Incorporating Design Knowledge into the Software Development Process using Normalized Systems Theory

Peter De Bruyn, Philip Huysmans, Gilles Oorts,
Dieter Van Nuffel, Herwig Mannaert and Jan Verelst
Normalized Systems Institute (NSI)
University of Antwerp
Antwerp, Belgium
{peter.debruyn,philip.huysmans,gilles.oorts,dieter.vannuffel,
herwig.mannaert,jan.verelst}@ua.ac.be

Arco Oost
Normalized Systems eXpanders factory (NSX)
Antwerp, Belgium
{arco.oost}@nsx.normalizedsystems.org

***Abstract*—The knowledge residing inside a firm is considered to be one of its most important internal assets in obtaining a sustainable competitive advantage. Also in software engineering, a substantial amount of technical know-how is required in order to successfully deploy the organizational adoption of a technology or application. In this paper, we show how knowledge on the development of evolvable software can be managed and incorporated into a knowledge base, to enable the more productive construction of evolvable systems. The Normalized Systems (NS) theory offers well-founded knowledge on the development of highly evolvable software architectures. This knowledge is captured in the form of Normalized Systems elements, which can be regarded as design patterns. In this paper, it is discussed how Normalized Systems elements facilitate the management of state-of-the-art knowledge in four processes: (1) knowledge creation, (2) knowledge storage/retrieval, (3) knowledge application, and (4) knowledge transfer. Based on this discussion, it is shown how lessons can be drawn from the NS approach for the management of software engineering knowledge.**

***Keywords*-*Normalized Systems*; *Design Patterns*; *Knowledge Management*.**

I. Introduction

As we highlighted in our previous work on which this paper further elaborates [1], an important movement within the strategic management literature, the resource-based view of the firm (RBV) states that internal resources (e.g., money, patents, buildings, geographical location,etc.) are the key elements for organizations in order to obtain a sustainable competitive advantage [2]. More specifically, the *knowledge* residing inside a firm is frequently considered to be its most important internal asset [3]. Further, focusing on the case of software adoption and development within organizations, the prevalence of the available knowledge becomes even clearer and knowledge management practices have in this respect been acknowledged frequently [4]. Indeed, information technology in general can be considered as a knowledge-intensive or complex technology innovation, requiring a substantial amount of know-how and technical knowledge by the adopting firm [5]. As a result, the degree of expertise or advanced knowledge of best-practices regarding a certain software technology becomes a decisive factor in the possibility for an organization to successfully deploy and manage it. Consequently, a firm should either already (i.e., prior to the adoption) possess the advanced knowledge required to operate the software technology or engage in *organizational learning* during exploitation.

Organizational learning is generally regarded as the result of individual learning experiences of members of an organization, which become incorporated into the behavior, routines and practices of the organization the individuals belong to [5]. According to Levitt and March [6], such an organizational learning can occur in two general ways: (1) “learning by doing”, which involves a learning process by self-experienced trial-and-error and (2) learning from the direct experiences of other people. While the first type of learning is typically a very profound and thorough way of knowledge gathering, it can be time-consuming, expensive and error-prone in the earliest stages. At this point, know-how, experiences and best-practices formulated by other users (i.e., the second type of organizational learning) come into play. Inside organizations, such knowledge transfers in software development can occur in many different ways, including, for example, explicit knowledge bases or experience repositories [7], “yellow pages” enabling search actions for accessible knowledgeable people [8] and mentoring programs [9]. At the inter-organizational or industrial level, the gathered knowledge can benefit from experience based on many different development projects.

In this paper, we explore how knowledge is managed within Normalized Systems (NS) theory (outlined in Section III). Furthermore we will indicate how this approach is deemed to offer additional benefits in terms of knowledge management compared to other software engineering approaches. In order to do so, the widely accepted framework of Alavi and Leidner [10] (summarized in Section II) will be used to base this claim and position how NS supports knowledge management in the development process of evolvable

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

software. Specifically, the role of knowledge in NS will be demonstrated according to four knowledge processes [10]:

- Knowledge Creation
- Knowledge Storage/Retrieval
- Knowledge Application
- and Knowledge Transfer.

This analysis will be presented in Section IV, after which an overview and a discussion regarding some significant differences between NS patterns and more commonly known design patterns, will be offered in Section V. Finally, we conclude the the paper in Section VI.

This paper is an extension of our previous work [1], as it offers a more in-depth analysis of knowledge management practices regarding NS and includes additional illustrating examples, guided by the widely accepted work of Alavi and Leidner [10].

II. Knowledge Management and Information Technology

Over the last decades, knowledge and knowledge management (KM) have been the subject of research within several disciplines, such as knowledge engineering, artificial intelligence, social science, management science, information science, etc. [11]. Due to this multidisciplinary character of the research topic, knowledge and knowledge management have been defined in numerous ways. In spite of this variety of definitions, the goal of knowledge management can most ordinarily be defined as to facilitate the flow of knowledge. To define what knowledge management covers, Tuzhilin identified some common aspects among the variety of KM definitions [12]. In its essence, the management of knowledge should include the acquisition, conversion, structuring and organizing and sharing of knowledge. These essential and agreed upon components of knowledge management are very similar to the framework formulated by Alavi and Leidner [10], of which the core aspects will be summarized below. Given the fact that Alavi and Leidner [10] have performed an in-depth, overarching and widely cited overview and analysis of knowledge management aspects present during the use of information systems, we choose to discuss and employ this framework in the current paper. Hence, in this section, we highlight this particular framework to analyze and discuss how information technology and IT artifacts in general can be used to engage in knowledge management. Alavi and Leidner distinguish four main processes of knowledge management facilitated by information systems: (1) knowledge creation, (2) knowledge storage/retrieval, (3) knowledge application and (4) knowledge transfer. We will highlight each of these processes briefly in the following sub-sections. The concepts of this framework are represented in Figure 1. During our discussion, we will systematically relate each of the processes to the this figure. Also, we will illustrate the application of this framework in previous work by showing how other authors have used this framework to analyze their knowledge management efforts by using information systems [13]. Later on, we will use this framework as our starting point for analyzing how NS theory enables an efficient way of knowledge management.

A. Knowledge Creation

Before one can only begin to point out the importance of knowledge management, knowledge needs to be *created*. Such knowledge creation can both entail the development of new content or the replacement or improvement of already existing knowledge within the organization. Although this creation process always fundamentally starts from an individual, interactions between individuals are an equally important factor in the knowledge creation process [14]. These interactions are represented by the conversion types of knowledge presented by Nonaka, which are based on the differentiation of explicit and tacit knowledge [14]. Based on the conversion of knowledge between these two types, he distinguishes four possible modes of knowledge creation (although they are mentioned to be often interdependent and intertwined in reality) [10]: (1) *socialization* (the transfer of one's personal tacit knowledge to new tacit knowledge of another person), (2) *externalization* (the conversion of tacit knowledge to new explicit knowledge, such as formulated in best practices), (3) *internalization* (the conversion of explicit knowledge to one's tactic knowledge, such as truly understanding some read findings) and (4) *combination* ("the creation of new explicit knowledge by merging, categorizing, reclassifying, and synthesizing existing explicit knowledge").

Each of these knowledge creation modes are also visually represented in Figure 1. More specifically, arrow E represents socialization, arrows C represent externalization, arrows D represent internalization and arrow F represents combination. For each of the discussed knowledge creation modes, facilitating conditions or environments can be considered. Also, for these processes, the interaction with information systems and technology is twofold. On the one hand, the knowledge creation modes facilitate the amassment of knowledge on technologies and information systems used within an organization. On the other hand, information systems can be used to facilitate each of these knowledge creation modes in several ways (e.g., by the use of information systems for collaboration support).

