Analyzing the Evolvability of Modular Structures: a Longitudinal Normalized Systems Case Study

Philip Huysmans, Peter De Bruyn, Gilles Oorts,
Jan Verelst, Dirk van der Linden and Herwig Mannaert

Normalized Systems Institute
University of Antwerp
Antwerp, Belgium
Email: {philip.huysmans,peter.debruyn,gilles.oorts,
jan.verelst,dirk.vanderlinden,herwig.mannaert}@uantwerp.be

***Abstract*—The evolvability of organizations as a whole is determined by the evolvability of different enterprise architecture layers. This paper presents a longitudinal case study, performed within an infrastructure monitoring company, on how Normalized Systems Theory enables this evolvability, at least, at the level of its information systems. By describing the different versions of the case organization's information system throughout time, we are able to analyze the characteristics of the system which facilitate this goal. In particular, the increasingly fine-grained structure of the system allows for multiple dimensions of variability. This analysis is then generalized and described in terms of modular systems. Based on this generalization, several implications for other enterprise layers are presented.**

Keywords–Normalized Systems; modularity; evolvability; case study

I. INTRODUCTION

In today's ever-changing and competitive markets, enterprises need to be able to respond ever so quickly to changing market demands. One could argue that this evolvability is one of the main requirements for an enterprise to be competitive in the current global economy. Certain scholars have argued that in order to create a sustainable competitive advantage, changes need to be applied *at a constant rate* [1]. This, however, means an organization is in a constant state of flux, and a fixed baseline on which new changes can be etched is ever absent. This considerably complicates the implementation of changes and the agility of organizations.

An additional challenge to organizational evolvability is that evolvability is required at multiple enterprise layers. For example, to attain a truly evolvable enterprise, its organizational structure, business processes and information systems need to be able to easily implement changes. As all these layers are intertwined, a single change in one of these layers will insurmountably result in multiple changes in one or more of the other layers. As a result, it is clear one should always study organizational evolvability as the accumulation of evolvability within all these inseparable layers. Likewise, enterprises should always strive for organizational evolvability within all layers.

Most enterprise architecture approaches propose a generic way of working towards evolvability. For example, Ross et al. [2] propose that, after a team and vision are established, an AS-IS architecture is developed. Next, a TO-BE architecture should be defined which enables the established vision. The transition between AS-IS and TO-BE architectures then needs to be planned and executed. Most frameworks do not provide a more concrete way of working [3].

At the lower organizational levels however, more detailed progress has been made in enabling evolvability. This is especially true for the lowest level, i.e., the information systems that support the other enterprise layers in the execution of their tasks. In this regard, Normalized Systems (NS) theory was introduced as an approach to build evolvable artifacts, such as information systems [4]. Although this approach is proven to be theoretically sound [5] and practically viable [6], [7], few cases have been published.

In this paper, we will therefore document a case to illustrate how Normalized information systems support organizational evolvability by allowing rapid and extensive changes to the software. The specific case is chosen because it concerns a software application that evolved extensively throughout time and allows to clearly illustrate why NS theory requires a fine-grained modular structure. NS theory has also proven to be relevant to design artifacts in other organizational layers as well [3], [8], [9]. Therefore, we will generalize the findings and reflect on the potential implications for modular structures in later sections of the paper.

The paper is structured as follows: first we introduce the Normalized Systems theory in Section II. Next, we describe the case study and the evolution of the discussed application in detail in Section III. The case reflections, generalizations and implications are discussed in the Discussion in Section IV, followed by a Conclusion in Section V.

II. NORMALIZED SYSTEMS

The case that is discussed in this paper is based on the body of thought of Normalized Systems (NS) theory. Therefore, we will briefly introduce this theory in this section. For a more comprehensive description, we refer to previous publications, such as [4], [5], [10], [11].

