Using Normalized Systems Expansion to Facilitate Software Migration - a Use Case

Christophe De Clercq

Research and Development
fulcra bv, Belgium
Email: christophe.de.clercq@fulcra.be

Jan Verelst

Department of Management Information Systems
Faculty of Business and Economics
University of Antwerp, Belgium
Email: jan.verelst@uantwerpen.be

***Abstract*—Applications with evolvability issues, becoming less and less modifiable over time, are considered legacy. At some point, refactoring such applications is no longer a viable solution, and a rebuild lurks around the corner. However, without a clear architecture that will enforce evolvability, the new application risks becoming non-evolvable over time. Re-building an existing application offers little business value; migrating from old to new can be complicated. Normalized Systems (NS) theory aims to create software systems exhibiting a proven degree of evolvability. One would benefit from building legacy systems according to this theory if legacy systems are to be rebuilt. In this paper, we will present a real-life use case of an application exhibiting non-evolvable behaviour and how this application is being migrated gradually into an evolvable application through NS-based software expansion. We will also address the extra value that NS-based software expansion brings in the migration scenario, allowing the combination of old and new features in the newly built application.**

Keywords*—*NS; Rejuventation; Software Migration

I. Introduction

The research on agile software development has increased in the last few years. This research has helped to improve the agile development methods, but there has not been much attention paid to making the software more agile.

Agile Architecture, as defined by key agile frameworks such as Scaled Agile Framework (SAFe) [1], is a set of values and principles that guide the ongoing development of the design and architecture of a system while adding new capabilities. This definition describes more of a process than a guarantee that the system being built will be agile, meaning the ability to change. An agile architecture is an architecture that can change. It is a feature of a system that requires deliberate design. Therefore, agile architecting is a better term to describe an agile approach to architecture, and agile architecture should indicate the intentionality to create a dynamic system.

Normalized Systems (NS) theory aims to increase software agility by designing software systems with agile architectures. Software evolvability, or how easily software can be modified, can be achieved by following a set of theorems that lead to a specific and evolvable software architecture. NS theory has been developed and improved over time. It is fully based on theoretical foundations and has been applied in several software projects. Previous research has documented the theoretical contributions of NS theory well, but there are few studies on real-life cases where NS theory has been used. This paper reports on a development project that shows the viability of the NS theory method for creating evolvable software and emphasizes the advantages of a real-life NS development project. We show how NS can help with an information system migration use case, and how it can make the target system adaptable. The paper is organized as follows: Section II explains the basics of NS, and Section III summarises software migration strategies. Section IV presents the use case, and Section V discusses the benefits of NS in this scenario. We conclude the paper in Section VI.

II. Fundamentals of NS theory

Software should be able to evolve as business requirements change over time. In NS theory [2], the lack of Combinatorial Effects measures evolvability. When the impact of a change depends not only on the type of the change but also on the size of the system it affects, we talk about a Combinatorial Effect. The NS theory assumes that software undergoes unlimited changes over time, so Combinatorial Effects harm software evolvability. Indeed, if changes to a system depend on the size of the growing system, these changes become harder to handle (i.e., requiring more work and therefore lowering the evolvability of the system).

NS theory is built on classic system engineering and statistical entropy principles. In classic system engineering, a system is stable if it has Bounded Input leading to Bounded Output (BIBO). NS theory applies this idea to software design, as a limited change in functionality should cause a limited change in the software. In classic system engineering, stability is measured at infinity. NS theory considers infinitely large systems that will go through infinitely many changes. A system is stable for NS, if it does not have Combinatorial Effects, meaning that the effect of change only depends on the kind of change and not on the system size.

NS theory suggests four theorems and five extendable elements as the basis for creating evolvable software through pattern expansion of the elements. The theorems are proven formally, giving a set of required conditions that must be followed strictly to avoid Combinatorial Effects. The NS theorems have been applied in NS elements. These elements offer a set of predefined higher-level structures, patterns, or “building blocks” that provide a clear blueprint for implementing the core functionalities of realistic information systems, following the four theorems.