As an example of academic efforts to identify knowledge creation, one can cite the work Lee et al. [13], who studied the knowledge management of a Korean automobile company. More specifically, they reviewed the process of making engineering changes to the finished design of automobiles from a knowledge-management perspective. They however found that little attention is payed to the knowledge creation within the knowledge-intensive process of making engineering changes. The knowledge that is amassed from a design change is documented in separate documents, without

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

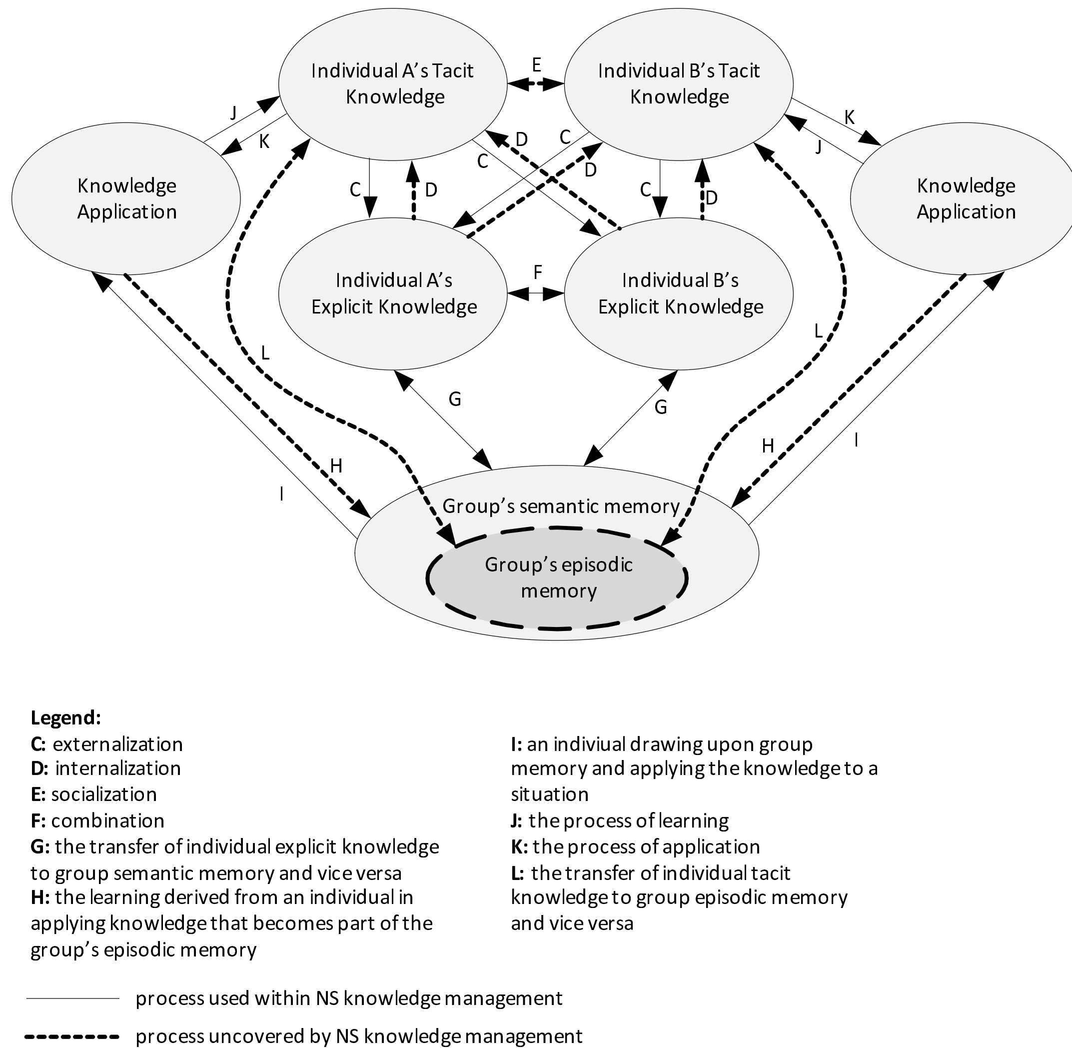

Figure 1. Visual representation of different knowledge management processes, adapted from Alavi and Leidner [10].

being captured in a knowledge base or incorporated into (other) workflows. Lee et al. therefore argue that due to the importance of the knowledge creation process, it should be supported better by proper knowledge management and a more elaborate Knowledge Management System (KMS) [13].

B. Knowledge Storage/Retrieval

To fully leverage its value, knowledge needs to be diffused across individuals within a company, research area,etc. once it has been created. Otherwise, companies lose the knowledge by simply forgetting the newly created knowledge or by forgetting it exists [15], [16]. However, this can be prevented by what literature identifies as individual and organizational memories, which are used within the knowledge storage and retrieval process [10]. Whereas the individual memory is simply referring to the knowledge of a single person, organizational memory is defined as "the means by which knowledge from the past, experience, and events influence present organizational activities" [17]. Such organizational memory can be facilitated by several means, including written documentation, databases with structured information, etc. Also, to a certain extent, organizational memory may extend traditional individual memories by including components such as organizational culture, structure, information archives, and so on. It may be subdivided in semantic memory (i.e., general, explicit and articulated memory) and episodic memory (i.e., context-specific and situated knowledge) [18]. Organizational memory is claimed to have both possible positive effects (e.g., by being able to reuse good solutions in the form of standards and procedures and by avoiding to make mistakes again) and negative effects (e.g., decision-making biases or status quo tending behavior). The authors of [10] mention database management techniques, document management

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

and groupware applications as typical IT means to develop and maintain (i.e., store) the “memory” of an organization. Additionally, IT artifacts in terms of design patterns have been claimed to provide useful means to store and retrieve organizational knowledge and best-practices. This is obvious within the domain of software engineering, as for instance initiated by the work of Gamma et al. [19]. The use of design patterns facilitates the translation of individual knowledge to organizational knowledge and the opposite translation (see the bidirectional arrow G in Figure 1) by offering a central organizational knowledge base (i.e., the design patterns). The design patterns within this knowledge repository can be accessed by all programmers for both storage and retrieval of the latest iteration of the design patterns. In this way, the central knowledge base acts as the organizational memory and thereby “helps in storing and reapplying workable solutions in the form of standards and procedures, which in turn avoid the waste of organizational resources in replicating previous work” [10]. In Section V, we will further elaborate on this specific way of enabling knowledge storage and retrieval by means of design patterns and how these more traditional design pattern approaches differ from the Normalized Systems theory patterns on which we focus in the remainder of the paper.

In Figure 1, the various kinds of knowledge repositories are represented by the ovals, as they represent both knowledge repositories or memories at the organizational and individual level. Obviously, as the aim in organizational knowledge management is to leverage the storage and retrieval of the organizational memory, the main focus should be placed on the ovals labeled as containing group memory. The processes of knowledge storage and retrieval, are represented by the bidirectional arrow G.

In the previously discussed research of Lee et al. [13], knowledge is stored in an intranet system. All engineering change requests (ECRs) and engineering change orders (ECOs) are automatically stored in a repository. The problem with this repository is however that it does not provide the functionality to navigate for relevant knowledge. Another important aspect that the repository lacks is the possibility of linking specific engineering changes with related problems, solutions,etc. These shortcomings clearly show the different levels and possible implementations of knowledge storage management [13].

C. Knowledge Application

The only way an organization can valorize the knowledge stored in its organizational memory, is by eventually applying the knowledge. Such applications might be realized by practices ranging between a continuum from directives (i.e., a set of specific rules, standard and procedures to make the tacit knowledge of specialists explicit for efficient communication, including non-specialists) to self-contained task mechanisms (i.e., a group of individuals with prerequisite knowledge in several domains which becomes combined in the considered team without explicitly formulated routines or procedures).

As typical examples of the usage of IT systems, corporate intranets are mentioned as useful means to access and maintain directives. Similarly, workflow automation systems and rule-based expert systems are suggested as interesting IT artifacts to enable the efficient automation of captured organizational knowledge and procedures.

Also, a large amount of codified best-practice might generate a new problem as the organizational members need to become competent in choosing the adequate best-practices to be employed.

