Easily Evolving Software Using Normalized System Theory A Case Study

Gilles Oorts, Kamiel Ahmadpour, Herwig Mannaert and Jan Verelst
Normalized Systems Institute (NSI)
University of Antwerp
Antwerp, Belgium
{gilles.oorts,kamiel.ahmadpour,herwig.mannaert,jan.verelst}@uantwerp.be

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

***Abstract*—Software agility is characterized by inevitable software changes and ever-increasing software complexity. Unless change accommodations are rigorously taken into account, the implementation of these changes may lead to exorbitant costs. This is in particular true for long-lived systems. For such systems, there is a need to explicitly address evolvability concerns during their design phase. This to carry out software evolution efficiently and reliably during their lifecycle, and prolong the productive life of the software systems. Normalized Systems (NS) theory has been recently proposed as an approach to develop agile and evolvable software. In this paper we discuss the practical advantages of the NS approach using a case study regarding the revision of a budget management application. Furthermore, advantages such as knowledge transfer through the NS development process are also discussed in this paper.**

Keywords–Normalized Systems theory; Evolvable Software; Adaptive Software; Agile Software; Case Study

I. Introduction

In ever-increasing volatile environments, evolvability is considered as one of the most important characteristics of information systems. As information systems support the operations and decision-making of organizations, software applications also need to support the changes on the business-side of organizations. However, organizations normally find it difficult to synchronize changing requirements needs with their software applications. This is because the current software development paradigms do not fully take into account the changeability of business needs over the life cycle of software systems. This problem is also characterized by the Law of Increasing Complexity as proposed by Lehman, which states that the structure of software tends to become more and more complex over time because of changes made to the software [1]. Furthermore, software applications are traditionally built to last several years -or even decades- in order to justify (high) development costs. With regard to changing business requirements, this often leads to decisions to either not implement the changes because they are too expensive, or to eventually (after several years or decades) totally scrap the application and start the development of a new “up-to-date” application.

Recently, Normalized Systems (NS) theory has been proposed as a way to deal with ever-changing requirements for software by building evolvable information systems, based on the systems theoretic concept of stability [2]. As recent research shows, these systems are capable of incorporating changes more easily and with less effort by means of a careful design of the software architecture [3], [4]. Therefore, changes can be made immediately and the life cycle of software applications is greatly extended, up to a point that they can be used and revised infinitely. In this paper we will discuss how changing business requirements can be easily implemented into an application developed according to the NS theory. This will be illustrated by means of a case regarding the revision of a budget management application. The initial development of this budget application is described in [4], which focused on illustrating the NS development methodology used in developing the application. In this paper we provide a clear understanding of the NS advantages in dealing with changes to this initial application by comparing both versions of the software, the scope of changes and the amount of time and effort spend on implementing all the updates in the new version. As the case description requires an understanding of the NS theory, a brief review of the NS theory is provided in Section II. For a more thorough description of the NS theory, we refer to a wide number of previous works (for example, [2], [3], [5]). In Section III, we will first provide a short description of the initial software application, followed by the changed business requirements. How these changes were implemented according to the NS theory is discussed in Section IV. In Section V, we will discuss the advantages of NS development, some observations, and contributions of the paper. We end the paper with a brief conclusion regarding this paper in Section VI.

II. Normalized Systems theory

NS theory is concerned with how information systems can be deterministically designed and developed based on the systems theoretic concept of stability. According to NS, the main obstacle to evolvability is the existence of so-called combinatorial effects. In this condition, the amount of effort to make a specific change in the system is not only related to the change but also to the size of the system. Therefore the effort to apply a specific change increases as the system grows [2]. According to the systems theory, stability refers to a system in which a bounded input function results in bounded output values, even as $t \rightarrow \infty$ (with t representing time). When applied to information systems, this means that applying a specific change to the information system should always require the same effort [3]. According to NS theory, the avoidance of all combinatorial effects in software leads to evolvable software, as this means ripple effects of changes do not increase over time and, as such, with the size of the system. To eliminate combinatorial effects, NS theory proposes a set of four theorems and five elements that can be expanded into fully functional applications through pattern expansion. This set of theorems and elements are the foundation of NS theory, and will be discussed in the next sections.

A. NS Theorems

- *Separation of Concerns (SoC)*, requiring that each change driver (concern) has to be separated from other concerns. This theorem allows us to isolate the impact of each change in its own entity;
- *Data Version Transparency (DVT)*, requiring that data entities can be modified (e.g., additional data can be sent between components), without having an impact on other entities;
- *Action Version Transparency (AVT)*, refers to a condition in which an action entity can be upgraded without impacting the calling components;
- *Separation of States (SoS)*, implies that actions or steps in a workflow are separated from each other in time by keeping a state after every action or step.