The NS theory is theoretically founded on the concept of stability from systems theory. According to systems theory, stability is an essential property of systems. For a system to be stable, a bounded input should result in a bounded output, even if an unlimited time period $T \rightarrow \infty$ is considered. For

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-438-1

information systems, this means that a bounded set of changes (selected from the so-called *anticipated changes* within NS theory) should result in a bounded impact to the system, even for $T \rightarrow \infty$ (i.e., an unlimited systems evolution is considered). In other words, stability reasoning expresses how the impact of changes to an information system should not depend on the size of the system, but only on size and property of the changes that need to be performed. If this is not the case, a so-called *combinatorial effect* occurs. It has been formally proven that any violation of any of the following theorems will result in combinatorial effects that negatively impact evolvability [5]:

- *Separation of Concerns*, which states that each concern (i.e., each change driver) needs to be encapsulated in an element, separated from other concerns;
- *Action Version Transparency*, which declares an action entity should be updateable without impacting the action entities it is called by;
- *Data Version Transparency*, which indicates a data entity should be updateable without impacting the action entities it is called by;
- *Separation of States*, which states all actions in a workflow should be separated by state (and called in a stateful way).

The application of the NS theorems in practice has shown to result in very fine-grained modular elements which may, at first, be regarded as complex. Although it quickly becomes clear to developers how every element is constructed very similarly, it is very unlikely to attain these strictly defined elements without the use of higher-level primitives or patterns. Therefore NS theory proposes a set of five elements (action, data, workflow, connector and trigger) that serve as patterns. Based on these elements, NS software is generated in a relatively straightforward way through the use of the NS expansion mechanism. For this purpose, dedicated software (called NS expanders) was built by the Normalized Systems eXpanders factory (NSX).

III. Case Study

The case we discuss in this paper is that of an organization which provides hardware and software for infrastructure monitoring (e.g., power supplies, air conditioning, fire detection systems and diesel generators). The infrastructure is monitored and managed from a central site called the Network Operating Center (NOC). In this center, status information from different facility equipment of geographically dispersed sites is gathered. The status information is sent by controllers which are embedded in the infrastructure. Different types of these controllers are developed and marketed by the organization. For example, the Telecom Site Controller (TSC) is developed specifically for telecom infrastructure, and the Monitoring Control Unit (MCU) is developed specifically for DC power supplies which contain AC/DC converters and batteries to handle power failures. The organization argued that the software to perform and manage the infrastructure monitoring could not be purchased as a commercial off-the-shelf package, because of the extensive customizations needed for the proprietary controllers and protocols. Consequently, a custom application was developed. We will describe the evolution of this application as a longitudinal case study by means of four phases the system has gone through.

A. Phase 1: SMS v1

Functionality and technology: The original version of the Site Management System (which we refer to as SMS v1) was deployed at the organization itself for a client from the railroad sector, as well as on-site for different clients from, a.o., the fiber glass sector. Initially, SMS v1 only supported the proprietary TSC controllers developed by the organization itself. Later on, MCU controllers were added, but only for certain clients. After initial deployment, more sites were added, and the application provided monitoring of around 280 sites.

SMS v1 was developed using Visual Basic and MS Access technology. The lack of a client-server architecture in the technology stack forced the organization to adopt suboptimal solutions for using the system in a distributed way: the MS Access database was remotely accessed through a network drive, and remote usage of the Visual Basic application was done by using a Virtual Network Computing (VNC) connection.

Structure: An explicit and deliberate structure of the application did not exist. Rather, the application was regarded as one single monolithic module, which relied on a database module. As a result, no code reuse was present: application code was duplicated in an unstructured way for every deployment instance.

Evolvability: The coarse-grained structure of the application resulted in certain quality deficiencies. The evolvability of SMS v1 was hard to determine: changes made to a certain code base were not always introduced in other code bases, which resulted in distinct (inconsistent) code bases. As a result, any change could have a different impact on a specific SMS v1 code base.

The arduousness of changing the code base was aggravated by the amount of configuration parameters which were hardcoded. For example, it was assumed that all TSCs in a network were configured in the same way (e.g., the alarms from the air conditioning system are registered on input 1). This resulted in a lack of flexibility during deployment. Moreover, the growth, increased usage, and multi-user access of the application resulted in performance issues. The used technology was not designed for scalability, and was focused on single-user access.

B. Phase 2: SMS v2

Functionality and technology: As the initial version of SMS provided adequate functionality, the need for a new version was largely motivated by non-functional requirements. As reported above, the scalability and flexibility of the original application was unsatisfactory. In 2005, an external software development company was approached. Based on the existing functionality, a new application was developed from scratch and deployed in 2006. We will refer to this application as Site Management System version 2 (SMS v2).

Because the lack of a client-server architecture was experienced as an obstacle for a scalable, flexible and multi-user application, a radical change of the application architecture was adopted. Instead of using the proprietary Microsoft technologies of SMS v1, a standard and web-based architecture was adopted. More specifically, the Java 2 Enterprise Edition (J2EE) platform was used, with Enterprise JavaBeans version 2.1 (EJB2) and Cocoon framework as characterizing components.

