Facilitating Software Migration using Normalized Systems Expansion - A Detailed Case Study

Christophe De Clercq
Research and Development
Fulcra bv, Belgium
Email: christophe.de.clercq@fulcra.be

Geert Haerens
Antwerp Management School, Belgium
Engie nv, Belgium
Email: geert.haerens@engie.com

***Abstract*—Applications with evolvability issues that become 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, the new application risks becoming non-evolvable over time without a clear architecture that will enforce evolvability. Rebuilding an existing application offers little business value; migrating from old to new can be complicated. Normalized Systems theory aims to create software systems exhibiting a proven degree of evolvability. According to this theory, one would benefit from building legacy systems if they were 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. The paper demonstrates that software expansion facilitates phased software migration without the downsides of fragile manual-built gateways and results in a future-proof and evolvable new software system.**

Keywords*—*NS; Expansion; Rejuvenation; Software Migration

I. Introduction

This article extends a previous contribution originally presented at the Sixteenth International Conference on Pervasive Patterns and Applications (PATTERNS) 2024 [1].

Research on agile software development has increased in the last few years. This research has helped improve agile development methods, but little attention has been paid to making software more agile.

Agile architecture, as defined by key agile frameworks such as Scaled Agile Framework (SAFe) [2], 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 based on theoretical foundations and has been applied in several software projects. Previous research has documented the theoretical contributions of NS theory well [3] [4] [5] [6], but there are fewer studies on real-life cases where NS theory has been used [7]. 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 summarizes software migration strategies. Section IV presents the use case, and Section V will explain the migration approach. Section VI looks at the migration mechanism via carefully designed gateways called Transformers. We conclude with Section VII discussing the benefits of NS in this scenario and conclude the paper in Section VIII.

II. Fundamentals of NS theory

Software should be able to evolve as business requirements change over time. In NS theory [8], 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, suppose changes to a system depend on the size of the growing system. In that case, these changes become more challenging to handle (i.e., requiring more work and lowering the system's evolvability).

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, which leads 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 to strictly follow 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 [8] 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 four 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 slow the development process. Secondly, the rules must be followed constantly and robotically, as a violation will introduce 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 [9] [10] 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 system states 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 function correctly. Cross-cutting concerns cut through every element, so they require careful implementation to avoid introducing Combinatorial Effects.

C. Element Expansion

An application comprises 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 application's business logic 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 code's generated structure 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.

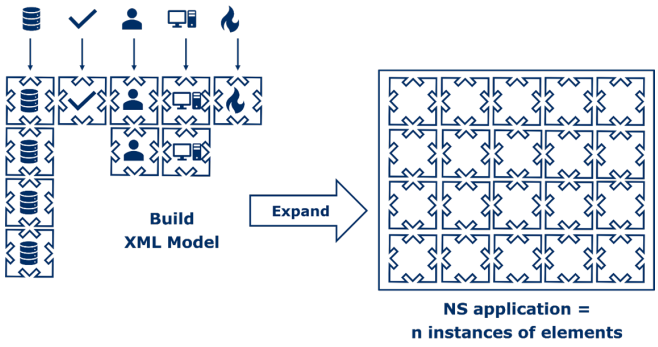

Fig. 1. Requirements expressed in an XML description file, used as input for element expansion.

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 re-inject them when 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 technology change (use a new persistence framework) is supported.

Software rejuvenation aims to routinely carry out the harvesting and injection process to ensure that the constant enhancements to 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 constantly improving the code templates, developing new templates that make the elements compatible with new technologies, and meticulously coding the business logic. The changes in the elements can be applied to all expanded applications, giving the concept of code reuse a new meaning. All developers can use a modification on a code template by one developer on all their applications with minimal impact, thanks to the rejuvenation process (see Figure 2).