The extent to which practitioners depend on the application of centrally available knowledge is demonstrated by Lee et al. [13]. When valuable knowledge about the incorporation of design changes is not readily available, engineers have to solely rely on their own tacit knowledge and off-line communication with colleagues to deal with challenging engineering changes. This shows that the application of knowledge highly depends on the available knowledge stored in a repository or transferred between agents.

D. Knowledge Transfer

The fourth knowledge management process discussed by Alavi and Leidner is the transfer of knowledge [10]. This process is considered to be an important process in knowledge management, as it provides the transfer of knowledge between individuals, groups, organizations and other sources. In spite of the acknowledgment of its importance, the dissociation of knowledge transfer from knowledge sharing is still unclear and both terms are often used interchangeably in academic literature [20]. Transfer and sharing can however be differentiated based on some parameters. While knowledge transfer is considered to be focused and direct communication to a receiver, sharing is far more a way of diffusing knowledge widespread (i.e., to multiple people via for example a repository). These two definitions are however just two extremes of an hypothetical continuum in which characteristics of both terms can be combined [21]. Others authors also point out that knowledge sharing is about exchanging tacit knowledge [22], while knowledge transfer exchanges more explicit knowledge [23], [24].

Within the case study of Lee et al. [13], one out of three problems related to a change in the final design of a car where not new. Because of significant differences between car models, knowledge on a specific problem (requiring a design change) on one component or car model cannot be easily transferred to another component or car model. The knowledge in this case can therefore be classified as very context-specific, consistent with the definition of episodic knowledge by El Sawy et al. [13], [18]. This example

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

therefore shows how the type of knowledge also impacts the transfer of knowledge.

III. Normalized Systems

The Normalized Systems (NS) theory starts from the postulate that software architectures should exhibit *evolvability* due to ever changing business requirements, while many indications are present that most current software implementations do not conform with this evolvability requisite. Evolvability in this theory is operationalized as being the absence of so-called *combinatorial effects*: changes to the system of which the impact is related to the size of the system, not only to the kind of the change which is performed. As the assumption is made that software systems are subject to unlimited evolution (i.e., both additional and changing requirements), such combinatorial effects are obviously highly undesirable. In the event that changes are dependent on the size of the system and the system itself keeps on growing, changes proportional to the systems size become ever more difficult to cope with (i.e., requiring more efforts) and hence hampering evolvability. Normalized Systems theory further captures its software engineering knowledge by offering a set of four theorems and five elements, and enables the application of this knowledge through pattern expansion of the elements. The theorems consist of a set of formally proven principles which offer a set of necessary conditions which should be strictly adhered to, in order to obtain an evolvable software architecture (i.e., in absence of combinatorial effects). The elements offer a set of predefined higher-level structures, primitives or "building blocks" offering an unambiguous blueprint for the implementation of the core functionalities of realistic information systems, adhering to the four stated principles [25].

A. Theorems

Normalized Systems theory proposes four theorems, which have been proven to be necessary conditions to obtain software architectures in absence of combinatorial effects [26] :

- *Separation of Concerns*, requiring that every change driver (concern) is separated from other concerns by separating it in its own construct;
- *Data Version Transparency*, requiring that data entities can be updated without impacting the entities using it as an input or producing it as an output;
- *Action Version Transparency*, requiring that an action entity can be upgraded without impacting its calling components;
- *Separation of States*, requiring that each step in a workflow is separated from the others in time by keeping state after every step.

In terms of knowledge management, as mentioned explicitly in [27], it must clearly be noted that the design theorems proposed are not new themselves; in fact, they relate to well-known (but often tacit or implicit) heuristic design knowledge of experienced software developers. For instance, well-known concepts such as an integration bus, a separated external workflow or the use of multiple tiers can all be seen as manifestations of the Separation of Concerns theorem [27]. Consequently, the added value of the theorems should then rather be situated in the fact that they (1) make certain aspects of that heuristic design knowledge explicit, (2) offer this knowledge in an unambiguous way (i.e., violations against the theorems can be proven), (3) are unified based on one single postulate (i.e., the need for evolvable software architectures having no combinatorial effects) and (4) have all been proven in a formal way in [26].

B. Normalized Systems Elements as Patterns

The theorems stated above illustrate that traditional software primitives do not offer explicit mechanisms to incorporate the principles. As (1) each violation of the NS theorems during any stage of the development process results in a combinatorial effect, and (2) the systematic application of these theorems results in very fine-grained structures, it becomes extremely challenging for a human developer to consistently obtain such modular structures. Indeed, the fine-grained modular structure might become a complexity-issue on its own when performed "from scratch". Therefore, NS theory proposes a set of five elements as encapsulated higher-level patterns complying with the four theorems:

- *data elements*, being the structured encapsulation of a data construct into a data element (having get- and set-methods, exhibiting version transparency,etc.);
- *action elements*, being the structured encapsulation of an action construct into an action element;
- *workflow elements*, being the structured encapsulation of software constructs into a workflow element describing the sequence in which a set of action elements should be performed in order to fulfill a flow;
- *connector elements*, being the structured encapsulation of software constructs into a connector element allowing external systems to interact with the NS system without calling components in a stateless way;
- *trigger elements*, being the structured encapsulation of software constructs into a trigger element controlling the states of the system and checking whether any action element should be triggered accordingly.

More extensive descriptions of these elements have been included in other papers (e.g., [25]–[27]). As these elaborated descriptions would offer little to no value to this paper, they were not included here. Each of the elements is a pattern as it represents a recurring set of constructs: besides the intended, encapsulated core construct, also a set of relevant cross-cutting concerns (such as remote access, logging, access control, etc.) is incorporated in each of these elements. For each of the patterns, it is further described

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

in [27] how they facilitate a set of anticipated changes in a stable way. In essence, these elements offer a set of building blocks, offering the core functionalities for contemporary information systems. In this sense, the NS patterns might offer the necessary simplification by offering pre-constructed structures that can be parametrized during implementation efforts. This way the NS patterns dictate the source code for implementing the pattern.

Regarding these patterns, it can be noted that their definition and identification are based on the implications of the set of theorems. For instance, the theorems Separation of Concerns (SoC) and Separation of States (SoS) indicate the need to formulate a workflow element. Contrary (and in addition) to an action element, such a workflow element allows the stateful invocation of action elements in a (workflow) construct. The SoS principle indeed requires this kind of stateful invocation and the SoC principle demands that the concern of invocation is handled by a separate construct. Next, each of the five patterns themselves contain knowledge concerning all the implications of the theorems referred to in Section III-A. Finally, each of these patterns has been described in a very detailed way. Consider for instance a data element in a Java Enterprise Edition (JEE) implementation (a widely used platform for the development of distributed systems) [28]. In [27] it is discussed how a data element `Obj` is associated with a bean class `ObjBean`, interfaces `ObjLocal` and `ObjRemote`, home interfaces `ObjHomeLocal` and `ObjHomeRemote`, transport classes `ObjDetails` and `ObjInfo`, deployment descriptors and EJB-QL for finder methods. Additionally, methods to manipulate a data element's bean class (create, delete, etc.) and to retrieve the two serializable transport classes are incorporated. Finally, to provide remote access, an agent class `ObjAgent` with several lifecycle manipulation and details retrieval methods is included. It can be argued that these elements incorporate the main concerns which are relevant for their function.

Moreover, the complete set of elements covers the core functionality of an information system. Consequently, as such detailed description is provided for each of the five elements, an NS application can be considered as an aggregation of a set of instantiations of the elements. Consider for example the implementation of an observer design pattern [19]. In order to implement this pattern in NS, three data elements (i.e., `Subscriber`, `Subscription` and `Notification`) are required. A `Notifier` connector element will observe the subject, and create instances of the `Notification` data element. These `Notification` data elements will be sent to every `Subscriber` that has a `Subscription` through a `Publisher` connector element. The sending is triggered by a `PublishEngine` trigger element which will periodically activate a `PublishFlow` workflow element. Consider that each NS element consists of around ten classes [25]. The seven identified elements therefore result in around seventy classes used to implement the design pattern, whereas the original implementation of the design pattern consists of two classes and two interfaces. Consequently, it is clear that, in order to prevent combinatorial effects, a very fine-grained modular structure needs to be adhered to.