A. NS Theorems

NS theory [2] is based on four theorems that dictate the necessary conditions for software to be free of Combinatorial Effects.

- Separation of Concerns
- Data Version Transparency
- Action Version Transparency
- Separation of States

Violation of any of these 4 theorems will lead to Combinatorial Effects and, thus, non-evolvable software under change.

B. NS Elements

Consistently adhering to the four NS theorems is very challenging for developers. First, following the NS theorems leads to a fine-grained software structure. Creating such a structure introduces some development overhead that may be considered slowing down the development process. Secondly, the rules must be followed constantly, robotically, as a violation will lead to the introduction of Combinatorial Effects. Humans are not well suited for this kind of work. Thirdly, the accidental introduction of Combinatorial Effects results in an exponential increase of rework that needs to be done.

Five expandable elements [3] [4] were proposed, which make the realization of NS applications more feasible. These elements are carefully engineered patterns that comply with the four NS theorems, and that can be used as essential building blocks for various applications: data element, action element, workflow element, connector element, and trigger element.

- **Data Element**: the structured composition of software constructs to encapsulate a data construct into an isolated module (including get- and set methods, persistency, exhibiting version transparency, etc.).
- **Action Elements**: the structured composition of software constructs to encapsulate an action construct into an isolated module.
- **Workflow Element**: the structured composition of software constructs describing the sequence in which action elements should be performed to fulfil a flow into an isolated module.
- **Connector Element**: the structured composition of software constructs into an isolated module allowing external systems to interact with the NS system without calling components statelessly.
- **Trigger Element**: the structured composition of software constructs into an isolated module that controls the states of the system and checks whether any action element should be triggered accordingly.

The element provides core functionalities (data, actions, etc.) and addresses the Cross-Cutting Concerns that each of these core functionalities requires to properly function. As Cross-Cutting Concerns cut through every element, they require careful implementation to not introduce Combinatorial Effects.

C. Element Expansion

An application comprises a set of data, action, workflow, connector, and trigger elements that define its requirements. The NS expander is a technology that will generate code instances of high-level patterns for the specific application. The expanded code will provide generic functionalities specified in the application definition and will be a fine-grained modular structure that follows the NS theorems (see Figure 1).

The business logic for the application is now manually programmed inside the expanded modules at pre-defined locations. The result is an application that implements a certain required business logic and has a fine-grained modular structure. As the generated structure of the code is NS compliant, we know that the code is evolvable for all anticipated change drivers corresponding to the underlying NS elements. The only location where Combinatorial Effects can be introduced is in the customized code.

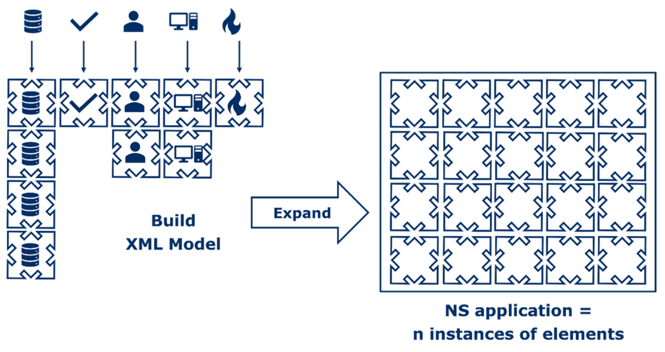

Fig. 1. Requirements expressed in an XML description file, used as input for element expansion.

D. Harvesting and Software Rejuvenation

The expanded code has some pre-defined places where changes can be made. To keep these changes from being lost when the application is expanded again, the expander can gather them and put them back when the application is re-expanded. Gathering and putting back the changes is called harvesting and injection.

