On the interconnection of cross-cutting concerns within hierarchical modular architectures

# On the Interconnection of Cross-Cutting Concerns within Hierarchical Modular Architectures

Herwig Mannaert, Peter De Bruyn, Jan Verelst

***Abstract*—Modularity is often employed to increase the flexibility and adaptability of a system. A well-known issue during the design of modular systems is the emergence of ripple effects propagating throughout the system when one module is changed which is dependent on other modules within the system. While several techniques or approaches have been proposed to mitigate these effects, they often neglect the integration and interconnection of cross-cutting concerns (i.e., functionalities which are required across different parts of the main functional dimension) within the system. This paper argues that the proper integration of cross-cutting concerns is important in order to avoid ripple effects and preserve the adaptability of a modular system. Based on a set of possible integration architectures which offer different alternatives to integrate cross-cutting concerns within a modular structure, we introduce the Integration Design Matrix (IDM) as an instrument to systematically analyze the cross-cutting concern provisioning within specific modular artifacts. A set of design guidelines is proposed to optimize the integration of the cross-cutting concerns, applicable within the context of the IDM. We illustrate our approach by means of examples in several domains.**

***Index Terms*—Modular architecture, cross-cutting concerns, integration design matrix.**

## I. Introduction

MANY human-made artifacts are designed as hierarchical modular systems. While many different modularity interpretations [1] or measurement types [2] exist, modular systems are in essence defined as being composed of a set of interacting subsystems (modules) over multiple aggregation layers or levels [3]. These artifacts can be both tangible and intangible. A suitable example of the first one may be a house (with rooms and bricks as its modules), whereas a software system (having packages, classes, methods, etcetera as its modules) may be an illustration of the latter one. Its application seems logical and is well documented within engineering [4], but has also been considered relevant for topics outside this area, such as management [5]. Modular architectures are believed to be associated with several beneficial characteristics including complexity reduction (by breaking down the problem into a set of subproblems which are easier to handle), adaptability (the system can be changed by adding, removing or replacing only certain parts) and the potential for re-use (modules with a general purpose can be optimized and serve as best practice building blocks for multiple systems) [3], [6], [7], [1]. The modularity of a product is often assumed to be related to the organizational architecture [8], [9] and both the modularity concept [10], [11] as well as the (standardized) interfaces arising between them [12], [13] are considered as one way to enable (the adoption of) innovations. However, realizing the integration of complex modular systems is challenging in practice [14] and the design does not always seem to result in the acclaimed benefits such as adaptability [15], [16].

TABLE I
Examples of modular aggregation levels and cross-cutting concerns in the context of housing and software.

| | **Housing** | **Software** |
|---|---|---|
| Modular aggregation levels | • city<br>• neighbourhood<br>• house<br>• room<br>• wall/floor/ceiling<br>• brick/device | • community<br>• application<br>• package<br>• entity<br>• class<br>• data structure, method/routine |
| Cross-cutting concerns | • electrical power<br>• heating<br>• water supply<br>• Internet connection<br>• multimedia access | • persistency/database access<br>• access control/security<br>• remote service access<br>• transactional integrity<br>• logging/archiving |

Moreover, while some design knowledge for high quality hierarchical modular systems exists (cf. infra), generally applicable prescriptive guidance seems limited. For instance, available metrics to measure the degree of modularity in a system seem to produce inconsistent results [17]. In particular, very little attention is paid to the integration of so-called cross-cutting concerns: concerns which are relevant for multiple (often most) parts within the system and are therefore said to "cut across" the complete hierarchical structure. Consider for instance concerns such as security, persistency and remote access within the context of a software application: all of them are needed in almost all parts of a software system. While the idea of cross-cutting concerns is generally accepted within certain software programming paradigms, its existence has (to the best of our knowledge) not yet been widely acknowledged within other areas. We argue in this paper that the concept of cross-cutting concerns is relevant in the design of modular systems in many other fields as well. For instance, utilities in housing, such as heating, electricity and isolation might be considered as relevant candidates to be identified and analysed as cross-cutting concerns for that domain. Moreover, we posit that the way in which these cross-cutting concerns are integrated throughout a system plays an important role in the extent to which a modular system may (not) realize its envisioned benefits (such as adaptability).

Table I provides examples of some relevant aggregation levels and cross-cutting concerns with the housing and soft-

ware domain. Similarly as we performed in this introduction, we will provide illustrations from these domains throughout the rest of this paper in order to clarify our reasoning and illustrate its applicability. The remainder of this paper will be structured as follows. Section II will cover some related work, including some concepts which will be used later on. In Section III, we will explore the different possible ways to integrate and connect cross-cutting concerns. Based on the benefits and drawbacks of these different architectures, we will provide some recommendations or design guidelines in Section IV, and present various indications from practice supporting their relevance and validity in Section V. In order to support the practical application of these design guidelines, we will propose a tool (matrix notation) in Section VI to represent and optimize existing modular architectures in that regard. Our conclusions are offered in Section VII.

## II. Related Work

This paper attempts to make contributions to the design of modular structures, and their cross-cutting concerns in particular. Later on, we propose a matrix notation to facilitate our recommendations. We therefore review some of the existing literature on these domains.

### A. Modular design principles

Regarding the design of hierarchical modular structures in general, important work has been done by Alexander [18], Simon [3] and Baldwin and Clark [6], [7]. Two important concepts in this regard are coupling and cohesion. *Coupling* is a measure for the degree to which modules are dependent on one another. High coupling is considered harmful for a design as it reduces the possibility to (re)use, adapt or study the modules in a system independently. *Cohesion* is a measure for the degree to which a module is confined to one concern or functionality. High cohesion is considered as beneficial for a design as the clear focus of the module decreases a module's complexity and increases its ability to be recombined with other modules. Therefore, the typical advice for a good modular design is to have *low coupling and high cohesion*. While not all authors explicitly use the same terminology, many of their ideas do align with these concepts. For instance, Simon [3] argues that complex systems which are not hierarchic and (near) decomposable (i.e., having strong interactions within each component and negligible interactions between components) are most likely too complex for human understanding. Baldwin and Clark [7] equally adhere to the idea that intermodular dependencies should be minimal and state that an explicit formulation of remaining intermodular dependencies ("design rules") is necessary in order to realize the potential benefits of modular structures. Other authors have formulated domain specific criteria for the design of modules, mostly in alignment with the low coupling/high cohesion reasoning. For instance, Myers [19] proposed a specific set of types of coupling and cohesion for software applications and Parnas [20] proposed the idea of 'information hiding' by prescribing that software modules should only have access to each others interface, thereby lowering coupling.

Some typical observations occurring during the design of modular systems and some (implicit) heuristic design principles for modular systems can be related to these fundamental concepts.

*1) Coupling:* Consider for example *ripple effects*, a generally observed phenomenon when changing modular structures [21], [22], [16], [23]. Such effects occur when an initial change to a module necessitates (a large amount of) changes to other modules, in their turn possibly requiring adaptations to other modules, etcetera. One could argue that such ripple effects are actually a manifestation of a (highly) coupled modular design in which many modules are dependent on one another such that changes to a module cannot be performed in isolation, resulting in the occurrence of an undesired multiplication or leverage effect when performing such a change.

*2) Cohesion:* In software engineering, most designers know that *duplications* within a modular structure are pernicious and need to be avoided [24], [23]. Duplication typically implies that a certain recurring part or functionality is combined with other parts of modules throughout the system. The modules containing the duplications therefore combine at least two concerns or functionalities, demonstrating low cohesion. These modules then become implicitly coupled as changes to the duplicated functionality need to be performed at all relevant modules in order to safeguard consistency, resulting again in the occurrence of an undesired multiplication or leverage effect when performing such a change. Duplication should be clearly distinguished from the mere *reuse* of a module, whether in software or physical systems. Here, the reuse of a module or part in a black box is desired and could only propagate multiplication or leverage effects if its replacement would violate (i.e., forcing a change to) the external interface. In contrast, duplicating logic in software or integration wiring in physical systems would indeed trigger propagation effects in case of changes.

### B. Cross-cutting concerns

Regarding the specific idea of cross-cutting concerns within hierarchical modular structures, we are only aware of explicit discussions taking place within the software engineering community. In particular, cross-cutting concerns, i.e., concerns that are present or *cut across the functional structure*, are an essential concept within the *Aspect-Oriented Programming (AOP)* paradigm as introduced by Kiczales [25]. Here, reusable code related to concerns such as logging or security are defined within so-called advices. These advices can be inserted at particular points (called joinpoints) within the regular code. As a consequence, the actual code which is compiled or executed is the result of regular code with advices woven into it at the specified joinpoints.

While other domains do not seem to explicitly mention the existence of cross-cutting concerns, some of them tend to recognize them in an indirect way. For instance, within the housing domain, Keymer [26] and Slaughter [27] discuss design strategies for the creation of more adaptable buildings. Some of them stress the importance of an adequate design of services distribution (plumbing, wiring, electricity,

air conditioning, etcetera), encouraged by the fact that these are (or will be) typically needed at many places in a building and play a crucial role regarding the extent of the impact when carrying out changes. The fact that the distribution of services or utilities is needed at many places across the main functional structure, and may trigger multiplication or leverage effects when implementing changes, coincides with our basic definition and motivation for the analysis of architectures in terms of cross-cutting concerns. Similarly, Eppinger and Browning [28] briefly discuss an investigation regarding the systems engineering and system integration aspects of the PW4098 jet engine. The outcome of the investigation resulted in a modular systems with six regular subsystems as well as two "more spatially distributed" subsystems which were "more functionally integrative across the engine" (i.e., the mechanical components and the externals and controls). Such subsystems that cannot be easily confined to one location in the modular structure are highly similar to cross-cutting concerns. As another example, some authors adopting a system approach within management literature employ the notions '*subsystems*' versus '*aspect systems*' (or '*aspectsystemen*') [29], [30]. When considering a system as a set of related objects, a subsystem will focus on a subset of these objects (considering all types of relations) whereas an aspect system will focus on a subset of types of relations (considering all objects). In that context, the production department within an organization could be seen as an example of the former and an analysis of the social relationships between employees as an example of the latter. As a consequence, also the idea of aspect systems stresses the importance of being able to look in a cross-cutting way at specific concerns in a hierarchical system.