Structure: SMS v2 was developed following industry best practices, which imposed a certain structure: different concerns need to be implemented in different constructs (e.g., java classes). For example, EJB2 prescribes that for each bean, local and home interfaces need to be defined, and that RMI-access to the bean must be provided by an agent class. Similarly, a certain structure was imposed by the Cocoon framework, which was employed for the web tier of the application. The actual business logic (e.g., checking the status of a TSC controller) was also implemented by separate classes. This prevents the inclusion of business logic in framework-specific classes: the controller class contains the actual description of the controller, such as the IP address and port, protocol, or phone number for dial-up access. Consequently, the parameters for each controller were clearly separated from other concerns, and could be configured separately, providing the required flexibility.

This way of working leveraged existing knowledge in the software engineering field, which is distributed in several ways. First, design patterns describe generally accepted solutions on how code should be structured in certain situations. For example, the Strategy pattern from the Gang of Four pattern catalog describes a structure to implement a “family of algorithms”, and make them interchangeable [12, p. 315]. This structure was applied in the SMS v2 application, and enabled the loading of the correct implementation class of a specific controller (e.g., TSC or MCU). Second, the usage of frameworks enforces certain industry best practices. For example, the Model-View-Controller design pattern [13] can be implemented in any object-oriented language by the programmer. Frameworks such as Cocoon enforce programmers to adhere to this pattern, thus eliminating a certain design freedom. Similarly, the usage of EJB2 also encourages a programmer to separate certain concerns. For example, by using object-relational mapping, a separation between logic and persistence is enforced.

These examples illustrate how applying existing software engineering knowledge enabled non-functional requirements such as flexibility and scalability by prescribing a finer-grained structure of software constructs. However, the application exhibited an even finer-grained structure than prescribed by the design patterns and frameworks. For example, a specific class was created to trigger certain tasks at certain intervals (e.g., checking if an alarm was generated). Separating this functionality is not prescribed by design patterns or frameworks: by considering it as business logic, it could be included in the implementation classes. Nevertheless, separating this rather generic functionality in its own constructs allowed the reuse its code in different contexts. As a result, the structure of software primitives for various entities started to exhibit a similar structure.

Evolvability: While developing SMS v2, NS theory was not yet formulated and the expanders were not yet developed. As a result, the recurring code structure (which contained many constructs) needed to be recreated manually. While still requiring some effort, this was relatively easy as each particular controller was rather similar.

Using a recurring structure resulted in other advantages as well. Due to experience with similar code structures used for other controllers (or even, similar code structures in other applications), the performance of the application under different loads (e.g., number of status messages sent) could be accurately estimated. As a result, scaling the application across different sites could be managed.

C. Phase 3: PEMM v1

Functionality and technology: Around 2007, SMS needed to support new functional requirements. First, the range of supported controllers was to be extended. For example, support was added for OLE for Process Control (OPC) servers. An OPC server groups communication from multiple controllers, which allows easier hardware setup. Second, specific functionality for certain controller types was to be supported. For example, a TSC controller provides configuration management for physical site access control. By sending configuration messages, access codes for specific sites with keypad access can be set. Third, various output options were to be provided. In case of certain alerts, an SMS could be sent to the operator, in addition to the regular monitor-based output. Because of the size of the new functional requirements, the application was renamed in Power Environmental Monitoring and Management (PEMM). We will refer to it as PEMM v1.

The technology stack of PEMM v1 was similar to the technology stack of SMS v2: a J2EE architecture with EJB2 and Cocoon framework. The versions of the different components were updated to more up-to-date versions.

Structure: The consistent and systematic separation of different concerns in several projects had resulted in a recurring software structure. For example, gathering and persisting data for certain entities required several constructs for creating a Create-Read-Update-Delete-Search (CRUDS) interface (i.e., jsp and html pages), java classes for the application server, and relation database table specifications. As a result, the required constructs in use could be reused for every new instance. In order to facilitate this reuse, a set of *pattern expanders* was created, which create the software constructs based on a configuration file. For example, the constructs for the data entities are created by the data element pattern expander. Parameters for the data entity are specified in a XML configuration file, also called a *descriptor file*. For a data element, for following parameters need to be defined:

- Basic name of the data element instance.
- Context information (i.e., package and component name)
- Data field information (i.e., names and data types for the various attributes of the entity)
- Relationships with other elements

For the PEMM v1 application, such descriptor files were created for, e.g., controllers, alarms, sites, etc.

