other temporal operators,  $\diamond$  (eventually) and  $\square$

---

<sup>183</sup> The system temporal logic that we present here is a system-adaptation of the simplest temporal logic used in theoretical computer science, which is called LTL (Linear Temporal Logic; see [8], [11], [45], [54], [64] or [69]). However there exists plenty of other more expressive temporal logics (see [11] or [54]) that can also be adapted to a systems engineering context if necessary, depending of the level of expressivity that may be requested.

<sup>184</sup> Using here the formalism of Definition 0.1.

<sup>185</sup> As we will see below,  $O(x, y, z)$  stands for a predicate that will fix the initial value  $x(t_0)$ ,  $y(t_0)$  and  $q(t_0)$  of the input, output and internal states variables to  $x$ ,  $y$  and  $q$  at the initial moment  $t_0$  of the considered timescale.



(always), which can be expressed using the previous operators (hence they do not extend the power of expressivity of the underlying logic), since they allow to state more easily temporal properties:

- Additional truth-functional logical operators:
  - $f \text{ OR } g = \text{NOT} (\text{NOT } f \text{ AND NOT } g)$ ,
  - $f \rightarrow g = \text{NOT } f \text{ OR } g$  ( $f$  implies  $g$ ),
- Additional temporal operators:
  - $\Diamond f = \text{TRUE} \cup f$  ( $f$  will be eventually true at some moment of time in the future),
  - $\Box f = \text{NOT} (\Diamond \text{NOT } f)$  ( $f$  is true at any moment of time).

To end our presentation of system temporal logic, we must of course define the *semantics* of the different previous logical operators. In other terms, we need to explain when a formal system  $S$  whose input, output and internal states sets are  $X$ ,  $Y$  and  $Q$  and whose timescale is  $T$ , will satisfy to a temporal formula constructed with these operators, which is expressed by writing  $S \models f$ , which reads that the system  $S$  satisfies to the formula  $f$  or equivalently that  $f$  is satisfied by  $S$ .

This satisfaction relationship  $\models$ , which provides the semantics of all system temporal formulae, can then be defined inductively according to the following properties (where we systematically set below  $S[t]$  for the system that has the same behavior than  $S$ , but whose initial moment is  $t$  instead of  $t_0$ ):

- $S \models \text{TRUE}$  for any system  $S$ ,
- $S \models O(x,y,q)$  if and only if  $x(t_0) = x$ ,  $y(t_0) = y$  and  $q(t_0) = q$  (when  $x, y, q \neq \emptyset$ )<sup>186</sup>,
- $S \models f \text{ AND } g$  if and only if  $S \models f$  and  $S \models g$ ,
- $S \models \text{NOT } f$  if and only if one does not have  $S \models f$ ,
- $S \models X f$  if and only if  $S[t_0+] \models f$ ,
- $S \models f \cup g$  if and only if  $\exists t \in T$  such that  $S[t] \models g$  and  $S[u] \models f$  for all  $u \in T$  with  $u < t$ .

It is also interesting to provide explicitly the semantics of the two additional temporal operators that we introduced above, as it can be deduced from the previous definitions:

- $S \models \Box f$  if and only if for all  $t \in T$ , one has  $S[t] \models f$ ,
- $S \models \Diamond f$  if and only if there exists  $t \in T$  such that  $S[t] \models f$ .

Our formalism allows to express all usual temporal properties of systems. To be more specific, let us now see how to express a system performance property in this system temporal logic framework. To this purpose, we need first to introduce the two following logical predicates that are here modeling intervals, respectively of input and output values:

$$X(x_0, x_1) = \text{AND } O(x, \emptyset, \emptyset) \text{ for all } x \in [x_0, x_1] ,$$

$$Y(y_0, y_1) = \text{AND } O(x, \emptyset, \emptyset) \text{ for all } y \in [y_0, y_1] .$$

---

<sup>186</sup> Where  $t_0$  stands for the initial moment of the considered timescale  $T$ .

A typical performance property stating for instance that a system must always have its inputs lying between two values, a and b, and its outputs lying between two other values, c and d, as soon as it enters into the internal state q, will then be expressed as follows:

$$\text{Performance} = \Box ( O(\emptyset, \emptyset, q) \rightarrow X(a,b) \text{ AND } Y(c,d) ).$$

We can also provide the example of a maintainability property that is generically stated as follows, expressing simply here that a system that satisfy such a property must always go back to a “normal” state when it enters in a non-“normal” state at a certain moment of time:

$$\text{Maintainability} = \Box ( \text{NOT } O(\emptyset, \emptyset, \text{“normal”}) \rightarrow \Diamond O(\emptyset, \emptyset, \text{“normal”}) ).$$

In the same way, a safety property would finally be generically defined by expressing that a system shall never be in a non-safe state, which can be stated as a system temporal logic invariant:

$$\text{Safety} = \Box ( \text{NOT } O(\emptyset, \emptyset, \text{“non-safe”}) ).$$



# Appendix B – Classical Engineering Issues

We shall now present several classical engineering issues on illustrative examples. These issues are indeed representative of the typical problems addressed by systems architecting. We classified them in two categories: on one hand, *product problems*, referring to purely architectural flaws leading to a bad design of the product, and on the other hand, *project problems*, referring to organizational issues leading to a bad functioning of the project. An overview of these different problems is presented in Table 13 below. The details of our examples & analyzes will be found in the sequel of this appendix.

- **Product problems**
  - **Product problem 1 – The product system model does not capture reality**
    - *Typical issue:* the system design is based on a model which does not match with reality
    - *Example:* the failure of Calcutta subway
  - **Product problem 2 – The product system has undesirable emergent properties**
    - *Typical issue:* a complex integrated system has unexpected and/or undesired emerging properties, coming from a local problem that has global consequences
    - *Example:* the explosion of Ariane 5 satellite launcher during its first flight
- **Project problems**
  - **Project problem 1 – The project system has integration issues**
    - *Typical issue:* the engineering of the system is not done in a collaborative way
    - *Example:* the huge delays of the Airbus A380 project
  - **Project problem 2 – The project system diverts the product mission**
    - *Typical issue:* the project forgets the mission of the product
    - *Example:* the failure of the luggage management system of Denver airport

Table 13 – Examples of typical product and project issues addressed by systems architecting

## B.1 Product problem 1 – The product system model does not capture reality

To illustrate that first product architecture issue, we will consider the Calcutta subway case<sup>187</sup> which occurred when a very strong heat wave (45°C in the shadow) stroke India during summer time. The cockpit touch screens of the subway trains became then completely blank and the subway drivers were therefore not able anymore to pilot anything. As a consequence, the subway company stopped working during a few days, which lead moreover to a huge chaos in the city and to important financial penalties for the subway constructing company, until the temperature came back to normal, when it was possible to operate again the subways as usually.

To understand what happened, the subway designers tested immediately the touch screens, but these components worked fine under high temperature conditions. It took then several months to

---

<sup>187</sup> This case is not public. We were thus obliged to hide its real location and to simplify its presentation, without however altering its nature and its systems architecting fundamentals.

understand the complete chain of events that lead to the observed dysfunctioning, which was – quite surprisingly for the engineers who made the analysis – of systemic nature as we will now see.