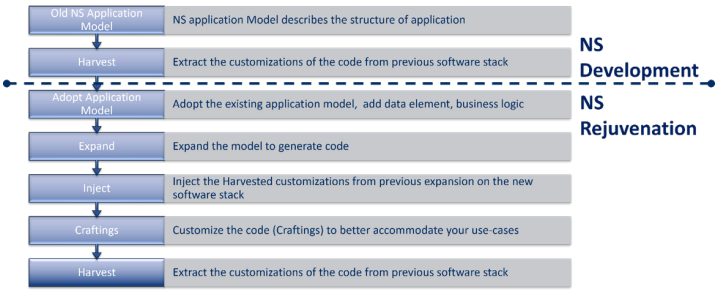

Fig. 2. NS development and rejuvenation.

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 [11]).

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 3 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)

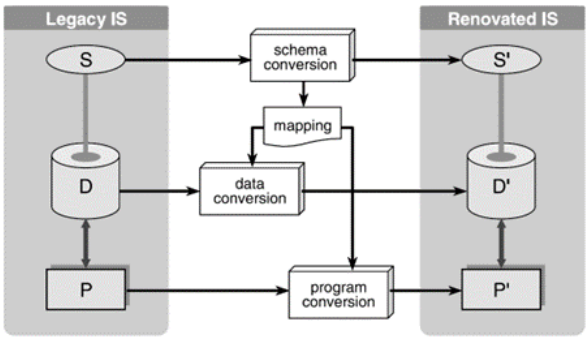

Fig. 3. Conceptual schema conversion strategy (from [11]).

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 [11], a legacy information system is defined as any 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 on which they depend, and they 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 it and move it to a new platform.

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.
- Composite database: migrate data and functionality together and go live when both are migrated.
- Chicken Little strategy: like a composite database but keep 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 [12] 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 [13] 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 PHP web server 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 critical 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 4.

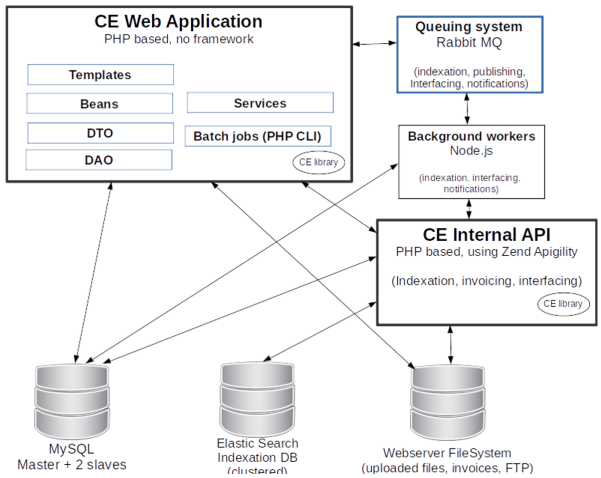

Fig. 4. CE2 VMS technical architecture.

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 [14] 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 many 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 problematic 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 correctly. 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. These are all technical obstacles that needed to be replaced by other technologies to enable the scaling of the platform, i.e., to connect ever more job-seekers and job-suppliers by simply adding resources.

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.

V. Migration Approach

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? The following sections will explain the new setup, how NS expansion was introduced, the differences between CE2 and CE3, and how transformers, carefully designed gateways, deal with the migration from CE2 to CE3. Note that the meaning of transformers in this paper is unrelated to the notion of transformers in today's popular Large Language Models (LLMs) [16] and Generative AI.

A. The New setup

In 2021, a new system, CE3 VMS, was proposed. 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 exposing 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 following 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.

B. 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 [15]. 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 the stakeholders did a qualitative comparison. The result of this comparison (see Figure 5) was that an expansion-based method using the NSX tools, was preferred. It should be noted that this was a qualitative comparison that needs to be verified again once implementation starts and finishes (see Section VII).