After such pattern expansion, the application can be compiled and deployed, similar to a regular application. Pattern expansion allows developers to quickly create a software structure which separates many concerns. Developing such structure from scratch would imply a disproportionate effort when compared to the effort required for programming the actual business logic. As a result, separating many concerns is often omitted, which results in code of poorer quality. Such pattern expansion is only feasible when every data element has an identical structure.

Evolvability: Because of identical structures within the code base, applying changes to the code becomes predictable, or even deterministic. We discuss four main groups of changes.

First, functional changes could be added to existing elements by *marginal expansion*. For example, adding a data attribute for a controller could be specified in an additional data descriptor. A marginal expansion recognizes the element for which the additional descriptor is specified, and adds the necessary code in the existing code base. As a result, certain functional changes can be made without overwriting customizations, and are coined *anticipated changes* [10, p. 95]:

- an additional data field;
- an additional data entity;
- an additional action entity;
- an additional version of a task.

In PEMM v1, an example of a marginal expansion was the addition of a comment field to an alarm data element. Adding the comment field through marginal expansion not only adds a field in the database table, but also adds that field in the java bean, in all CRUDS screens, etc.

Second, new functionality can be added to the application by *generating new elements*. These can be integrated in the existing code base by providing relations to the existing elements in the descriptor files. In PEMM v1, the following functionality was added by expanding new elements into the existing application:

- FAQs: a FAQ element allows customers to input knowledge concerning specific alarms. These FAQs are made available to operators who need to monitor and manage the alarms.
- Asset management: in order to keep track of various assets, an asset element was added to allow a technician to add the serial number of used or newly added assets to a certain site.
- Service log: a service log element records the history of all service interventions made on a particular site.

Third, functionality which cannot be implemented by expanding new elements or by applying anticipated changes, needs to be realized through *customizations*. Non-standard functionality, such as user interface screens, reporting or authentications needs to be programmed separately. Implementation classes for actions (e.g., checking a controller) are typical examples of such customizations, which are added to the code base as separate files. Another example for PEMM v1 is the reporting functionality, which is an implementation class for an action element which generates a file to import alarm data in reporting tools. Separate files which are necessary in the code base (such as the implementation classes) can be easily located by programmers, since they always occur at the same location within the element structure. However, customizations can also be made by overwriting code in the generated files. Such customizations are harder to track, as white-box inspection or separate documentation is required to know where they are located. In PEMM v1, the following customizations in the expanded code needed to be made:

- Authorization: in an NS application, a base component is added to configure user authorizations. In the PEMM application, custom business rules were added based on these configurations. For example, certain users could acknowledge collections of alarms, instead of acknowledging each alarm separately. In multi-tenant deployments of PEMM, this ability needed to be restricted to alarms from certain sites.
- User Interface: examples of user interface screens which were added to the PEMM application as customization are: (1) trending charts, which show certain measurements over time; (2) Alarm overview screens: color-coded tables which provide a high-level view of active alarms; (3) Map view: a map which shows sites with active alarms. This map view is linked to the alarm overview screens.

D. Phase 4: PEMM v2

Functionality and technology: Around 2012-2013, the organization decided to switch from the proprietary TSC and MCU controllers and protocols towards industry-standard controllers (Beckoff) and protocols (SNMP, Modbus over IP and OPC DCOM DA). The introduction of these requirements triggered an update of the PEMM application using newer versions of the NS expanders (now called PEMM v2), which incorporated a new mechanism to facilitate the extraction (*harvesting*) and addition (*injection*) of customizations. While the functional changes could have been implemented in PEMM v1 without combinatorial effects, they required customizations which would need to be redone in the case of a regeneration at a later point in time. Therefore, the decision was made to migrate to the new version of expanders. As the new version of expanders can use the same descriptor files and expand a similar source code structure, porting the application to a new expander version required much less effort in comparison with a rewrite. Using the new expander versions also implied the incorporation of new frameworks (e.g., Knockout) and new versions of the existing frameworks (e.g., Struts 2), compilers (JDK) and servers (e.g., Jonas application server) with their accompanying fixes and functional enhancements. Moreover, the programming team changed: a developer of the infrastructure monitoring organization, familiar with these new frameworks, was appointed to work on the new application.