Finally, in general modularity literature, one could argue that (in an indirect way) the idea of "design rules" as proposed by Baldwin and Clarck [7] could be interpreted as one way to facilitate the plugging in of cross-cutting concern functionalities all across a modular system by adopting explicitly agreed upon interfaces to those cross-cutting concerns. Apart from that, the discussion of cross-cutting concerns or a similar idea within general modularity theory seems limited. As a consequence, the need for and reflection on cross-cutting concerns within general modularity thinking and its applications does not seem completely absent in extant literature but is treated in a rather implicit way resulting in very few available prescriptive design guidelines.

### C. Matrix notations

Regarding the use of graphical or matrix-based representation forms to study modular artifacts, we have been preceded by many others. For instance, Steward [31] introduced the so-called Design Structure Matrix (DSM) which visualizes (maps) and studies the dependencies between design parameters. The notation has later on be leveraged for the purpose of analyzing processes, organizations and product architectures in a variety of domains [32], [33], [28] and is still being further developed and applied in contemporary research [34], [35], [36]. In such a DSM, the parts or design parameters of a system are depicted on both the horizontal and vertical axis of the matrix and x's are placed on all cells where a particular part or design parameter impacts or depends on another part or design parameter. In a more general approach, more diverse values can be used within the matrices (e.g., numeric values between 0 and 1 indicating the strength of an interaction, colors to represent certain dimensions, etcetera). All kinds of clustering and sequencing algorithms can be applied to these matrices in order to group parts into modules and minimize non-diagonal (or at least intermodular) dependencies (i.e., coupling) within the system.

Whereas DSMs are square matrices typically focusing on elements of the same project domain, one of its proposed extensions allows the mapping of elements of different project domains. More specifically, a Domain Mapping Matrix (DMM) is a rectangular $m \times n$ matrix mapping two different project domains, where $m$ is the size of $DSM_1$ (of project domain 1) and $n$ is the size of $DSM_2$ (of project domain 2) [37]. Further, combinations between DSMs and DMMs exist such as the House of Quality (HoQ) [38], [39] which consists of multiple matrices, including a roof depicting the dependencies between technical requirements (similar to a DSM) and a matrix mapping technical requirements onto customer requirements (being a DMM). Consequently, intra-domain representations (i.e., DSMs), inter-domain representations (i.e., DMMs) and combinations of both (i.e., the House of Quality) can be identified when considering matrix based notations for system modeling and analysis, possibly even extended to multi-domain matrices (MDMs) when also applying computational methods to combined intra- and inter-domain matrices [40].

Given their focus on the mapping of dependencies, the study of design change propagations throughout systems has become one of the main usages of DSMs and DMMs. For instance, Baldwin and Clark [7] show a problem-solving path on a DSM illustrating the cycling within an interconnected task structure through which a designer might need to go when working on a modular system. Further, Koh et al. leveraged DSMs and DMMs to assess the changeability of complex engineering systems and to use these change forecasts to prioritize component modularisation [41], [42]. In order to assess the impact on planned changes on other components, they combine the *Change Propagation Impact* with the *Change Propagation Likelihood*. While our work shares the emphasis on the impact of change, we only consider change (propagation) for certain identifiable *anticipated changes*, and assume change propagation to be deterministic and binary, i.e., the anticipated change is propagated or not.

The applicability of these matrix notations for system modeling and analysis for a multitude of domains has been accurately documented in literature. For instance, the work of Eppinger and Browning [28] provides a broad overview of several domains in which the DSM has been applied, including examples within the domains we introduced in the previous section, i.e., housing and software. For example, regarding the former, a discussion is presented which uses DSMs to analyze an earlier proposed layer taxonomy by Brand [43] to decompose a typical building based on the expected range of change. The test largely confirmed the relevance of the dif-

ferent layers but equally identified four different dependency types among components, including "low dependencies inside of the layer and high dependencies outside" indicating that often many of the components in the modularized environment where still dependent on outside concerns. Regarding the latter, it was illustrated how DSMs can be used to optimize the modular architecture of software and to provide visibility into undesirable and cyclical dependencies between Java Jar files.

As modularity is an inherently recursive concept, modeling efforts such as matrix notations can be applied on a variety of aggregation levels, ranging from very fine-grained levels up to very coarse-grained aggregation levels, as is also apparent from the examples that have previously been documented in literature [28]. In this context, Maier has presented a detailed framework for the level of detail or granularity in models and engineering design, and discussed related concepts like abstraction, hierarchy, and aggregation [44]. Nevertheless, the choice of the appropriate level of granularity depends to a large extent to the context, application domain and purposes for which the model is used and few prescriptive and unambiguous guidelines in this regard have been found in literature. Hence, we consider the formulation of strict and generally applicable guidelines for the identification of the appropriate granularity level to be noncritical as it is highly context-dependent and we can further build upon the earlier identification of various aggregation levels with a corresponding granularity in many application domains where it has been proven to be rather non-problematic in the past.

As our matrix notation will focus on the design of cross-cutting concern within the modular structure of an artifact, our approach could be considered as an intra-domain representation. However, our notation differs from the existing DSM by the fact that we consider —a limited number of— aggregation levels on both axes, only consider a static (non-dynamic) perspective, and specifically focus on the integration of the various cross-cutting concerns or utility services within and between the aggregation levels.

## III. Cross-cutting concern integration architectures

The initial decomposition (or aggregation) of a modular system is typically performed in one single dimension, which we call the *main functional dimension* or *concern*. For a general hierarchical modular system, this dimension is schematically represented in the horizontal plane of Figure 1: the individual cuboids represent individual modules (e.g., the rooms in a house or classes in a software system), the bars the connections/interactions between the modules (e.g., the walls between the rooms or a method call between classes). When considering cross-cutting concerns, we focus on functionalities which are of a different kind, nature or dimension than the ones typically considered in the main functional dimension (e.g., electricity, security, etcetera) as they are required (by definition) across different parts of the main functional dimension. Introducing a new cross-cutting concern therefore leads to an integration problem as multiple functional dimensions need to be combined. This integration can be performed in different ways. Based on our earlier work [45], this section presents several ways to do so, which we call *integration architectures*.

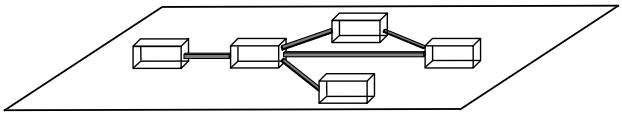


Fig. 1. A modular system considered from one main dimension [45].

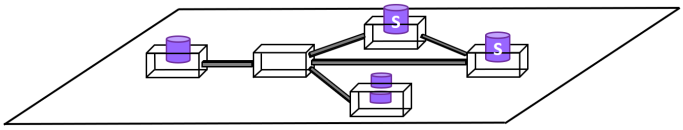


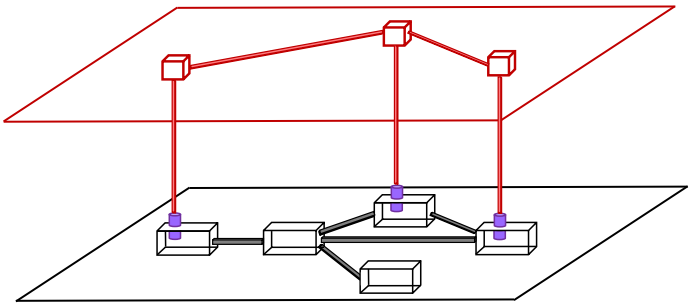

Fig. 2. Embedded integration architectures [45].

### A. Embedded architectures

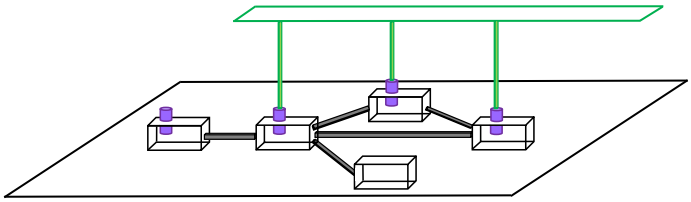
A first option is to embed the actual implementation of the cross-cutting concern within or around the main functional modules. We refer to this implementation as the *embedded integration architecture* or *architecture 1*. This architecture is graphically represented in Figure 2 by the colored cylinders added to some of the cuboids. For example, a fireplace or electricity generator could be placed in every room (or some rooms) of a house. Or a method taking care of persistency could be added to one or multiple software classes. Often, as a field matures, standard solutions for generic cross-cutting concerns (shared across many systems) might arise. For instance, a standardized electricity generator or persistency method can be designed, incorporating best practices and becoming re-used in many modules along many systems. Therefore, we distinguish between an embedded *dedicated* (i.e., non-standardized) integration architecture and an embedded *standardized* integration architecture, which we label as architectures *1A* and *1B*, respectively. In Figure 2, we differentiate between both by leaving the cylinders of dedicated cross-cutting concern implementations empty, whereas the standardized implementations are indicated with an "S".

### B. Relayed architectures

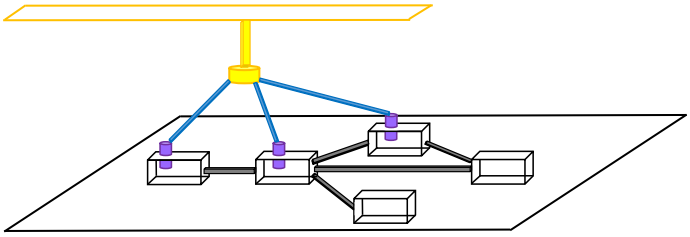
A second option is to have the cross-cutting concern part within or surrounding the main functional modules to act as a proxy or "connector" to a provider framework. This framework provides the actual implementation of the cross-cutting concern at another location, i.e., outside the considered main modules, allowing for a more elaborate implementation and often benefitting from so-called *advantages or economies of scale*. We refer to this implementation as the *relay integration architecture* or *architecture 2*. In Figure 3, we represent relayed implementations of a cross-cutting concern by using another plane. Think for example of an elaborate electricity distribution network in a house to which the different rooms in a house connect via sockets in its walls. Or various classes in a software system calling an elaborated persistency framework



(a) Relay to dedicated framework integration architecture.