The analysis indeed revealed that the starting point of the problem were the bogies, that is to say the mechanical structure which is carrying the subway wheels. Each subway wagon is supported by two bogies, each of them with four wheels. The important point is that all these bogies were basically only made with metal. This metal expanded under the action of high heat, leading to an unexpected behavior of the bogies that we are now in position to explain.

One must indeed also know that to each bogie is attached a braking system. These braking systems are in particular regulated by the central subway computer where a control law was embedded. The control law obliges each local braking system on each bogie to exert a braking force which shall be between two safety lower and upper borders<sup>188</sup>, when braking is initiated. The role of the central computer is thus to ensure that the two safety borders are always maintained during braking, which is achieved by relaxing or increasing the braking force on a given braking system.

The key point was that the underlying control law was not valid at high temperature. This control law was indeed designed – and quite robust in that case – in a Western environment where strong heat never occurs. Hence nobody knew that it was not correct anymore in such a situation.

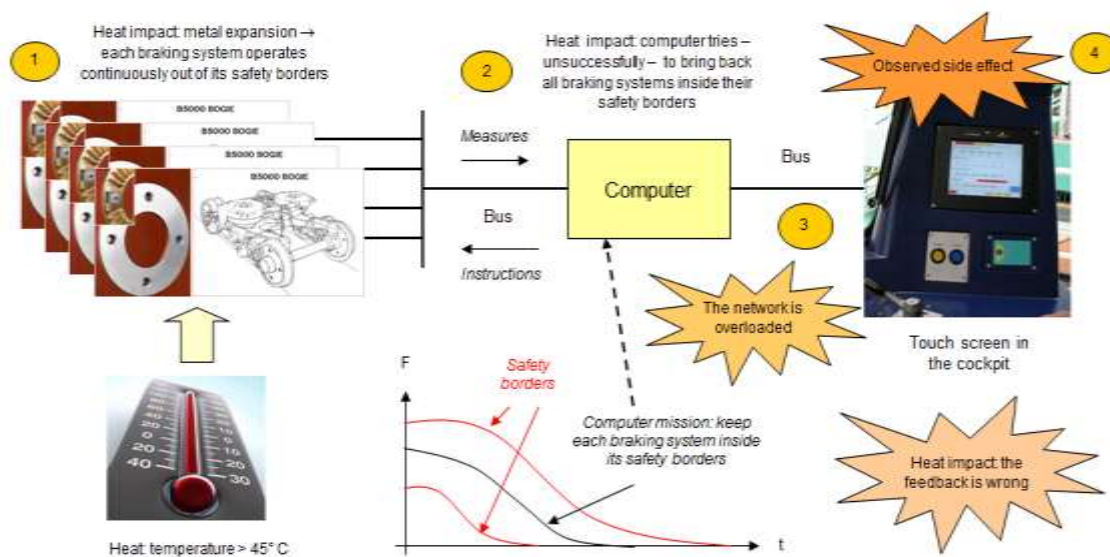


Figure 62 – The Calcutta subway case

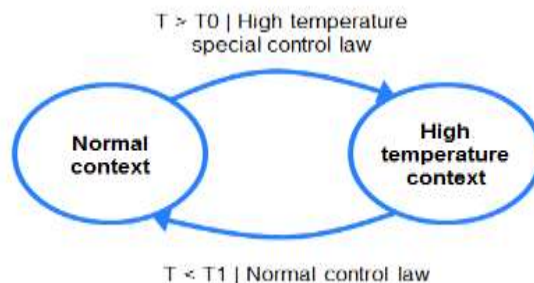
What happened can now be easily explained. The high temperature indeed provoked the same metal expansion among the different subway bogies. Hence all bogies were continuously working out of their safety borders during braking. But the computer was not aware of that situation and continued to try to bring back all braking systems inside their safety borders, applying its fixed control law that was unfortunately false in this new context. As a consequence, there was a permanent exchange of messages between the central computer and the numerous braking systems along the subway. It resulted in an overload of the network which was dimensioned to support such a heavy traffic. The

<sup>188</sup> Braking forces can indeed not be neither too strong (in order to avoid wheels destroying rails), nor too weak (in order to avoid wheel slip which will result in no braking at all).

observed effect on the touch screen was thus just a side effect of this overload, due to the fact that the touch screen is also connected to the central computer by the same network, which explains why nothing wrong was found at the touch screen level.

We can thus see that this case highlights a typical modeling problem, here the fact that the braking control law was false in the Indian high temperature context, but also an integration issue<sup>189</sup>, that ultimately led to an operational failure through a “domino” effect<sup>190</sup>, where an initial local problem progressively propagated along the subway and resulted in a global breakdown of the system. One shall thus remember that it is a key good practice to permanently check and ensure the consistency between a system model and the reality it models, since reality will indeed always be stronger than any model, as illustrated by the Calcutta subway case.

Note finally that this kind of modeling problem is typically addressed by systems architecting, which proposes an answer through “operational architecting” (we refer to both Chapter 3 and Chapter 4 for more details). Such an analysis indeed focuses on the understanding of the environment of the concerned system. In the Calcutta subway, a typical operational analysis would have consisted in considering India as a key stakeholder of the subway and then trying to understand what is different in India, compared to the Western countries where the subway was initially developed. It is then easy to find that very strong heat waves do statistically occur in India each decade which creates an Indian specific “High temperature context” that shall be specifically analyzed. A good operational systems architecting analysis shall then be able to derive the braking system lifecycle, presented in Figure 63, with two states that are respectively modeling normal and high temperature contexts and two transitions that do model the events<sup>191</sup> that create a change of state and of braking control law.



**Figure 63 – The missing operational analysis in the Calcutta subway case**

<sup>189</sup> In that case, all components are indeed working well individually. The problem is a subway system level problem that cannot be found in one single component, but rather in the bad integration of all involved components.

<sup>190</sup> Another classical example of a domino effect is provided by O. de Weck [28] who described the collapse of the redesign of the US F/A-18 aircraft fighter for the Swiss army. This aircraft was initially designed in 1978 for the US Navy as a carrier-based fighter and attacker, with 3,000 flight hour's expectation, missions with average duration of 90 minutes and maximal acceleration of 7.5 g and 15 years of useful life. As a consequence of the neutral policy, inland, small size and mountainous nature of Switzerland, this country wanted a based-land interceptor aircraft, with 5,000 flight hour's expectation, missions with average duration of 40 minutes and maximal acceleration of 9.0 g and 30 years of useful life. Engineers analyzed that it was sufficient to change some non-robust fatigue components near the engine, made in Aluminum, in order to meet Swiss requirements. These components were then redesigned in Titanium. Unfortunately a shift of the center of gravity of the aircraft was then created and one needed to reinforce the fuselage to solve that issue. This other change lead to transversal vibrations that required other reinforcements of weights and modifications in the flight control system. Many changes continued to propagate within the aircraft up to impacting the industrial processes and the organization of the construction factory. 500 grams changes lead thus finally to 10 million dollars modifications that were never expected.