Structure:

Again, a large portion of the code base could be generated using the expanders resulting in a very similar general structure as before. This time however, *anchors* were added to the structure, which allow the harvesting/injection mechanism to work. This mechanism solves the issue of overwriting customizations in case of regeneration. Customizations to the code base (e.g., GUI elements) can now be made in three ways:

- insertions: customizations are put between predefined anchors in the expanded code base;
- extensions: customizations are contained within separate files, added in the file structure in predefined directories;
- overlays (discouraged): customizations overwriting expanded files, but not being captured by the harvesting/injection mechanism.

In case of a regeneration, the harvesting mechanism checks the anchors for insertions and predefined directories for extensions, which are both stored ("harvested"). This allows the harvested code changes to be injected in the newly expanded code base, and extended files to be added in the appropriate

directories. The harvesting mechanism therefore leads to a clean separation between the expanded code base and its customizations. Given the recurrent structure of the expanded code base, the main complexity of the application becomes determined by the customizations, rather than the expanded code base itself. In PEMM v2, customizations only represent a small fraction of the overall code base: 5 percent.

Evolvability: The harvesting/injection mechanism enabled new dimensions of evolvability: as customizations could be applied to a newly generated code base, both could start to evolve independently. The obsoleteness of marginal expansions illustrates the usefulness of the mechanism. Additional data attributes could now be simply added to the descriptors, upon which a completely new code base (including injected customizations) could be generated. Similarly, a new code base could be generated based on new technology versions. For example, when a new version of the presentation framework provides new features which are included in the expanders, a new code base could be generated and injected with the customizations. The application is then enhanced with the new features, without requiring additional effort. As a result, the technologies used in the application could easily be kept up to date.

IV. DISCUSSION

In this section, we respectively discuss some case reflections (Section IV-A), introduce a generalization of the four case phases (Section IV-B) and discuss the implications of the case findings for other enterprise layers (Section IV-C).

A. Case Discussion

Certain noteworthy reflections in relation to the current state-of-the-art in software engineering can be made based on the case documented in the previous section.

First, a tendency has been observed to deprecate or "throw away" large portions of code when functional requirements or team members change. This "*not invented here syndrome*" [14] typically results in a lot of rework and little reuse. An illustration of this phenomenon is the redevelopment of the application after phase 1. The case further illustrates how, in subsequent phases, this inclination can be mitigated by applying a fine-grained, reusable code structure. Many functional changes have been implemented, and different programmers have been working on the application, without the need to deprecate the existing code base. This reuse enabled a focus on applying functional changes, rather than reworking architectural aspects of the code. As a result, the application has been updated regularly during 7 years, without requiring large code deprecations.

Second, the contrast between the widely used design patterns and software elements as proposed by NS Theory should be noted. While design patterns might incorporate substantial design knowledge regarding evolvability, their concrete implementation is still left to the programmer. Applying multiple design patterns simultaneously results in a complex structure, which, if created by hand, is error-prone and difficult to maintain. This has been acknowledged by various scholars reviewing the state-of-the-art of design patterns: "*general design principles can guide us, but reality tends to force trade-offs between seemingly conflicting goals, such as flexibility and maintainability against size and complexity*" [15, p. 88].

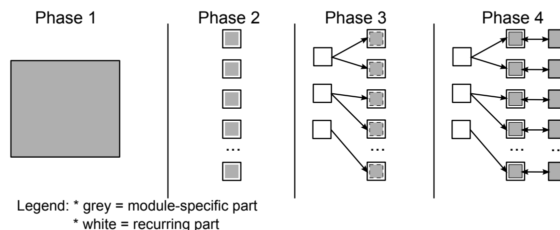

Figure 1. 4 phases, each one with their distinct variability dimensions.

Phase 2 of the case, where the code structure needed to be recreated manually, illustrates this. However, in subsequent phases, code reuse was enabled by the expansion mechanism. As a result, manual coding was no longer required to apply existing design knowledge, leading to a consistent and correct application of the accumulated knowledge. The mechanism of applying design knowledge through the use of expanders has been discussed in [16].