The application can be re-expanded for different reasons. For example, the code templates of the elements are improved (fix bugs, make faster, etc.), new Cross-Cutting Concerns (add a new logging feature) are included, or a change in technology (use a new persistence framework) is supported.

Software rejuvenation aims to carry out the harvesting and injection process routinely to ensure that the constant enhancements on the element code templates are incorporated into the application.

Code expansion produces more than 80% of the code of the application. The expanded code can be called boiler-plate-code, but it is more complex than what is usually meant by that term because it deals with Cross-Cutting Concerns. Manually producing this code takes a lot of time. Using NS expansion, this time can now be spent on the constant improvement of the code templates, the development of new code templates that make the elements compatible with new technologies, and on meticulous coding of the business logic. The changes in the elements can be applied to all expanded applications, giving the concept of code reuse a new meaning. A modification on a code template by one developer can be used by all developers on all their applications with minimal impact, thanks to the rejuvenation process.

III. Fundamentals of Software Migration Strategies

Software systems are supposed to change over time as the business environment changes. When a system has issues following the changes, it is marked as legacy.

In [5], a legacy information system is defined as any information system that significantly resists modification and change. The main reasons for becoming legacy are the lack of system flexibility (the very definition of legacy) and the lack of skills to change the system.

Information Systems are closely linked with the technologies they depend on, which also evolve. These changes are not driven by the business context but by the progress and shifts in technology and its market. When some technologies lose their support from the providers, their expertise will also disappear, leading to a shortage of skilled resources to make the necessary changes to the information system.

If a system is outdated but the business still needs to change and improve, the only solution is to redesign the system and move it to a new platform.

Formally, re-engineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Re-engineering generally includes some form of reverse engineering (to achieve a more abstract description) followed by some more form of forward engineering or restructuring (from [5]).

Usually, the re-engineering of a new system will involve not only current functionalities but also future functionalities. Re-engineering provides the old and new requirements, while migration builds and uses the new system that replaces the legacy one.

Figure 2 shows the three activities that are part of the migration process:

- The transformation of the conceptual information schema (S)
- The data transformation (D)
- The programming code transformation (T)

The order of the three migration activities can vary, affecting when the target system is ready for end users. The literature defines the following generic methods:

- Database first: migrate data first, then migrate programming gradually, and go live when all programming migrations are done.
- Database last: migrate programming first, go live when all data is migrated.

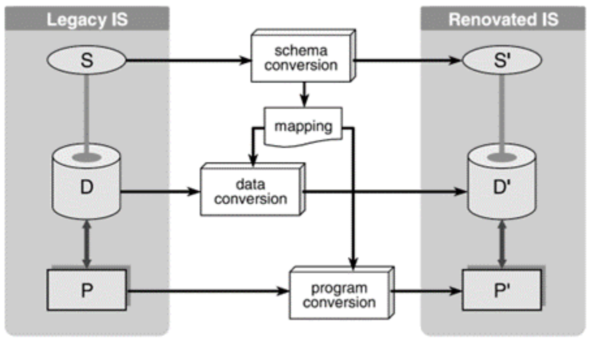

Fig. 2. Conceptual schema conversion strategy (from [5])

- Composite database: migrate data and functionality together, and go live when both are migrated.
- Chicken Little strategy: like a composite database but keep both legacy and replacement systems running simultaneously.
- Big bang methodology: develop a new system, stop the old system, migrate data, and start a new system.
- Butterfly methodology: big bang with data synchronization techniques to reduce data migration time and downtime.

Each of these strategies has advantages and disadvantages. We refer to [6] for more details.

IV. Use Case: Connecting-Expertise

This paper presents a case study of migrating a legacy information system using NS principles and NS expansion/rejuvenation, which helped overcome some of the limitations of the selected migration strategy.

We begin by providing a functional view of the legacy system, followed by a technical view. We then discuss the legacy system's evolvability problems, justify the need for a new system, and describe how the transition from old to new occurred.