<sup>191</sup> Here the fact that the temperature is higher (resp. lower) than some threshold  $T_0$  (resp.  $T_1$ ) when the braking system is in “Normal” context (resp. “High temperature” context).

Such a diagram is typically an (operational) system model. It looks apparently very simple<sup>192</sup>, but one must understand that introducing the “High temperature” context and the transition that leads to that state will allow avoiding a stupid operational issue and saving millions of euros...

## B.2 Product problem 2 – The product system has undesirable emergent properties

The second product-oriented case study that we will discuss is the explosion of the very first satellite launcher Ariane 5 which is well known due to the remarkable work of the Lions commission, who published a public detailed and fully transparent report on this accident (see [56]). This case was largely discussed in the engineering literature, but its main conclusions were rather focused on how to better master critical real-time software design. We will here present a systems architecting interpretation of that case, which, in the best of our knowledge, was never made up to now.

Let us now remember what happened on June 4, 1996 for the first flight of Ariane 5. First of all, the flight of this satellite launcher was perfect from second 0 up to second 36 after take-off. At second 36.7, there was however a simultaneous failure of the two inertial systems of the launcher that lead at second 37 to the activation of the automatic pilot which misunderstood the error data transmitted by the inertial systems. The automatic pilot corrected then brutally the trajectory of Ariane 5, leading to a mechanical brake of the boosters and thus to the initiation of the self-destruction procedure of the launcher that exploded at second 39.

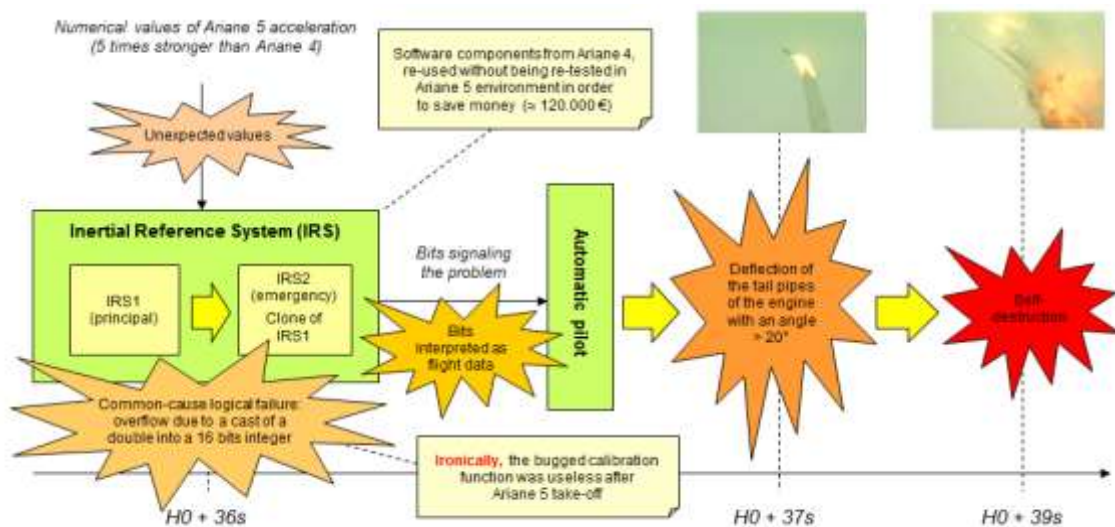


Figure 64 – The Ariane 5 case

As one can easily guess, the cost of this accident was tremendously high and probably reached around 1 billion euros. One knows that the direct cost due to the satellite load lost was around 370 million euros. But there was also an induced cost for recovering the most dangerous fragments of the launcher (such as the fuel stock) that crashed in the (quite difficult to access) Guyana swamps, which took one month of work. Moreover there were huge indirect costs due to Ariane 5 program delaying:

<sup>192</sup> This simplicity is unfortunately an issue for systems architecting. Most of people will indeed agree to the fact that one cannot manipulate partial differential equations without the suitable studies in applied mathematics. But the same people will surely think that no specific competency is required to write down simple operational models such as the one provided by Figure 63, which is unfortunately not the case. We indeed do believe that only good systems architects can achieve such an apparently simple result, which will always be the consequence of a good combination of training and personal skills.



the second flight was only performed one year later and it took three more years to perform the first commercial flight of the launcher, by December 10, 1999.

As already stated, the reason of that tragic accident is fortunately completely analyzed through the Lions commission report (cf. [56]). The origin of the accident could indeed be traced back to the reuse of the inertial reference system (IRS)<sup>193</sup> of Ariane 4. This critical complex software component perfectly worked on Ariane 4 and it was thus identically reused on Ariane 5 without being retested<sup>194</sup> in the new environment. Unfortunately Ariane 5 was a much powerful launcher than Ariane 4 and the numerical values of Ariane 5 acceleration – which are the inputs of the inertial reference system – were five times bigger than for Ariane 4. These values were thus coded in double precision in the context of Ariane 5, when the inertial reference system was designed to only accept single precision integers as inputs. As a consequence, due to the fact that this software system was coded in C<sup>195</sup>, an overflow occurred during its execution. The error codes resulting from that software error were then unfortunately interpreted as flight data by the automatic pilot of Ariane 5, which corrected – one second after receiving these error codes – the trajectory of the launcher from an angle of more than 20°, resulting quite immediately in the mechanical breaking of the launcher boosters and one second after, in the initiation of the self-destruction procedure.

The Ariane 5 explosion is hence a typical integration issue<sup>196</sup>. All its components worked individually perfectly, but without working correctly altogether when integrated. Hence one shall remember that a component of an integrated system is never correct by itself. It is only correct relatively to the set of its interfaced components. When this set evolves, one must thus check that the target component is still properly integrated with its environment, since the fact that the IRS module fulfilled Ariane 4 requirements cannot ensure it fulfils Ariane 5 requirements.

This example also shows that an – usually not researched – emergent property of integration can be death. The Ariane 5 system was indeed *incorrect by design* since the launcher could only explode as it was integrated. In other words, Ariane 5 destruction was embedded in its architecture and it can be

---

<sup>193</sup> An inertial reference system is a software based system that continuously calculates the position, orientation and velocity of a moving object. In the context of Ariane 5, it is thus a critical since most of the other systems depend on its calculations.

<sup>194</sup> The engineer in charge of the IRS proposed to retest it in the new Ariane 5 environment, but it was decided not to follow that proposal in order to save around 120.000 euros of testing costs.

<sup>195</sup> The C programming language is very permissive and does not provide automatic type control. Contrarily a C program will always convert any input value into the type that it manipulates. In the Ariane 5 context, the inertial reference system thus converted automatically all double integer inputs into single integers, according to standard C language rules. This purely syntactic type conversion destroyed the physical meaning of the involved data, leading thus to the observed overflow.