These are just brief definitions of the NS theorems, as these have previously been extensively discussed in other work (e.g., [2], [3], [5]). It has to be mentioned that none of these theorems are completely new, and even relate to heuristic knowledge of developers [5], [6]. However, formulating this knowledge as theorems aimed at identifying combinatorial effects will help to build information systems that contain a minimal number of combinatorial effects. Only when the design of an application completely adheres to the NS theorems, one can profit from the software evolvabilty that the NS theory offers.

B. Normalized Systems Elements

As the systematic application of the NS theorems results in a very fine-grained modular structure, NS theory proposes to build information systems based on the aggregation of instantiations of five higher-level software patterns or elements, being:

- *a data element*, representing an encapsulated data construct with its get- and set-methodss to provide access to its information in a data version transparent way. Cross-cutting concerns (for instance access control and persistency) are considered to be a part of the data element;
- *an action element*, containing a core action representing a single change driver or functional task;
- *a workflow element*, containing the sequence in which a number of action elements should be executed in order to fulfill a flow;
- *a trigger element*, controlling the states (both regular and error states) and checking whether an action element has to be triggered accordingly;
- *a connector element*, ensuring external systems are able to interact with the NS system without allowing elements to be called in a stateless way.

The above mentioned NS elements are the essential building blocks for a NS application and provide the core functionality of an information system. They can then be easily extended later (cf., description of extensions in Section IV). A functional analyst will formulate instantiations of the NS elements that are the foundations of a NS application [4]. At run time, these instances are instantiated once more (i.e., constitute a double instantiation) to form specific occurrences of, for example, a budget [4].

The NS elements have been described more extensively in [2], [3], [5] and the implementation of a data element in a Java Enterprise Edition (JEE) has been described in a previous work [5]. The definition and identification of the NS elements is based on the implications of the set of NS theorems [7]. For example, the definition of the workflow element is based on the Separation of Concerns (SoC) and Separation of States (SoS) theorems. In a workflow element, we can invoke action elements in a completely stateful manner and as mentioned earlier, keeping track of every action's state, leads to Separation of States (SoS). Similarly, each of the five NS elements constitutes one possible solution for implementing all four NS theorems, thus eliminating all combinatorial effects.

Each of these five elements provides a general reusable solution to a commonly occurring problem within a given context. Therefore, they can be considered as a design pattern, containing a core construct and several cross-cutting concerns (such as remote access, logging, access control, etc.). This architecture provides protection from combinatorial effects while allowing for a set of anticipated changes to be applied to a system [5]. As such, the five NS elements can be used to build an evolvable information system that satisfy the four NS theorems.

C. NS Pattern Expansion

The use of NS elements as design patterns is supported by the NS pattern expansion mechanism, which enables the conversion of NS element instances defined by the developer into fully functional code. Without using such expanders, it would be near to impossible to achieve the fined-grained modular structure prescribed by the NS theorems. Therefore the NS expanders are considered to be an essential part of designing an application using NS theory.

As such, pattern expansion is one of the four phases in the NS development process [4]. First a comprehensive functional analysis is performed to identify the NS element instances. Coding these instantiations is the next step of this process and it is done using some special "descriptor files". A descriptor file is a text or XML-based file which constitutes the input for the NS expanders.

For example, in case of a data element instance, one needs to provide the following parameters in the descriptor file in order to be able to work with the expanders:

- *Basic name of the data element instance:* Each data element instance needs to have a unique name which needs to be provided in the descriptor file (e.g., Budget).
- *Context information:* It provides the package and component name of the data element instance.
- *Data field information:* Each data element instance can contain one or more data fields and all the information about these data fields (such as their name and data type) needs to be provided in the descriptor file;
- *Relationships with other element instances:* It is necessary to address all the relationships of current data element instances with other element instances.

In the next step, through the process of expansion, the descriptor files get expanded into the code of a functional application. This show how a minimum of input information in the descriptor files can be used to transform into a fully functional application. The NS pattern expansion is done by software (called NS expanders) developed especially for this purpose by the NS eXpanders factory (NSX).