Third, the lack of an expansion mechanism jeopardizes true black-box reuse of software modules. As argued in [4, p.178]: "*in many cases, the problem is not that the component cannot be found in a repository, or cannot be reused at this point in time, the main problem is that this would create dependencies of which the future implications are highly uncertain.*" This implies that white-box inspection remains necessary in order to safely reuse modules in an evolving system, since not all dependencies of a module can be considered to be visible in its interface (i.e., hidden dependencies exist). The expansion mechanism allows a one-time inspection of a modular structure and, because of the systematic duplication of that structure, a deterministic construction (guaranteeing the absence of additional dependencies) of (re-)expanded code. Moreover, re-expansions, such as illustrated in PEMM v2, even allow the removal of newly discovered hidden dependencies in the elements within older NS-based applications.

B. Generalizing the four phases to modular structures

The four versions of the SMS/PEMM system as described in Section III can actually be analyzed in terms of general modularity structures as well. That is, in each of the four phases, a more NS-like approach was adopted in which the software code was modularized in a more fine-grained way. Reflecting on the essence of each of these phases may help us in applying NS reasoning to modular systems in general and to other application domains, such as modular organizational artifacts. Figure 1 provides a visual overview of these four phases in general modularity reasoning. We will now briefly discuss each of them separately, while paying specific attention to the *variability dimensions* and *recombination potential* in each of the phases. The former specifies the dimensions along which the evolution of the system takes place. The latter quantifies the number of different ways in which the system can be arranged, based on its modular structure.

1) Phase 1: one monolithic block.: In the first phase, the application could more or less be considered as one monolithic block. While any application itself obviously consists out of a set of modular primitives as provided by the programming language used (e.g., functions, structs, classes), no special attention was given to a purposeful delineation of parts within

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-438-1

the overall application. This is similar to a general system in which no deliberate modularization is introduced. Flexibility in such design is clearly limited as the *recombination potential* basically only equals 1. This means that no parts of the system can be re-used (within one system or between several systems) or separately adapted and combined with other parts. As a consequence, adaptations are mostly not contained into one well-defined part of the overall system. Therefore, the *variability dimension* is the system as a whole and Figure 1 therefore only represents one grey box for this phase.

2) Phase 2: identifying a first set of modules.: In the second phase, the application was more purposefully structured, i.e., different parts were deliberately separated (e.g., classes for local interfaces, home interfaces and agents). As the modularization was already quite fine-grained and similar functionality was required several times, a certain repetitiveness in the code base became clear. Therefore, in order to clarify this way of working in general modularity reasoning, the second column of Figure 1 first of all represents a system as consisting out of several subsystems or modules. Additionally, this phase already exhibits a special kind of modularity as each of these modules consists out of a manually constructed (mostly) recurring part (i.e., the white part) and a part specific for that module (i.e., the grey part). This means that the *variability dimension* is redirected towards the set of individual modules: the goal of modularity is that each of the modules can be adapted (e.g., upgraded to a newer version) and be plugged in into the system as a whole (i.e., recombined with one another). Based on such modular design, the *recombination potential* becomes k^N when having N modules with each k versions. Increasing N or k therefore significantly increases the recombination potential of such system.

However, it needs to be mentioned that subdividing a system and creating versions of each of these subsystems in order to increase the recombination potential is only successful if done wisely. That is, problems arise due to coupling at the *intramodular level* if content is duplicated among modules. Coupling at the *intermodular level* may arise if dependency rippling occurs. Regarding the former, applying a change to a duplicated part requires each of the modules (in which the duplicated part is embedded) to be adapted. Regarding the latter, ripple effects may cause that a change in one modules requires all other modules (using this first module) to change as well in order to be still able the use first module. The NS theorems formulated in general modularity reasoning can therefore be argued to focus on designing systems which eliminate both such intramodular coupling (e.g., via Separation of Concerns) and intermodular coupling (e.g., via Version Transparency) [9].

3) Phase 3: reusing modular structures in a systematic way.: As from the third phase, the software developers stopped creating the recurring part manually and started generating these recurring parts. Therefore, the left part of the third column of Figure 1 shows general and explicitly predefined parts which can be used as the "background" for modules. More specifically, predefined structures for three types of modules are provided in this example (e.g., in a software context: an empty data, action, and flow element). In the right part of the third column, the "background" of the module is combined with the module-specific part in order to arrive at a fully functional module. When analyzing the *recombination potential* in this case, one has to consider both the versions of the predefined modules (e.g., an ameliorated predefined structure to encapsulate processing functions), as well as the versions of the module-specific parts (e.g., an updated processing function with a new encryption algorithm). Therefore, in case we consider only one predefined module type j, the recombination potential becomes $l \times k^N$ when having l versions of the predefined module type j, N instantiated modules of type j, and k versions of each module-specific part. It should be noticed that, for each new version of a predefined modular structure applied to an already existing module, the module-specific part should again be incorporated into this modular "background" manually. Therefore, this recombination potential cannot fully be realized at this stage of modularization yet.