<sup>196</sup> Moreover they were also a number of software engineering mistakes – presented below – that also illustrate the fact that the hardware-software integrated nature of the launcher was not really taken into account by the designers. *Issue 1 – software specificity misunderstanding*: only physical (which are statistical) component failures were considered, but logical (which are systematic) component failures are of totally different nature. Note that this kind of software failure can only be addressed by formal model checking or dissimilar redundancy strategy (which consists in developing two different versions of the software component by two different teams on the basis of the same specification in order to ensure a different distribution of bugs within the two versions); *issue 2 – poor software documentation*: the conditions for a correct behavior of the IRS module were not explicitly documented in the source code; *issue 3 – poor software architecture*: the raise of a local exception in a software component shall normally never imply its global failure.

seen as a purely logical consequence<sup>197</sup> of its integration mode. This extremal – and fortunately rare – case illustrates thus well the real difficulty of mastering integration of complex systems!

Note finally that systems architecting can provide a number of methodological tools to avoid such integration issues. Among them, one can typically cite interface or impact analyses. In the Ariane 5 context, a simple interface type check would for instance allowed seeing that the input types of the inertial reference system were simply not compatible with the expected ones, which would probably permit avoiding a huge disaster!

### B.3 Project problem 1 – The project system has integration issues

Our first “project architecture” issue is the initial Airbus 380 delivery delay, since it is mainly public (see [82] for an extensive presentation of that case study). Let us recall that this aircraft is currently the world’s largest passenger airliner. Its origin goes back in mid-1988 when Airbus engineers began to work in secret on an ultra-high-capacity airplane in order to break the dominance that Boeing had on that market segment since the early 1970s with its 747. It took however a number of years of studies to arrive to the official decision of announcing in June 1994 the creation of the A3XX program which was the first name of the A380 within Airbus. Due to the evolution of the aeronautic market that darkened in that moment of time, it is interesting to observe that Airbus decided then to refine its design, targeting a 15–20 % reduction in operating costs over the existing Boeing 747 family. The A3XX design finally converged on a double-decker layout that provided more passenger volume than a traditional single-deck design, perfectly in line with traditional hub-and-spoke theory as opposed to point-to-point theory that was the Boeing paradigm for large airliners, after conducting an extensive market analysis with over 200 focus groups.

In the beginning of 2000, the commercial history of the A380 – the new name that was then given to the A3XX – began and the first orders arrived to Airbus by 2001. The industrial organization was then put in place between 2002 and 2005: the A380 components are indeed provided by suppliers from all around the world, when the main structural sections of the airliner are built in France, Germany, Spain, and the United Kingdom, for a final assembly in Toulouse in a dedicated integration location. The first fully assembled A380 was thus unveiled in Toulouse by 18 January 2005 before its first flight on 27 April 2005. By 10 January 2006, it flew to Colombia, accomplishing both the transatlantic test, and the testing of the engine operation in high-altitude airports. It also arrived in North America on 6 February 2006, landing in Iqaluit, Nunavut in Canada for cold-weather testing. On 4 September 2006, the first full passenger-carrying flight test took place. Finally Airbus obtained the first A380 flight certificates from the EASA and FAA on 12 December 2006.

During all that period, orders continued to arrive from the airline companies, up to reaching a bit less than 200 cumulated orders, obtained in 2007. The first deliveries were initially – in 2003 – planned for end 2006, with an objective of producing around 120 aircrafts for 2009. Unfortunately many industrial difficulties – that we will discuss below – occurred and it was thus necessary to re-estimate

---

<sup>197</sup> The IRS component of Ariane 5 consisted in two exactly similar software modules (which, as explained in the previous footnote, was of no use due to the logical nature of the failure that repeated similarly in each of these modules). The 36 seconds delay that separated take-off from the crash of the IRS can thus be decomposed in two times 18 seconds that are necessary for each module to logically crash. As a consequence, we can typically state the following architectural “theorem” which illustrates the “incorrection” by design of Ariane 5. *Theorem*: Let us suppose that the IRS component of Ariane 5 has  $N$  similar software modules. The launcher shall then be destroyed at second  $18*N+3$ .

sharply downward these figures each year<sup>198</sup> (cf. Figure 65). The very first commercial A380 was finally produced by end 2007 and instead of 120, only 23 airliners were delivered in 2009.

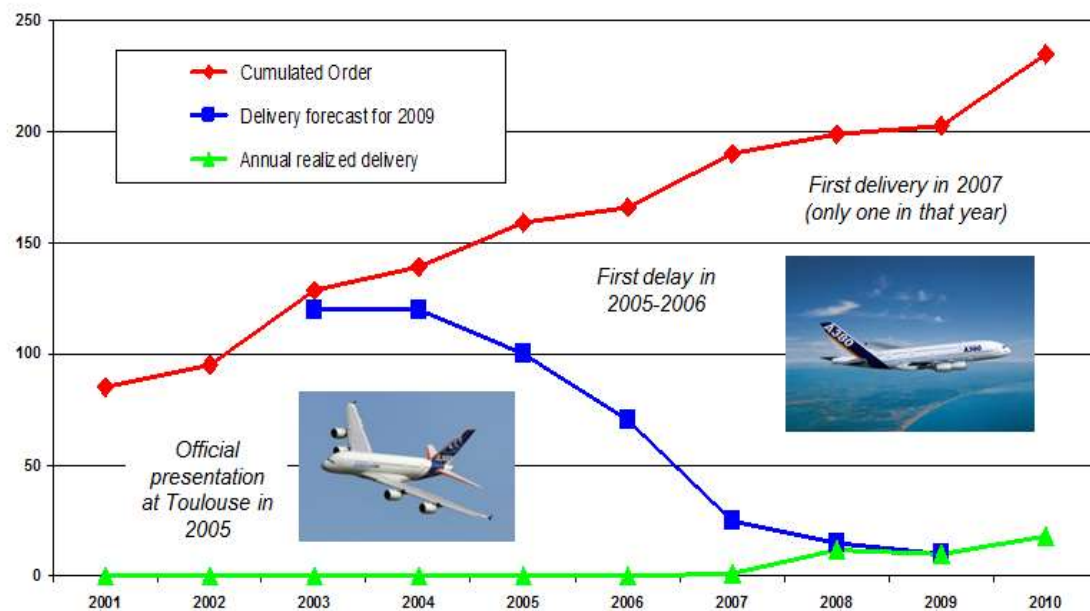


Figure 65 – The Airbus 380 case

These delays had strong financial consequences, since they increased the earnings shortfall projected by Airbus through 2010 to € 4.8 billion. It is thus clearly interesting to try to better understand the root causes of such an important failure.

The source of these delays seems to be connected to the incoherence of the 530 km (330 miles) long electrical wiring, produced in France and Germany. Airbus cited in particular as underlying causes the complexity of the cabin wiring (98,000 wires and 40,000 connectors), its concurrent design and production, the high degree of customization for each airline company, and failures in configuration management and change control. These electrical wiring incoherencies were indeed only discovered at the final integration stage in Toulouse<sup>199</sup>, which was of course much too late...