C. Pattern Expansion

As stated before, in practice, the very fine-grained modular structure implied by the NS principles seems very unlikely to arrive at without the use of higher-level primitives or patterns. The process of defining these patterns and transforming them into code is shown in Figure 2. As NS proposes a set of five elements which serve for this purpose, this figure shows how the actual software architecture of NS conform software applications can be generated relatively straightforward. First the requirements of the application are translated in instantiations of the five NS elements. To achieve generated software code, these instantiations need to be created. Therefore, the instantiations are coded into so-called descriptor files, which are text- or XML-based files describing the inputs for the expanders. For example, in case of the data element pattern, the pattern expansion mechanism needs a set of parameters including the basic name of the data element (e.g., `Invoice`), context information (e.g., component and package name) and data field information (e.g., data type). The expanders then generate skeleton source code for all these instantiations, together with all deployment and configuration files required to construct a working application on one of several technology stacks, such as Java Enterprise Edition. For the invoice example, this would be the set of classes and data fields: the bean class `InvoiceBean`, interfaces `InvoiceLocal` and `InvoiceRemote`,etc. As the code generation process is typically very fast, this allows for interactive sessions to use the generated application to validate the correctness of the descriptor files. Next, extensions can be added to the generated code, but only in very specific pre-defined locations in the generated code to ensure that the extension do not compromise the control of combinatorial effects. Extensions can be inserted typically in the implementation class of an action element, or more generally in pre-specified anchors in the code. Next, these extensions are harvested by automated tools and stored separately from the skeleton code. When a new version of the expanders is built, for example with new frameworks in the web tier or in the persistence tier, or with minor upgrades, the application is re-generated by first expanding the skeleton code and then injecting the extensions.

In terms of knowledge management, it should be noted that the patterns and the expansion mechanism should not be considered as separate knowledge reuse mechanisms: rather, the pattern expansion facilitates the re-use of knowledge embedded in the patterns, as each expansion of the patterns

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

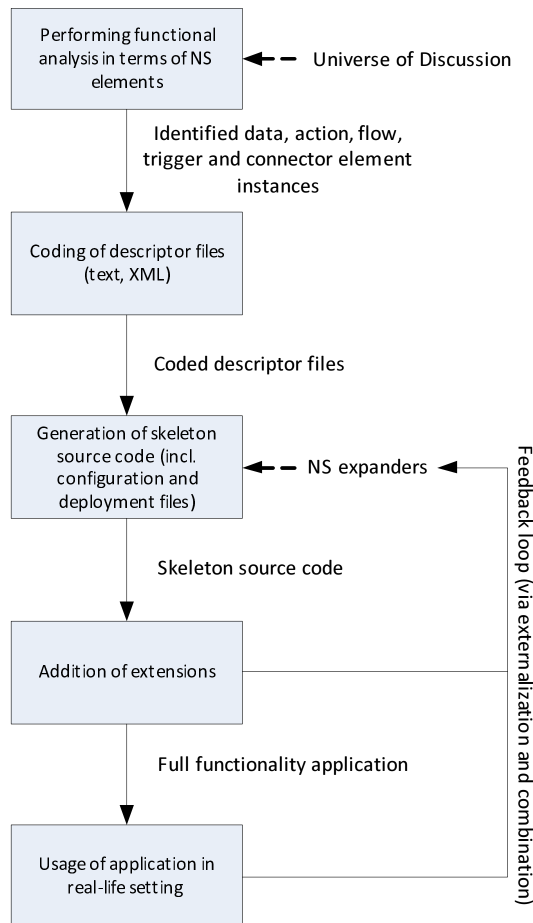

Figure 2. Visual representation of the Normalized Systems development process.

results in a new application of the knowledge encapsulated in the pattern. Through this, pattern expansion facilitates both types of learning discussed earlier (i.e., "learning by doing" and learning from experience of other people) by utilizing the knowledge contained in the patterns.

Also, the information codified in a pattern may not be sufficient to adequately transfer the intended knowledge. This is even the case when using the design patterns proposed by Gamma et al. [19]. For example, it has been claimed that the *Dependency Inversion Principle* helps to gain a better understanding of the *Abstract Factory* pattern [29]. Similarly, the structure of the NS patterns can only be understood when the NS theorems are taken into account.

IV. Normalized Systems Patterns as Knowledge Management

In the previous sections, we explained four processes that are widely regarded to be the essential processes of knowledge management. We also outlined how Normalized Systems theory employs a set of NS patterns to represent a fine-grained modular structure which can be systematically expanded to provide an evolvable software architecture. In this section, we discuss how the use of NS patterns seems to facilitate each of the four essential processes of knowledge management as identified by Alavi and Leidner [10].

A. Knowledge Creation

As discussed in section III-B, Normalized Systems theory relies on the use of patterns to capture design knowledge. One of the main purposes of the use of patterns and the associated pattern expansion mechanism, is to easily incorporate new knowledge into the patterns themselves, and the expanded NS applications in a second stage. Therefore, Normalized Systems theory can be considered to easily facilitate knowledge creation using IT artifacts (i.e., elements as design patterns). This opportunity for knowledge creation can be interpreted from two distinct perspectives.

First, improvements (i.e., new content) or changes (i.e., replacement of already existing knowledge such as typical bug fixing or a new kind of algorithm) regarding the actual functional parts of the system (i.e., the so-called 'tasks') are easily incorporated in the whole system (i.e., transformed from tacit into explicit knowledge). This because functional parts that are different change drivers are separated according to NS principles, meaning a single functional part is the only place where any modifications have to be made and the remainder of the system can easily interact with the new task (and hence, use this knowledge). In NS terms, we could call this kind of changes and expertise inclusions, knowledge dispersion at the "*sub-modular level*" as only changes and new knowledge are incorporated at the sub-modular level of the tasks (and not in the modular structure of the elements). In order to illustrate this first kind of knowledge creation in NS, consider the developments on the connector element. A *user connector* element allows a user to interact with the application, for example by offering create, read, update, delete and search (CRUDS) functionality on a data element. Such connector elements are expanded based on the parametrization of the data elements, resulting in separate CRUDS screens for every data element. In certain applications, the end users requested that CRUDS functionality for different data elements is combined within one page. This could be achieved, but only by adding extensions to the expanded code from the connector element. These extensions were performed by the same team of programmers over and over again. After several iterations, different ways of integrating CRUDS functionality emerged, which were referred to by the programmers of these extensions using specific names. For example, a screen where a linked data element is added below another data element is referred to as a "waterfall screen". For such a waterfall screen, a reoccurring extension needs to be made every time. Once the specific code for creating this screen is separated from other concerns, it can be added to the connector element. Therefore, the user connector element

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

will be updated to provide the expansion such waterfall screen without needing any extension. According to the programmers, this can only be achieved because all other concerns are removed from within the functional class of the connector element, allowing them to focus solely on the organization of the user interface components.

Second, knowledge can be incorporated at the "*modular level*" as well. This kind of knowledge inclusion would include change (e.g., an extra separated class in the pattern) and modifications (e.g., improved persistence mechanism) regarding the internal structure of an element (the pattern). Indeed, once the basic structure or cross-cutting concern implementation of an element is changed due to a certain identified need or improvement, the new best-practice knowledge can be expanded throughout the whole (existing) modular structure and used for new (i.e., additional) instantiations of the elements. In order to further illustrate this second kind of knowledge creation based on NS patterns, consider the following example, based on real-life experience from developers using NS.