(b) Relay to standardized framework integration architecture.



(c) Relay to framework gateway integration architecture.

Fig. 3. Relay integration architectures [45].

taking care of persisting all kinds of data. Also here, the relayed integration architecture can make use of a dedicated (i.e., self-designed, non-standardized) or a standardized framework to provide the cross-cutting concern. That is, one can design its own electricity distribution or persistency framework or make use of an already existing and standardized distribution network or persistency framework (e.g., Java Persistency API). Therefore, we distinguish between a relay to a *dedicated* (i.e., non-standardized) framework and a relay to a *standardized* framework, which we label as integration architectures *2A* and *2B*, respectively. Integration architecture 2A is visualized in Figure 3a. Note that this option does not only require modular design decisions regarding the main functional dimension, but also regarding the cross-cutting concern dimension (i.e., the cuboids in the cross-cutting concern plane) as well as the connection between both (i.e., the vertical bars). Integration architecture 2B is visualized in Figure 3b. As in this case the main functional dimension connects to an (external) standardized framework, the internal modular design thereof does not need to be considered and an empty cross-cutting concern plane is used. Connecting the main modules to this plane is based on accessing a standardized interface, like a socket tapping into electrical power provided at a standard voltage and frequency, or some annotation lines configuring the data class to use the persistency framework.

In integration architectures 2A and 2B, modules still connect individually to a provider framework. When using a *relay to a framework gateway* (which we define as integration architecture *2C*), an intermediate level or gateway module is added. This intermediate module is the single point which connects to the provider framework and allows all main modules to switch collectively to another provider framework at once (instead of individually) as now the proxy modules in the main functional dimension only refer to the framework gateway. Integration architecture 2C is visualized in Figure 3c, in which the intermediate gateway module is represented by the broad yellow cylinder. Consider for instance a central power supply or converter in a building which provides the possibility to change the nature of the external power (e.g., solar panels or the national electricity grid). Or a dedicated class in a software application, to which all other classes refer, ensuring the connection to a provider of naming and directory services (this is less realistic for a persistency framework).

Figure 4 provides an overview of the different integration architectures we discussed above. It should be clear that, typically, multiple cross-cutting concerns for a system can or should be considered concurrently and the chosen cross-cutting concern integration architecture can differ for every individual cross-cutting concern for every individual module. For instance, Figure 4 represents up to 4 cross-cutting concerns integrated by using 4 different integration architectures (the encapsulation of the connections in additional small cuboids will be discussed in the next section). Table II provides examples of each of the provided integration architectures for our two running cases: the electricity concern in housing structures and the persistency concern in software architectures. For the housing electricity case, each example distinguishes two *hierarchical or aggregation levels* to which the relevant cross-cutting concern is provided, as mentioned between brackets. The software persistency case only describes the hierarchical or aggregation level of the class.

## IV. Design Guidelines for Integrating Cross-Cutting Concerns

While the previous section discussed some cross-cutting concern integration architectures, it did not cover the question whether one architecture is superior to another. Additionally, other aspects than the choice of integration architectures might influence the quality of the cross-cutting concern design within a modular structure. This section proposes a set of design guidelines for this purpose. We use existing design principles as discussed in Section II-A to motivate our guidelines. It is important to mention in this context that our three design guidelines should be applied together as their partial application might in fact deteriorate a design. Indeed, the underlying motivation of the design guidelines is *to limit the impact of change*, and as such they are only necessary conditions.

### A. Encapsulation

*Guideline:* The integration of a cross-cutting concern should be performed in an encapsulated way. This means, first, that *every cross-cutting concern (i.e., its actual implementation or the relay connecting to the actual centralized implementation) should have its own separate module*. Stated

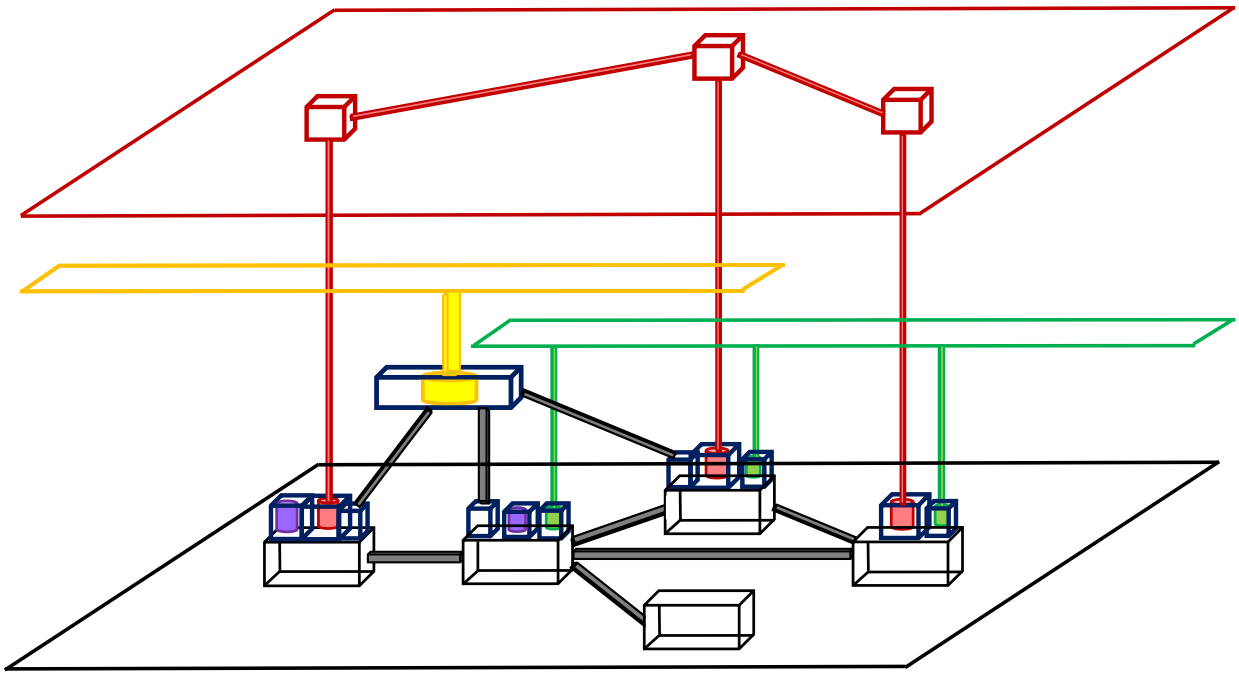


Fig. 4. Integration architectures for cross-cutting concerns [45].

TABLE II
EXAMPLES OF IMPLEMENTATIONS OF CROSS-CUTTING INTEGRATION ARCHITECTURES

| | **Housing (electrical power)** | **Software (persistency)** |
|---|---|---|
| **Embedded dedicated integration architecture (1A)** | • every house has its own electricity generator *(aggregation level = house)*<br>• every device requiring electricity to operate has a battery embedded within it *(aggregation level = device)* | • every software class contains a custom made module (e.g., method) which completely takes care of the persistency for that class *(aggregation level = software class)* |
| **Embedded standardized integration architecture (1B)** | • every house has its own standardized (commoditized) electricity generator *(aggregation level = house)*<br>• every device requiring electricity to operate has a standardized (commoditized) battery embedded within in *(aggregation level = device)* | • every software class contains a standardized module (e.g., method) which completely takes care of the persistency for that class *(aggregation level = software class)* |
| **Relay to dedicated framework integration architecture (2A)** | • every house taps electricity from the shared (dedicated) solar panel electricity grid provided by the local community *(aggregation level = house)*<br>• every device requiring electricity to operate taps electricity from a dedicated electricity grid within the house by means of cables *(aggregation level = device)* | • every software class contains a relay module (e.g., method) calling into a dedicated framework at a higher aggregation level (e.g., a software entity) covering persistency *(aggregation level = software class)* |
| **Relay to standardized framework integration architecture (2B)** | • every house taps electricity from the standardized national electricity distribution network *(aggregation level = house)*<br>• every device requiring electricity to operate taps electricity from a standardized wireless electricity grid within the house *(aggregation level = device)* | • every software class contains a relay module (e.g., method) calling into an immanent standardized framework covering persistency (e.g., Java Persistency API) *(aggregation level = software class)* |
| **Relay to framework gateway integration architecture (2C)** | • the electricity for each house is provided through a gateway at the level of its community which may switch between electricity provided by the community or the national electricity grid *(aggregation level = house)*<br>• every device requiring electricity to operate connects via sockets to a gateway at the level of the house which can switch between electricity provided by the house's own electricity generator (e.g., based on solar panels) or the national electricity distribution network (e.g., nuclear energy) *(aggregation level = device)* | • every software class contains a relay module (e.g., method) calling into a dedicated gateway (which may switch or adapt to changes in the persistency provider framework) *(aggregation level = software class)* |

otherwise, the cross-cutting concern should not be joined with the main functional concern in one atomic module. Second, this means that this separate module should *only interact with other modules through a predefined and stable interface* [1].

*Motivation:* First, the isolation of the cross-cutting concern into a separate module increases cohesion. Indeed, when the cross-cutting concern parts and main functional parts are combined into one module, this module combines two functionalities of a different kind. In contrast, having separate modules for the cross-cutting concern and main functional parts results in modules focusing on one type of functionality and therefore exhibiting a higher degree of cohesion. Second, only allowing the cross-cutting concern module to interact via a clearly defined interface lowers coupling as only the interface then represents the dependency with other modules. The impact of a change is also better predictable (i.e., limited to interactions via the interface), thereby preventing ripple effects in the main functional module and beyond.