The origin of this problem could be traced back to the fact that German and Spanish Airbus facilities continued to use CATIA® version 4, while British and French sites migrated to version 5. This caused overall configuration management problems, at least in part because wire harnesses manufactured using aluminum, rather than copper, conductors necessitated special design rules including non-standard dimensions and bend radii. This specific information was not easily transferred between

<sup>198</sup> Airbus announced the first delay in June 2005 and notified airlines that deliveries would be delayed by six months. This reduced the total number of planned deliveries by the end of 2009 from about 120 to 90–100. On 13 June 2006, Airbus announced a second delay, with the delivery schedule slipping an additional six to seven months. Although the first delivery was still planned before the end of 2006, deliveries in 2007 would drop to only 9 aircraft, and deliveries by the end of 2009 would be cut to 70–80 aircraft. The announcement caused a 26 % drop in the share price of Airbus' parent, EADS, and led to the departure of EADS CEO, Airbus CEO, and A380 program manager. On 3 October 2006, upon completion of a review of the A380 program, Airbus new CEO announced a third delay, pushing the first delivery to October 2007, to be followed by 13 deliveries in 2008, 25 in 2009, and the full production rate of 45 aircraft per year in 2010.

<sup>199</sup> There exists a video where one can see a poor technician in Toulouse that is unable to connect two electrical wires coming from two different sections of the aircraft, due to a lack of 20 centimeters of wire.

versions of the software, which lead to incoherent manufacturing and at the very end, created the integration issue in Toulouse. On a totally different dimension, the strong customization of internal equipment also induced a long learning curve for the teams and thus other delays.

Independently of these “official” causes, there are other plausible deep causes coming from cultural conflicts among the dual-headed French & German direction of Airbus and lack – or break – of communication between the multi-localized teams of the European aircraft manufacturer.

As systems architects, we may summarize such problems as typical “project architecture” issues. The issues finally observed at product level are indeed only consequences of lack of integration within the project, that is to say project interfaces – to use a system vocabulary – that were not coherent, which simply refer, in more familiar terms, to project teams or project tools that were not working coherently altogether. It is thus key to have a robust project architecture in the context of complex systems development since the project system is always at least as complex as the product system it is developing. Unfortunately it is a matter of fact that the energy spent with technical issues is usually much more equivalent than the energy spent on organizational issues, which often ultimately lead to obviously bad project architectures in complex systems contexts, resulting at the very end into bad technical architectures in such contexts.

#### **B.4 Project problem 2 – The project system diverts the product mission**

As a last example of different types of project issues, we will now consider the case of the Denver airport luggage management system failure, which is fortunately well known due to the fact that it is completely public (see for instance [27] or [72]).

Denver airport is currently the largest airport in the United States in terms of total land area and the 6<sup>th</sup> airport in the United States (the 18<sup>th</sup> in the world) in terms of passenger traffic. It was designed in order to be one of the main hub for United Airlines and the main hub for two local airlines.

The airport construction officially started in September 1989 and it was initially scheduled to open on October 29, 1993. Due to the very large distance between the three terminals of the airport and the need of fast aircraft rotations for answering to its hub mission, the idea of automating the luggage management emerged in order to provide quick plane inter-connections to travelers. United Airlines was the promoter of such a system which was already implemented in Atlanta airport, one of their other hubs. Since Denver airport was intended to be much wider, the idea transformed in using the opportunity of Denver’s new airport construction to improve Atlanta’s system in order to create the most efficient & innovative luggage management system in the world<sup>200</sup>. It was indeed expected to have 27 kilometers of transportation tracks, with 9 kilometers of interchange zones, on which were circulating 4.000 remote-controlled wagons at a constant transportation speed of 38 km/h for an average transportation delay of 10 minutes, which was completely unique.

The luggage management system project started in January 1992, a bit less than 2 years before the expected opening of the airport. During one year, the difficulties of this specific project were hidden since there were many other problems with more classical systems. However at beginning 1993, it became clear that the luggage management system could not delivered at schedule and Denver’s

---

<sup>200</sup> The idea was to have a fully-automated luggage transportation system, with new hardware & software technologies, that was able to manage very large volumes of luggage.

major was obliged to push back the opening date, first to December 1993, then to March 1994 and finally again, to May 15, 1994.

Unfortunately the new automated luggage management system continued to have strong problems. In April 1994, the city invited reporters to observe the first operational test of the new automated baggage system saw instead disastrous scenes where this new system was just destroying luggage, opening and crashing their contents before them. The major was then obliged to cancel sine die the opening date of the airport. As one can imagine, no airline – excepted United – wanted to use the new “fantastic” system, which obliged to abandon the idea of a global luggage management system for the whole airport. When the airport finally opened in February 28, 1995, thus only United Airlines terminal was equipped with the new luggage management system, when the other terminal<sup>201</sup> that also opened was simply equipped with totally classical systems (that is to say tugs and carts).

In 1995, the direct additional costs due to this failure were of around 600 M\$, leading to more than one billion dollars over cost at the very end. Moreover the new baggage system continued to be a maintenance hassle<sup>202</sup>. It was finally terminated by United Airlines in September 2005 and replaced by traditional handlers manually handling cargo and passenger luggage. A TV reporter who covered the full story concluded quite interestingly<sup>203</sup> that “it took ten years, and tons of money, to figure out that big muscle, not computers, can best move baggage”.

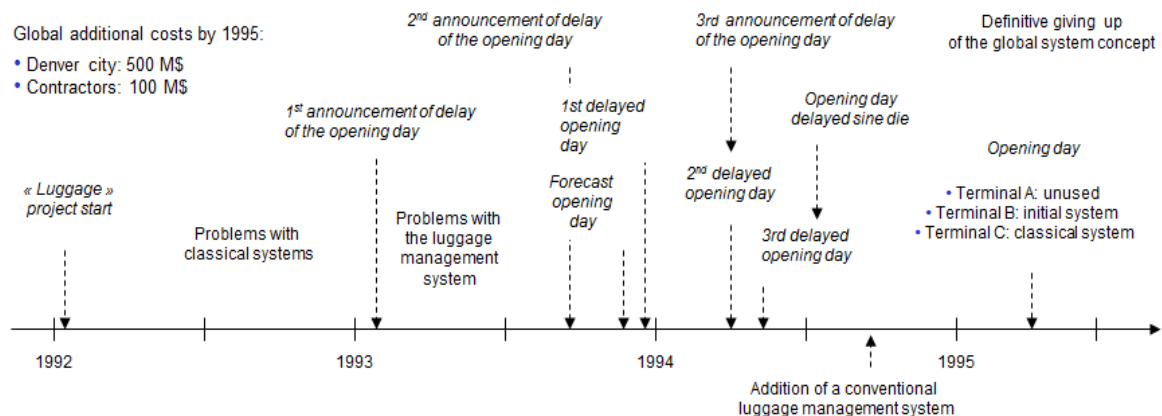


Figure 66 – The Denver luggage management system case

When one looks back to that case, it is quite easy to understand why this new automatic luggage system collapsed. There were first too many innovations<sup>204</sup>. It was indeed both the first global automated system, the first automatic system that was managing oversize luggage (skis!), the first system where carriages did not stop during their service<sup>205</sup>, the first system supported by a computer

<sup>201</sup> It was initially not possible to open the last airport terminal due to the time required for changing the already constructed automatic luggage management system to a standard manual one.