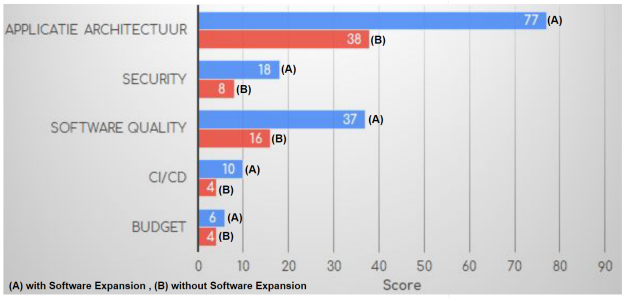

Fig. 5. Implementing CE2 with or without Software Expansion.

C. 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 to use the corresponding CE2 VMS functionalities. Data already in CE2 VMS is accessed/stored via APIs on CE3 VMS. Only when new functionalities on CE3 VMS introduce new data types will the data 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 CE3 VMS, called a CE3 data element. Another is for data in CE2 VMS that CE3 VMS exposes through a CE3/CE2 data

element. According to NS principles, the CE3/CE2 data elements transform the less anthropomorphic CE2 data elements into a data structure. The CE2 data element will aggregate a certain amount of CE3/CE2 data elements. Figure 6 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.

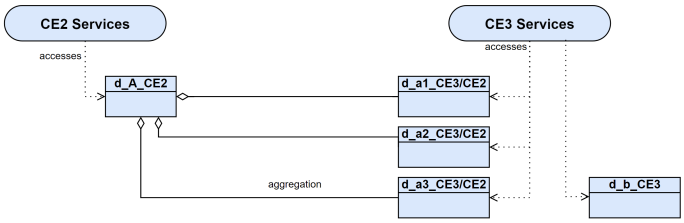

Fig. 6. Transformation of data objects between CE2 and CE3.

D. The Transformer Cross-Cutting Concern

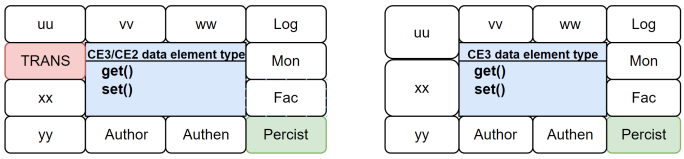

Fig. 7. Transformer as a Cross-Cutting Concern of the CE3/CE2 data element type.

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 and CE2 data models to be unambiguously mappable. This was ensured during the design of the CE3 data model. Figure 7 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 get the CE2 data through the CE3/CE2 data element and save it into a CE3 data element. After this migration task, the feature that needs this data will only use the native CE3 data element, smoothly transitioning from one system to the other. Figure 8 explains the process.

E. 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 then be applied, and the transformers will no longer be 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.

VI. Inside the Transformers

This section will take a closer look at the transformers, including their coding. We start by explaining how transformers are activated where required, followed by the main classes that make up the transformers. We continue by listing where transformations are required and end by describing the main types of transformers.

A. Activating Transformations

In Section V-C, we have explained the difference in the data model used in CE3 and CE2 and that transformers translate one model to another. When a DataElement is created in CE3 with a homologue in CE2, linking a transformation to the CE3 DataElement is a matter of indicating in the model of the DataElement that you want to have transformations included. Example: In CE2, the notion of "Bids" exists. As there are some anthropomorphic issues with this literal, the decision was taken to stop using "Bids" and replace it by "Proposal." The DataElement for "Proposal" in CE3 gets flagged with the need for a transformation, and the fully qualified name of the corresponding CE2 data element (com. connecting expertise.ce2.Bids) is given as a parameter. The same is done for each attribute associated with CD3's "Proposal": indicate to what field in the CE2's "Bids" the transformation must happen.

By default, one-to-one mapping/transformation is included such that the name of an attribute of CE3's "proposal" to a different attribute name but with the same content type in CE2's "Bid" For example "proposal_nr" to "bids_nr".