A. Functional perspective

Connecting-Expertise is a company that provides a software platform called CE VMS that helps to improve and simplify the sourcing, assigning, and management of an organization's workforce. Connecting Expertise uses a software platform to connect job-seekers and job-suppliers quickly and efficiently.

When a job-seeker (seeking a human resource for a job) and a job-supplier (supplying a human resource for a job) find each other on the platform, the platform handles the necessary administrative steps to make someone work effectively, such as creating assignments, creating and processing timesheets, and invoicing based on timesheets.

The business model of Connecting-Expertise combines a buyer-funded model, where a job-seeker pays a license or a fee per hour worked by a consultant to use the platform, and a vendor-funded model, where a job-supplier pays per hour worked by a consultant.

B. Technical perspective

The first version of CE VMS dates from 2007. CE VMS's core comprises a web server that uses PHP and a MariaDB MySQL backend DB. The application has components such as DTO/DAO classes (for data storage, access, and exchange), HTML view templates, and CLI scripts for running background processes.

In 2017,, some CE VMS kernel features were separated and moved to a new PHP server with a Zend Apigility API framework. This setup is called CE2 VMS. The APIs are only for internal use (not accessible by the job-seekers and suppliers systems) and even though the features provided by the API are not part of the CE VMS kernel, both kernel and API framework use common code (like the data access logic, as they both connect to the same database). The shared code is in a library that both the kernel and the APIs use, but some code, like DTO and DTA classes, exist in both the kernel and the library.

The queuing system is a key component of the current system, as it transfers tasks that take a long time from the web application to specialized processing servers. The tasks that take a long time are placed in a queue processed by node.js scripts. These scripts will invoke the relevant (internal) APIs, communicate with the DB, and even call external APIs of CE2 VMS users' systems. An overview of the technical architecture can be found in Figure 3 .

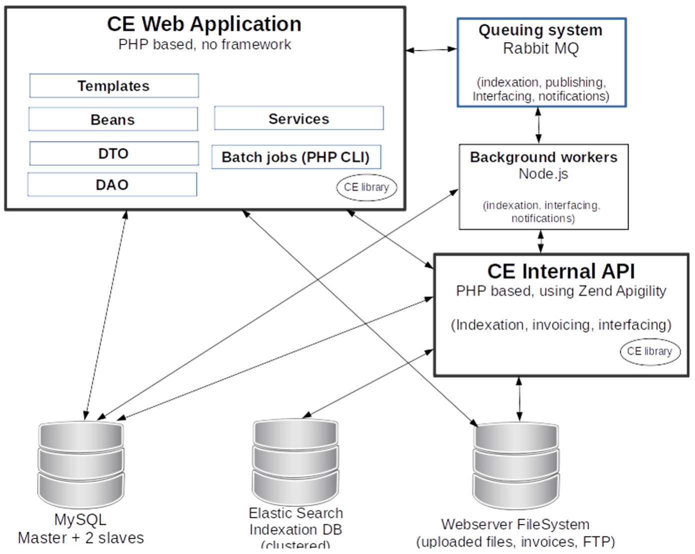

Fig. 3. CE2 VMS technical architecture.

C. Maintainability and evolvability issues

The following sections will describe the main problems affecting the system's maintainability and evolvability: the code base, code quality, technical architecture, scalability, and functionality. Each of these areas will be explained in more detail below.

1) Code base: The code base was developed without proper coding standards that were maintained and followed. First, the SOLID principles [7] were suggested as a coding standard at some point, but the standard is not systematically applied and verified, leading to many violations. Second, current coding practices led to highly coupled code because of the use of global variables and the absence of interfaces. Third, many classes are long and complex, and a lot of unused code has not been removed. Fourth, consistent naming conventions for database elements and attributes are missing. Finally, we reiterate the previous point of code duplication between the kernel and the libraries and the lack of standard frameworks that could help structure the system and the code.