<sup>202</sup> Its nickname quickly became the “luggage system of hell”!

<sup>203</sup> That one must always remember that a system can be formed of people only doing manual operations. In most of cases, good systems architectures are hybrid with both automatic and manual parts. In Denver’s case, the best solution was however purely manual, which United Airlines took ten years to understand.

<sup>204</sup> As a matter of fact, it is interesting to know that Marcel Dassault, the famous French aircraft engineer, was refusing all projects with more than one innovation! A strategy that worked quite well for him.

<sup>205</sup> Suitcases were automatically thrown with a catapult... which easily explains the high ratio of crashes.

network and the first system with a fleet of radio-localized carriages. There was also an underlying huge increasing of the complexity: compared to the similar existing Atlanta system, the new luggage management system was 10 times faster, had 14 times the maximal known capacity and managed 10 times more destinations. Moreover the project schedule was totally irrelevant with respect to the state-of-the-art: due to the strong delay pressure, no physical model and no preliminary mechanical tests were done, when the balancing of the lines required two years in Atlanta.

It is thus quite easy to understand that the new luggage management system could only collapse. In some sense, it collapsed because the project team totally lost of sight the mission of that system, which was just to transport luggage quickly within the airport and at the lowest possible cost, with a strong construction constraint due to the fact that there were only less than 2 years to implement the system. A simple systems architecture analysis would probably concluded that the best solution was not to innovate and to use simply people, tugs and carts, as usual. This case study illustrates thus quite well a very classical project issue where the project system forgets the mission of the product system and replaces it by a purely project-oriented mission<sup>206</sup> that diverts the project from achieving the product system mission. Any systems architect must thus always have in mind this example in order to avoid the same issue to occur on its working perimeter.

---

<sup>206</sup> In Denver's luggage management system case, "creating the world most innovative luggage management system" indeed became the only objective of the project (which is not a product, but a project system mission).

## Appendix C – Some Systems Modeling Good Practices

- **Good practice 1:** an architectural model shall be done **to solve a specific problem**.
- **Good practice 2:** modeling activities shall be fully **part** of any system design project and this from the very beginning of the project
- **Good practice 3:** be clear about the advantages provided by modeling and adopt a **simplicity principle** coupled with a permanent **methodological doubt** (why do we need that? what is it for?)
- **Good practice 4:** the natural evolution of a system makes **very difficult to define its « true » operational reality**: never hesitate therefore to use all the methods and tools which seem adapted to settle all operational uncertainties
- **Good practice 5:** it is necessary to integrate all **technical and human dimensions** of the system to be built, and this from the very beginning of the project
- **Good practice 6:** the more a system modeling is done in an **iterative way** – based on a permanent questioning between the stakeholders and the systems architect – the more it will be effective
- **Good practice 7: “black-box” modeling, without any feedback from the customer, must be banished:** stakeholders must be involved in the architectural process
- **Good practice 8:** the **validation** of an architectural model is a **permanent process** which must create trust in the model, from the points of view of both the architect and the domain engineers & customers who will verify & validate it
- **Good practice 9:** the achievement, as soon as possible, of a **“simple, but not simplistic”, preliminary model** which “works” is fundamental in order to make the architectural approach credible by the stakeholders: details can typically be added later if necessary (a coarse-grained model can turn out to be sufficient)
- **Good practice 10:** a **complete & coarse-grained coherent model** is more important than an accumulation of details, which are often not relevant to the project objectives
- **Good practice 11: systems architect expertise is primary:** one must avoid beginners who know only how to use modeling tools (which is very different from architecting)
- **Good practice 12:** the developed architectural models shall be considered as **assets of the organization** and managed coherently, in configuration & traceability and in space & time, by the organization in charge of the system design.

Table 14 – Some good systems modeling practices<sup>207</sup>

---

<sup>207</sup> Adapted from a personal communication of Professor Jacques Printz.





# Acknowledgements

CESAMES Association and the CESAM Community would like to thank the numerous companies and architects without whom the CESAM Framework would have never existed. All concepts and methods presented in this pocket guide were indeed prototyped and progressively developed during various missions in close contact with their real concrete systems and problems.

The second acknowledgement goes to the Ecole Polytechnique and to the sponsors of its industrial chair “Engineering of Complex Systems”, that is to say Dassault Aviation, Naval Group, Direction Générale de l’Armement (DGA) and, last but not least, Thales. This pocket guide is indeed, for CESAMES Association and the CESAM Community, the outcome of a long and progressive research and maturity process that was initiated 15 years ago with the creation of the chair.

# References

- [1] Abts C., Boehm B., Brown W. A., Chulani S., Clark B.K., Horowitz E., Madachy R., Reifer D.J., Steece B., *Software Cost Estimation with COCOMO II* (with CD-ROM), Englewood Cliffs, Prentice-Hall, 2000
- [2] Aiguier M., Golden B., Krob D., *Complex Systems Architecture and Modeling*, [dans Posters of the first international conference on "Complex Systems Design & Management" (CSDM 2010), M. Aiguier, F. Bretaudeau, D. Krob, Eds.], 16 pages, 2010
- [3] Aiguier M., Golden B., Krob D., *Modeling of Complex Systems II : A minimalist and unified semantics for heterogeneous integrated systems*, Applied Mathematics and Computation, **218**, (16), 8039-8055, doi : 10.1016/j.amc.2012.01.048, 2012
- [4] Aiguier M., Golden B., Krob D., *An adequate logic for heterogeneous systems*, [dans Proceedings of the 18th International Conference on Engineering of Complex Computer Systems (ICECCS' 2013), Y. Liu, A. Martin, Eds.], IEEE, 2013
- [5] ANSI/GEIA, *ANSI/GEIA EIA-632 – Processes for engineering a system*, 2003
- [6] ANSI/IEEE, *ANSI/IEEE 1471-2000 – Recommended Practice for Architecture Description of Software-Intensive Systems*, 2000
- [7] Antonopoulos J., *The great Minoan eruption of Thera volcano and the ensuing tsunamis in the Greek archipelago*, Natural hazards, 5 (2), 153-168, 1992
- [8] Artale A., *Formal Methods – Lecture III: Linear Temporal Logic*, Free University of Bolzano
- [9] Aslaksen E.W., *The changing nature of engineering*, McGraw-Hill, 1996
- [10] Aslaksen E., Belcher R., *Systems engineering*, Prentice Hall, 1992
- [11] Baier C., Katoen J.P., *Principles of Model Checking*, MIT Press, 2008
- [12] Barwise J., *Handbook of mathematical logics*, North-Holland, 1972
- [13] Bellman R.E., *Dynamic Programming*, Princeton University Press, 1957
- [14] Berrebi J., Krob D., *How to use systems architecture to specify the operational perimeter of an innovative product line ?*, [dans "Proceedings of INCOSE International Symposium of 2012 (IS 2012)", M. Celentano, Ed.], 15 pages, INCOSE, 2012
- [15] Blanchard B.S., Fabricky W.J., *Systems engineering and analysis*, Prentice Hall, 1998
- [16] Bliudze S., Krob D., *Towards a Functional Formalism for Modelling Complex Industrial Systems*, [dans "European Conference on Complex Systems" (ECCS'05), P. Bourguine, F. Képès, M. Schoenauer, Eds.], (article 193), 20 pages, 2005
- [17] Bliudze S., Krob D., *Towards a Functional Formalism for Modelling Complex Industrial Systems*, ComPlexUs, Special Issue : Complex Systems - European Conference - November 2005 - Selected Papers - Part 1, **2**, (3-4), 163-176, 2006