If an attribute requires a more custom transformation, the attribute of the DataElement must be flagged with the hasCustomTransformation option. The expanded code for the transformer will then include the boilerplate code to call the

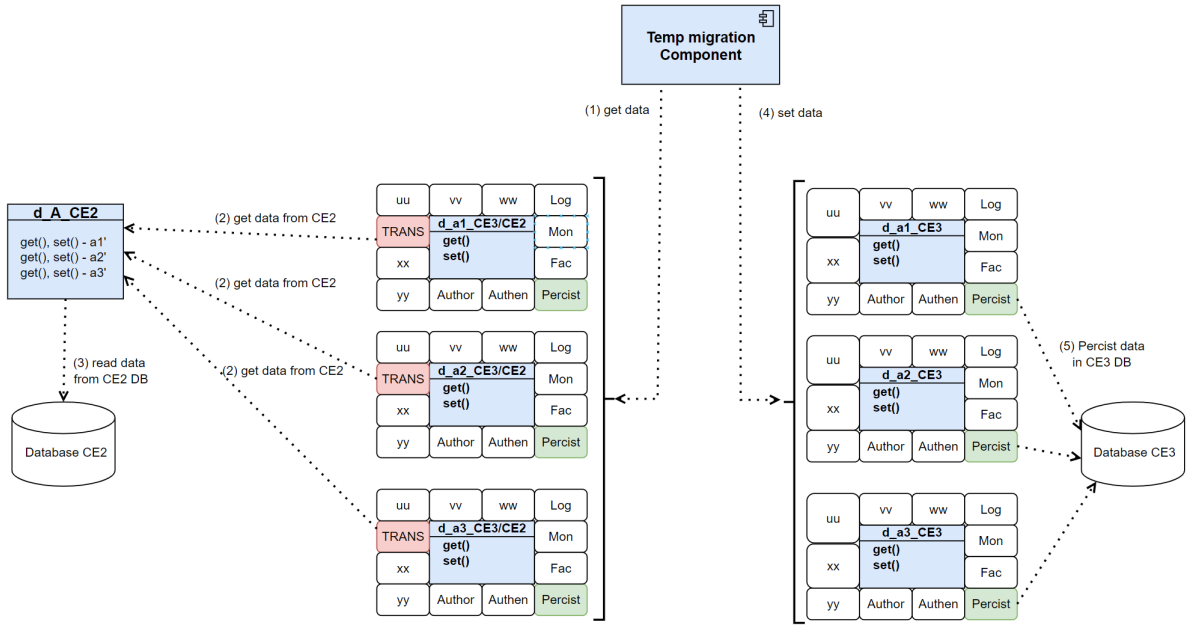

Fig. 8. Migration of data from CE2 VMS to CE3 VMS.

transformation and provide anchor points (locations to add custom code in the expanded code) to code the transformation manually.

Custom transformations are required when no pattern is found between the mapping of CE3 to CE2,and thus, they cannot be generalized in expandable code. However, sometimes a pattern can be found, and then it is interesting to generalize this pattern in an expansion template and provide it as a configuration option of the CE3 DataElement.

For example, language translation is a transformation that is potentially applicable to all DataElements and has a general pattern. The indication of a need for language transformation of the values of attributes of a CE3 DataElement will expand the required code for performing this task.

B. Expaned Transformation Code

Three transformation Java classes will be created based on the configuration of the DataElements that require transformation.

- **<DataElement>TransformationCoordinator.java**: Transformation may require a custom sequence of operations, e.g., fetch a different DataElement, do a transformation operation on it, and then only perform a transformation on the actual textless DataElement>. The transformer coordinator class holds this logic.
- **To<DataElement>Transformer.java**: This transformer class will take CE2 data as input and provide CE3 data as output.
- **From<DataElement>Transformer.java**: This transformer class takes CE3 data as input and provide CE2 data as output

C. Usage of Transformations

The method invocations to transformation classes discussed in the previous section get injected into the DataElementBean, the heart of the logic layer. There, operations are called, which allow the manipulation of the DataElement in CE3 but require transformations for proper reflection into CE2.

