Chapter 10

LESSONS FROM THE UNITY OF ARCHITECTING

Rich Hilliard

Abstract: This chapter discusses insights for the harmonization of Software Engineering and Systems Engineering practices through the lens of one particular practice area: Architecting. These insights derive from the *unity of architecting*—the observation that whether one is architecting software, or systems, or enterprises, there is an essence which these endeavors share. This essence is elucidated through a case study of ISO/IEC/IEEE 42010:2011, the international standard for Architecture Description (originally IEEE 1471:2000) in the form of lessons learned from the development and use of that Standard over the past 15 years. These lessons are then applied more generally to Software and Systems Engineering. I focus on two areas: the role of *first-class concerns* as a unifying dimension in the engineering of any complex system; and on the need for *knowledge mechanisms*, of which methods and practices are one form, to enable continuing progress to be made. I demonstrate this by sketching how Essence and similar method frameworks can be improved by incorporating first-class concerns and lessons learned.

1. INTRODUCTION

This chapter offers a case study and lessons learned from Architecting as a basis to examine the commonalities, differences and hoped-for harmonization of Software and System Engineering, via what I will call the *unity of architecting* in these (and other) fields. *This chapter is not about how to architect.*[1] This chapter is about lessons learned from experiences in Architecting and how those lessons may be useful to addressing the topic of this book on Software Engineering in the Systems context. The chapter is directed to practitioners interested in applying these lessons and to method

[1] For that, see *All about ISO/IEC/IEEE 42010*, and references therein: http://www.slideshare.net/dJdU/all-about-isoiecieee-42010-2014r5.

To appear in Software Engineering in the Systems Context, editors:
Ivar Jacobson and Harold "Bud" Lawson

developers, educators and researchers in the hope of furthering the understanding of foundations and their practical application to our field.

First, I look at the case of the first formal standard for Architecture Description, IEEE *Recommended Practice for Architectural Description of Software-intensive Systems* (IEEE Std 1471:2000). I examine the initial scope and goals of this effort, the design choices made, experiences using the Standard, and its subsequent adoption by ISO and joint revision as *Systems and software engineering — Architecture description* (ISO/IEC/IEEE 42010:2011).

Next, I distill some lessons learned from Architecting with relevance for Software Engineering and System Engineering and for bridging gaps between those disciplines. Several of these insights are codified as what I call the *unity of architecting* (discussed below).

Finally, I apply these lessons to offer some practical suggestions for how Software and System Engineering harmonization can proceed, looking in particular at: *first-class concerns*, which was a central concept in IEEE 1471 and motivated many of the best practices codified in the Standard; and *knowledge mechanisms* – of which methods and practices are one form – and their role in mature engineering disciplines. I demonstrate this by sketching how Essence can be improved through the introduction of first-class concerns and guidelines for knowledge mechanisms.

2. ARCHITECTING: A CASE STUDY

IEEE Std 1471:2000, *Recommended practice for the Architectural Description of Software-intensive Systems*, now superseded by ISO/IEC/IEEE 42010:2011, *Systems and software engineering — Architecture description*, specifies best practices for Architecture Description. If Google hits, unit sales and literature citations are any indication, the Standard has been very influential.

Work on IEEE 1471 began in 1995. As chartered by the IEEE Computer Society, its initial scope was "software architecture" because the early and mid 1990s were a very fertile time for work in that field (Kruchten, Obbink, and Stafford, 2006; Shaw and Clements, 2006). However, as work progressed, its authors realized that the best practices being codified were not limited to Software Architecture, but equally applicable to Systems Architecture and Enterprise Architecture settings. This scope was reflected IEEE 1471:2000, *Recommended Practice for Architectural Description of Software-intensive Systems*, the first formal standard addressing the architecture of **systems**, where

> "the term *system* encompasses individual applications, systems in the traditional sense, subsystems, systems of systems, product lines, product families, whole enterprises, and other aggregations of interest" *(IEEE Std 1471, 4.1)*

The application of the Standard to Systems Engineering was first discussed in (Maier, Emery, and Hilliard, 2004) and is now reflected in standards such as ISO/IEC/IEEE 15288 (6.4.4) for system life cycle processes.

ISO adopted the IEEE standard in 2007 as ISO/IEC 42010. Subsequently, ISO and IEEE produced a joint revision, published as ISO/IEC/IEEE 42010, *System and software engineering — Architecture description*, extending the ideas of the first edition to architecture description languages and architecture frameworks. The conceptual model and best practices of the Standard have found widespread use in software, systems and enterprise architecting.[2] As a foundation for architectural thinking, the Standard has been applied to Architecture Evaluation (ISO/IEC 42030, in progress), Conformity Assessment,[3] tools for architects (such as SysML, UPDM and MEGAF (Hilliard et al., 2012)) and to Architecture Frameworks (such as TOGAF).

3. LESSONS LEARNED

This section will distill and discuss some of the lessons learned from ISO/IEC/IEEE 42010 (most of which were already present in the 2000 edition, IEEE 1471). The next section will sketch how these lessons can be applied to harmonizing Systems and Software Engineering practices.

3.1 *Pick your Battles*

Some products, including systems, standards, methods and practices, *try to do too much*. This often results in these products being unnecessarily large and complex, built upon assumptions or prerequisites that the user must accept to use the products. When the assumptions of the product do not match those of the user, the product is less likely to be adopted, and less likely to be used properly or effectively. Users will often avoid a "big bang"

[2] *ISO/IEC/IEEE 42010 Annotated Bibliography*, http://www.iso-architecture.org/42010/docs/bibliography-42010.pdf for many references to applications of the Standard.

[3] *Architecture Description Conformity Assessment*, http://softsysarchitect.net/adca.

change to what they are doing versus introduction of small improvements—and rightly so.

An alternative approach is to stay focused: minimize assumptions; do one thing really well; and anticipate that any product will be used in combination with other products. Developers of systems, standards, methods, practices, tools and other products can facilitate those combinations by *Picking their Battles* to avoid over-specification.