- [18] Bliudze S., Krob D., *Modeling of Complex Systems - Systems as data-flow machines*, Fundamenta Informaticae, Special Issue : Machines, Computations and Universality, **91**, 1-24, 2009
- [19] Boehm B., *Software Engineering Economics*, Englewood Cliffs, Prentice-Hall, 1981
- [20] Booch G., Jacobson I., Rumbaugh J., *The Unified Modeling Language Reference Manual*, Second Edition, Addison-Wesley, 2004
- [21] Börger E., Stärk R., *Abstract state machines*, Springer, 2003
- [22] Caseau Y., Krob D., Peyronnet S., *Complexité des systèmes d'information : une famille de mesures de la complexité scalaire d'un schéma d'architecture*, Génie Logiciel, **82**, 23-30, 2007
- [23] Cha P.D., Rosenberg J., Dym C.L., *Fundamentals of modeling and analyzing engineering systems*, Cambridge University Press, 2000
- [24] Chalé Gongora H. G., Doufene A., Krob D., *Complex Systems Architecture Framework. Extension to Multi-Objective Optimization*, 3rd international conference on "Complex Systems Design & Management" (CSDM 2012), Y. Caseau, D. Krob, A. Rauzy, Eds.], Springer Verlag, 105-123, 2012
- [25] Cousot P., Cousot R., *Abstract Interpretation*, Symposium on Models of Programming Languages and Computation, ACM Computing Surveys, 28(2):324-328, June 1996
- [26] Dauron A., Douffène A. , Krob D., *Complex systems operational analysis - A case study for electric vehicles*, [dans Posters of the 2nd international conference on "Complex Systems Design & Management" (CSDM 2011), O. Hammami, D. Krob, J.L. Voirin, Eds.], 18 pages, 2011
- [27] de Neufville R., *The Baggage System at Denver: Prospects and Lessons*, Journal of Air Transport Management, Vol. 1, No. 4, Dec., 229-236, 1994
- [28] de Weck O., *Strategic Engineering – Designing systems for an uncertain future*, MIT, 2006
- [29] de Weck O., Krob D., Liberti L., Marinelli F., *A general framework for combined module- and scale-based product platform design*, [dans Proceedings of the "Second International Engineering Systems Symposium", D. Roos, Ed.], 15 pages, MIT, 2009
- [30] de Weck O.L., Roos D., Magee C.L., *Engineering systems – Meeting human needs in a complex technological world*, The MIT Press, 2011
- [31] Doufène A., Krob D., *Sharing the Total Cost of Ownership of Electric Vehicles : A Study on the Application of Game Theory*, [dans Proceedings of INCOSE International Symposium of 2013 (IS2013)], 2013
- [32] Doufène A., Krob D., *Model-Based operational analysis for complex systems - A case study for electric vehicles*, [dans Proceedings of INCOSE International Symposium of 2014 (IS2014)], 2014
- [33] Doufène A., Krob D., *Pareto Optimality and Nash Equilibrium for Building Stable Systems*, IEEE International Systems Conference, 2015
- [34] Friedenthal S., Moore A.C., Steiner R., *A Practical Guide to SysML : the Systems Modeling Language*, Morgan Kaufmann OMG Press, 2012

- [35] Giakoumakis V., Krob D., Liberti L., Roda F., *Optimal technological architecture evolutions of Information Systems*, [dans Proceedings of the first international conference on "Complex Systems Design & Management" (CSDM 2010), M. Aiguier, F. Bretaudeau, D. Krob, Eds.], 137-148, Springer Verlag, 2010
- [36] Giakoumakis V., Krob D., Liberti L., Roda F., *Technological architecture evolutions of Information Systems: trade-off and optimization*, Concurrent Engineering: Research and Applications, Volume 20, Issue 2, 127-147, 2012
- [37] Gruhl W., *Lessons Learned, Cost/Schedule Assessment Guide*, Internal presentation, NASA Comptroller's office, 1992
- [38] Grady J.O., *System Requirements Analysis*, Elsevier, 2006
- [39] Grady J.O., *System Verification – Proving the Design Solution Satisfies the Requirements*, Elsevier, 2007
- [40] Hofstadter, Douglas R., *Gödel, Escher, Bach: An Eternal Golden Braid*, Basic Books, 1979
- [41] Honour E.C., *Understanding the value of systems engineering*, INCOSE 2014 International Symposium, Vol. 14, 1207–1222, Toulouse, June 20-24, 2014, France, INCOSE, 2004; accessible at <http://www.seintelligence.fr/content/images/2015/12/ValueSE-INCOSE04.pdf>
- [42] IEEE, *IEEE 1220-2005 – Standard for Application and Management of the Systems Engineering Process*, Institute of Electrical and Electronics Engineers, 2005
- [43] INCOSE, *Systems Engineering Handbook, A guide for system life cycle processes and activities*, INCOSE, January 2011,
- [44] ISO/IEC/IEEE, *ISO/IEC/IEEE 15288:2015 – Systems and software engineering -- System life cycle processes*, May 2015
- [45] Katoen J.P., *LTL Model Checking*, University of Twente, 2009
- [46] Korzybski A., *Science and Sanity: an Introduction to Non-Aristotelian Systems and General Semantics*, International Nonaristotelian, 1950
- [47] Kossiakoff A., Sweet W.N., *Systems engineering – Principles and practice*, Wiley, 2003
- [48] Krob D., *Modelling of Complex Software Systems : a Reasoned Overview*, [dans ``26th IFIP WG 6.1 International Conference on Formal Methods for Networked and Distributed Systems" (FORTE'2006), E. Najm, J.-F. Pradat-Peyre, V. Vigié Donzeau-Gouge, Eds.], Lecture Notes in Computer Science, **4229**, 1-22, Springer Verlag, 2006 (Invited speaker)
- [49] Krob D., *Architecture of complex systems : why, what and how ?*, [dans Proceedings of ``COgnitive systems with Interactive Sensors (COGIS'07)", H. Aghajan, J.P. Lecadre, R. Reynaud, Eds.], Stanford University, 1 page, 2007 (Invited speaker)
- [50] Krob D., *Comment sécuriser la conception et le déploiement des logiciels métiers par une démarche d'architecture collaborative ?*, [in ``Journée Française des Tests Logiciels", B. Homès, Ed.], 23 pages, CFTL, 2008
- [51] Krob D., *Éléments d'architecture des systèmes complexes*, [in "Gestion de la complexité et de l'information dans les grands systèmes critiques", A. Appriou, Ed.], 179-207, CNRS Editions, 2009