A CE3 DataElement contains methods for performing basic operations on the data. When transformers are attached to a CE3 DataElement, these basic operations need to call the transformers to access the corresponding CE2 DataElement. Examples of operations requiring transformer integration

- *Projections*: Representation of the DataElement in an NS application, with “Info” (the most essential attributes), “Details”, and “DataRef” (data reference) as 3 possible sub-projections.
- *SearchResult*: When data gets searched the requested projections get wrapped in a result class.
- *CrudsResult*: When a CRUD operation is invoked, the end product gets wrapped in a result class.
- *Diagnostics*: When something went wrong in the data layer, this needs to be transformed to a correct error in the logic layer.
- *Finder*: Information to search information, the result of this is a SearchResult.
- *QueryFilter*: A different representation to search information, the result of this is a SearchResult.
- *SortField*: column names on which can be sorted, these can vary in name. If additional methods should be added to DataElements, it suffices to do so on the DataElement template, including the option to link a transformer to them. A rejuvenation cycle will then include these extra methods and corresponding transactions in the entire code

base, thus eliminating the introduction of a combinatorial effect.

D. Types of Transformations

Depending on what a CE3 DataElement represents, different transformations will be required to link with the corresponding CE2 DataElement(s). We will now elaborate the main types of transformation.

1) Table renames: A Data Element in CE3 with a corresponding Data Element in CE2 that only differs in name requires a simple table rename transformation. For example: “bids” and “proposal” are names of tables corresponding with the ”bids” DataElement in CE2 the ”proposal” DataElement in CE3. Both represent the same thing; they are just named differently. Having activated a transformation at DataElement level between the two, results in expanded code that allows the conversion of one into the other. In Listing 1, we see the code that takes care of the conversion of ProposalDetails (CE3) toward BidsDetails (CE2) and the vise-versa.

```
//From
...
public static com.connectingexpertise.ce2.BidsDetails
    ↪ transformDetails(ProposalDetails details) {...}
...
//To
...
public static ProposalDetails transformDetails(
    ↪ ParameterContext<com.connectingexpertise.ce2.
    ↪ BidsDetails> detailsParameter) {...}
...
```

Listing 1. Table renames or mapping between two different DataElements of CE2 and CE3.

2) Column renames: The CE3 DataElement for ”proposal” has an attribute ”request”, while the corresponding CE2 DataElement ”bids” has a similar attribute called ”rfq”. Mapping from ”request” to ”rfq” and vise versa required a mapping of columns. Both represent the same attribute; they are just named differently. Having activated a transformation at the attribute level results in expanded code that allows one conversion into the other. In Listing 2, we first see getting the attribute ”request” defined in CE3’s Proposal and setting that value into the attribute ”Rfq” of CE2’s Bids, and vice versa.

```
//From
...
transformedDetails.setRfq(FromRequestTransformer.
    ↪ transformDataRef(details.getRequest()));
...
//To
...
transformedDetails.setRequest(ToRequestTransformer.
    ↪ transformDataRef(details.getRfq()));
...
```

Listing 2. Column mapping between two different CE2 and CE3 DataElements.

Note the **From<item>Transformer**, taking CE3 data as input and given CE2 data as output, and the **To<item>Transformer**, taking CE2 data as input and giving CE3 data as output.