*Illustration and reflections:* In order to represent the application of the encapsulation guideline visually, consider Figure 4. Here, in contrast with Figures 2 and 3, we have placed the cylinders depicting the cross-cutting concerns outside of the main concern cuboid to symbolize the fact that the cross-cutting concern is isolated in a separate module. The small bold cuboids encapsulating the cylinders represent the module to be "shielded" by its interface.

In order to apply this guideline to a specific situation, consider for instance the heating of the rooms of a house. Here, a non-encapsulated variant could be the use of fireplaces incorporated within the walls of the rooms (i.e., the fireplace and walls constitute one inseparable whole). It is clear that, in case the implementation of this cross-cutting concern would need to be changed (e.g., upgrading to a more modern fireplace), the impact of that change would not be limited to the fireplaces themselves but would also entail ripple effects in the room or functional module, as it would require drilling into the walls, the areas surrounding the chimneys, etcetera. This effect would not occur when an encapsulated variant is chosen, such as a (un)pluggable electric heater. Here, the electric heater is a separate module (it is not inextricably incorporated within the wall) which interacts with the other modules via a clear interface (its electricity plug). The electric heater can be changed without impacting the wall: switching the electric heaters to more modern electric heaters would only require the heaters themselves to be replaced. Similarly in a software context, persistency functionality added amidst variables and methods of the main funcional classes (representing for instance the domain entities of an information system like *Customer*, *Order* and *Invoice*), would be non-encapsulated, as a change in the implementation of the persistency would require coding efforts within each of those modules leading in general to ripple effects. On the other hand, isolating persistency code within distinct methods or classes which can be used ("called") by other methods or classes would be an encapsulated approach, confining the impact of a change in implementation to these dedicated modules.

As we consider the possibility of multiple cross-cutting concerns, our advice for encapsulation results in the design of main modules surrounded by a set of non-main or peripheral modules each implementing one cross-cutting concern in an encapsulated way. For instance, within a housing context, one could consider the living area within each room as a main module and the heating, water provisioning, electricity, etcetera as the non-main modules of a room (encapsulating the cross-cutting concern implementation or its relay). In a software context, it would imply the introduction of various peripheral classes (encapsulating code dealing with concerns like persistency, access control, and transactions) around the main functional modules or domain classes.

### B. Interconnection

*Guideline:* The cross-cutting concern module surrounding a main module, should not implement the cross-cutting concern itself (i.e., embedded). Instead, *the cross-cutting concern implementation should be realized in a relayed way via an interconnection*. As a consequence, the cross-cutting concern module surrounding the main module will act as a proxy.

*Motivation:* Having embedded cross-cutting concerns leads to the duplication of their implementation at all places where the concern is required. As discussed in Section II-A, coupling and cohesion advise against this. Indeed, an embedded implementation would result in ripple effects in case the cross-cutting concern implementation needs to be updated as it is spread out over multiple locations. Adopting proxies which connect to a centralized cross-cutting concern implementation would avoid this, allowing to update the cross-cutting concern at one location. Further, we note that embedded implementations are not always feasible (e.g., due to space limitations) and that the centralized production or provisioning of a cross-cutting concern allows for typical *advantages of centralisation*, like increased efficiency due to *economies of scale*.

Still, one might argue that two types of coupling remain. First, whereas embedded architectures are somewhat autonomous (i.e., independent) in terms of their cross-cutting concern provisioning, the use of an interconnection implies that the proxies become dependent on the proper working of the centralized cross-cutting concern implementation: its failure might endanger the proper working of all decentralized main modules. Such 'single point of failure' might for instance be mitigated by making sure that a central redundant emergency provider is in place, thereby leveraging another

[1]The idea of encapsulation is heavily inspired by the software community where is it a widely acknowledged principle and practice. However, we are aware that translating this idea of fully predefined and stable interfaces to other domains such as mechanical designs should not be considered as trivial or straightforward [46]. More specifically, a fully defined and blackbox interface for mechanical designs should take into account multiple categories of interactions (e.g., physical size, power consumption, electro-magnetic fields, etcetera). Nevertheless, we wish to stress that the idea of "design rules" as proposed by Baldwin and Clark [7] is similarly proposing explicitly predefined and stable interfaces for modularity in general. Further, whereas we are aware that it might be challenging to currently realize the design guideline in a certain application domain, part of our contribution also aims to provide an overview of potential innovation and improvement opportunities for modular designs (of all kinds) in the near or distinct future.

advantage of scale, i.e., increasing overall reliability[2]. Second, the proxy modules themselves might have connections which are specific for the chosen centralized cross-cutting concern provisioning. In that case, a change to another centralized cross-cutting concern provider might still require adapting all proxies. This dependency can be resolved by employing a gateway as an intermediary module to which both the proxies and the actual cross-cutting concern implementation connect: the cross-cutting concern provider can then be changed without requiring changes to every individual proxy.

*Illustration and reflections:* In order to represent the application of the interconnection guideline visually, consider again Figure 4 where the previously discussed integration architectures are shown in an encapsulated way. The red, green and yellow planes represent the relayed implementation of the cross-cutting concern connecting to the proxy modules via the red, green and yellow vertical interconnections. Therefore, these three architectures (2A, 2B, 2C) comply with the interconnection guideline. As the yellow architecture (2C) employs a gateway, this option seems to get the ultimate preference.

In order to apply this guideline to a specific situation, consider again the heating of a house. In case an embedded architecture is opted for, the heat production is fully taken care of (and therefore duplicated) in each room of the house. When trying to change the heating provisioning (e.g., upgrading to a more modern fire place), each room with heating would be impacted. Such ripple effect would not occur when the heating for the rooms is provided in a relayed way by using centralized heat generation (e.g., using electricity to heat water) which is then distributed (e.g., through pipes) to all relevant rooms. In this case, changing the heat generation mechanism (e.g., using gas instead of oil or electricity to heat the water) would only require an adaptation of the central heat generator but would not impact the radiators in each individual room (still having hot water flowing through their pipes). The heat generation itself would at the same time also be done in a more efficient way by exploiting economies of scale. Of course, the absence of an impact with respect to the connectors or radiators is only valid for *a set of anticipated changes*, i.e., any heating system that allows for the heat to be distributed by a traditional fluid. Similarly at the software level, persistency could be fully taken care of by means of a method in each class (in which case it would be dedicated) or could be provided by methods merely connecting to an existing persistency framework taking care of the actual persistency functionality (in which case it would use an interconnection), the latter resulting in a much smaller impact when the specific implementation of persistency would change. However, this impact would still be spread out across the functional structure or domain entities, and could be further reduced by introducing a central gateway module absorbing a possible change of the actual provider framework.

[2]Remark that we suggest the possibility of introducing some kind of duplication with the purpose of increasing reliability, whereas we earlier argued that duplication within modular structures should be avoided in order to enable, for instance, adaptability. However, it is important to be aware that the idea of duplication as mentioned in Section II-A and afterwards mainly refers to situations in which the number of duplications is proportional to the size of the modular system in scope (e.g., fire places in every individual room of a house, resulting in more (less) fireplaces for bigger (smaller) houses) and, as a consequence, mostly larger than two (as would be the case when duplicating due to reliability reasons). The duplication for reliability as we propose would also be situated at a high and central aggregation level resulting in, again, a duplication which is not proportional to the overall size of the modular system. Conceptually, one could even argue that in case of duplication for reliability reasons two different concerns can be distinguished: a first central module being responsible for provisioning the cross-cutting concern by default and a second central module acting as a back-up for the cross-cutting concern, thereby not representing a genuine duplication as meant in Section II-A. Therefore, we do not consider both viewpoints to be contradictory.

### C. Downpropagation

*Guideline:* The *integration of a cross-cutting concern should be implemented up to the lowest aggregation level which is possible*. With the lowest aggregation level possible, we mean the lowest level within the hierarchical system at which modules can still be changed, added or removed, and at which the cross-cutting concern is relevant. At that level, a cross-cutting relay concern module (e.g., a proxy) should be present next to the main functional module.

*Motivation:* Insufficient downpropagation of the integration of a cross-cutting concern requires its provisioning and implementation at a (higher) aggregation level. This requires awareness of the structure of the lower aggregation level. In particular, this implies that coupling is present between the concerning aggregation levels as changes up to the lowest aggregation level can lead to changes in the provisioning of the cross-cutting concern at higher aggregation levels. In contrast, when integrating the cross-cutting concern at the lowest aggregation level, the cross-cutting concern is immediately provided (embedded or via a proxy) for each module at the lowest aggregation level. Changes at this lowest aggregation level will therefore not require adaptations at higher levels in order to stay assured of the provisioning of the cross-cutting concern.

*Illustration and reflections:* Figures 2 till 4 visually represent the possible integration architectures when focusing on one aggregation level (although the relay architectures might imply that the actual implementation of the cross-cutting concern is implemented at a higher aggregation level via a relay). Therefore, these same alternatives can be considered at each of the various aggregation levels of a modular system (as we illustrated for instance in Table I). Following our guideline, the integration of cross-cutting concerns (and hence, a choice regarding their possible integration architectures) should be implemented up to the lowest aggregation level which is possible, for instance by using proxies (and not merely reside within the higher aggregation levels). It is clear that this guideline implies the adherence to the two other guidelines. The combination of downpropagation with embedded implementations would lead to large amounts of duplications, while downpropagation without encapsulation would lead to massive impacts in case of a change.