Pick your Battles is so pervasive, it could be called a "meta-strategy" – simply good engineering practice. It is closely related to *Divide and Conquer* and *Separation of Concerns* (to be discussed extensively in the rest of this chapter). It underlies other practices, such as: *Establish System Boundaries* and *Bound the Problem*. It is useful to recognize that any system of interest has a boundary—that boundary may change as more is learned, but at any point it is useful to distinguish a definite "inside" and "outside" and underlies another lesson (*Architecture is contextual*, see below).

There are many applications of *Pick your Battles* in ISO/IEC/IEEE 42010. The rest of this section will highlight a few.

3.2 Architecture vs. architecture description

The Standard defines the ***architecture*** *of a system* as

> "fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution" (ISO/IEC/IEEE 42010:2011, 3.2)

Beyond that definition, the Standard has very little to say about *architecture* at all! The Standard does not address what a "good" architecture is. Rather than focus on architectures, whose properties may vary with application domain, with technologies, and in other ways, the Standard focuses on *best practices for documenting architectures*. An *architecture description* (AD) is an explicit work product expressing an architecture of a system, usually via models, text and graphics. The best practices as codified in the Standard focus on what makes a good AD (not on what makes a good architecture). Making good ADs should improve the ability of architects and system stakeholders to communicate and understand one another in achieving good architectures and hopefully better systems.

3.3 System theory agnostic

As noted above, in the Standard an architecture is defined relative to a system. *So, what is a system?* The Standard takes no position on what constitutes a system—leaving this to users of the Standard. As systems will vary with domain of application and other considerations, it is intended that

users of the Standard will "bring their own" system theory—defining *system* and criteria for systems as they choose (see Lawson 2015). In the Standard, system is used as a placeholder; if the user considers considers ***S****system* is used as a placeholder; if the user considers *S* to be a system, the Standard offers best practices for documenting an architecture for *S*. Following the Standard, in this chapter I will use **system** to include software, systems and enterprises without prejudice.

3.4 Life cycle, process and method agnostic

Just as the Standard is agnostic to what constitutes a system, it also does not take a position on the life cycle, processes or methods that the architect should use. This philosophy is somewhat different from much of current practice—where there is a major emphasis on defining life cycles, establishing processes for system and software construction and providing detailed definitions of suitable methods. Instead, the Standard is intended to be compatible with and usable in a variety of settings: including various life cycles, various process models and with various architecting methods. This allows users of the Standard to choose life cycle processes, methods and practices appropriate to their particular situation. This degree of freedom recognizes that there are many possible usages for an AD. The Standard is therefore intended to be *usage neutral* over a range of AD uses, including: as basis for design and development; to baseline an existing system; to compare and conduct tradeoffs among alternate architectures; as a reference architecture; as a product line architecture; as basis for acquisition or procurement; as planning tool; as input to tools or other analyses, etc. (see 4.4 of the Standard for extensive list of usages).

3.5 Multiple views

The *Pick your battles* meta-strategy, in particular with reference to the notion of Separation of Concerns, is also embodied in the best practices of the Standard itself—an architecture description is specified in terms of multiple *architecture views*, such that each view:

- is addressed to identified system stakeholders and
- answers identified concerns of those stakeholder.

In fact, this tenet of Architecture Description—*stakeholder- and concern-driven views*—is a fundamental organizing principle, following from and formalizing much of current practice in Architecting. This is discussed in detail below—for its applicability to Software and Systems Engineering in general (not only to Architecting).

3.6 Architecture is contextual

As in the built world, architecting takes place in an environment—because systems exist in environments. System engineering has traditionally focused upon the operational environment. Software engineering has focused on the users in the operational environment and the stakeholders in the development environment. Architects often must consider the *wider environment* (Lawson, 2010). In the Standard, the *environment of a system* is:

> "context determining the setting and circumstances of all influences upon a system" which invariably "includes developmental, technological, business, operational, organizational, political, economic, legal, regulatory, ecological and social influences." (ISO/IEC/IEEE 42010, 3.8)

A consequence of establishing a system boundary (discussed above) is recognizing that some things are within the architect's span of control while some things are outside of that control. In addition to establishing a system boundary, it is critical in successful architecting to recognize the particulars of the environment and understand those influences upon the system.

One practical approach to understanding that environment is via understanding and identification of the system's *stakeholders*—anyone with a stake in the system is by definition part of its environment; and further, to identify and analyze the areas of *concern* of each of those stakeholders.

3.7 Interest-relativity

As noted immediately above, complex systems have many stakeholders who are parties (individuals, groups and organizations) with interests in the system. In the Standard, these interests are called *concerns*, following Dijkstra's introduction of the phrase "separation of concerns" into Software Engineering in 1974. (More on this history in *Moving Ahead*, below.) Concerns span both the "problem space" and "solution space" as well as the "horizontal and vertical dimensions" of a system (Jacobson and Lawson 2015).

Understanding of stakeholders and their concerns is the basis of successful architectures because the diversity of stakeholders and their concerns creates the richness of the environment (as above) which in turn determines the complexities which architects face and which must be handled by any solution.

acceptability, accessibility, accountability, accuracy, adaptability, administration, affordability, agility, assurance, auditability, authentication, autonomy, availability, backup, behavior, benefit, business alignment, business goals, business strategies, capacity, certification, communication, compatibility, completeness, complexity, compliance to regulation, conceptual integrity, concurrency, confidentiality, configurability, configuration management, consistency, continuity of operation, control, correctness, cost, credibility, customer experience, customizability, data accessibility, data integrity, data privacy, degradation, dependability, deployment, disaster recovery, disposability, distribution, documentation, durability, ease of learning, ease of use, economy of mechanism, effectiveness, efficiency, environmental protection, error handling, evolvability, extensibility, failure management, fault tolerance, feasibility, fidelity, flexibility, functionality, generality, implementability, information assurance, integrity, inter-process communication, interchangeability, interference, internationalization, interoperability, intuitiveness, known limitations, learnability, legal, licensing, localizability, logistics, maintainability, manageability, mobility, modifiability, modularity, monitoring, network topology, openness, operability, operating costs, optimizability, organization, performance, persistence, platform compatibility, portability, predictability, price, privacy, provability, quality of service, recoverability, regulatory compliance, reliability, repeatability, reporting, reproducibility, resilience, resource constraints, resource management, resource utilization, response time, responsiveness, reusability, robustness, safety, scalability, schedule, security, serviceability, simplicity, stability, state change, structure, subsystem integration, supportability, survivability, sustainability, system features, system properties, system purpose, technological constraints, testability, throughput, timeliness, traceability, trustworthiness, understandability, usability, usage, user-friendliness, vendor lock-in, versatility, workflow management ...