For instance, one way to adopt a model-view-controller (MVC) architecture in a JEE distributed programming environment is by adopting (amongst others) the Struts framework. In such MVC architecture, a separate controller is responsible for handling an incoming request from the client (e.g., a user via a web interface) and will invoke (based on this request) the appropriate model (i.e., business logic) and view (i.e., presentation format), after which the result will eventually be returned to the client. Struts is a framework providing the controller (ActionServlet) and enabling the creation of templates for the presentation layer. Obviously, security issues need to be handled properly in such architecture as well. Applied to our example, these security issues in Struts were handled in the implementation of the Struts Action itself in a previous implementation of our elements. In other words, the implementation class itself was responsible for determining whether or not a particular operation was allowed to be executed (based on information such as the user's access rights, the screen in which the action was called, etc.). As a result, this "security function" became present in all instantiations of an action element type (i.e., each session). Moreover, this resulted in a combinatorial effect as the impact of a change such as switching towards an equivalent framework (i.e., handling similar functions as Struts), would entail a set of changes dependent on the number of instantiated action elements (and hence, on the size of the system). In order to solve the identified combinatorial effect, the Separation of Concern theorem has to be applied: separating the part of the implementation class responsible for the discussed security issues (i.e., a separate change driver) in its own module within the action element. In our example, a separate interceptor module was implemented, next to the already existing implementation class. This way, not only the combinatorial effect was excluded, but the new knowledge in terms of a separate interceptor class was applied to all action elements after isolating the relevant implementation class parts and executing the pattern expansion. Additionally, all new applications will use the new action element.

Considering the underlying idea of design patterns and the NS element, namely to transform tacit knowledge into explicit knowledge, one can readily understand why theories using design patterns (such as NS) mostly rely on "externalization" and "combination" regarding the relevant knowledge management aspects. Thereby, both these knowledge creation processes refer to the definition of new explicit knowledge, be it from existing tacit knowledge (i.e. externalization) or existing explicit knowledge (i.e. combination). First, the use of *externalization* is demonstrated by the fact a lot of good programming practices (i.e., best-practices) are incorporated in the structure of the elements themselves. Indeed, while the NS theorems prescribe a set of necessary conditions in order to attain evolvable and easily adaptable software architectures, the elements provide a constructive proof and explicit way of working regarding how to achieve this in reality, which is generally conceived to be only attainable by very highly experienced and skilled programmers. For instance, designing software architectures in such a way that the cross-cutting concerns are integrated in a fine-grained modular way is considered to be rather challenging. The formulation of the elements in combination with the expansion mechanism allow a way to externalize this experience and apply it at large scale. Second, one example of the use of *combination* to formulate new explicit design knowledge within NS, is the elimination of combinatorial effects within a software application. Whenever violations of the four NS principles are discovered within new software, programmers report the violations and their effects to their colleagues and supervisors. This way, a solution can be found for eliminating the violations (using both tacit and explicit knowledge).

B. Knowledge Storage/retrieval

Knowledge storage and retrieval should ensure that a certain expertise within companies is retained and placed easily at the disposal of the relevant people within the organization, in order to be applied at a later stage. In the Normalized Systems approach, a major part of the knowledge is stored within the NS elements. These elements offer a standardized way to create (i.e., generate) software applications by prescribing a set of predefined and systematically re-used modules. Consequently, the use of design patterns (i.e., the NS elements) facilitates the translation of individual knowledge to organizational knowledge by offering a central organizational knowledge base (i.e., the design patterns). The design patterns within this knowledge repository can be accessed by all programmers for both storage and retrieval of the latest iteration of the design pattern, and can hence

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

be considered to be a part of the organizational memory. With Normalized Systems, this advantage of re-using the “standard” design patterns is in fact exceeded by the benefit of using a solution that —from an evolvability viewpoint— is proven to be optimal.

To further position knowledge used within Normalized Systems, we can refer once more to the classification of El Sawy et al. [18], which breaks down organizational memory into semantic memory and episodic memory. As the design patterns formulate a sound software structure that is generally applicable to every software application, this knowledge should be classified as semantic knowledge. The opposite of this general and explicit semantic knowledge is the so-called episodic knowledge, which is defined as context-specific knowledge. As Normalized Systems formulates general software architecture principles for software, this type of organizational knowledge is not part of the general Normalized Systems patterns. This context-specific knowledge in incorporated in so-called extensions that are added to the software after the expansion based on the five recurrent elementary structures. Because the patterns are detailed enough to be instantiated, no manual implementation of the patterns (as is the case with the design patterns proposed by Gamma et al. [19]) is required. Consequently, an identical code structure reoccurs in every application which is created using the expansion of NS elements. The commonality of the structure of the patterns makes that once one understands the patterns, one understands all its instantiations as well. In this way, it could be argued that —at least partially— the pattern structure becomes the documentation. Therefore, no source code level documentation is required and all knowledge is stored in the NS patterns. Such advantages can only be achieved for semantic knowledge, since episodic knowledge is different is various contexts.

C. Knowledge Application

In the Normalized Systems theory rationale, the knowledge present in the NS elements is applied by employing the elements as a design template for evolvable software. Each NS compliant software application is an aggregation of a set of instantiation of one of the five NS elements. Therefore, the knowledge of NS contained in the NS elements and their accompanying expansion mechanism can be considered to be prescriptions or directives as defined by Grant [3] (i.e., a set of rather unambiguous and specific standards or rules used to guide the actions of persons). When writing a software application, a programmer retrieves the latest version of the software design patterns from the knowledge repository. Afterwards, the element instances are parametrized and configured in descriptors files (e.g., the relevant fields, relationships,etc. for a data element are specified). Hence, by combining their tacit knowledge with the described structure of the NS elements, the programmers build evolvable software.

The use of the NS elements and theorems indeed results in evolvable and easily adaptable software architectures. For instance, an important characteristic of these structures is that they separate technology-dependent aspects from the actual implementation, resulting in the fact that one can easily switch the underlying technology stack of the software. One transition that has been performed, is changing the underlying implementation architecture from Enterprise Java Beans (EJB) version 2 to EJB version 3. Because these standards encapsulate the business logic of an application, they use a different way of communicating between agents and beans. Therefore, this transition normally is a labor-intensive and difficult task. Using the architecture described in this paper, this transition can however be achieved rather easily by using the pattern expansion mechanism. This is because the expanders that perform the expansion are very similar for different technologies. This is done by clearly separating functional requirements of the system (i.e., input variables, transfer functions and output variables) from constructional aspects of the system (i.e., composition of the system). Whereas all constructional aspects are described in patterns, functional aspects are separately included in descriptor files (such as data elements, action elements,etc.). As each pattern can be conceived a recurring structure of programming constructs in a particular programming environment (e.g., classes), one can conclude that the functional/constructional transformation then becomes located at one abstraction level higher than before.

An important result from the application of knowledge is that it is often combined with a learning process. By building software using the expansion of NS elements, the programmers improve both their tacit knowledge on building (evolvable) software and explicit knowledge that will be incorporated in the design pattern (i.e., NS elements). This increased tacit knowledge (“experience”) will over time also contribute to the definition of changes to the design patterns. The inherent way of working implied by the NS expansion mechanism (i.e., expanding software architectures by systematically instantiating the NS elements, and incorporating new bits knowledge again into this core of patterns) also efficiently copes with the issue articulated in Section II-C, namely that the automated ways of working should be continually kept up-to-date.

D. Knowledge Transfer

Within a knowledge system, knowledge is transferred from where it is available (i.e., a repository) to where it is needed. For Normalized Systems, the knowledge repository of the NS design patterns (NS elements) needs consistent updating to reflect the most recent software architecture for evolvable software. This is done by transferring the new explicit design knowledge created by individuals to the group semantic knowledge repository of NS elements. The use of this repository can be characterized as impersonal

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

and formal, which promotes a faster and further distribution and is a good way to transfer knowledge that can be readily generalized to other contexts (which is the case for NS) [10]. Analogously to the discussion in Section II, the exchange of explicit knowledge (i.e., NS elements) in NS theory can be classified most appropriately as knowledge transfer. The use of a NS repository however bears closer resemblance to knowledge sharing process. This shows that the exchange of NS knowledge should be placed on the previously discussed continuum between knowledge transfer and knowledge sharing.