The NS expanders expand the descriptor files into skeleton source code for all the identified NS element instantiations. Furthermore, the NS expanders also provide all deployment and configuration files required to construct a working application on a supported technology stack. The skeleton source code facilitates a top-down design approach, where a functional system with complete high-level structures is designed and coded, and this system is then progressively expanded to fulfill the requirements of the project. These expansions are called NS extensions and will be discussed later in this paper. The classes of the skeleton code represent the modular structure of the defined NS element. For the Budget element instance, the NS expanders will for example generate a set of classes and data fields such as the bean class BudgetBean and its related local and remote interfaces (BudgetLocal and BudgetRemote).

Because of the NS expansion mechanism, applying changes to the application only requires us to provide new descriptor files and a re-expansion of these updated files will provide a new version of the application. This process will be shown in the section discussing the implementation of the changes to the application presented in the next paragraphs.

III. The NS Budget Management Application Case

Over the last years, several applications have been built based on NS theory and its development methodology. The most extensive description of this methodology can be found in [4], in which it is discussed by using the development of a budget management application for a local Belgian government as an exemplar. After using this application for only one year, both regulatory and requirements changes required the application to be changed. As the initial application was built according to the NS principles discussed in the previous section, these changes could be implemented rather easily. In the next paragraphs, we will first explain the functionality and design of the initial application, followed by an overview of the change requests.

A. The Initial Budget Management Application

Budget tracking and management are important aspects within the administration of the local government. Budgets need to be awarded, specified, managed and utilized for the local government to function properly and fulfill its services to the citizens. To accomplish this, the overall available budget is divided into very fine-grained sub-budgets. This however drastically complicates the budget assignment, reservation, fixations, changes, etc. To cope with this complexity -and simultaneously realizing the much-needed integration of budget management with project management and budget reporting- a project was started at the end of 2012 to develop a stand-alone application to capture the budget management functionalities.

The challenges of the budget application development are discussed in [4]. First, the new application needed to match the flexibly and user-friendliness of Excel pivot tables (which were previously used). To cope with this challenge, the development of the initial application was focused on budget management functionalities and its user-friendliness. As will be discussed in this paper, the initial application would, as such, be a sound basis for further extending the application to include other requirements such as budget reporting, simulation and project management, etc.

The context-specific and fine-grained composition of the budgets was another challenge of the application development. Budgets need to be defined in a range of different levels of the specific government. Therefore budgets can be managed on a scale from general to highly specific. On the most fine-grained or specific level, budgets are defined by a combination of the following six parameters: department, activity, article, domain, product and budget year. However, budgets can also be defined based on a subset of these parameters, meaning budgets can be defined on several levels.

After going through a functional analysis and NS software development process, the final application architecture is shown by the set of unchanged, changed and removed data element instances in Fig. 1. This figure clearly shows how the application is structured around a central data element instance, being a “Budget”. A current budget is defined by the aggregation of “Budget Changes” made to the budget over time. The parameters that can be used to define a budget -“Department”, “Activity”, “Article”, “Domain”, “Product” and “Budget year” instances- are shown on the left side of Fig. 1. These “Articles” can be grouped into “Economic groups”. In the initial application, Economic groups make up a “Budget estimate”. This estimate used to be utilized to draw up a target budget at the beginning of a budget year. The data elements instances on the right of the Budget instance in Fig. 1 are used for managing budgets. “Budget fixations” are used secure a part of a budget for a specific purpose. These Budget fixations are assigned to a specific “Supplier” and can be called in “Budget calls”, so the budgets can be partially spent when needed. Budget calls are associated with “Invoices” and “Work orders” to track the spending of “Budget calls”. Other aspects of the application will be discussed in the next section, as they are part of the changes made to the initial application.

B. Change Requests for the Application

As mentioned before, the government officials had several change requests after having used the budget management applications for some time. Additionally, changes in legislation required the introduction of a “Purchase file” data instance to enable long-term (i.e., more than one year) tracking of purchases of departments. Thereby, this case show how an application that was in use for only one year already needed functional changes. This meant that a lot of software features had to either be added, changed or removed. One developer even expressed that one could argue the change requests were so extensive and concerned the foundation of the application that you could consider it as a new application. NS theory however allows for far-reaching changes to be made to an application, and the renewed application is therefore considered a second version of the budget application. In the following paragraphs, we will briefly review some of the changes that have been made to the application, for which Fig. 1 can be used as a reference.