- [52] Krob D., *Eléments de systématique – Architecture de systèmes*, [in Complexité-Simplexité, A. Berthoz - J.L. Petit, Eds.], Editions Odile Jacob, 2012
- [53] Krob D., *Eléments de modélisation systématique*, [in L’Energie à découvert, R. Mossery - C. Jeandel, Eds.], CNRS Editions, 2013
- [54] Kröger F., Merz S., *Temporal Logic and State Systems*, Springer, 2008
- [55] Lamport L., *Specifying systems – The TLA+ language and tools for hardware and software engineers*, Addison-Wesley, 2003
- [56] Lions J.L., *Ariane 5 – Flight 501 Failure – Report by the Inquiry Board*, ESA, 1996
- [57] Maier M.W., Rechtin E., *The art of systems architecting*, CRC Press, 2002
- [58] Manna Z., Pnueli A., *The Temporal Logic of Reactive and Concurrent Systems*, Springer, 1992
- [59] Marwedel P., *Embedded system design*, Kluwer, 2003
- [60] Meadows D.H., Meadows D.L., Randers J., Berhens W.W. III, *The Limits to Growth*, Universe Books, 1972
- [61] Meinadier J.P., *Ingénierie et intégration de systèmes*, Lavoisier, 1998
- [62] Meinadier J.P., *Le métier d’intégration de systèmes*, Lavoisier, 2002
- [63] Miles L.D., *Techniques of value analysis and engineering*, McGraw-Hill, 1972
- [64] Murray R.M., *Linear Temporal Logic*, California Institute of Technology, 2012
- [65] Murschel A., *The Structure and Function of Ptolemy's Physical Hypotheses of Planetary Motion*, Journal for the History of Astronomy, xxvii, 33–6, 1995
- [66] NASA, *Systems Engineering Handbook*, 2007-edition, NASA/SP-2007-6105, 2007
- [67] Novikova T., Papadopoulos G.A., McCoy F.W., *Modelling of tsunami generated by the gian late bronze age eruption of Thera, south Aegean see, Greece*, Geophysical Journal International, 186 (2), 665-680, 2011
- [68] Parnell G.S., Driscoll P.J., Henderson D.L., *Decision marking in systems engineering and management*, Wiley, 20011
- [69] Pnueli A., *The temporal logics of programs*, IEEE 54<sup>th</sup> Annual Symposium on Foundations of Computer Science (FOCS), 46-57, IEEE, 1977
- [70] Printz J., *Productivité des programmeurs*, Hermès, 2001
- [71] Sage A.P., Armstrong J.E., *Introduction to systems engineering*, Wiley, 2000
- [72] Schloh M., *The Denver International Airport automated baggage handling system*, Cal Poly, Feb 16, 1996
- [73] Severance F.L., *System modeling and simulation – An introduction*, Wiley, 2001
- [74] Sillitto H., *Architecting systems – Concepts, principles and practice*, College Publications, 2014
- [75] Simon H., *The Architecture of Complexity*, Proceedings of the American Philosophica, 106 (6), 467-482, December, 1962

- [76] Simpson T.W., Siddique Z., Jiao J.R., *Product platform and product family design, Methods and applications*, Springer Verlag, 2006
- [77] The Open Group, *TOGAF® Version 9.1 – The Book*, The Open Group, 2011
- [78] Turner W.C., Mize J.H., Case K.H., Nazemetz J.W., *Introduction to industrial and systems engineering*, Prentice Hall, 1978
- [79] Valerdi R., *The Constructive Systems Engineering Cost Model (COSYSMO), Quantifying the Costs of Systems Engineering Effort in Complex Systems*, VDM Verlag Dr. Muller, 2008
- [80] von Bertalanffy K.L., *General System Theory : Foundations, Development, Applications*, George Braziller, 1976
- [81] White S.A., Miers D., *BPMN Modeling and Reference Guide, Understanding and Using BPMN, Future Strategies*, 2008
- [82] Wikipedia, *Airbus A380*, [https://en.wikipedia.org/wiki/Airbus\\_A380](https://en.wikipedia.org/wiki/Airbus_A380)
- [83] Wikipedia, *COCOMO*, <https://en.wikipedia.org/wiki/COCOMO>
- [84] Wikipedia, *COSYSMO*, <https://en.wikipedia.org/wiki/COSYSMO>
- [85] Wikipedia, *Deferent and epicycle*, [https://en.wikipedia.org/wiki/Deferent\\_and\\_epicycle](https://en.wikipedia.org/wiki/Deferent_and_epicycle)
- [86] Wikipedia, *Enterprise architecture framework*, [https://en.wikipedia.org/wiki/Enterprise\\_architecture\\_framework](https://en.wikipedia.org/wiki/Enterprise_architecture_framework)
- [87] Wikipedia, *Nash equilibrium*, [https://en.wikipedia.org/wiki/Nash\\_equilibrium](https://en.wikipedia.org/wiki/Nash_equilibrium)
- [88] Wikipedia, *Occam's razor*, [https://en.wikipedia.org/wiki/Occam%27s\\_razor](https://en.wikipedia.org/wiki/Occam%27s_razor)
- [89] Wikipedia, *Organon*, <https://en.wikipedia.org/wiki/Organon>
- [90] Wikipedia, *OSI model*, [https://en.wikipedia.org/wiki/OSI\\_model](https://en.wikipedia.org/wiki/OSI_model)
- [91] Wikipedia, *Predicate*, [https://en.wikipedia.org/wiki/Predicate\\_\(mathematical\\_logic\)](https://en.wikipedia.org/wiki/Predicate_(mathematical_logic))
- [92] Wikipedia, *Systems architecture*, [https://en.wikipedia.org/wiki/Systems\\_architecture](https://en.wikipedia.org/wiki/Systems_architecture)
- [93] Wikipedia, *Systems biology*, [https://en.wikipedia.org/wiki/Systems\\_biology](https://en.wikipedia.org/wiki/Systems_biology)
- [94] Wikipedia, *Systems engineering*, [https://en.wikipedia.org/wiki/Systems\\_engineering](https://en.wikipedia.org/wiki/Systems_engineering)
- [95] Wikipedia, *Systems psychology*, [https://en.wikipedia.org/wiki/Systems\\_psychology](https://en.wikipedia.org/wiki/Systems_psychology)
- [96] Wikipedia, *Systems theory*, [https://en.wikipedia.org/wiki/Systems\\_theory](https://en.wikipedia.org/wiki/Systems_theory)
- [97] Wikipedia, *Trade-off*, <https://en.wikipedia.org/wiki/Trade-off>
- [98] Wikiquote, *Systems engineering*, [https://en.wikiquote.org/wiki/Systems\\_engineering](https://en.wikiquote.org/wiki/Systems_engineering)
- [99] Winskel G., *The formal semantics of programming languages – An introduction*, The MIT Press, 1993
- [100] Zeigler B.P., Praehofer H., Kim T.G., *Theory of modeling and simulation – Integrating discrete events and continuous dynamic systems*, Academic Press, 2000