Figure 10.1 Common concerns for systems and software

3.8 Role of concerns

> Haec autem ita fieri debent ut habeatur ratio firmitatis utilitatis venustatis.
>
> *(Well building hath three conditions: firmness, commodity, and delight.)*
>
> *Marcus Vitruvius Pollio*

Architects must address a wide range of concerns for a successful system. Even Vitruvius recognized three key concerns (in the quote above). Figure 10.1 lists a number of concerns common to software and systems as examples, but is by no means exhaustive.

A fundamental goal of the Standard, beginning with the 2000 edition, IEEE 1471, was to codify and motivate the use of multiple views in architecture descriptions which was already in practice (Perry and Wolf, 1992; Kruchten, 1995) and was inspired by earlier work in Requirements Engineering (Ross 1977; Finkelstein and Sommerville, 1996). The motivation for multiple views was simple: each architecture view answered different, complementary questions and taken together expressed the whole architecture.

Concerns in the Standard name the areas of interest held by the various stakeholders of an architecture. The Architect engages with stakeholders, elicits and collects their stated concerns, identifies other concerns from experience and considers these areas of interest as a part of understanding: what needs to be done; what should be documented and ultimately how to address the issues that arise in architecting the system.

In ADs conforming to the Standard, concerns must be identified (i.e., recorded); and each concern is linked with the stakeholders holding that concern. To create the views of the architecture, the Architect selects viewpoints for use in the AD, such that each concern must be *framed* by at least one viewpoint.

By creating models and organizing them into views within the AD, the Architect *addresses* the concerns—otherwise, the work is not yet finished! Each view in an AD models the architecture of interest in a different manner, using a well-defined set of conventions which may include: selected notations, models kinds, and associated methods—collectively termed the *viewpoint* of the view. Each *viewpoint* establishes the conventions for a type of view (Finkelstein et al., 1992; ISO/IEC 10746-2, 1996).

Different kinds of models[4] are suited for different concerns. For example, a UML class diagram is useful for concerns such as *What are the types of critical information in this enterprise?* However, when the concern is *Transaction throughput*, a class diagram is of little value. For concerns such as *System behavior*, a static UML package diagram is useless. In these examples, the relation of concerns to representations should be obvious, however in many cases the relation is less so; making concerns and their linkage to the views and their models in an AD part of the Standard's best practices.

[4] Following definitions of Minsky and Ross, *M* is a model of *S* if *M* can be used to answer questions about *S*.

3.9 Separating viewpoints from views

One design tenet of the Standard, starting with IEEE 1471:2000, was to make explicit the separation of the viewpoint from the view itself; the viewpoint capturing the conventions for constructing, interpreting and analyzing a type of view; the view an expression of a system's architecture relative to an identified set of concerns and conforming to its governing viewpoint. First-class viewpoints were first introduced in (Ross, 1977). For some history and discussion, see also (Hilliard, 1999).

Making concerns explicit means we can reason about these matters *before any modeling takes place*: referring to the examples above, a model meeting the conventions of a class diagram will *never* allow the interested parties to analyze throughput. A picture of packages and their static dependencies will *never* allow us to analyze expected or actual system behavior. So, establishing the concerns relevant to a viewpoint creates minimum expectations on what it can and cannot be useful for—even before any modeling begins. The linkage:

Concerns → Viewpoints → Views

establishes a minimal chain of expectations for the stakeholders of an AD that it will 1) raise the relevant concerns; 2) demonstrate how it will account for those concerns via the viewpoints, and lastly 3) address those concerns via the views. Another way to think about this is in terms of the lifetime of a Concern:

Identify → Frame → Address

This is discussed below. Concerns also provide stakeholders a basis for traceability, a "semantic table of contents" into the Architecture Description: if one is interested in χ, then look for a view adhering to VPif one is interested in concern *C*, then look for a view adhering to viewpoint *VP* which frames *C*.

Concerns enable the Architect to determine "the right tools for the job" in terms of which types of representation are suitable (or not) for a given system. This is a powerful idea that has application not only to Architecture Description but across Software and Systems Engineering (as discussed below).

3.10 Open and extensible

> All problems in computer science can be solved by another level of indirection.
>
> *David John Wheeler*

Early in the development of IEEE 1471 (1995–2000), a key design question was: *What views should the Standard require all architecture descriptions to include?*

There were already a variety of architecture views in use and many debates on the importance of each. The IEEE 1471 solution was *not* to prescribe a particular set of views, but instead (via "another level of indirection") to introduce the idea of a *viewpoint*: conventions on a type of view (as above), and to prescribe the rules for specifying new viewpoints. Each AD must document the viewpoints it uses. They may be off-the-shelf, highly reusable, or brand-new and unique to this particular project.

This approach is contrary to what is often found in the world of architecture frameworks. Choosing an architecture framework is often *All or Nothing*. One chooses a framework and uses what it provides, recognizing that no framework anticipates *all possible concerns for a domain of interest.*

Among the many published architecture frameworks, it is trivial to identify gaps (usually apparent in their static meta models). The ontology of frameworks has evolved as our understanding of enterprises, information systems and software has evolved. The earliest frameworks knew nothing of object-orientation; later frameworks invariably included objects. Early frameworks did not include notions like service— today, no self-respecting framework would ignore service constructs. There is no reason to believe this evolution will not continue. An architecture framework is, at best, a "starter set" of concerns, stakeholders, viewpoints and model kinds for Architects within the domain of interest (Emery and Hilliard, 2009).

Rather than pretending to cover all possible concerns once-and-for-all and the requisite viewpoints to frame them, the philosophy should be to give architects the starter set and a means to extend the framework in a *principled* manner. In the Standard, the available "units of extension" are the Viewpoint and the Model Kind: each providing specific forms of representation, with its own mini-meta model (*What*) to address one or more identified concerns (*Why*), via associated methods and practices (*How*). Generalizing from this, it is useful to consider any practice in terms of *Why–What–How* elements (see *Moving Ahead*).