In order to apply the downpropagation guideline to a specific situation, consider again the heating of a house. Here, the heating is now traditionally integrated up to the aggregation level of the rooms, e.g., one or more radiator(s) in each room to be heated. This is not yet the lowest thinkable hierarchical or aggregation level as each room is typically constructed out of

bricks. Therefore, one might argue that the heating integration is traditionally not implemented up to the lowest aggregation level possible and the implementation of the cross-cutting concern connector or proxy (e.g., the radiator) is duplicated in every room. In the hypothetical case where one might want to enlarge every room with a set of additional bricks (i.e., enlarging the rooms), the radiators might not be of a sufficient capacity to heat the whole room and might need to be changed. Moreover, introducing an additional radiator in this room (or in an additional room) would in general imply drilling into the walls for connecting pipes. Stated differently, the cross-cutting concern provisioning is coupled with the size and shape of the room, and a change would easily lead to ripple effects in the main structure. This coupling would not be present when small heating radiators would be able to be integrated at the lowest aggregation level (i.e., the brick) as the enlargement of the rooms with additional bricks would automatically imply the increase of heating capacity of each room[3]. We remark that the idea of the "lowest possible aggregation level" should be interpreted in a pragmatic way. As stated in Section II-C, we only consider the obvious aggregation levels which are relevant to the architecture and at which real functional changes occur. For housing, this is obviously still the case for a room or its bricks (e.g., rooms or bricks can be added or changed) but no longer for extremely small parts such as individual 'molecules' or 'atoms'. In theory, one could argue that these are —in the limit— modules of a house as well, but as changes to molecules and atoms are typically not considered relevant for the design of a house, we do not consider them as relevant aggregation levels in this context. Similarly at the software level, concerns like persistency should be relayed or connected at the lowest level of domain classes to avoid ripple effects in the handling of this concern at a higher level. In the latter case, modules handling the concern at an higher level, e.g., a component, often become aware of the lower level domain entities like *Customer* or *Order*. Changing or adding domain entities could therefore lead to modifications and even structural changes in the code at higher levels, e.g., the component or application, handling and/or dispatching the concern.

## V. The Concept of Integrated Elements and Some Indications from Practice

Having proposed a set of design guidelines for the integration of cross-cutting concerns into hierarchical modular structures, we now investigate the practical realization and feasibility of these design guidelines. In a first subsection, we present the concept of *integrated elements*, both in software and in construction, as an optimal way to realize those guidelines. While a reference implementation of such elements has been implemented and published in software, it is more hypothetical in other domains, such as construction. Therefore, we discuss in the later subsections some recent evolutions and realizations that can be regarded as (partial) implementations of these concepts, and as indications of the relevance of our proposed approach in such application domains.

[3]One might argue that such a design might still necessitate adaptations at a higher aggregation level when additional central capacity is required to provide heating for all rooms which are now enlarged: for example, a second central heat generator might need to be added at a centralized location. However, similarly as we set out in Footnote 1, mainly the adaptations due to duplications proportional to the size of the system are the ones that generate ripple effects being harmful for a system's adaptability. Clearly, the number of locations where adaptations are required for the upscaling of the provisioning of a centralized cross-cutting concern is not proportional to the size of the system. On the contrary, the centralisation is in general aligned with economies of scale and more appropriate to deal with scalability issues.

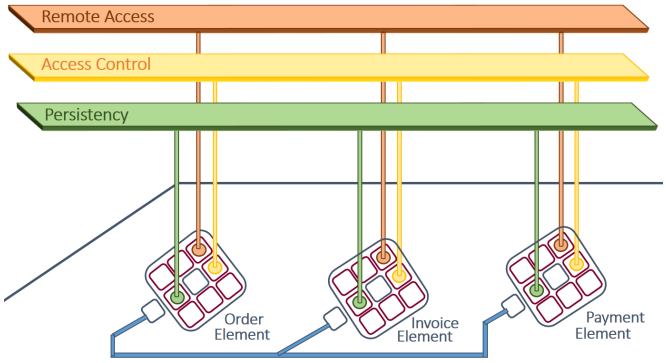



Fig. 5. A software element integrating cross-cutting concerns.

### A. Toward Integrated Elements

The design guidelines imply that the integration of cross-cutting concerns into hierarchical modular structures should be done through encapsulated interconnections that are propagated down to fine-grained modular building blocks. In the case of software, we have implemented and published the architecture and automatic generation or expansion of so-called elements in our previous work [45], [47]. The structure of these software elements is schematically represented in Figure 5 for three functional data entities (*Order*, *Invoice*, and *Payment*), and three cross-cutting concerns, i.e., *Persistency*, *Access Control*, and *Remote Access*. It realizes the design guidelines of the previous section in the following way:

- *Encapsulation:* The lines of code that are specific for a concern, i.e., the colored little disks, are encapsulated within separate classes or modules, i.e., the small rounded rectangles. These classes interact with other classes through version-transparent interfaces that are independent of the technology, allowing to change the internal technology without impacting the other classes.
- *Interconnection:* The lines of code in the colored little disks do not implement the concern themselves, but merely serve as a relay or interconnection to the solution framework, i.e., the large flat coloured cuboid, that is being used for the actual implementation of that concern.
- *Downpropagation:* The lines of code interconnecting the functional structure with the frameworks implementing the cross-cutting concerns, are propagated down to the basic functional or domain entities, in this case the data entities. This allows the addition or modification of these domain entities without impacting existing interconnection code that would be located at a higher level.

As explained in [45], [47], such an element structure needs to be defined for the basic building blocks of software systems,

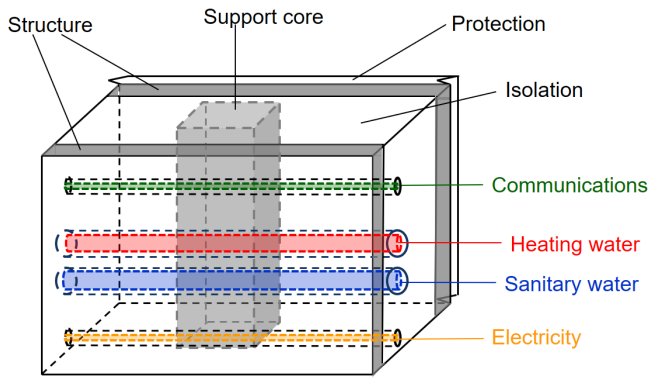



Fig. 6. A concept element for integrating concerns in construction [45].

i.e., data entities, processing tasks, workflow orchestrators, and input/output connectors.

The structure of an analogous *concept element* for housing or construction is schematically represented in Figure 6. This basic functional building block, *an integrated brick*, accommodates four cross-cutting concerns, i.e., *Communications*, *Heating water*, *Sanitary water*, and *Electricity*. It realizes the design guidelines of the previous section in the following way:

- *Encapsulation:* The parts that are specific for a concern, i.e., the colored small pipelines, are encapsulated within slightly larger pipelines. These larger pipelines, connected to the brick structure, are independent of the internal technology. In this way, changes in an internal technology, e.g. other types of communication wires, electricity conductors or heating fluids, become possible without impacting the rest of the brick structure.
- *Interconnection:* The small colored pipelines do not provide the utility themselves, but merely serve as a relay or interconnection to a centralized utility provider. The utilities are for instance provided by an external telecommunications network accessed through a communications gateway, by a central heating system, or by a connection to the water or electricity infrastructure.
- *Downpropagation:* The pipelines interconnecting the functional structure with the infrastructure providing the utilities, are propagated down to the basic building entities, in this case the bricks of the walls. This allows the addition or removal of (parts of) walls without impacting other (parts of) walls through drilling and plumbing.

As in the case of software elements, such an element structure needs to be defined for the basic primary structures of housing, i.e., inner walls, outer walls, floors and ceilings.

### B. Some Elements in Construction

When we first proposed in [45] the concept element of Figure 6, we also hypothesized in early 2016 on analogous element structures for roads, encapsulating pipelines for communications, water supply and drainage, and electricity. Though we were not aware of any implementation at that time, an initiative to develop such roads[4] was announced in late 2016. Currently, several of such roads have been successfully realized.

While solar photovoltaic power generation started off by using separate photovoltaic panels mounted on top of the construction structures, we currently see a trend toward integrating photovoltaic cells into the construction units. For instance, integrated solar tiles have become available for regular houses, as in the Solar Roof®[5], and photovoltaic cells have been integrated in the hull structure of the trunk of the Crew Dragon® spacecraft[6]. The possibility has also been mentioned to integrate battery storage into the solar tiles.

The Hivehaus® modular living space initiative[7], considering a house as a modular aggregation of hexagonal compartments, provides an example of integrating cross-cutting concerns in standardized building blocks at a more course-grained modular level. In these compartments, the distribution of auxiliary facilities or utilities has been integrated upfront, and adheres to *encapsulation* and *interconnection*. But of course, the more coarse-grained modular level, i.e., the *lack of downpropagation*, restricts the design freedom of the house to an aggregation of the —in this case hexagonal— modular building blocks.

Another example of compartmentalized building blocks integrating utilities or cross-cutting concerns are containerized data center solutions. These solutions integrate the various auxiliary facilities at roughly the same modular level, e.g., a 40 feet container. While electricity is provided externally, most other utilities, including air-conditioning, and batteries and diesel generators for emergency power supply, are embedded within the functional module. Though this lack of interconnection may be less favorable to achieve advantages of scale, it does provide evolvability with respect to the most prominent type of change in this case, i.e., increasing data center capacity by adding additional containers without impacting the existing infrastructure of the containerized data centers.

### C. Modular Structures in Airports

Airport terminals are in general large buildings exhibiting some modular structure. Typical add-on modules to airport terminals are jet bridges. Such a jet bridge, as represented in Figure 7, is connected to the airport terminal, but subject to an independent manufacturing processes. It seems therefore logical that a typical concern or utility in the jet bridge, i.e., air conditioning or cooling, is realized in an embedded —and not standardized— way, as represented in Figure 7. Nevertheless, air conditioning is a cross-cutting concern throughout the entire airport terminal, and the various parts or corridors of the terminal are probably connected to a central air conditioning system. Hence, it could be beneficial to apply the interconnection guideline, and to tap into the central system for the air conditioning concern of the jet bridges. This could bring scale advantages (i.e., higher efficiency and lower redundancy) to the cooling. It could also lead to an improved encapsulation, as appropriate connections to the global circulation system could

[4]See https://www.plasticroad.eu/

[5]See https://www.tesla.com/solarroof

[6]See https://www.spacex.com/vehicles/dragon/

[7]See https://www.hivehaus.co.uk/

Fig. 7. A jet bridge containing an embedded air conditioning system.