4) Phase 4: expanding elements and harvesting customizations.: In the last phase, depicted in the fourth column of Figure 1, the module-specific parts can be isolated and separately stored (i.e., harvested, as represented in the right side of the column) before a regeneration of the recurring predefined modular structures is performed. Therefore, the module-specific parts do not have to be incorporated manually in this modular "background" any longer. Instead, these parts are automatically injected into the general parts at predefined locations. This enables to achieve the mentioned *recombination potential* in the previous phase in reality. Stated otherwise, two different *variability dimensions* have to be considered. First, we have the different versions of the module-specific parts. Second, there are different versions of the module-generic parts. This means that classical version numbering in such cases becomes rather useless: it does not make a lot of sense anymore to consider a fixed "version" of the modular system as it is the result of the combination of two different variability dimensions (i.e., all general parts and module-specific parts can have different versions).

C. Implications for other enterprise layers

While the core of this paper discussed how NS Theory specifically modularizes the software layer within an enterprise architecture, the previous subsections reflected on the implications for the software engineering field and modular systems in general. Based on these reflections, some preliminary implications for other enterprise layers can be discussed.

First, the case illustrates how software evolvability was enabled by allowing changes to small modules, rather than updating one large, monolithic design (cfr. removing the need for code deprecation), as discussed in Section IV-A. In current enterprise architecture approaches, AS-IS and TO-BE versions are typically designed. Here, the focus is mainly directed to two separate, monolithic designs as opposed to gradual changes to small, individual modules. For truly evolvable enterprise architecture layers, the evolution of smaller modules should be addressed.

Second, the usage of repetitive structure instantiations through expanders was contrasted with the documentation of more generic design patterns (cfr. Section IV-A). On different organizational layers, recurring structures or patterns have been proposed as well [17]. Initial explorations of organizational patterns ("elements"), conform with NS Theory, have already been presented [3], [9]. For instance, De Bruyn [9] conceptually suggests a set of possible cross-cutting concerns and ele-

ments at this level. Currently, these approaches have however not yet been implemented in practice and should be further elaborated in future research. Ultimately, this would lead to the interesting phenomenon of clearly described variability dimensions within organizational layers, which provides decision makers with clear options for evolving the organization [18].

Third, we discussed how NS Theory demonstrates the difficulty of designing truly black-box modules (cfr. Section IV-A). It has been argued by various scholars that several other layers within organizations can be considered as modular structures as well [19]. Based on these arguments, several attempts have been made in the past to apply NS reasoning to such layers, such as business processes and enterprise architectures [8], [3], [9]. These efforts mainly concentrated on identifying and proving the existence of combinatorial effects in a diverse set of organizational layers and functional domains [8], [3], [20], [21], demonstrating a similar issue: it is hard to create truly black-box, fine-grained modules on these levels as well.

Fourth, the concept of recombination potential demonstrates the relevance of addressing the difficult research challenges outlined in the previous implications. Achieving a larger recombination potential in organizational artifacts such as products, processes and departments would (1) enable mass customization of products, which currently still results in high costs and a large complexity [22]; (2) provide a systematic approach to versioning artifacts, which is a large issue when implementing innovation at a steady pace [1]; and (3) aid in executing complex mergers and acquisitions, by considering organizational departments as modular options [18].

V. Conclusion

In this paper, we discussed a longitudinal case study of an NS application. We focused on how an increasingly fine-grained software structure enabled different types of evolvability. Such a description contributes to the NS knowledge base, since it illustrates the theoretical implications of NS Theory. Following this description, we generalized our findings towards generic modular structures. Since different enterprise architecture layers have been considered as modular structures before, we applied the resulting insights to other layers. This effort contributes to ongoing research in the Enterprise Engineering field, by integrating the current paper with previous research, and exploring future research challenges.