3.11 Ontology-based

An explicit *ontology* is helpful in the effective and practical implementation of the lessons learned above. The identification of stakeholders and concerns as part of an architecture description (AD), and as a part of the specification of viewpoints and model kinds, encourages the explicit linkage of problem elements and solution elements in the minds of architects and a system's stakeholders. As noted in the previous section, making entities first-class facilitates extensibility. The Standard goes further

in this regard, to define an Architecture Description *meta model* which (1) reflects the ontology of AD constructs and their relationships; and (2) provides a meta-syntactic basis for extension. A core part of the ontology of the Standard is shown in **Error! Reference source not found.**.

In addition to being informative to users, an explicit meta model is useful for imposing syntactic and semantic rules (cf. the Meta-Object Facility in relation to UML). In the Standard, one of the requirements on conforming architecture frameworks is to build upon the Standard:

> An architecture framework shall establish its consistency with the provisions of the conceptual model [*presented in part in* ***Error! Reference source not found.***]

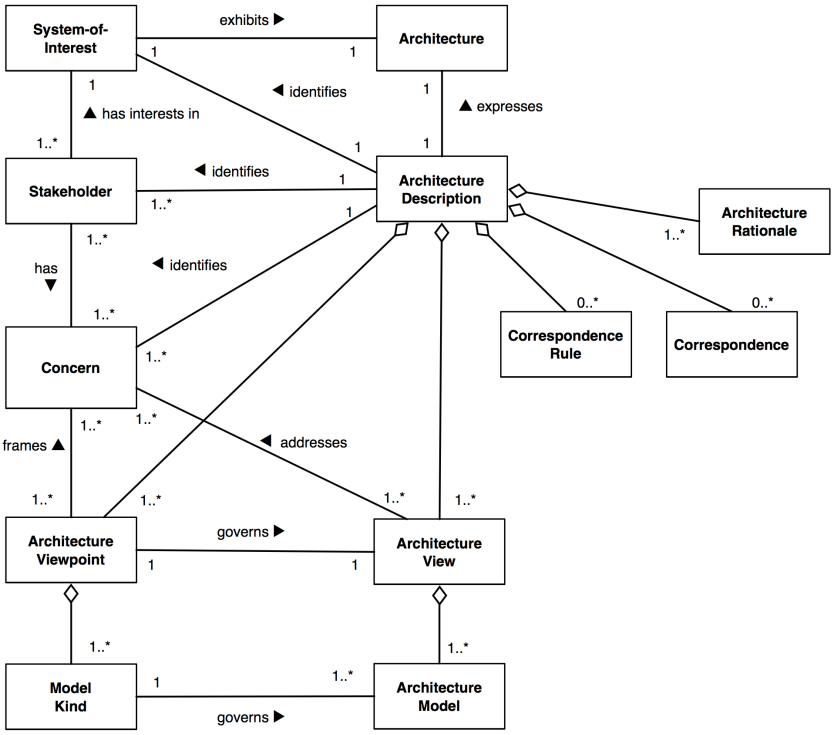

Figure 10.2 A portion of the Architecture Description meta model

Many architecture frameworks (such as DODAF, NAF, TOGAF) provide a meta model of their intended subject matter. The challenge in this approach is to cover the full range of entities in their domain of interest. An alternate approach, in the spirit of *Pick your Battles*, would be small, focused and

composable meta models organized around viewpoints or concerns—precisely opposite to the usual strategy of large architecture frameworks' meta models (discussed above) which attempt to codify the elements of their domains once-and-for-all, and offer no provisions for extension.

Following Jean Bézivin (Bézivin, 2005), the first-class constructs in the Standard can be arranged into meta model layers as shown in figure 10.3.

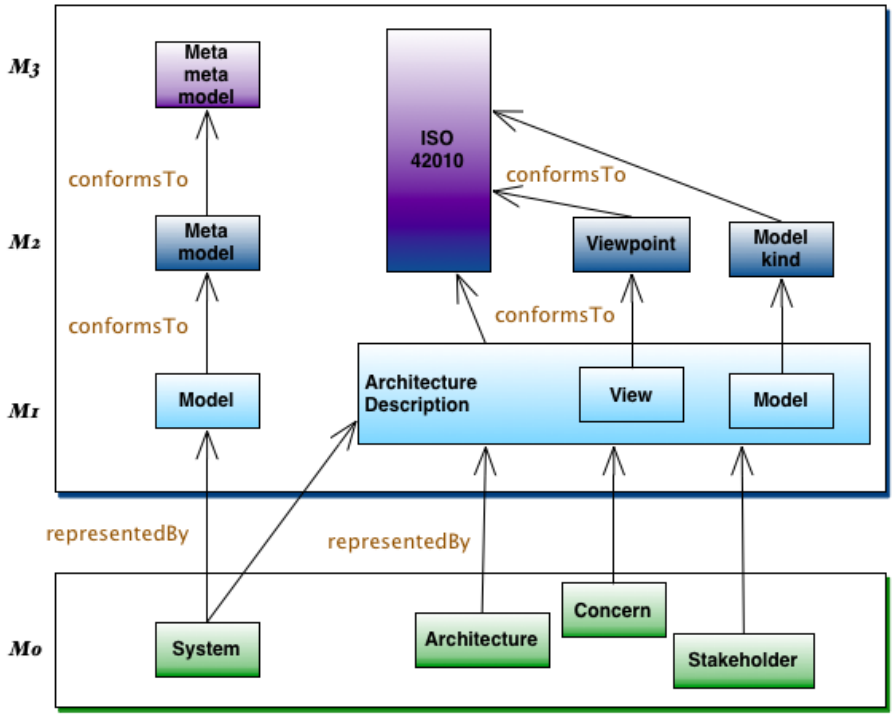

Figure 10.3 Architecture Description elements arranged by meta model layer

4. THE BIGGEST LESSON: THE UNITY OF ARCHITECTING

The message from the case study above, is the *unity of architecting*: from the oldest times to today, from building and civil architecture, through today's fields of software, system and enterprise architecture. In developing and using the Standard, reflecting on architecture from Vitruvius through Fred Brooks (Brooks, 2010) and John Zachman, it was observed that there is significant commonality to Software, System and Enterprise architecting. To

paraphrase Zachman, *architecting is architecting is architecting* (Zachman, 2007). This section outlines some of those commonalities.