Fig. 8. Two airport gates with embedded security and baggage claim.

avoid impacts on the jet bridges when upgrading or modifying the central cooling system. This is in contrast with the situation as depicted in Figure 7, where a change in the embedded air conditioning system could have a significant impact on the jet bridge.

Though in general lacking physical barriers, airport gates can be considered as the modules within an airport terminal, and can be argued to be the main functional modules. Of course, a lot of utilities or auxiliary facilities are shared between —cut across— these functional modules. Consider for instance check-in counters and security checkpoints, restaurants and bars, as well as toilets and washrooms. Though these utilities do not exhibit *downpropagation* in most airports, we can see at the Berlin Tegel airport that this is indeed possible. As is visible in Figure 8, where both the security checkpoints and the baggage claim areas have been downpropagated to units comprising two gates. The embedded implementation of these concerns is often experienced as pleasant, as there is no danger of long waiting lines due to other flights. However, as can be seen in Figure 9, the facilities for washrooms and food and beverage have been *downpropagated* as well and are embedded in the two-gate unit. While this results in a rather limited offering for these utilities to the passengers waiting at these gates, the introduction of interconnections for


Fig. 9. Two airport gates with embedded food, beverage, and washrooms.

these concerns (granting passengers access through a relay connection to a centralized concourse with plenty of bars, restaurants, and washrooms) could easily address this issue.

## VI. The Integration Design Matrix

So far, we have discussed the importance of cross-cutting concerns and how they can be integrated within a modular system. In a typical system, multiple cross-cutting concerns and multiple aggregation levels are relevant. First, this calls for a way to preserve the general overview. Second, a systematic conformity assessment of our design guidelines of Section IV is desirable and can provide relevant recommendations for the system at hand[8]. Therefore, we propose in this section the *Integration Design Matrix (IDM)* which visualizes many of the aspects discussed before. In a first subsection, we present the definition and interpretation of the IDM. Two other subsections discuss two use cases of the IDM: the architecting of concerns throughout various aggregation levels, and the architecting of multiple concerns at a specific aggregation level. In both subsections, we will indicate the opportunities the IDM offers for evaluation, exploration, and ideation of architectures.

### A. Definition and Interpretation of the IDM

Table III provides a general representation of an IDM[9] An IDM covers the design decisions for the integration of one cross-cutting concern into an hierarchical modular structure. Therefore, the IDM is similar to a Design Structure Matrix (DSM), but features hierarchical aggregation levels for its rows and columns instead of constituent subsystems/activities. These hierarchical aggregation levels are domain-specific, but

[8]As mentioned before, our three design guidelines should be applied together as their partial application might in fact deteriorate a design: integrating the heating of a house at the level of a brick (i.e., applying downpropagation) without encapsulation, for example, would lead to the complete destruction and rebuilding of walls when changes to the heating would occur.

[9]In earlier work [48], [49], [50], [51] we already demonstrated the relevance of the different integration architectures at multiple aggregation levels in a simplified table for some specific application domains (i.e., logistics and housing utilities). The IDM presented in this paper, builds on that idea but presents a more structured elaboration.

| **Cross-cutting concern** | L1 | L2 | ... | Ln |
|---|---|---|---|---|
| aggregation Level L1 | | | | |
| aggregation Level L2 | | | | |
| aggregation Level ... | | | | |
| aggregation Level Ln | | | | |

TABLE III
GENERIC REPRESENTATION OF THE INTEGRATION DESIGN MATRIX.

| **Heating** | L1 | L2 | L3 | L4 |
|---|---|---|---|---|
| District : L1 | | 1 | 1 | ? |
| House : L2 | | | 1 | ? |
| Room : L3 | | | 1 | |
| Brick : L4 | | | | |

TABLE IV
AN INTEGRATION DESIGN MATRIX FOR HEATING DISTRIBUTION.

in principle quite straightforward to identify for a domain engineer, e.g., districts/neighbourhoods/buildings/rooms in construction, or cloud/application/component/entity in software systems. And while reading the entries in a row of a DSM reveals the output or influence from other elements to which this element is subject, these entries within an IDM indicate whether this aggregation level provides a certain concern or utility to the other levels.

The IDM representation enables engineers to preserve a general overview of how a specific concern or utility is provided and distributed throughout the aggregation levels of the architecture, and enables a systematic conformity assessment of our design guidelines of Section IV:

- Every entry located *on the diagonal* of the IDM implies an *embedded integration architecture* of type 1A or 1B.
- *Moving upwards* from the diagonal entails the application of the *interconnection* guideline and corresponds to an interconnected or relay integration architecture of type 2A/2B/2C. This could facilitate the realization of *economies of scale* as the concern or utility would be provided at a higher aggregation level and would be distributed to the lower levels.
- *Moving to the right* from the diagonal entails the application of the *downpropagation* guideline and corresponds to the distribution of the provided concern or utility to lower aggregation levels. This could facilitate the ease of changing and the evolution of the structure of the lower aggregation level, as it would remain connected to the utility distribution.
- *The entries below* the diagonal imply that the concern or utility is produced at a lower aggregation level, in order to be aggregated and made available at a higher level. Though these architectures, providing utilities in a distributed way, are in principle outside the scope of this contribution, *moving to the left* from the diagonal would imply *interconnection*, while *moving downwards* from the diagonal would correspond to *downpropagation*.
- *At every entry* in the IDM, the proper *encapsulation* needs to be investigated. Insufficient encapsulation of embedded installations would lead to structural impacts on the corresponding modules. In the case of interconnection or down-propagation, such an improper encapsulation could even lead to an avalanche of ripple effects due to a change in the installation at an higher level.

As in a traditional DSM matrices, engineers could decide to limit the values of the IDM entries to be binary, (i.e., the architecture is present or not), or to allow more diverse values to be used within the matrices. Possible non-binary values could be the letter identifying the specific architecture, i.e., A, B, or C. Another possibility is to indicate whether the architecture exhibits a proper encapsulation through the conditional encirclement of the value, e.g., ⓵ or Ⓐ.

There are several ways for an engineer within a particular application domain to use an IDM. First, it can be used to sketch an overview of the currently used integration architectures for a specific utility or cross-cutting concern. Second, the IDM highlights the extent to which the current cross-cutting concern implementation (does not) adhere to our proposed design guidelines. Third, as multiple cross-cutting concerns are typically relevant for a given system, similar matrices can be drawn for various cross-cutting concerns in order to get an overview of the various integration architectures at a certain aggregation level. Finally, it can be used for exploration and ideation, as every remaining entry in an IDM may represent a novel approach, while its position in the matrix entails several possible advantages and disadvantages.

### *B. Architecting Concerns throughout Aggregation Levels*

As stated in the previous subsection, the IDM can be used to sketch an overview of the current integration of a cross-cutting concern throughout the aggregation levels. Table IV presents an IDM for the integration or distribution of heating throughout hierarchical construction levels, distinguishing four aggregation levels: the city or district level, the individual house or building, the rooms in a house or building, and the building blocks or bricks of those rooms. First, we can identify traditional heating systems for housing in the *active entries*.

- $I_{3,3}$ represents embedded heating in a room. A first example is the use of a fireplace. This is clearly neither standardized (architecture 1A), nor encapsulated, as removing the fireplace for another solution will result in a major impact. Another example is the use of mobile heaters. These are typically available off-the-shelve (architecture 1B), and as they can be connected to the electricity system, pretty well encapsulated.
- $I_{2,3}$ represents a central heating system. This interconnection architecture clearly offers advantages of scale, and can be considered to be well standardized and encapsulated (architecture 2B). Indeed, a large amount of central heating systems, both oil and gas heating, can be

introduced and connected to the existing heat distribution pipes. However, the down-propagation is limited to the level of the room, and this implies that structural changes to the room, i.e., removing or extending a wall, may lead to additional impacts on the heating distribution system.

In the first row or aggregation level, we find entries related to modern evolutions regarding city or district heating[10] [52]:

- $I_{1,2}$ corresponds to centralized heating systems distributing heat to houses in a whole town or district. This can clearly bring additional economies of scale, or enable the use of resources not available to individual houses like biomass or geothermal energy. Providing an appropriate implementation of a heat exchanger, this could realize the required encapsulation to shield the individual houses from impacts due to changes in the central system.
- $I_{1,3}$ represents the fact that the heating, produced centrally at the city or district level, is downpropagated to the level of the rooms. This can be done easily by connecting the above mentioned heat exchanger to a typical central heating system inside the house. As this heat exchanger would decouple the rooms from the actual provider of the heating, i.e., a change in the district heating system could be accomodated by the heat exchanger at the level of the house, this would correspond to a 2C architecture.

Finally, we can also identify some entries for exploration and ideation, as a first step toward innovation.

- $I_{2,4}$ corresponds to the downpropagation of the heat distribution as conceptually represented in Figure 6. This thorough downpropagation would allow to perform structural changes to the rooms (i.e., removing or modifying a wall) without an impact on the heat distribution. However, without proper encapsulation of the pipes transporting the fluids, this could trigger change propagations on the advanced construction bricks.
- $I_{1,4}$ represents the fact that a centralized city or district heating system, connected through a heat exchanger to the central heating system of the house, could be combined with the thorough downpropagation as represented in Figure 6. Such a deep downpropagation could possibly combine the advantages of major economies of scale with extreme flexibility toward structural changes in the construction of the houses and rooms.