3) Cardinality mapping: CE2 supports three languages. All translations are spread throughout the codebase. Adding a fourth language would require a full revision of the codebase, effectively introducing a combinatorial effect. In CE3, the issue can be solved by adding an attribute with a language code postfix (e.g., _NL, _FR, _EN) to a one-to-many table containing translations. Upon insertion, the transformer will create the necessary detail records with the default translation. During update, it will map the language attribute to the corresponding translation row. As long as the transformer is in place, CE3 will be limited to using only the CE2 languages. Once the transformer is gone (functionality fully migrated from CE2 to CE3), many languages can be added without introducing a combinatorial effect. Listing 3 shows some code related to translations. Note that the original DataElement data will need to fetched and mapped in order not to lose this information when making changes to DataElementTranslation.

```
//From
...
public class FromCountryTranslationTransformer {
  public static com.connectingexpertise.ce2.CountryDetails
      ↪ transformDetails(CountryTranslationDetails details
      ↪ ) {
    com.connectingexpertise.ce2.CountryDetails
        ↪ transformedDetails = new com.connectingexpertise
        ↪ .ce2.CountryDetails();
    transformedDetails.setId(details.getId());

    CrudsResult<com.connectingexpertise.ce2.CountryDetails>
        ↪  originalDetailsCrudsResult = com.
        ↪ connectingexpertise.ce2.CountryLocalAgent.
        ↪ getCountryAgent(Context.emptyContext()).
        ↪ getDetails(details.getId());
    if (originalDetailsCrudsResult.isError()) return null;
    com.connectingexpertise.ce2.CountryDetails
        ↪ originalDetails = originalDetailsCrudsResult.
        ↪ getValue();

    // anchor:value-fields-transformDetails:start
    if (details.getLanguage().getName().equals("nl")) {
        transformedDetails.setCountryNl(details.getName());
        transformedDetails.setCountryFr(originalDetails.
            ↪ getCountryFr());
        transformedDetails.setCountryEn(originalDetails.
            ↪ getCountryEn());
    } else if (details.getLanguage().getName().equals("fr")
        ↪ ) {
        transformedDetails.setCountryNl(originalDetails.
            ↪ getCountryNl());
        transformedDetails.setCountryFr(details.getName());
        transformedDetails.setCountryEn(originalDetails.
            ↪ getCountryEn());
    } else {
        transformedDetails.setCountryNl(originalDetails.
            ↪ getCountryNl());
        transformedDetails.setCountryFr(originalDetails.
            ↪ getCountryFr());
        transformedDetails.setCountryEn(details.getName());
    }
    // anchor:value-fields-transformDetails:end

    1 // anchor:parent-fields-transformDetails:start
    transformedDetails.setUuid(originalDetails.getUuid());
    transformedDetails.setIsoCode(originalDetails.
        ↪ getIsoCode());
    transformedDetails.setPostcodelist(originalDetails.
        ↪ getPostcodelist());
    transformedDetails.setCompanyNumberRequired(
        ↪ originalDetails.getCompanyNumberRequired());
    transformedDetails.setRequiresVatNumber(originalDetails
        ↪ .getRequiresVatNumber());
```

```
    transformedDetails.setSelfregistration(originalDetails.
        ↪ getSelfregistration());
    transformedDetails.setSelfregistration(originalDetails.
        ↪ getSelfregistration());
    transformedDetails.setSortOrder(originalDetails.
        ↪ getSortOrder());
    transformedDetails.setUseRrnr(originalDetails.
        ↪ getUseRrnr());
    // anchor:parent-fields-transformDetails:end

    // anchor:custom-other-fields-transformDetails:start
    // anchor:custom-other-fields-transformDetails:end

    return transformedDetails;
  }
}
...
//To
...
public class ToCountryTranslationTransformer {
  public static CountryTranslationDetails transformDetails(
      ↪ ParameterContext<com.connectingexpertise.ce2.
      ↪ CountryDetails> detailsParameter, String language)
      ↪ {
    Context context = detailsParameter.getContext();
    com.connectingexpertise.ce2.CountryDetails details =
        ↪ detailsParameter.getValue();
    CountryTranslationDetails transformedDetails = new
        ↪ CountryTranslationDetails();
    transformedDetails.setId(details.getId());

    // anchor:value-fields-transformDetails:start
    if (language.equals("nl")) {
        transformedDetails.setName(details.getCountryNl());
    } else if (language.equals("fr")) {
        transformedDetails.setName(details.getCountryFr());
    } else {
        transformedDetails.setName(details.getCountryEn());
    }
    // anchor:value-fields-transformDetails:end

    CrudsResult<DataRef> resolvedLanguageDataRef =
        ↪ LanguageLocalAgent.getLanguageAgent(context).
        ↪ resolveDataRef(DataRef.withName(language));
    if(resolvedLanguageDataRef.isError()) {
        transformedDetails.setLanguage(null);
    } else {
        transformedDetails.setLanguage(
            ↪ resolvedLanguageDataRef.getValue());
    }

    transformedDetails.setCountry(ToCountryTransformer.
        ↪ transformDataRef(details.getDataRef()));

    // anchor:custom-other-fields-transformDetails:start
    // anchor:custom-other-fields-transformDetails:end

    return transformedDetails;
  }
}
...
```

Listing 3. Handling translation in the transformer.