Key ingredients of Architecting are:

Architecting faces multiple stakeholders with diverse concerns.
This has been discussed above.

Architecting spans problem and solution.
This, too, has been mentioned above. Architecting involves understanding, often negotiating or even reformulating, the problem while working on possible solutions.

Architecting is multi- (and cross-) disciplinary.
Frequently, this is a consequence of the two items above. The multi-disciplinary nature of architecting was recognized very early:

> "The ideal architect should be a man of letters, a skillful draftsman, a mathematician, familiar with historical studies, a diligent student of philosophy, acquainted with music, not ignorant of medicine, learned in the responses of jurisconsults, familiar with astronomy and astronomical calculations."
>
> *Vitruvius,* De Architectura

Architecting is decision-making.
The essence of Architecting (and more generally Software Engineering and Systems Engineering) is making decisions. The slogan *Decisions Are Your Main Deliverable* is appropriate.[5]

Architecting is not a phase or a stage, but a life cycle practice.
Unlike some life cycles that locate Architecture in-between Requirements and Design, Architecting is an on-going effort because concerns arise throughout the life cycle—each decision addresses some concerns, while possibly introducing new ones.

Architecting involves getting to the fundamentals.
The architect attends to the essence of the system being architected. This must be achieved in the context of the system's environment including all of its stakeholders and concerns.

As argued elsewhere (Hilliard, Rice, and Schwarm, 1996), variations do appear; however, these variations are the result of the varying stakeholders

[5] Eltjo R. Poort, http://www.infoq.com/articles/driving-agile-architecting-with-cost-and-risk

and concerns across systems—not due to some intrinsic differences between Software Architecture, System Architecture and Enterprise Architecture. These varying stakeholders and concerns introduce varying vocabularies, and therefore varying techniques and methods needed to reach a solution.

In each case, the basic process is similar (Hilliard, 2009a): *Analyze–Synthesize–Evaluate* (see figure 10.4). Variation is not at this level, but determined by the substantive differences of stakeholders, concerns and the implications of those differences. The implications of the variation are that to be successful: architects need to be cognizant of the different disciplines and methods in different domains—hardly breaking news, but the basis for making a more systematic approach to our methods and practices.

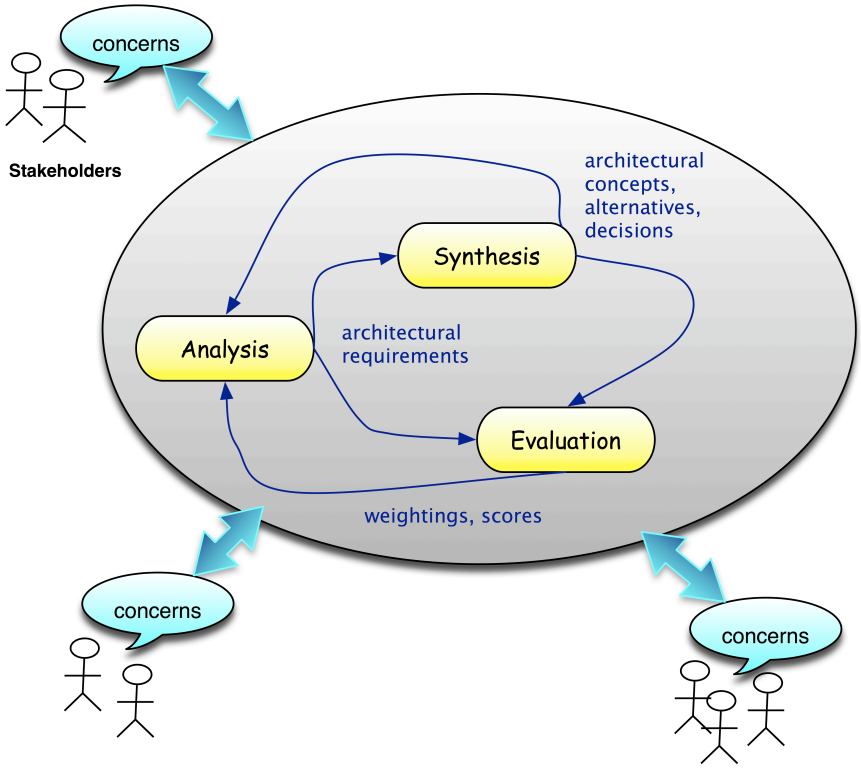

Figure 10.2 The activities of architecting (following Hofmeister et al., 2007)

Alas, this unity is not always acknowledged in recent conceptions of architecture. For example, the attitude, often seen in Enterprise Architecture (EA) blogs and discussion groups, is that EA is different from, apart from, and can learn nothing from, older fields, which have *architecture* in their

names. The insights from the Standard are just the opposite: that EA is continuous with more mature fields, such as Systems Architecture and Software Architecture; that all of these fields still have things to learn from "real" architecture (the architecture of built forms); and that all of these fields can still learn from one another.

Recognizing these commonalities and putting in place knowledge mechanisms for capturing and sharing this knowledge are topics in the next section. To summarize:

1. Architecting is architecting is architecting.
2. Common problems can benefit from common solutions and from other fields that have "been there" already.
3. Knowledge mechanisms can aid architects in sharing methods and practices (Hilliard, 2009b).

5. MOVING AHEAD

This section applies the lessons learned above to the harmonization of Systems and Software Engineering. First, I focus on the role of **concerns**, as one potential unifying notion. While, their role has been recognized informally for some time, some argue for giving them status as *first-class entities* (Sutton and Rouvellou, 2004; Hilliard, 2013). The previous section argued for the centrality of concerns in understanding and motivating the architecture of a system. Elsewhere I have argued that concerns are a missing dimension more generally in thinking about Software and Systems Engineering methods.[6] I argue that concerns permeate not only the architecture, but all aspects of software and system development, and therefore need to be mirrored in our methods and tools and thinking about Systems and Software Engineering.

The second focus is on the nature of knowledge sharing mechanisms and their role in improving methods and practices.

5.1 Toward a theory of concerns

"Concerns are what we care about in software."