The first significant change was the addition of a project management functionality to the application. Because the NS methodology was used, this functionality could be left out in the initial application to be easily added later on. The project management functionality that needed to be added involved project monitoring through observation of the state of work orders. These work orders get initiated in the system for specific tasks and are linked to the budget fixation they belong to. When an invoice is received for a work order, the invoice lines are linked to specific work order lines. To fully implement the new project management functionality, new data element instances for "Consultant" and "Profile" needed to be added as well.

Another change request was the extension of invoice management. Invoices were made more detailed by adding a data element instance "Invoice line" and by linking invoices directly to the budget fixation they belong to. Previously invoices were defined on three levels: for fixations, budget calls and work orders. For simplification and centralization reasons, this was reduced to only one level in the revised application.

The third major change in the new application is the inclusion of purchase orders. These orders needed to be added to the application as the purchasing department needed to be able to control purchases according to the granted budgets. Additionally, a element instance for "Purchase File" was added to hold all the information on specific purchase orders.

Furthermore, it was made possible to further specify an activity by adding the data element instances "Action plan", "Policy domain" and "Management domain". And as budget estimates were not used in the application, the corresponding element instance was removed.

These changes show that only the fundamental functionality of the application stayed unchanged in the new version (i.e., the budget, budget change and budget-defining data element instances). With the exception of four element instances, all initial element instances needed to be "touched" and several element instances needed to be added to provide the requested functionality changes.

IV. Implementation of the Changes

As the goal of NS theory is to design software in an evolvable way, it should be no surprise that the change requests discussed in the previous section could be implemented quickly and easily by just a single developer. By taking evolvability of software into account at design time, NS applications can effortlessly be extended through the descriptor files and the expansion mechanism.

In the descriptor files of the NS element instances, one can for instance easily change the data model of an application (i.e., the relationships between element instances). For example, although the "Work order" and "Invoice" retained the same name in the second version of the budget application, their definition and position in the data model changed completely (cf. previous section). The changes could however be applied by simply re-defining the relationships of these element instances and the instances they are linked with in their descriptor files.

Similarly, adding or removing NS element instances from an application can also be done by just writing or removing descriptor files for these instances. The "Work order Line", "Invoice Line" and "Consultant" element instances for example could be added to the application by creating new descriptor files containing information such as their description, relationships, etc. Additionally, relationships to these new instances need to be added in existing element instances that are coupled with the new element instances. As these changes can be done rather easily, implementing all the required changes to the descriptor files of the budget application took less than 1 man-day.

Although the NS expansion process delivers a fully working application that includes all defined NS element instances, the functionalities of the application most likely still need to be extended to provide context-specific functionality. This is done through manually programming customizations, either within anchor points in the expanded files (called "injections") or in separate files (called "extensions"). The additional functionality added to the budget application needed two important extensions and/or injections.

As the budget application is very data-intensive (i.e., it exists of only NS data element instances), a lot of data validations needed to be implemented in the application. For example, budget calls can not exceed the available budget, budgets need to be unique, etc. Implementing these validations took 3 man-days.

The second important type of extensions that needed to be implemented were graphical extensions. These included colored status boxes, projections and HTML screens to implement these projections. Projections are views that can be defined on NS data element instances that show data in a specific way. These projections need to be defined so information can be projected to the end user in the most useful way in specific use cases. For example, when a user is looking up information on a specific budget, this screen also needs to display overall information on all budget calls made on this budget. From this overall information, the user can then select a budget call to get more information on. It however does not need to show detailed information on all budget calls in the budget screen, as this would lead to an information overload on this screen. Furthermore, different projections can be defined depending on user roles. For the 25 end users of the budget application, several roles are defined in the application (e.g., super user, administrator, manager, employee). As such, managers or members of a specific department can be shown more detailed information than regular employees. Implementing the custom graphical elements for the second version of the application took 5 man-days in total, of which the projections took 3 days to implement.

V. Discussion

From the description of the case in the previous section, one can make several observations.

First off all, the benefit of reduced effort of changing NS applications seems to come at an extra cost or effort at design time of the application. However, the additional effort to design in an evolvable way is negligible. Previous research has shown that the NS expansion process is very efficient