4) Deprecated attributes: Some CE3 DataElements contain attributes that are considered to be deprecating the new model. However, as long as the functionality is not fully migrated from CE2 to CE3, the data is still stored in CE2. Consequently, these attributes still need to be completed in CE2 when editing an entry. Otherwise, they will be nulled, which impacts the workings of CE2. Creating DataElements containing depreciated attributes always have a default stored in the CE2 database. This is achieved by retrieving the current CE2 element and filling the deprecated values with the existing ones. Listing 4 shows some sample code.

5) Data value mappings: The typing of attributes in CE2 has not been consistent; e.g., instead of consistently storing a boolean as TRUE/FALSE, a boolean is sometimes stored as strings with ”y” and ”n” as values. In the new CE3 data model, this is now harmonized, and the transformation will map to the corresponding type in CE2. A similar transformation can be found in the code below (Listing 5).

```
...
// fill in original CE2 values
if (details.getId() != null && details.getId() != 0L) {
  CrudsResult<com.connectingexpertise.ce2.BidSkillDetails>
      ↪ oldDetailsResult = com.connectingexpertise.ce2.
      ↪ BidSkillLocalAgent.getBidSkillAgent(Context.
      ↪ emptyContext()).getDetails(details.getId());
  if (oldDetailsResult.isSuccess()) {
        BidSkillDetails oldDetails = oldDetailsResult.
            ↪ getValue();
        transformedDetails.setFreeSkillName(oldDetails.
            ↪ getFreeSkillName());
        transformedDetails.setType(oldDetails.getType());
        transformedDetails.setQuestionType(oldDetails.
            ↪ getQuestionType());
        transformedDetails.setDescription(oldDetails.
            ↪ getDescription());
        transformedDetails.setJustificationMissingSkill(
            ↪ oldDetails.getJustificationMissingSkill());
        transformedDetails.setSkill(oldDetails.getSkill());
        transformedDetails.setCustomerScore(oldDetails.
            ↪ getCustomerScore());
  }
}
...
```

Listing 4. Filling in deprecated attributes in CE2 for database consistency.

```
...
1 transformedDetails.setOverBudget(details.
    ↪ getHasBudgetExceeded() != null ? (details.
    ↪ getHasBudgetExceeded() ? "Y" : "N") : null);
4 transformedDetails.setRfq(FromRequestTransformer.
    ↪ transformDataRef(details.getRequest()));
5 transformedDetails.setUuid(details.getUuid());
7 if (DataRefValidation.isDataRefDefined(details.
    ↪ getExternalAnonymizationStatus())) {
     transformedDetails.setExternalAnonymizationStatus(
         ↪ details.getExternalAnonymizationStatus().
         ↪ getName());
   }
...
// anchor:custom-other-fields-transformDetails:start
2 transformedDetails.setLonglist(details.getIsLonglisted()
    ↪ ? 1 : 0);
3 transformedDetails.setVat(details.getIsVatExcluded() !=
    ↪ null ? (details.getIsVatExcluded() ? "EXCLUDED" : "
    ↪ INCLUDED") : null);
6 transformedDetails.setState(transformState(details.
    ↪ getExternalStatus().getName()));
// anchor:custom-other-fields-transformDetails:end
...
6 public static String transformState(String state) {
        String transformedState;
        switch (state) {
          case "PUBLISHED":
                transformedState = "SUBMITTED";
                break;
          case "PENDING_PUBLICATION":
                transformedState = "PENDING_SUBMIT";
                break;
          case "CREATED":
                transformedState = "INITIATED";
                break;
```

```
            default:
                  transformedState = state;
                  break;
            }
            return transformedState;
}
...
```

Listing 5. Filling in deprecated attributes in CE2 for database consistency.