2) Code Quality: The code has quality problems because there are no coding standards. First, there is no testing plan to test each class or component of the application. Second, doing functional acceptance tests is hard because the code is complex. One needs to know many technical details (like how the queue works, DB queries, and manual running of background jobs to do end-to-end tests). Third, security coding practices are not used, so the code is vulnerable to common security risks like SQL injection because input data is not validated properly. Finally, releasing a new version is a big deal instead of a routine, often needing last-minute fixes, even when acceptance testing seems good.

3) Technical Architecture: The technical architecture documentation (the infrastructure, system software, and networking used) is not consistent, complete, or coherent. This might account for the redundancies observed, such as using two different indexing databases, two worker systems, two invoicing systems, and a custom approach to connecting with external systems. The reason for having two different technical environments for serving the BE and UK markets is not justified and leads to double maintenance. There is a strong dependency between the code base and the underlying technical infrastructure. Changing underlying technical components (such as the DB) is very difficult because of the lack of abstraction of the technologies used (tight coupling between c code and Maria DB).

4) Scalability: A system that can cope with a growing amount of work by adding resources has scalability. The current environment has some components that are hard to scale. First, the DB (MariaDB – MySQL) is not clustered (no load balancing option, and it is on the same server as the web server, which means they share the server resources). Second, the file storage area for timesheet uploads is only accessible from the web server, so all background processes that need these files (like the background invoicing process) must also run on the web server (which also shares the resources). Third, the Xapian indexation system does not work across the network, and it has to run on the web server, just like the current job executer (Jenkins). There is also resource sharing here. Lastly, the application does not use caching mechanisms, which leads to unnecessary DB queries.

5) Functionality: The system is complicated to set up for new clients. They frequently need new application settings, reports, or even application functions. This makes it hard to expand the application to more customers (for example, in a new country). The system also has a limitation on the currency: some system modules only support the Euro.

D. The Need for Change

Connecting-Expertise needs to enable integration with the backend systems of job-seekers and suppliers to remain competitive as a platform. However, this development is hindered by current issues of evolvability. Connecting-Expertise faces a challenge: how can CE2 VM offer integration with external systems, along with existing and new functionalities, without affecting the current CE2 VMS platform and creating a whole new CE platform from scratch?

1) New setup: In 2021, a new system, called CE3 VMS, was being put forward. It consists of a set of external APIs that provide integration functionalities with job-seeker and supplier systems. These APIs call a new set of internal APIs, which expose the new CE data model.

As we discussed, the CE2 VMS data model is inconsistent and lacks anthropomorphism. For CE3 VM, a new data model that follows the NS evolvability principles is being put forward. Connecting-Expertise decided to create a set of APIs that would enable external integration and calls toward the CE3 VMS. These APIs would interact with internal APIs that expose existing CE2 VMS functionalities, new CE3 VMS functionalities, and the new CE3 VMS data model. In the next sections, we will explain the reason for an NS approach, the new CE3 VMS data model, the conversion from CE3 VMS to the CE2 VMS data model, the overall transition strategy from CE2 VMS to CE3 VMS, and the benefit of rejuvenation.

2) NS Expansion approach: Connecting-Expertise realized that their platform had issues with adaptability. Connecting-Expertise liked the NS approach but was not completely convinced about using NS Expansion with the NSX tools [8]. Two methods were compared: building the new CE3 system following the NS principles or the CE3 system with the NSX tools. Essentially, this means deciding between working with or without software expansion. All stakeholders were informed about both methods and a qualitative comparison was done by the stakeholders. The result of this comparison (see Figure 4) was that an expansion-based method using the NSX tools, was preferred. It should be noted that this was a qualitative comparison, which needs to be verified again once implementation starts and/or finishes (see Section V).