V. Discussion

In this section, in order to provide a summarizing overview of our analysis presented above, we will first discuss to which extent the knowledge management practices within Normalized Systems cover all aspects as identified by Alavi and Leidner [10], based on Figure 1. Next, we will present some reflections with respect to design patterns in general and how the use of elements within Normalized Systems seems to enhance the existing practices of design patterns regarding knowledge management on several domains.

A. Overview of knowledge management aspects of Normalized Systems

To recapitulate the NS knowledge management processes, we will discuss these processes according to the representation introduced by Alavi and Leidner [10], as shown in Figure 1. This figure has been adapted to represent whether or not the defined knowledge processes are used within Normalized Systems theory. This is done by indicating those processes which are not covered by the Normalized Systems theory by dotted lines. Consequently, full lines indicate the knowledge processes that are used within NS knowledge management.

Regarding the knowledge management creation processes (i.e., *arrows C, D, E, and F*), we can notice that the Normalized Systems theory primarily enables the externalization and combination processes (i.e., the processes indicated by the *arrows C* and *F* respectively). These processes aim to make implicit knowledge and best practices explicit (as is done by the formulation of the NS theorems and elements) and to combine already existing explicit knowledge among several members of the group (e.g., discussing additional concerns which need to be separated or improvements of the current elements). While the processes of socialization (i.e., *arrow E*) and internalization (i.e., *arrow D*) might occasionally occur in the NS community, those aspects are not explicitly managed within the Normalized Systems rationale. Indeed, the aim of the Normalized Systems approach is to design evolvable software architectures based on formally proven and tested (and hence, explicit) principles and their implications.

The bidirectional interaction between an individual's explicit knowledge repository and the group's memory is visualized trough *arrow G*. In NS reasoning, such explicit knowledge (e.g., the formally known need to separate a certain external technology in a distinct module) becomes embedded in the group's memory by incorporating it in the general structure of the NS elements and might subsequently offer new insights regarding the explicit knowledge of another person as well. Also *arrows K* and *J* are relevant in a NS context. The first represents the application of a developer's new tacit insights (i.e., new possible improvements of the elements) into a trial-version of the elements, while the latter may occur in the situation where the real-life implementation of software in an organizational setting may point out that a certain part of of a 'task' implemented in an action element evolves independently in a realistic setting, thus constitutes a separate change driver and should consequently be separated.

The transfer of individual tacit knowledge to group's episodic memory (i.e., *arrows L*), is a type of knowledge transfer that is not used in NS knowledge management. This simply because the knowledge repository does not include any type of episodic memory, rendering the transfer of knowledge non-existent. Arguably the most important transfer of knowledge within NS theory is the expansion of NS elements into evolvable software. This transfer is shown by *arrows I*, which represent the repeated use of design patterns (i.e., the NS elements) for building agile software. In the opposite direction, directly learning from the application of the NS elements is not supported in the knowledge management for NS theory (i.e., *arrows H*). As the Normalized Systems rationale stipulates a deterministic and proven way of constructing evolvable software based on the NS design theory, it does not allow new knowledge to be formulated directly from the application of the NS elements. Instead, new knowledge should always be rigorously verified by traditional knowledge creation processes of externalization and combination before being added to the existing knowledge base of NS elements.

Finally, the extension of the NS knowledge management to multiple groups will add an extra layer of complexity to the management of knowledge. However, the centrality of the current knowledge base and the limited size of developers working on the development of the NS elements are the reasons these challenges are not the main point of interest at this moment.

B. Knowledge Management using Design Patterns

As discussed in the introduction, knowledge management also plays an important parts in software engineering. The specific use of design patterns in object-orientation during the 90's, exemplified by the seminal work of Gamma et al. [19], was incited by the fact that modern computer literature regularly failed to make tacit (success determining) knowl-

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

edge regarding low-level principles of good software design explicit [30]. Patterns provide high-level solution templates for often-occurring problems. The patterns proposed by Gamma et al. [19] were conceived as the bundling of a set of generally accepted high-quality and best-practice solutions to frequently occurring problems in object-orientation programming environments. For instance, in order to create an one-to-many dependency between objects so that when the state of one object changes, all its dependents are notified and automatically updated, the observer pattern (i.e., an overall structure of classes giving a description or template of how to solve the concerned problem) was proposed [19]. As a consequence, the use of these patterns can be considered as specifically aimed at facilitating (inter-)organizational learning by learning from direct experiences of other people — in this case experienced software engineers —, and being one specific way of knowledge transfer.

According to Schmidt [31], design patterns have been so successful because they explicitly capture knowledge that experienced developers already understand implicitly. The captured knowledge is called implicit because it is often not captured adequately with design methods and notations. Instead, it has been accumulated through timely processes of trial and error. Capturing this expertise allows other developers to avoid spending time rediscovering these solutions. Moreover, the captured knowledge has been claimed to provide benefits in several areas [32]. Such benefits include (a) documentation of software code, (b) knowledge reuse when building new applications, and (c) incorporation of new knowledge in existing software applications. In this section, we focus on these benefits, and discuss how NS elements can be considered to be an improvement on this way of working.

1) Documentation: Patterns provide developers with a vocabulary which can be used to document a design in a more concise way [19], [32], [33]. For example, pattern-based communication can be used to preserve design decisions without elaborate descriptions. By delineating and naming groups of classes which belong to the same pattern, the descriptive complexity of the design documentation (e.g., a UML class diagram) can be reduced [33]. Consequently, the vocabulary offered by patterns allows a shift in the abstraction level of the discussions. This usage of design patterns is mostly applied at the conceptual level, and neglects the source code documentation. However, the abstract nature of patterns, i.e., as a solution template, means that it is possible to implement a certain design pattern using different alternatives. Therefore, it has been argued that the addition of source-code level documentation of the pattern usage is required to perform coding and maintenance tasks faster and with fewer errors [34].

In NS, the structure of the five software patterns could be described in a similar way. The focus would then be on the different concerns which need to be separated in each element. As discussed in Section III, each concern needs to be encapsulated in a separate module (e.g., a class in the object-oriented paradigm). Consequently, the different concerns dictate the modular structure of the element. As a result, this documentation could provide similar insights as obtained by traditional design patterns. However, the NS elements are described in such detail that they can be expanded, resulting in working code. In Section IV-B, we discussed how the reoccurring code structure in itself becomes the documentation for expanded software: because a certain piece of code is identical in every expanded instance, a programmer only needs to inspect this piece of code once in order to understand how that particular piece works. This eliminates the need for including documentation in every instance of source code. In conclusion, the pattern expansion allows documentation at the pattern level to be sufficient, eliminating the need for code-level documentation.

2) Using knowledge to build new applications: Several authors propose the usage of design patterns to create new software applications (e.g., [35]). Earlier we discussed how patterns provide high-level solution templates, and consequently, do not dictate the actual source code. As a result, knowledge concerning the implementation platform remains important. A correct and efficient implementation of a design pattern requires a careful selection of language features [31]. Clearly, design patterns alone are not sufficient to build software. As a result, the implementation of a design pattern during a software development process remains essentially a complex activity [31]. Developing software for a concrete application then requires the concrete experience of a domain and the specifics of the programming language, as well as the ability to abstract away from details and adhere to the structure prescribed by the design pattern. Nevertheless, certain companies and researchers attempt to integrate the knowledge available in design patterns in other approaches, in order to create automated code generation. For example, so-called software factories attempt to create software similar to automated manufacturing plants [36]. This should drastically improve software development productivity. However, such approaches have not yet reached widespread adoption.