References

[1] A. Van de Ven and H. Angle, An Introduction to the Minnesota Innovation Research Program. New York, NY: Oxford University Press, 2000.

[2] J. W. Ross, P. Weill, and D. C. Robertson., Enterprise Architecture as Strategy – Creating a Foundation for Business Execution. Harvard Business School Press, Boston, MA, 2006.

[3] 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.

[4] H. Mannaert and J. Verelst, Normalized Systems—Re-creating Information Technology Based on Laws for Software Evolvability. Kermt, Belgium: Koppa, 2009.

[5] H. Mannaert, J. Verelst, and K. Ven, "Towards evolvable software architectures based on systems theoretic stability," Software: Practice and Experience, vol. 42, no. 1, January 2012, pp. 89–116. [Online]. Available: http://dx.doi.org/10.1002/spe.1051

[6] G. Oorts, P. Huysmans, P. De Bruyn, H. Mannaert, J. Verelst, and A. Oost, "Building evolvable software using normalized systems theory: A case study," in System Sciences (HICSS), 2014 47th Hawaii International Conference on, Jan 2014, pp. 4760–4769.

[7] G. Oorts, K. Ahmadpour, H. Mannaert, J. Verelst, and A. Oost, "Easily evolving software using normalized system theory - a case study," in Proceedings of ICSEA 2014 : The Ninth International Conference on Software Engineering Advances. Nice, France: ICSEA, 2014, pp. 322–327.

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

[9] P. De Bruyn, "Generalizing normalized systems theory: Towards a foundational theory for enterprise engineering," Ph.D. dissertation, University of Antwerp, 2014.

[10] 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, 2011, pp. 1210–1222.

[11] P. Huysmans, G. Oorts, and P. De Bruyn, "Positioning the normalized systems theory in a design theory framework," in Proceedings of the Second International Symposium on Business Modeling and Sofware Design (BMSD), Geneva, Switzerland, July 4-6 2012, pp. 33–42.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design patterns: elements of reusable object-oriented software. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[13] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view controller user interface paradigm in smalltalk-80," J. Object Oriented Program., vol. 1, no. 3, Aug. 1988, pp. 26–49. [Online]. Available: http://dl.acm.org/citation.cfm?id=50757.50759

[14] R. Katz and T. J. Allen, "Investigating the not invented here (nih) syndrome: A look at the performance, tenure, and communication patterns of 50 r & d project groups," R&D Management, vol. 12, no. 1, 1982, pp. 7–20. [Online]. Available: http://dx.doi.org/10.1111/j.1467-9310.1982.tb00478.x

[15] G. Hohpe, R. Wirfs-Brock, J. W. Yoder, and O. Zimmermann, "Twenty years of patterns' impact," Software, IEEE, vol. 30, no. 6, Nov 2013, pp. 88–88.

[16] P. De Bruyn, P. Huysmans, G. Oorts, D. Van Nuffel, H. Mannaert, J. Verelst, and A. Oost, "Incorporating design knowledge into the software development process using normalized systems theory," International Journal On Advances in Software, vol. 6, no. 1 and 2, 2013, pp. 181–195.

[17] J. Dietz, Enterprise Ontology: Theory and Methodology. Springer, 2006.

[18] C. Y. Baldwin and K. Clark, "The option value of modularity in design," Harvard NOM Research Paper, vol. 3, no. 11, 2002.

[19] 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, 2010, pp. 259–283.

[20] J. Verelst, A. Silva, H. Mannaert, D. A. Ferreira, and P. Huysmans, "Identifying combinatorial effects in requirements engineering," in Advances in Enterprise Engineering VII, ser. Lecture Notes in Business Information Processing, H. Proper, D. Aveiro, and K. Gaaloul, Eds. Springer Berlin Heidelberg, 2013, vol. 146, pp. 88–102.

[21] E. Vanhoof, P. Huysmans, W. Aerts, and J. Verelst, "Evaluating accounting information systems that support multiple gaap reporting using normalized systems theory," in Advances in Enterprise Engineering VIII - Fourth Enterprise Engineering Working Conference (EEWC 2014), ser. Lecture Notes in Business Information Processing, D. Aveiro, J. Tribolet, and D. Gouveia, Eds., vol. 174. Springer, 2014, pp. 76–90.

[22] J. H. Gilmore and B. J. Pine II, "The four faces of mass customization." Harvard Business Review, vol. 75, no. 1, 1997, pp. 91 – 101.