3) CE3 VMS Data Model: CE3 VMS does not rebuild existing functionalities. Instead, it uses the CE3 VMS data model to call existing functionalities (as a data exchange format) and converts the CE3 VMS data model to the CE2 VMS data model so that the corresponding CE2 VMS functionalities can be used. Data already in CE2 VMS is accessed/stored via APIs on CE3 VMS. Only when new functionalities on CE3 VMS introduce new data types, the data will be stored and accessed in the CE3 VMS-specific database.

CE3 VMS uses two types of data elements. One is for CE3 VMS native data, which can only be accessed and used by

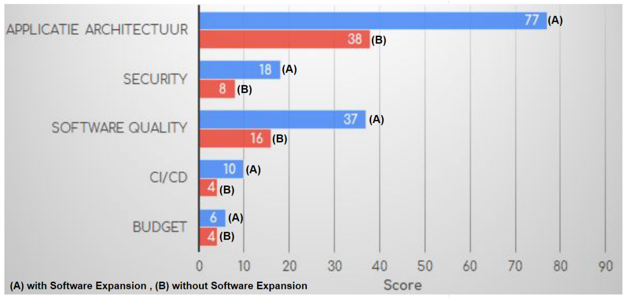

Fig. 4. Implementing CE2 with or without Software Expansion.

CE3 VMS, called a CE3 data element. Another is for data in CE2 VMS that CE3 VMS exposes through a CE3/CE2 data element. The CE3/CE2 data elements transform the less anthropomorphic CE2 data elements into a data structure according to NS principles. The CE2 data element will be aggregating a certain amount of CE3/CE2 data elements. Figure 5 shows an example modelled in ArchiMate. The diagram shows a data object d_A_CE2 that is an aggregation of d_a1_CE3/CE2, d_a2_CE3/CE2 and d_a3_CE3/CE2, and accessible via CE2 and CE3, while data object d_b_CE3 is only accessible via CE3. Transformers are used to convert the CE2 data object and CE2/CE3 data objects.

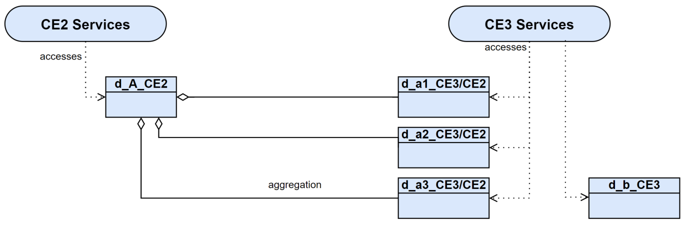

Fig. 5. Transformation of data objects between CE2 and CE3.

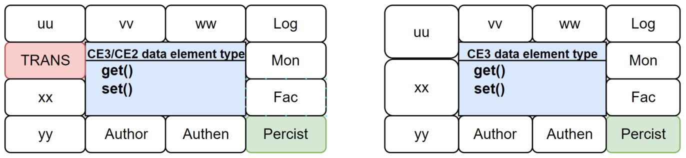

Fig. 6. Transformer as a Cross-Cutting Concern of the CE3/CE2 data element type.

4) The Transformer Cross-Cutting Concern: The transformers deal with a Cross-Cutting Concern that affects both CE2 and CE3. They are special classes that belong to the CE3/CE2 data elements of CE3 VMS.

All the expanded CE3/CE2 data elements have a transformer inside them as a Cross-Cutting Concern. The transformer's role is to map the CE3 data model to the CE2 data model. When an instantiated CE3/CE2 data element performs persist/retrieve actions, the transformer will change the CE3 data into the CE2 format - like an ETL operation - and then do the persist/retrieve action on the CE2 database. This approach requires the CE3