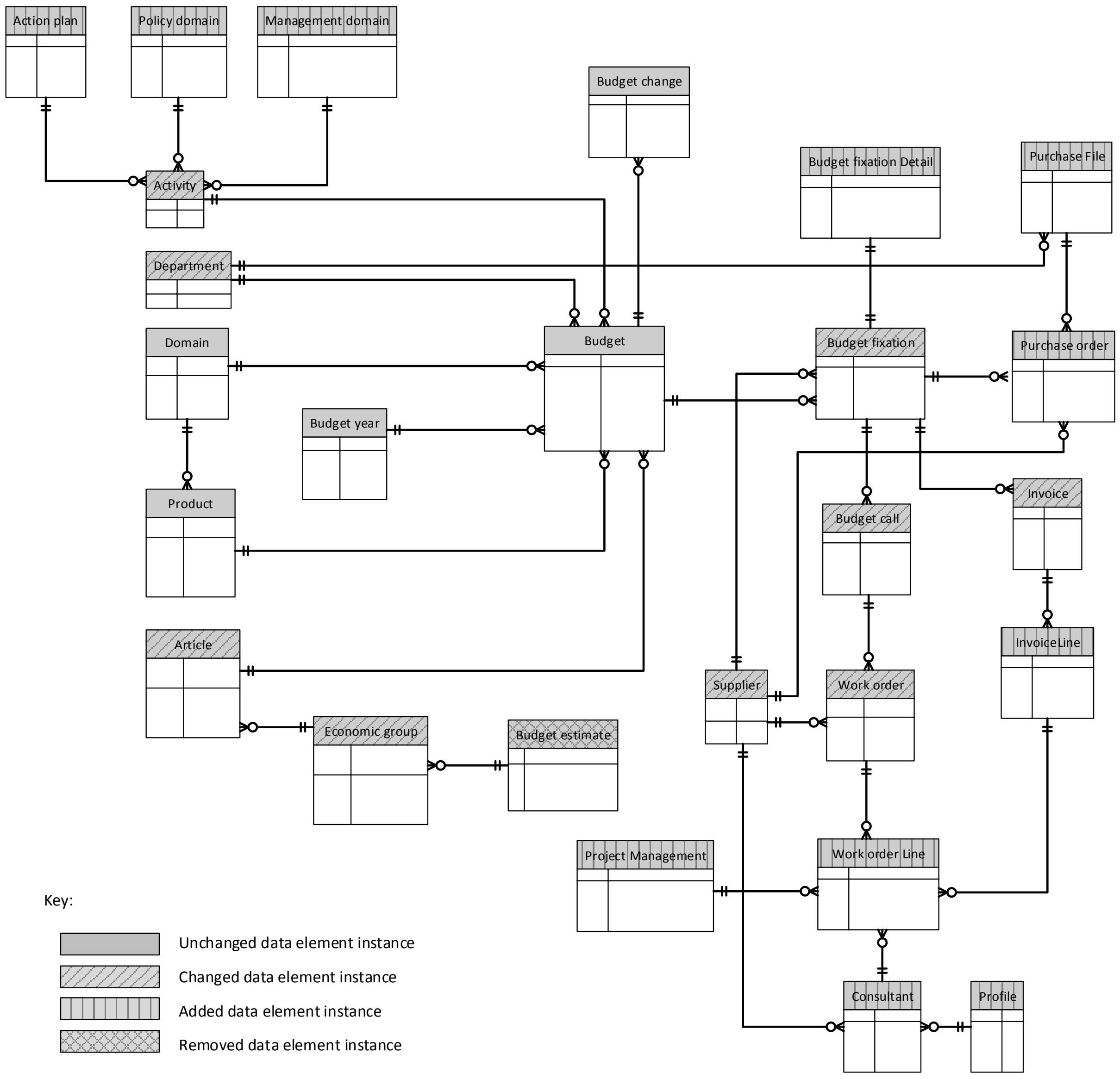

Figure 1. Entity Relationship Diagram Showing the Architecture and Changes to the Budget Application

and fast and even provides a way of developing software faster than traditional development methodologies [4]. This is because the developer does not need to concern himself with the software architecture or boilerplate code once the NS element instances are defined in descriptor files. The only prerequisite is that additional knowledge on the NS theorems, elements and expanders is required for developers to be able to develop software that is fully according to NS theory.

Once an application has been built according to NS principles, the case description also shows it can be easily changed. The total development time of the thorough changes to the budget application was only about 9 man-days. According to the developer, the entire job was very clear to him. And he reckoned the amount of effort he had to spent on re-developing the application was much less than something they normally do when an application is not designed based on NS theory.

A third observation is that because the rapid development of the new versions of an application, issues that are otherwise proportionally irrelevant, can become even more time-consuming than the development itself. For the revision of the budget application, there was a lot of effort needed to convert and input the old Excel-data in the new application. This is because of missing data, inconsistencies, wrong data formats, etc. Overall this even took more effort than developing the new version of the application.