Sutton and Rouvellou, 2001

[6] R. Hilliard, *In search of the Higgs, or What's wrong with SEMAT?* http://www.slideshare.net/dJdU/in-search-of-the-higgs-or-whats-wrong-with-semat

Dijkstra (1974) coined the phrase "separation of concerns" in an often-cited passage:

> Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask ourselves whether, and if so: why, the program is desirable. But nothing is gained—on the contrary!—by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by "focusing one's attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.

Dijkstra's observation has led to continued, frequent usage of the phrase, "separation of concerns" in Software Engineering to the present. Sometimes that usage has been superficial; in other cases, there have been attempts to take concerns seriously. As a brief history, notice the progression toward treatment of concerns directly. Separation of Concerns (SoC) has influenced:

- programming strategies: methods of modularization, information hiding and abstraction and mechanisms supporting these in programming languages;
- exploration of the extra-functional properties of software (quality, cost, etc.) (Maier and Rechtin, 2000);
- multiple views and viewpoints in architecture, requirements and design (architecture discussed above) (IEEE Std 1016; Sawyer, Sommerville, and Viller, 2014);
- aspect-oriented programming and recently early aspects (Kiczales et al., 1997); and
- "advanced separation of concerns" or concern-oriented approaches (such as (Sutton and Rouvellou, 2004) and references therein).

Concerns reflect the reality that our systems are complex because the interested parties, i.e., the system stakeholders, have diverse interests. Unlike requirements which tend to be volatile, concerns persist. Software and system engineers know that concerns are important, because pundits like to make up cute reminders for *some* of the important ones (see Table 10.1).

While it is helpful to have reminders and checklists, these mnemonics do not begin to scratch the surface on the full range of concerns that the software engineer or system engineer might confront.

Instead of mnemonics, I argue it is time to treat concerns as first-class elements of our systems, methods, practices and processes.

Concerns, when not made explicit as areas of interest, are often confused with "non-functional requirements",[7] with "qualities" or "ilities" or with system characteristics or properties. As such, identifying the actual concerns is often ignored in contrast to is already "known" users and customers really need. Agile approaches often eschew any explicit discussion of such matters, believing that the "right" properties will emerge for the system by focusing on user stories. As a discipline, we are lacking an adequate vocabulary and ontology of these things (ilities, qualities, requirements, goals, motivations, desires, etc.)—a topic beyond the scope of this chapter.

Table 10.1 Some mnemonics for popular concerns

CRAMPS	Cost, Risk, Availability, Manageability, Performance, Scalability
PESTLE	Political, Economic, Social, Technological, Legal, and Ecological
POLDAT	Process, Organization, Location, Data, Application, Technology
OBASHI	Ownership, Business Process, Application, Systems, Software, Hardware, Infrastructure

5.2 Motivation

In the Architecture Description case study above, I showed the role of concerns in selecting viewpoints. Here I suggest their potential role in relation to more general Systems and Software Engineering concepts. Consider:

- Concerns give processes and tasks their context; e.g.: *We perform this task because it yields an understanding of system deployment.*
- Concerns give work products their relevance; e.g.: *This work product explains how user privacy will be protected in data stores.*
- Concerns determine the requisite skills; e.g.: *To achieve real-time performance, the project will need a rate-monotonic scheduling specialist.*

As such, concerns have the potential to bind together things in terms of their *relevance* to the process, work to be done, methods or ways of working, people, and work products produced. Systems and Software Engineering are

[7] "Non-functional requirements" is a terrible term for several reasons. See Lago, Avgeriou, and Hilliard (2010) for discussion.

concern-based—whether or not we acknowledge this. If we do acknowledge it, then it is only reasonable to assert that concerns must be identified, managed and modeled as part of Systems and Software Engineering.

5.3 Concerns in context

Concerns are of interest because they *cross-cut* other, familiar entities used to discuss software and systems engineering methods. I choose Essence to discuss here, for concreteness. However, concerns are a missing dimension in all of the method frameworks, including Essence,[8] SPEM, and ISO 24744. Figure 10.5 shows how concerns relate to these entities, using terminology from Essence. The figure, expressed as a UML class diagram, is a hybrid—it fuses elements from the Architecture Description meta model (cf. figure 10.2) depicted in green and the alphas from the Essence Kernel (Jacobson et al., 2012) depicted in purple, to show the central role of Concerns and their relations to other entities. System and Stakeholder are common to both. There are also two departures from the original diagrams:

- Work Product is added, to subsume things such as Views and Models (per the Standard).
- Way of Working should be understood to subsume things such as Viewpoints and Model Kinds (per the Standard).

Concerns classify the other core elements, and are distinct from any of those elements. Concerns should be considered another dimension to thinking about methods, addressing questions such as, *Why do we use this practice?*

[8] Essence includes Area Of Concern, which is neither first-class (allowing only a fixed set of values: Customer, Solution, and Endeavor), nor extensible, nor an alpha, and is too coarse-grained to be used here.

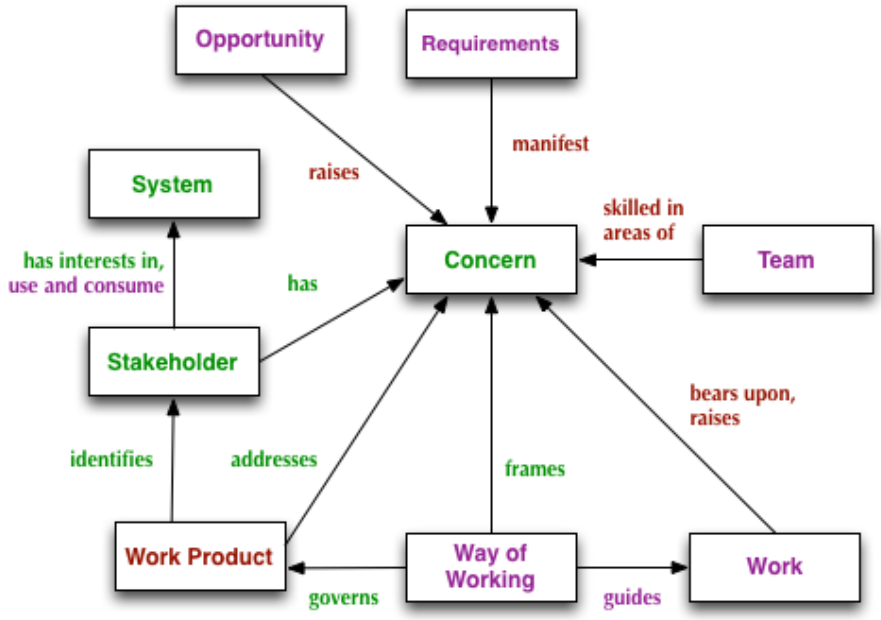

Figure 10.5 Concerns cross-cut familiar entities

5.4 A Life cycle of Concerns

Concerns, as with most entities, have a life cycle. In Essence terms, a progression of "a simple set of states that represent their progress and health." As noted above, the life cycle of concerns in the Standard is very simple:

Identify – Frame – Address

The pair Frame – Address is key to insuring that the architecture description is relevant to stakeholders with that concern. Before making it possible to address a concern, it must be established that that concern is *framed*: the architect must insure that notations are available that enable expression of concern-relevant notions. This is appropriate to the scope of the Standard and its use. One might think of that as *one use case* for Concerns within systems, focused on Architecting. In the context of figure 10.5, we generalize Viewpoint to Way Of Working to say that these *frame* Concerns, and generalize View to Work Product to say that these *address* Concerns.

In the larger context of Systems, we can imagine other use cases, within a broader life cycle, such as:

Identify – Specify – Frame – Manage – Interrelate – Address

First, is to **Identify**, or make explicit, individual concerns relevant to a system. In the Standard, concerns are documented as "atoms": typically names, or sometimes in the form of questions, that an architecture description must address, but no further structure is specified or assumed by the Standard. Within the larger context of Systems and Software Engineering, concerns could be linked to others, and may have internal structure. This is an area for future research.

As concerns are better understood, practices for **Specifying**, analyzing and classifying concerns may be possible. Sutton and Rouvellou (2004) and others suggest the possibility of a much richer structure or *concern space* and techniques for this. There may be an analogy here with the practice of *Dimensional Analysis* as found in mature engineering and scientific fields (Gibbings, 2011).

The **Framing** of concerns has been discussed in the case study above: insuring that all concerns are covered by at least one Way of Working. This is the basis for on-going **Management** of concerns throughout a project (or across related projects). Some relations will be preexisting or native. Others will evolve, and new ones will be added as a consequence of subsequent decisions, new knowledge, requirements and other particulars.

5.5 Principles of Concerns

Ideally, a (future) theory of concerns should allow us to entertain statements such as:

- Requirements, goals, visions and other intentions induce/manifest concerns.
- Often there are concerns nowhere manifest in the "formal" requirements.
- Concerns span the "problem space" and the "solution space".
- If *K* is a concern for system *S*, we would expect there to be: work pertaining to *K*, work products reflecting *K*, people skilled in/knowledgeable about *K* as part of the project.
- When two tasks, *t1* and *t2* pertain to concern, then the tasks are likely to (need to) coordinate or share work products.
- If two work products *w1* and *w2* both frame concern *L*, then there should be some traceability or consistency relation(s) between *w1* and *w2*.

To the extent our methods refer to processes, work to be done, and work products, our methods should be sensitive to concerns. *Without concerns, our methods and practices are empty.*

All of the above may seem quite elementary. We (should) take these observations for granted in Systems and Software Engineering; unfortunately, concerns are rarely handled explicitly. As with methods and practices more generally, one may get away with this in simple cases, but explicit treatment is warranted for complex, large and distributed projects. In these cases, one might envision a "web" of work products, methods and practices, people and skills highlighting their relevance with respect to the identified concerns. This web of concerns then aids stakeholders, as a *semantic index*, in navigating to items of interest and as a means of planning,

resourcing, managing, checking status, tracking progress and identifying gaps.

Putting mechanisms in place to make this possible is the subject of the next section.

6. KNOWLEDGE MECHANISMS = WAYS OF WORKING

If concerns are to be useful to Systems and Software Engineering, then they should be visible in our methods. The ideas above are equally applicable to various communities, including within and between Systems and Software Engineering. The opportunities for sharing would be increased through choosing common approaches to documenting relevant, (re)usable knowledge—*Ways of Working* in the terminology of Essence.

As suggested by the above case study, we should design our methods, tools and techniques as we would design our systems—to be open and extensible, and therefore composable. This is sensible given the *unbounded* range of potential concerns that software, system engineers and architects may face for a given system of interest. Our methods should be as *agile as possible*: establishing the minimum, essential elements to be captured, to be accessible, and relevant to the widest possible range of users and uses. Recognizing this reality of diverse concerns, our goal should be small, specific methods that can be "loosely coupled" with others to compose practices and processes (cf. the Unix tool philosophy or the "little languages" paradigm). These are more likely to be adopted and used than large, monolithic, "high-ceremony" processes. Approaches such as Essence, emphasizing composable primitives of methods are a step in the right direction (Jacobson et al., 2013, p. xxxiii).

In Architecting, various mechanisms are used, listed here in increasing order of complexity/size (Other disciplines, including Requirements Definition, Design, Quality Assurance have their own mechanisms.):[9]

[9] For templates for architecture viewpoints and architecture frameworks, embodying these ideas, see http://www.iso-architecture.org/42010/templates/.

- patterns and styles
- model kinds
- architecture viewpoints
- architecture description languages (Hilliard et al., 2012)
- architecture frameworks (Emery and Hilliard, 2009)

Based on the Architecting lessons learned in the previous section, I suggest that any mechanisms, or ways of working, should include at least three ingredients (*Why–What–How*) as part of their definition:

1. Why use this?
   - What is this way of working good for?
   - What concerns does it frame?
   - Who are the stakeholders for results produced?
2. What does it provide?
   - What results and outcomes does this produce?
3. How to use it?
   - What methods and techniques are available to guide or direct work?
   - How is it linked to other mechanisms?
   - What automated tools support this practice?

The Why–What–How pattern of "interlocking interrogatives" originated with Ross and Schoman, (1977).