Table V presents an IDM for the integration of the electricity distribution, representing four aggregation levels: the region, city, house, and room. We identify various entries:

- The entries on the first row represent the traditional electricity distribution: electricity is generated in centralized power plants and the distribution is managed over an entire region. Through an hierarchical system of distribution and electrical substations or transformer stations, this electricity is distributed to the cities, communities, and houses. Due to the lack of encapsulation at the houses, the AC voltage is connected to houses and rooms without a proper transformer. Therefore, it is not possible to change the distribution voltage to DC without impact all houses, though most devices in the houses are based on DC and solar power is generated in DC as well.
- $I_{3,3}$ and $I_{3,4}$ represent the situation where a house is self-sufficient from an electrical point of view. Though the use of diesel generators to realize this goal is considered to be outdated and only suited for rural areas, the combined use of solar panels and batteries could lead to a revival of such an embedded architecture, as seems to happen typically in the early days of new technologies.
- $I_{2,1}$ and $I_{3,1}$ represent architectures where utility services are being provided at a lower aggregation level and aggregated or collected at a higher level. These architectures are becoming quite relevant, as solar panels in farms and even on individual houses are starting to offer electricity to the grid, and being turned into *virtual power plants*.

[10]See http://www.cooldh.eu/

| **Electricity** | L1 | L2 | L3 | L4 |
|---|---|---|---|---|
| Region : L1 | 1 | 1 | 1 | 1 |
| City : L2 | 1 | | | |
| House : L3 | 1 | | 1 | 1 |
| Room : L4 | | | | |

TABLE V
AN INTEGRATION DESIGN MATRIX FOR ELECTRICITY DISTRIBUTION.

| **Persistency** | L1 | L2 | L3 | L4 |
|---|---|---|---|---|
| Cloud : L1 | | 1 | 1 | 1 |
| Application : L2 | | | 1 | 1 |
| Component : L3 | | | | 1 |
| Entity : L4 | | | | |

TABLE VI
AN INTEGRATION DESIGN MATRIX FOR SOFTWARE PERSISTENCY.

Table VI presents an IDM for the integration of the persistency concern in software applications, representing four aggregation levels: the cloud, the software application, component, and domain entity, e.g., *Order* or *Invoice*.

- $I_{2,3}$ and $I_{2,4}$ represent architectures where a persistency framework, like JPA (Java Persistency API), is being included in the application or application server, and used by the various components and domain entities. In accordance with our *downpropagation* guideline, we have a preference for $I_{2,4}$. As persistency frameworks are typically standardized, this corresponds to an architecture of type 2B, and as we have argued both in the design guidelines and in our previous work on software architectures [45], [47], it is very important to encapsulate the downpropagated interconnections properly.
- $I_{3,4}$ represents the possibility to include the persistency framework in the individual components of the application, enabling individual components to change their persistency framework independent from the other

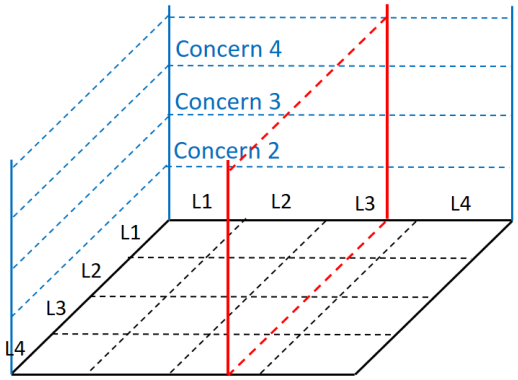



Fig. 10. Several IDMs for multiple concerns stacked on top of each other.

components. Of course, the interconnection needs to be downpropagated to the domain entities, and encapsulated.

- The entries on the first row represent the offering of persistency services in the cloud, enabling individual software applications, components, or even domain entities, to make use of a service API (Application Programming Interface) for persistent storage. This type of architecture is clearly emerging, both in dedicated service platforms providing an integrated *digital vault*, and in generic services offered by the main cloud providers.

For this concern, we currently do not see any relevant entries below the diagonal. For concerns like data mining, one could imagine such entries representing distributed data that is collected at individual applications, components, or entities, and submitted to centralized business intelligence systems.

### C. Architecting Multiple Concerns at an Aggregation Level

While an individual IDM only covers the integration design of one particular cross-cutting concern, multiple cross-cutting concerns are typically relevant for a given architecture. Examples are the presence of concerns like heating, water, electricity, and communication within a house or construction facility, or the relevance of persistency, access control and transactions in a software application. Therefore, for each system, a number of different matrices can be created, equal to the number of relevant cross-cutting concerns. This is schematically represented in Figure 10, where the various IDMs for the different concerns are visualised as blue planes in a third dimension orthogonal to the IDM. This figure also visualizes the possibility, represented in a perpendicular red plane, of taking a *level-slice*, which provides the overview for a certain aggregation level from which level the concerns or utilities are provided to that level for a set of concerns.

A matrix for such a slice is represented in Table VII for the concerns provided to an airport gate as described in Section V-C. The slice considers the aggregation level of the gate, and represents for the various concerns from which aggregation level (the *terminal L1* or the *gate L2*) the individual concerns are provided to the level of the gate. While these utility services are provided in most airports from the level of the terminal (as represented in the matrix at the left of Table VII), they are provided from within the level of the gate in Berlin Tegel (matrix at the right of Table VII).

| **Standard Gate** | L1 | L2 |
|---|---|---|
| Security | 1 | |
| Baggage | 1 | |
| Restaurants | 1 | |
| Washrooms | 1 | |

$\Longleftrightarrow$

| **Tegel Gate** | L1 | L2 |
|---|---|---|
| Secutity | | 1 |
| Baggage | | 1 |
| Restaurants | | 1 |
| Washrooms | | 1 |

TABLE VII
AN IDM SLICE FOR THE GATE LEVEL IN TWO AIRPORT TYPES.

| **Gate** | L1 | L2 | L3 |
|---|---|---|---|
| Secutity | | | 1 |
| Baggage | | | 1 |
| Restaurants | 1 | | |
| Washrooms | 1 | | |

TABLE VIII
AN INNOVATIVE IDM SLICE FOR THE GATE LEVEL IN AN AIRPORT.

As mentioned before, the Berlin Tegel architecture might be perceived as pleasant for security and baggage claiming, but not for restaurants and washrooms due to the limited offerings. In this case, the IDM can provide an environment for ideation and innovation, enabling the exploration of more diverse architectures for the provisioning of the various cross-cutting concerns. In Table VIII, we present again a bit slice for offering various utility services to gates, distinguishing 3 aggregation levels: the terminal, a small set of gates, and an individual gate. In the envisioned structure, security and baggage services would be provided at the gate level (entries $L_{1,3}$ and $L_{2,3}$), while restaurants and washrooms could be provided at the terminal level (entries $L_{3,1}$ and $L_{4,1}$). These centralized offerings would require an interconnection architecture, where passengers could gain access from the individual gates to a centralized concourse with restaurants and washrooms, with a simple electronic access system making sure that they can only return to their appropriate gate. One could also consider other concerns or utilities, e.g., fitness rooms or business lounges, and/or explore the possibility of providing utility services at the level of subsets containing a (small) number of gates.

Table IX presents another matrix for such a level-slide, representing the various cross-cutting concerns in software applications that are provided to the aggregation level of the domain entities (e.g., *Order* or *Invoice*). The possible aggregation levels to provide the cross-cutting concerns are, in accordance with Table VI, the cloud, the application, component, and domain entity. The entries in Table IX are consistent with the guidelines presented in this paper, and with the software architecture presented in our previous work [45], [47]. We have also made use of the possibility, already mentioned in Section VI-A, to represent architectural types instead of binary

| Domain Entity | L1 | L2 | L3 | L4 |
|---|---|---|---|---|
| Persistency | | (2B) | | |
| Transactions | | (2B) | | |
| Access Control | | (2B) | | |
| Naming Service | | (2C) | | |
| Data Security | | | | 1B |

TABLE IX
AN IDM SLICE FOR THE CONCERNS OF A SOFTWARE ENTITY.

values. We can identify the following architectures:

- Most cross-cutting concerns are provided from the application level to the various domain entities, by including a standardized framework like JPA (Java Persistency API) for persistency, EJB (Enterprise Java Beans) for transactions, or JEE (Java Enterprise Edition) access control. Using interconnections to standardized frameworks, these concerns are clearly implemented in a 2B architecture. As explained in [45], the interconnection code is properly encapsulated in separate classes implementing interfaces, resulting in a (2B) notation for the architecture.
- The naming service concern is implemented using a standardized JNDI (Java Naming and Directory Interface) framework. Due to the limited amount of programming interfaces that are required, the interconnection code is encapsulated in a set of central gateway classes, which corresponds to a (2C) architecture.
- The data access concern is implemented by inserting additional constraints directly into the various queries. This corresponds to a 1B architecture, as the inserted querying code is standardized, but implemented within the domain entities, and not properly encapsulated. Therefore, the integration architecture for this cross-cutting concern could and should be improved.

## VII. CONCLUSION

The proper design of cross-cutting concerns within a modular architecture is important and non-trivial. This paper presented a set of general integration architectures and proposed three design guidelines for that purpose. We also introduced the Integration Design Matrix as a possible tool for verifying the conformance to these guidelines and identifying interesting innovation opportunities in a systematic way. In doing so, we believe our work offers contributions for both theory and practice. Regarding theory, we motivated and illustrated that the relevance of cross-cutting concerns outreaches the traditional area of software to hierarchical modular systems in general. The integration architectures are formulated in general modularity terms as well and might therefore contribute to our knowledge on possible modular design options. While some of the proposed design guidelines by themselves are not new (e.g., encapsulation), their formulation in relation to the design of cross-cutting concerns is (to the best of our knowledge). Regarding practice, we remark that our design guidelines were inspired on existing concepts such as coupling and cohesion which have proven their value but might not always be easily transferable into practice. As a consequence, we believe that our design guidelines might potentially serve as more specific instantiations of these general concepts and might help to realize them in the context of cross-cutting concern integration. Further, the IDM could act as a tool allowing designers to manage the incorporation of cross-cutting concerns in a modular system in a more systematic way. It enables the generation of a quick overview of the current status of a modular design with respect to its cross-cutting concerns, and indicates opportunities for improvement or innovation, as empty entries represent opportunities for exploration and ideation. Future research will be directed towards the exploration of IDMs for different types of cross-cutting concerns for systems within different application domains. Whereas our aim was to provide a theoretical underpinning and supporting design tool for modular systems in general, it is clear that the specific elaboration of such matrices requires in-depth knowledge of the system under consideration and therefore necessitates the involvement of domain experts.