6) Complex and large data: In some cases, the data transformations are more extreme. CE2 contains files, often stored as php serialized objects in the database. Migration of all this data in a big-bang operation can be time-consuming and risky. For this reason, a particular transformation was implemented to migrate such complex objects from CE2 to CE3 when touched. This spreads the migration over a longer time. The inconvenience of touching such an object the first time will result in some extra delay due to the on-the-fly transformation toward CE3, is preferred over the risk of a big-bang migration. Practically, in CE3, an extra Asset Data Element is created. This Data Element contains the actual file. This asset is created the first time an entry in CE2 of such a file is read, and it does not exist yet in CE3. If it already exists in CE3, the file is searched and linked. This kind of transformation will often reside between custom anchors because of the different file types existing in CE2 with other structures.

VII. Discussion

In this section, we will discuss different aspects of the migration approach. We will start with the choice of NS expansion, followed by a comparison between this migration approach and a generic migration approach called Chicken Little [12], and a short discussion on the value of a phased migration. We will end by giving some basic numbers.

A. The Choice for NS Expansion

In Section V-B, 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 from using NS Expansion 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 enhanced, 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 active is called the Chicken Little approach (see [12]). 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 8) is thanks to 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.

D. Some Basic Numbers

The system currently contains 546 CE2 data elements, corresponding to database tables, and 416 CE3 data elements with 120 CE task elements and 48 CE3 workflows. The development team consisted of 1 to 3 back-end developers, with the lower number at the beginning and the end, and the higher number in the middle. At the end of 2022, the creation of a dedicated front-end, on top of the generated user interface, was initiated. This effort also involved 2 to 3 developers, where a decrease in back-end developers made room for an increase in front-end development. As is often the case in software development, the front-end development turned out to be less predictable than the back-end development. Due to the specific nature of front-end development, and the many stakeholders involved, it should probably be treated as a separate project.

VIII. Conclusion

This paper presented a real-life case where NS Expansion facilitates software migration. We introduced NS and NS Expansion and gave 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 significant drawbacks of a phased migration.

ACKNOWLEDGMENT

The authors thank Sven Beterams from Connecting-Expertise for sharing his application knowledge and Jan Hardy of NSX for explaining how the transformers are implemented. We would also like to thank Chetak Kandaswamy for collecting and structuring the material required to create this paper.

REFERENCES

[1] C. De Clercq and J. Verelst, "Using Normalized Systems Expansion to Facilitate Software Migration-a Use Case," The Sixteenth International Conference on Pervasive Patterns and Applications (PATTERNS 2024), pp. 6–12, April 2024.

[2] SAFe Framework, [Online], Available: www.scaledagileframework.com, [retrieved: December, 2024].

[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.

[4] 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.

[5] H. Mannaert, P. De Bruyn, and J. Verelst, "On the interconnection of crosscutting concerns within hierarchical modular architectures," IEEE Transactions on Engineering Management, 2020.

[6] H. Mannaert, K. De Cock, and P. Uhnak, "On the realization of metacircular code generation: The case of the normalized systems expanders," In Proceedings of the Fourteenth International Conference on Software Engineering Advances (ICSEA) (Vol. 2019, pp. 171-176).

[7] P.Huysmans, J. Verelst, H. Mannaert, and A. Oost, "Integrating information systems using normalized systems theory: four case studies," In IEEE 17th Conference on Business Informatics, Volume 1, pp. 173-180, 2015.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] Connecting-Expertise, [Online], Available : www.connecting-expertise.com, [retrieved: December, 2024].

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

[15] NSX, [Online], Available: www.normalizedsystems.org, [retrieved: December, 2024].

[16] A. Vaswani, N. Shazeer, N. Parman, et al., "Attention is all you need," 31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA 2017.