7. FINAL THOUGHTS

I have suggested that concerns pervade Software and Systems Engineering. Concerns make explicit what experienced practitioners often know tacitly. The case study of concerns in Architecting shows one "use case." I have argued that the lessons learned from that case study have wider applicability in those fields, and offer one way of sharing and harmonizing insights across those fields, with implications for how we define and document our practices and ways of working. The approach to concerns sketched here is still in its early stages, but can be elaborated in a number of directions.

Acknowledgements

Thanks to Anatoly Levenchuk, Bud Lawson, David Emery, Don O'Neill and Paris Avgeriou for valuable comments on earlier versions of this chapter.

References

Bézivin, J. (2005). "On the unification power of models". In: *Software & Systems Modeling* 4.2, pp. 171–188.

Brooks, F. P. (2010). *The Design of Design: Essays from a Computer Scientist*. New York: Addison-Wesley.

Dijkstra, E. W. (1974). On the role of scientific thought. Reprinted in *Selected writings on computing: a personal perspective* (1982).

Emery, D. and R. Hilliard (2009). "Every Architecture Description Needs a Framework: Expressing Architecture Frameworks Using ISO/IEC 42010". In: *Proceedings of the 2009 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA 2009)*. Ed. by R. Kazman et al. IEEE Computer Society Press, pp. 31–40.

Finkelstein, A. and I Sommerville (1996). "The Viewpoints FAQ". In: *Software Engineering Journal* 11.1, pp. 2–4.

Finkelstein, A. et al. (1992). "Viewpoints: a framework for integrating multiple perspectives in system development". In: *International Journal of Software Engineering and Knowledge Engineering* 2.1, pp. 31–57.

Gibbings, J. (2011). *Dimensional Analysis*. Springer-Verlag.

Hilliard, R. (1999). "Views and viewpoints in software systems architecture". In: *First Working IFIP Conference on Software Architecture*. Position paper. San Antonio.

— (2009a)."Architecture description in-the-large". In: Exploring Enterprise, System of Systems, System, and Software Architecture workshop at WICSA/ECSA 2009.

— (2009b). "Knowledge mechanisms in ISO/IEC 42010 (keynote)". In: *4th International Workshop on SHaring and Reusing Architectural Knowledge,* Vancouver, Canada.

— (2013). *Surveying the Twin Peaks*. 2nd International Workshop on the Twin Peaks of Requirements and Architecture, during ICSE, 21 May 2013, San Francisco.

Hilliard, R., T. B. Rice, and S. C. Schwarm (1996). "The Architectural Metaphor as a Foundation for Systems Engineering". In: *Proceedings of Sixth Annual International Symposium of the International Council on Systems Engineering*.

Hilliard, R. et al. (2012). "On the Composition and Reuse of Viewpoints across Architecture Frameworks". In: Proceedings of the Joint 10th Working IEEE/IFIP Conference on Software Architecture & 6th European Conference on Software Architecture (WICSA/ECSA). Helsinki, Finland: IEEE Computer Society.

Hofmeister, C. et al. (2007). "A general model of software architecture design derived from five industrial approaches". In: *The Journal of Systems and Software* 80.1, pp. 106–126.

IEEE Std 1016, IEEE Standard for Information Technology — Systems Design — Software Design Descriptions (1998).

IEEE Std 1471, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems (2000).

ISO/IEC 10746-2, Information technology – Open Distributed Processing – Reference model: Foundations (1996).

ISO/IEC/IEEE 42010, Systems and software engineering — Architecture description (2011).

Jacobson, I. et al. (2012). "The Essence of Software Engineering: The SEMAT Kernel". In: *Communications of the ACM* 55.12. ACM Queue Magazine, Volume 10 Issue 19, October 24, 2012, pp. 42–49.

— (2013). *The Essence of Software Engineering*. Addison-Wesley.

Jacobson, I. and Lawson, H. B. (2015). Software and Systems, appearing in this book.

Kiczales, G. et al. (1997). "Aspect-Oriented Programming". In: *Proceedings European Conference on Object-Oriented Programming, Springer-Verlag* 1241, pp. 220–242.

Kruchten, P. B., H. Obbink, and J. Stafford (2006). "The Past, Present, and Future for Software Architecture". In: *IEEE Software* 23.2, pp. 22–30.

Kruchten, P. B. (1995). "The "4+1" View Model of architecture". In: IEEE *Software* 12.6, pp. 42–50.

Lago, P., P. Avgeriou, and R. Hilliard (2010). "Guest editors' introduction, Software Architecture: Framing Stakeholders' Concerns". In: IEEE *Software* 27.6 (November/December), pp. 20–24.

Lawson, H. B. (2010). *A Journey Through the Systems Landscape*. Volume 1. Systems Series, College Publications, Kings College, London.

Lawson, H. B. (2015). "Attaining a system perspective", appearing in this book.

Maier, M. W., D. Emery, and R. Hilliard (2004). "ANSI/IEEE 1471 and systems engineering". In: *Systems Engineering* 7.3, pp. 257–270.

Maier, M. W. and E. Rechtin (2000). *The art of systems architecting*. 2nd. CRC Press.

Perry, D. E. and A. L. Wolf (1992). "Foundations for the study of Software Architecture". In: *ACM SIGSOFT Software Engineering Notes* 17.4, pp. 40–52.

Ross, D. T. (1977). "Structured Analysis (SA): a language for communicating ideas". In: *IEEE Transactions on Software Engineering* SE-3.1, pp. 16–34.

Ross, D. T. and K. E. Schoman Jr. (1977). "Structured Analysis for requirements definition". In: *IEEE Transactions on Software Engineering* SE-3.1, pp. 6–15.

Sawyer, P., I. Sommerville, and S. Viller (2014). *PREview: Tackling the Real Concerns of Requirements Engineering*. Technical report CSEG/5/1996. Lancaster University Computing Department.

Shaw, M. and P. Clements (2006). "The Golden Age of Software Architecture". In: *IEEE Software* (March/April), pp. 31–39.

Sutton, S. M. and I. Rouvellou (2001). "Concern Space Modeling in COSMOS". In: *Proceedings OOPSLA 2001*.

— (2004). "Concern Modeling for Aspect-Oriented Software Development". In: *Aspect-Oriented Software Development*. Ed. by R. E. Filman et al. Addison-Wesley, pp. 479–505.

Zachman, J. A. (2007). *Architecture is Architecture is Architecture*. http://zachmaninternational. com