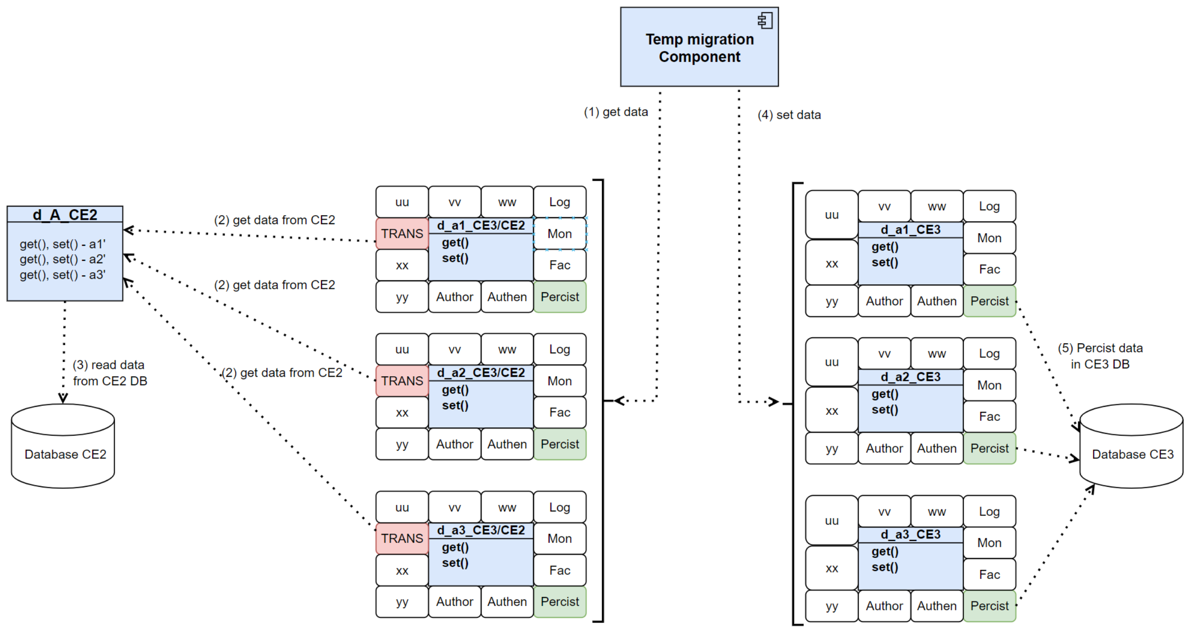

Fig. 7. Migration of data from CE2 VMS to CE3 VMS.

and CE2 data models to be unambiguously mappable. This was ensured during the design of the CE3 data model. Figure 6 shows the difference between the 2 data element types.

A feature available on CE2 VMS will use the data elements created on CE2 VMS. The same feature can be accessed from CE3 VMS through the CE3/CE2 data elements. When all users of this feature switch from using it on CE2 VMSand start using it on CE3 VMS (moving users from the old to the new platform for that feature), it is time to also move all the relevant data from the CE2 VMS database to the CE3 VMS database. The transformers will help with this migration.

A migration task would just get the CE2 data through the CE3/CE2 data element and save it into a CE3 data element. After this migration task is done, the feature that needs this data will only use the native CE3 data element, making a smooth transition from one system to the other. Figure 7 explains the process.

5) Rejuvenation and Transformation: To create CE3 VMS, a connection with CE2 VMS had to be embedded in the code. The parts of the code that handle this connection are in the transformation classes. These classes belong to the CE3/CE2 data elements. When setting up the meta-model used as the basis for the code expansion, data elements will be marked as either type CE3/CE2 or type CE3. All transformation classes are then included in the expansion. When a data structure does not need to be linked to both CE2 and CE3 anymore, it is enough to specify this in the meta-model and re-expand. CE3 data elements will be applied at that point, and the transformers are no longer required. The process of re-expansion that improves the element structures is called rejuvenation. In this case, the rejuvenation process eliminates all code and connections to CE2, removing the link to legacy.

V. Discussion

In this section, we will discuss different aspects of the migration approach. We will start with the choice of NS expansion, followed by the value of a phased migration. We will end by comparing this migration approach with a generic migration approach called Chicken Little [6].