The code expansion which occurs when using NS elements needs to be distinguished from this approach. Consider for example the action element. The functional class of such an element still needs to be programmed manually. However, the code for reoccurring concerns, such as remote access, can be expanded since this code is identical for different action element instantiations. Similarly, an instantiation of a data element needs to be functionally defined (i.e., through descriptor files which contain data field definitions). However, the concerns which reoccur in every data element instantiation are expanded. In Section IV-C, these are referred to as constructional elements. Consequently, the building of new applications applies code reuse to an

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

extent as large as possible: the common source code, which is similar for all element instantiations, is expanded, while functional requirements need to be provided by the programmer. As a result, an optimal way of using knowledge to build applications is applied, without restricting the programmer in addressing functional requirements.

3) Incorporating new knowledge in existing applications: Because of the increasing change in the organizational environment in which software applications are used, adaptability is considered to be an important characteristic. However, adapting software remains a complex task. Various studies have shown that the main part of the software development cost is spent after the initial deployment [37], [38]. Several design patterns focus on incorporating adaptability into their solution template. Empirical observations have been reported which confirm the increased adaptability when using design patterns [39]. Adaptations could be made easier in comparison with an alternative that was programmed using no design patterns, and achieved adaptability was retained more successfully because of the prescribed structure. Nevertheless, some researchers also report negative effects on adaptability, caused by the added complexity of the design patterns. By prescribing additional classes in comparison to simpler solutions, more errors have been introduced in some cases [39].

Both observations are consistent with the experiences obtained by developing NS. By separating all concerns within the elements, combinatorial effects are prevented, which allows improved adaptability. Since this is similar to how design patterns work, it shows how NS incorporates existing design knowledge. However, NS prescribes to separate more concerns than traditional patterns or methods. This leads to a very fine-grained, but complex structure. As described by Prechelt et al., such complexity reduces adaptability [39]. Therefore, the pattern expansion mechanism is a crucial component of NS, as discussed in Section III-C. We discussed how code expansion seems to be indispensable for knowledge reuse to separate concerns and prevent combinatorial effects. Moreover, the expansion mechanism also allows to make adaptations in a structured way. When changes or updates are applied to the elements, the expanded code can be updated by either re-expanding, or by using marginal expansion. Marginal expansion updates only parts of already expanded code, without replacing the expanded element as a whole. Consequently, newly generated knowledge (such as, e.g., a newly identified combinatorial effect) can be applied in existing applications as well.

C. Positioning NS as knowledge management

The approach of capturing knowledge using NS as described in this paper clearly deviates from the body of thought of other knowledge management approaches. For example, in the article by Tuzilin [12], the evolution of knowledge management systems from content management systems is discussed, and it is highlighted how the process of making tacit knowledge explicit was initially regarded to be optimal for capturing knowledge. However, as this proved to be an insurmountable challenge, future developments of knowledge management are expected to focus on the indexation of tacit knowledge. Consequently, when knowledge is needed, the responsible knowledge source can be identified, and can be shared without needing to make all tacit knowledge explicit. NS theory opposes this idea. The rate of reuse of the evolvable modular structure of software elements is too high to be supported by such communication-based approaches. Therefore, the knowledge captured in NS (i.e., being the modular structure of evolvable software patterns) is made explicit. Consider for example the implications of the Separation of Concerns theorem, as discussed in Section III-A. It implies that each concern needs to be separated in a separate module. Compare this to a non-normalized system, where multiple concerns are mixed in a module. These concerns are on a sub-modular level, and are not explicitly identified as different concerns. Nevertheless, knowledge of these concerns is vital, since they introduce combinatorial effects, and hence limit evolvability. The knowledge related to which concerns need to be separated is made explicit in NS through the modular structure, as available in the expanders.

Our discussed way of knowledge capture in NS is feasible since a specific kind of knowledge is focused on: the modular structure of software. For organizational knowledge, such a modular structure may not be well-suited, and different systems may be needed here (cf. infra). Nevertheless, many organizational issues are being studied as being modular structures. For example, coordination issues in supply chains have been claimed to be modularity issues [40]–[42]. Consequently, making knowledge concerning such issues explicit in a modular structure could be explored as well.

D. Contributions and Future Work

This paper could be claimed to have several contributions, while indicating several opportunities of future work. Regarding contributions, first, this paper might help in clarifying the particular way of how software applications are built according to the Normalized Systems way of thinking and —more specifically— how this enables the creation, storage/retrieval, application and transfer of knowledge. Our aim was to provide a practical overview, including examples, of how NS might enhance knowledge management in practice, based on a theoretically founded framework and its concepts. Second, this paper illustrates the possibility of readily applying the published framework of Alavi and Leidner [10] to analyze the knowledge management processes regarding a software development approach. To the authors' knowledge, no other researchers have described their software development approach based on this framework in such an extensive way. Therefore, this

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

paper illustrates the benefits researchers might realize by presenting their software development approach according to this framework for clarifying any related knowledge management benefits or issues, as well as the relevance of the considered framework for this purpose. Third, our analysis clearly highlighted how NS differentiates from the direction which is taken in other knowledge management approaches. In NS, knowledge is made explicit by capturing it into modular structures of evolvable software patterns, instead of pure textual descriptions. While we are not the first to argue that more efficient ways of knowledge management can be attained by the use of design patterns, we argued that the NS patterns could be hypothesized to be a sort of "enhanced design patterns" regarding documentation (i.e., only requiring documentation at the pattern level), knowledge usage (i.e., maximal code expansion) and new knowledge incorporation (i.e., including new knowledge by simple re-expansion or marginal expansions).

Regarding future work, we could first notice that, although the discussion in this paper is limited to Normalized Systems theory for software, the theory has recently been applied to both Business Process Management [43] and Enterprise Architecture [44] domains. As part of future research, the possible formulation and investigation of patterns on the level of business processes and enterprise architecture (and their knowledge management implications) can be studied. However, we already mentioned in Section V-C that the concerns (and hence modular structures) identified at these levels are of a different kind. Therefore, the knowledge management issues regarding the identification, storage and "deployment" of such modular structures at the organizational level should be investigated in the future as well.

Another area of future research concerns a more elaborated and detailed way of describing how the discussed knowledge management processes in NS relate to similar software development approaches. The Normalized Systems approach was shown to include and support the four widely adopted types of knowledge management processes. The question however remains how the support of these knowledge management processes by NS precisely relates to other software development paradigms and approaches. Such a comparison calls for rather extensive research efforts and is therefore suggested as part of future research.

VI. Conclusion

Creating, managing and applying knowledge is a crucial competence for organizations today. Therefore, knowledge management is a widely investigated and popular research topic. In this paper, we explored how Normalized Systems theory, and its use of elements and their expansion mechanisms in particular, support knowledge management in the development process of evolvable software. For this purpose, we employed the framework of Alavi and Leidner [10] to analyze how the four essential processes within knowledge management are facilitated in the Normalized Systems reasoning: (1) knowledge creation, (2) knowledge storage/retrieval, (3) knowledge application and (4) knowledge transfer. Our analysis shows that design patterns as a central knowledge repository facilitate the transfer of knowledge from an individual to others in an explicit and efficient way. All processes of Alavi and Leidner [10] seem to be supported by Normalized Systems reasoning. Some transformations are considered to be essential (e.g., new knowledge can be absorbed by arrow J in Figure 1 going from "knowledge application" to an individual tacit knowledge), while others are not directly (but rather indirectly) included in the NS rationale (e.g., not directly incorporating knowledge from applications into the group's memory, but through the knowledge creation processes of externalization and combination).

Further, we showed in this paper that the NS elements can be considered to be enhanced patterns for software development with benefits on three dimensions (i.e., less need for explicit documentation, more deterministic development of new applications and more convenient incorporation of new knowledge into existing applications). From interviews with developers, these benefits have shown to enhance the transfer of knowledge, success rate and the overall quality of NS developments.

Acknowledgment

P.D.B. is supported by a Research Grant of the Agency for Innovation by Science and Technology in Flanders (IWT).

References

[1] P. De Bruyn, P. Huysmans, G. Oorts, D. Van Nuffel, H. Mannaert, J. Verelst, and A. Oorst, "Using normalized systems patterns as knowledge management," in *Proceedings of the Seventh International Conference of Software Engineering Advances (ICSEA)*, Lisbon, Portugal, 2012, pp. 28–33.