## REFERENCES


[1] M. Efatmaneshnik, S. Shoval, and L. Qiao, "A standard description of the terms module and modularity for systems engineering," *IEEE Transactions on Engineering Management*, vol. 67, no. 2, pp. 365–375, 2020.

[2] A. Cabigiosu and A. Camuffo, "Measuring modularity: Engineering and management effects of different approaches," *IEEE Transactions on Engineering Management*, vol. 64, no. 1, pp. 103–114, 2017.

[3] H. Simon, *The Sciences of the Artificial*. MIT Press, 1996.

[4] F. Salvador, "Toward a product system modularity construct: Literature review and reconceptualization," *IEEE Transactions on Engineering Management*, vol. 54, no. 2, pp. 219–240, 2007.

[5] C. Campagnolo and A. Camuffo, "The concept of modularity within the management studies: a literature review," *International Journal of Management Reviews*, vol. 12, no. 3, pp. 259–283, 2010.

[6] C. Y. Baldwin and K. B. Clark, "Managing in an age of modularity," *Harvard Business Review*, vol. 75, no. 5, pp. 84–93, 1997.

[7] ——, *Design rules: The power of modularity*. The MIT Press, 2000, vol. 1.

[8] T. J. Marion and M. H. Meyer, "Organizing to achieve modular architecture across different products," *IEEE Transactions on Engineering Management*, vol. 65, no. 3, pp. 404–416, 2018.

[9] A. Querbes and K. Frenken, "Grounding the "mirroring hypothesis": Towards a general theory of organization design in new product development," *Journal of Engineering and Technology Management*, vol. 47, pp. 81 – 95, 2018.

[10] J. C. Ho and C.-S. Lee, "A typology of technological change: Technological paradigm theory with validation and generalization from case studies," *Technological Forecasting and Social Change*, vol. 97, pp. 128 – 139, 2015.

[11] T. Habib, J. N. Kristiansen, M. B. Rana, and P. Ritala, "Revisiting the role of modular innovation in technological radicalness and architectural change of products: The case of tesla x and roomba," *Technovation*, vol. 98, p. 102163, 2020.

[12] J.-Y. Ho and E. O'Sullivan, "Standardisation framework to enable complex technological innovations: The case of photovoltaic technology," *Journal of Engineering and Technology Management*, vol. 50, pp. 2 – 23, 2018.

[13] S. Bahrami, B. Atkin, and A. Landin, "Innovation diffusion through standardization: A study of building ventilation products," *Journal of Engineering and Technology Management*, vol. 54, pp. 56 – 66, 2019.

[14] M. Naghizadeh, M. Manteghi, M. Ranga, and R. Naghizadeh, "Managing integration in complex product systems: The experience of the ir-150 aircraft design program," *Technological Forecasting and Social Change*, vol. 122, pp. 253 – 261, 2017.

[15] M. Lehman, "Programs, life cycles, and laws of software evolution," in *Proceedings of the IEEE*, vol. 68, 1980, pp. 1060–1076.

[16] C. Eckert, P. Clarkson, and W. Zanker, "Change and customisation in complex engineering domains," *Research in Engineering Design*, vol. 15, no. 1, pp. 1–21, 2014.

[17] K. Hölttä-Otto, N. A. Chiriac, D. Lysy, and E. S. Suh, "Comparative analysis of coupling modularity metrics," *Journal of Engineering Design*, vol. 23, no. 10–11, pp. 790–806, 2012.

[18] C. Alexander, *Notes on the Synthesis of Form*. Harvard University Press, 1964.

[19] G. Myers, *Reliable Software through Composite Design*. Van Nostrand Reinhold Company, 1975.

[20] D. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[21] F. Haney, "Module connection analysis– a tool for scheduling of software debugging activities," in *Proceedings of Fall Joint Computer Conference*, 1972, pp. 173–179.

[22] E. Koh, N. Caldwell, and P. Clarkson, "A method to assess the effects of engineering change propagation," *Research in Engineering Design*, vol. 23, no. 4, pp. 329–351, 2012.

[23] H. Mannaert, J. Verelst, and K. Ven, "The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability," *Science of Computer Programming*, vol. 76, no. 12, pp. 1210–1222, 2011, special Issue on Software Evolution, Adaptability and Variability.

[24] M. Fowler, K. Beck., O. Brant, and D. Robert, *Refactoring: Improving the Design of Existing Code*. Addison Wesley Professional, 1999.

[25] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1997, pp. 220–242.

[26] M. Keymer, "Design strategies for new and renovation construction that increase the capacity of buildings to accommodate change," Master's thesis, Massachusetts Institute of Technology, 2000.

[27] E. Slaughter, "Design strategies to increase building flexibility," *Building Research & Information*, vol. 29, no. 3, pp. 208–217, 2001.

[28] S. Eppinger and T. Browning, *Design Structure Matrix Methods and Applications*. MIT Press Ltd, 2016.

[29] A. de Leeuw, *Bedrijfskundig management*. Van Gorcum, 2000.

[30] M. in't Veld, B. Slatius, and J. in't Veld, *Analyse van bedrijfsprocessen*. Noordhoff Uitgevers, 2015.

[31] D. Steward, "The design structure system: A method for managing the design of complex systems," *IEEE Transactions on Engineering Management*, vol. EM-28, no. 3, pp. 71–74, 1981.

[32] S. Eppinger, "Model-based approach to managing concurrent engineering," *Journal of Engineering Design*, vol. 2, no. 4, pp. 283–290, 1991.

[33] T. R. Browning, "Applying the design structure matrix to system decomposition and integration problems: A review and new directions," *IEEE Transactions on Engineering Management*, vol. 48, no. 3, pp. 292–306, 2001.

[34] T. R. Browning, "Design structure matrix extensions and innovations: A survey and new opportunities," *IEEE Transactions on Engineering Management*, vol. 63, no. 1, pp. 27–52, 2016.

[35] S. Son, J. Kim, and J. Ahn, "Design structure matrix modeling of a supply chain management system using biperspective group decision," *IEEE Transactions on Engineering Management*, vol. 64, no. 2, pp. 220–233, 2017.

[36] H. Son, Y. Kwon, S. C. Park, and S. Lee, "Using a design structure matrix to support technology roadmapping for product–service systems," *Technology Analysis & Strategic Management*, vol. 30, no. 3, pp. 337–350, 2018.

[37] M. Danilovic and T. Browning, "Managing complex product development projects with design structure matrices and domain mapping matrices," *International Journal of Project Management*, vol. 25, no. 3, pp. 300–314, 2007.

[38] J. Hauser and D. Clausing, "The house of quality," *Harvard Business Review*, vol. 66, no. 3, pp. 63–73, 1988.

[39] Y. Akao, *Quality Function Deployment: Integrating Customer Requirements into Product Design*. Productivity Press, 1990.

[40] U. Lindemann, M. Maurer, and T. Braun, *Structural complexity management: an approach for the field of product design*. Springer Science & Business Media, 2008.

[41] E. Koh, N. Caldwell, and P. Clarkson, "A technique to assess the changeability of complex engineering systems," *Journal of Engineering Design*, vol. 24, no. 7, pp. 477–498, 2013.

[42] E. Koh, A. Förg, M. Kreimeyer, and M. Lienkamp., "Using engineering change forecast to prioritise component modularisation," *Research in Engineering Design*, vol. 26, no. 4, pp. 337–353, 2015.

[43] S. Brand, *How Buldings Learn: What Happens After They're Built*. Penguin, 1994.

[44] J. Maier, C. Eckert, and P. Clarkson, "Model granularity in engineering design – concepts and framework," *Design Science, Cambridge University Press*, vol. 3, no. e1, pp. 1–39, 2017.

[45] H. Mannaert, J. Verelst, and P. De Bruyn, *Normalized systems theory : from foundations for evolvable software toward a general theory for evolvable design*. Koppa, 2016.

[46] D. E. Whitney, "Why mechanical design cannot be like vlsi design," *Research in Engineering Design*, vol. 8, no. 3, pp. 125–138, 1996.

[47] H. Mannaert, J. Verelst, and K. Ven, "Towards evolvable software architectures based on systems theoretic stability," *Software: Practice and Experience*, vol. 42, no. 1, pp. 89–116, 2012.

[48] P. De Bruyn, H. Mannaert, and P. Huysmans, "Exploring evolvable modular patterns for transportation vehicles and logistics architectures," in *Proceedings of the the Ninth International Conferences on Pervasive Patterns and Applications (PATTERNS 2017)*, Athens, Greece, 2017, pp. 46–51.

[49] P. De Bruyn, J. Faes, T. Vermeire, and J. Bosmans, "On the modular structure and evolvability of architectural patterns for housing utilities," in *Proceedings of the the Ninth International Conferences on Pervasive Patterns and Applications (PATTERNS 2017)*, Athens, Greece, 2017, pp. 40–45.

[50] P. De Bruyn, H. Mannaert, and P. Huysmans, "Exploring evolvable modular patterns within logistics," *International journal on advances in intelligent systems*, vol. 10, no. 3-4, pp. 290–299, 2017.

[51] P. De Bruyn, H. Mannaert, J. Faes, T. Vermeire, and J. Bosmans, "The modular structure of housing utilities : analyzing architectural integration patterns," *International journal on advances in intelligent systems*, vol. 10, no. 3-4, pp. 280–289, 2017.

[52] R. Wiltshire, *Advanced District Heating and Cooling (DHC) Systems*. Woodhead Publishing, 2016.