A. *The choice for NS Expansion*

In Section IV-D2, we explained why Connecting-Expertise chose to use NS Expansion compared to standard programming using the NS principles as guidelines. We asked the Connecting-Expertise's lead developer, Sven Beterams, if the estimated gains of using NS Expansion also materialized during project delivery. He confirmed that thanks to NS Expansion, the development went faster, the code quality improved considerably, and the data model was anthropomorphic and consistent. The development of the backend was greatly improved and the phased migration approach was made possible thanks to NS Expansion/Rejuvenation.

B. *Migration Approach*

The usage of the transformers plays an essential role in the migration from CE2 VMS toward CE3 VMS. The idea of gradually shifting functionalities from one system to another while keeping both live is referred to as the Chicken Little approach (see [6]). The main drawback of using this approach is the need for gateways between the source and target system. These gateways must be meticulously designed and consistently implemented, which can be daunting. NS Expansion mitigates the downsides of doing Chicken Little dramatically. The gateways are implemented using the transformer classes that are part of the data elements. Using NS Expansion ensures that each gateway/transformer is identical in structure and usage. The transformers can evolve, and all modifications and improvements can be quickly and easily redeployed using re-expansion/rejuvenation. When functionality is fully migrated from the source to the target system, there is no longer the need to keep the gateways in place. With classic coding practices, the manual removal of the gateways comes with risks. Accidental removal of too much could result in broken functionalities. Insufficient removal results in traces of legacy code in a brand-new system. With NS Expansion, it suffices to perform a rejuvenation cycle to replace the code templates that contain transformers with code templates without transformers. All traces of legacy are removed in a consistent and precise way.

C. Phased migration

Connecting-Expertise wanted to avoid a big-bang migration. The transformer approach facilitated this even more. The ease with which the final migration of data can be performed (as described in Figure 7) is thanks to the usage of the transformer Cross-Cutting Concern and the ability to rejuvenate the code and erase all links to legacy after final migration. Without the NS Expansion approach, this task would be much harder.

VI. Conclusion

This paper presented a real-life case where software migration is facilitated by NS Expansion. We introduced NS, NS Expansion, and a general overview of software migration approaches. We presented the Connecting-Expertise use case, where a mission-critical platform needed to evolve while keeping the existing system operational. We have shown that addressing the migration as a Cross-Cutting Concern, using transformer classes embedded in data elements, combined with NS Expansion and rejuvenation, can mitigate some of the major drawbacks of a phased migration.

Acknowledgment

The authors would like to thank Sven Beterams from Connecting-Expertise, for sharing his knowledge of the applications. We would also like to thank Chetak Kandaswamy for collecting and structuring the material required to create this paper.

References

[1] SAFe Framework, [Online], Available: www.scaledagileframework.com, [retrieved: April, 2024]

[2] H. Mannaert, J. Verelst, and P. De Bruyn, “Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design”, ISBN 978-90-77160-09-1, 2016

[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, Volume 76, Issue 12, pp. 1210-1222, 2011

[4] P. Huysmans, G. Oorts, P. De Bruyn, H. Mannaert, and J. Verelst, “Positioning the normalized systems theory in a design theory framework”, Lecture notes in business information processing, ISSN 1865-1348-142, pp. 43-63, 2013

[5] S. Demeyer and T. Mens, “Software Evolution“, ISBN 978-3-540-76439-7, 2008

[6] A. Sivagnana Ganesan and T. Chithralekha, “A Comparative Review of Migration of Legacy Systems“, International Journal of Engineering Research & Technology (IJERT), ISSN 2278-0181, Volume 6, Issue 02, February 2017

[7] R. Martin, ”Clean Architecture”, ISBN-13 978-0-13-449416-6, 2017

[8] NSX, [Online], Available: www.normalizedsystems.org, [retrieved: April, 2024]