[2] B. Wernerfelt, "A resource-based view of the firm," *Strategic Management Journal*, vol. 5, no. 2, pp. 171–180, 1984.

[3] R. M. Grant, "Toward a knowledge-based theory of the firm," *Strategic Management Journal*, vol. 17, pp. 109–122, 1996.

[4] F. Bjrnson and T. Dingsyr, "Knowledge management in software engineering: A systematic review of studied concepts, findings and research methods used," *Information and Software Technology*, vol. 50, no. 11, pp. 1055–1068, 2008.

[5] P. Attewell, "Technology diffusion and organizational learning: The case of business computing," *Organization Science*, vol. 3, no. 1, pp. 1–19, 1992.

[6] B. Levitt and J. G. March, "Organizational learning," *Annual Review of Sociology*, vol. 14, pp. 319–340, 1988.

[7] C. Chewar and D. McCrickaerd, "Links for a human-centered science of design: integrated design knowledge environments for a software development process," in *Proceedings of the Hawaii International Conference on System Sciences*, 2005, p. 256.3.

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[8] T. Dings yr, H. K. Djarraya, and E. Røyrvik, “Practical knowledge management tool use in a software consulting company,” *Communications of the ACM*, vol. 48, no. 12, pp. 96–100, 2005.

[9] F. Bjrnson and T. Dingsyr, “A study of a mentoring program for knowledge transfer in a small software consultancy company,” in *Product Focused Software Process Improvement*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, vol. 3547, pp. 245–256.

[10] M. Alavi and D. E. Leidner, “Review: Knowledge management and knowledge management systems: Conceptual foundations and research issues,” *MIS Quarterly*, vol. 25, no. 1, pp. 107–136, 2001.

[11] N. K. Kakabadse, A. Kakabadse, and A. Kouzmin, “Reviewing the knowledge management literature: towards a taxonomy,” *Journal of Knowledge Management*, vol. 7, no. 4, pp. 75–91, 2003.

[12] A. Tuzhilin, “Knowledge management revisited: Old dogs, new tricks,” *ACM Trans. Manage. Inf. Syst.*, vol. 2, no. 3, pp. 13:1–13:11, 2011.

[13] H. Lee, H. Ahn, J. Kim, and S. Park, “Capturing and reusing knowledge in engineering change management: A case of automobile development,” *Information Systems Frontiers*, vol. 8, pp. 375–394, 2006.

[14] I. Nonaka, “A dynamic theory of organizational knowledge creation.” *Organization Science*, vol. 5, no. 1, pp. 14–37, 1994.

[15] L. Argote, S. L. Beckman, and D. Epple, “The persistence and transfer of learning in industrial settings,” *Manage. Sci.*, vol. 36, no. 2, pp. 140–154, 1990.

[16] E. D. Darr, L. Argote, and D. Epple, “The acquisition, transfer, and depreciation of knowledge in service organizations: productivity in franchises,” *Manage. Sci.*, vol. 41, no. 11, pp. 1750–1762, 1995.

[17] E. W. Stein and V. Zwass, “Actualizing organizational memory with information systems,” *Information Systems Research*, vol. 6, no. 2, pp. 85–117, 1995.

[18] O. A. El Sawy, G. M. Gomes, and M. V. Gonzalez, “Preserving institutional memory: The management of history as an organizational resource.” *Academy of Management Best Papers Proceedings*, vol. 1, pp. 118–122, 1986.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1994.

[20] J. A. Kumar and L. Ganesh, “Research on knowledge transfer in organizations: a morphology,” *Journal of Knowledge Management*, vol. 13, pp. 161–174, 2009.

[21] W. R. King, T. R. Chung, and M. H. Haney, “Knowledge management and organizational learning,” *Omega*, vol. 36, no. 2, pp. 167–172, 2008.

[22] M. Polanyi, *The Tacit Dimension*. Routledge, London, 1967.

[23] M. Hansen, N. Nohria, and T. Tierney, “Whats your strategy for managing knowledge?” *Harvard Business Review*, vol. 77, no. 2, pp. 106–116, 1999.

[24] I. Pinho, A. Rego, and M. Pina e Cunha, “Improving knowledge management processes: a hybrid positive approach,” *Journal of Knowledge Management*, vol. 16, no. 2, pp. 215–242, 2012.

[25] H. Mannaert and J. Verelst, *Normalized systems: re-creating information technology based on laws for software evolvability*. Koppa, 2009.

[26] 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.

[27] ——, “Towards evolvable software architectures based on systems theoretic stability,” *Software: Practice and Experience*, vol. 42, pp. 89–116, 2012.

[28] Oracle. Java platform, enterprise edition. Last access date: 04.01.2013. [Online]. Available: http://www.oracle.com/technetwork/java/javaee/overview/index.html

[29] L. Welicki, J. Manuel, C. Lovelle, and L. J. Aguilar, “Patterns meta-specification and cataloging: towards knowledge management in software engineering,” in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006, pp. 679–680.

[30] J. Coplien, “The culture of patterns,” *Computer Science and Information Systems*, vol. 1, no. 2, pp. 1–26, 2004.

[31] D. C. Schmidt, “Using design patterns to develop reusable object-oriented communication software,” *Commun. ACM*, vol. 38, no. 10, pp. 65–74, 1995.

[32] D. Riehle, “Lessons learned from using design patterns in industry projects,” in *Transactions on pattern languages of programming II*, J. Noble and R. Johnson, Eds. Berlin, Heidelberg: Springer-Verlag, 2011, ch. Lessons learned from using design patterns in industry projects, pp. 1–15.

[33] G. Odenthal and K. Quibeldey-Cirkel, “Using patterns for design and documentation,” in *ECOOP*, 1997, pp. 511–529.

[34] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, and W. F. Tichy, “Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance,” *IEEE Trans. Softw. Eng.*, vol. 28, no. 6, pp. 595–606, 2002.

[35] C. Larman, *Applying UML and Patterns*. Prentice Hall, 1997.

[36] J. Greenfield and K. Short, “Software factories: assembling applications with patterns, models, frameworks and tools,” in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2003, pp. 16–27.

[37] D. Coleman, D. Ash, B. Lowther, and P. Oman, “Using metrics to evaluate software system maintainability,” *Computer*, vol. 27, no. 8, pp. 44–49, 1994.

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[38] R. L. Glass, “Maintenance: Less is not more,” *IEEE Software*, vol. 15, no. 4, pp. 67–68, 1998.

[39] L. Prechelt, B. Unger, W. Tichy, P. Brossler, and L. Votta, “A controlled experiment in maintenance: comparing design patterns to simpler solutions,” *Software Engineering, IEEE Transactions on*, vol. 27, no. 12, pp. 1134–1144, 2001.

[40] S. K. Ethiraj and D. Levinthal, “Bounded rationality and the search for organizational architecture: An evolutionary perspective on the design of organizations and their evolvability.” *Administrative Science Quarterly*, vol. 49, no. 3, pp. 404–437, 2004.

[41] C. Y. Baldwin and K. B. Clark, *Design Rules, Volume 1: The Power of Modularity*, ser. MIT Press Books. The MIT Press, January 2000.

[42] Y. K. Ro, J. K. Liker, and S. K. Fixson, “Modularity as a strategy for supply chain coordination: The case of u.s. auto,” *Engineering Management, IEEE Transactions on*, vol. 54, no. 1, pp. 172 –189, feb. 2007.

[43] D. Van Nuffel, “Towards designing modular and evolvable business processes,” Ph.D. dissertation, University of Antwerp, 2011.

[44] P. Huysmans, “On the feasibility of normalized enterprises: Applying normalized systems theory to the high-level design of enterprises,” Ph.D. dissertation, University of Antwerp, 2011.

2013, © Copyright by authors, Published under agreement with IARIA - www.iaria.org