A. Knowledge transfer

NS development also incorporates knowledge management processes that support capturing, storing, transferring and applying development knowledge. How this works is discussed in [7], based on the widely used theoretical framework of [8]. Basically, knowledge is captured and transferred through the use of NS expansion, as this process provides a way to incorporate new insights obtained from practical application of the theory into the NS knowledge base (i.e., the NS expanders). Captured knowledge can be newly normalized features in the NS elements that can be re-used in future applications, new general reflections on building Normalized software, etc. This way, newly normalized features can simply be provided to new applications through NS expansion and they do not need to be manually added after expansion. As such, any new version of the NS expanders can as well be used to re-expand older applications to provide additional functionality, graphical and more user-friendly enhancements, etc.

During the development of the revised budget application, several insights gained from previous NS projects could be used, including the initial development of the budget management application. One example of functionality that could be added more easily in the application revision are composite screens. These advanced screens provide an overview of data on different levels. They show for example the budget of a department, and by selecting an activity, domain, article, etc., one can drill down to a specific budget on the same screen. Before the start of the initial budget application, implementing such screen would take about 600 lines of code in manually programmed extensions. By incorporating some of the functionality of composite screens in the NS expanders, this was reduced to about 60 lines of manual code during the development of the initial budget application. In this revision of the application the effort needed was even further reduced to about 5 to 10 lines of code for each layer in a composite screen. The development of the revised budget application also lead to the addition of new knowledge to the NS expanders. The idea of projections (cf., previous section) that only show relevant information on a NS data element instance (e.g., when in a list of departments, one is only interested in total budget of the departments) is very useful in a large array of contexts and applications. Therefore they were added to the expanders after completion of the project.

B. Contributions and future research

This paper has several *contributions*. First, it shows the advantages of building software according to the NS design theory. These advantages can normally only be observed over long periods of time, when systems are required to evolve or be adapted. The case description in this paper however already shows for the first time how fundamental changes can be made to an application without excessive implementation effort (e.g., implemented by only one developer over a very limited amount of development days). Furthermore, the absence of combinatorial effects in NS applications will also make sure that the effort of implementing changes does not increase over time, as the application becomes larger and more complex. Second, the case description shows how the NS development process (discussed in detail in [4]) also supports the implementation of changes to an application by using the descriptor files and NS expanders.

Possibilities for *future research* include additional case studies to provide more information on how the NS theory realizes profound progress with regard to the evolvabilty of software.

VI. Conclusion

In this paper, we discussed how software can be easily revised and adapted when it is built according to NS theory. This was shown by means of describing the revision of a budget management application built for a local Belgian government, of which the NS development is previously discussed [4]. This case shows how, even after only being used for a single year, changes needed to be made to the software design because of regulatory and requirement changes. As such, the paper shows how these fundamental changes can be made easily and without much effort. Because the case application was built according to NS principles, one will also be able to implement future changes with the same ease and the application will thereby become evolvable.

References

[1] M. Lehman and J. Ramil, “Rules and tools for software evolution planning and management,” *Annals of Software Engineering*, vol. 11, no. 1, pp. 15–44, 2001.

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

[3] H. Mannaert, J. Verelst, and K. Ven, “The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability,” *Science of Computer Programming*, vol. 76, no. 12, pp. 1210 – 1222, 2011, special Issue on Software Evolution, Adaptability and Variability. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016764231000208X

[4] G. Oorts, P. Huysmans, P. D. Bruyn, H. Mannaert, J. Verelst, and A. Oost, “Building evolvable software using normalized systems theory: A case study,” *2014 47th Hawaii International Conference on System Sciences*, vol. 0, pp. 4760–4769, 2014.

[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, pp. 89–116, 2012. [Online]. Available: http://dx.doi.org/10.1002/spe.1051

[6] P. D. Bruyn, G. Dierckx, and H. Mannaert, “Aligning the normalized systems theorems with existing heuristic software engineering knowledge,” in *Proceedings of The Seventh International Conference of Software Engineering Advances (ICSEA)*, ser. ICSEA ’12, Lisbon, Portugal, 2012, pp. 84–89.

[7] P. D. Bruyn, P. Huysmans, G. Oorts, D. V. Nuffel, H. Mannaert, J. Verelst, and A. Oost, “Incorporating design knowledge into software development using normalized systems,” *International Journal On Advances in Software*, vol. 6, no. 1&2, pp. 181 – 195, 2013.

[8] 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. pp. 107–136, 2001. [Online]. Available: http://www.jstor.org/stable/3250961