Building Evolvable Software Using Normalized Systems Theory: A Case Study

Gilles Oorts
University of Antwerp
Normalized Systems Insitute
gilles.oorts@uantwerp.be

Philip Huysmans
University of Antwerp
Normalized Systems Insitute
philip.huysmans@uantwerp.be

Peter De Bruyn
University of Antwerp
Normalized Systems Insitute
peter.debruyn@uantwerp.be

Herwig Mannaert
University of Antwerp
Normalized Systems Insitute
herwig.mannaert@uantwerp.be

Jan Verelst
University of Antwerp
Normalized Systems Insitute
jan.verelst@uantwerp.be

Arco Oost
Normalized Systems eXpanders factory
arco.oost@nsx.normalizedsystems.org

Abstract

Normalized Systems (NS) theory has recently been proposed as an approach to develop agile and evolvable software by defining theorems and design patterns for software architectures. In this paper we discuss the NS development process, which is illustrated by means of an elaborate description of a case regarding a budget management application developed according to the theory. Advantages of the NS approach, such as swift application development through code expansion and the transfer of additional NS design knowledge to new applications, are equally discussed.

1. Introduction

Over the last decade, an ever-increasing amount of research conducted has been conducted on agile software development [3]. Although the progress made in this research domain has proven to be very valuable in improving agile development processes (e.g. [9], [4]), less attention has been paid to making the software itself more agile. In this paper, we describe a project in which the focus is on the evolvability of the software architecture itself. If an organization is to be competitive in current volatile and competitive economic conditions, it needs to be agile across its organizational structure, enterprise architecture and information systems [1]. Therefore it is important for organizations to focus on implementing software that supports changes in the organization, as this can be considered as an important step or precondition in establishing an agile organization.

Recently Normalized Systems (NS) theory has been proposed as a theory for making software more agile [13]. Here the ability for software to be easily changed is called software evolvability. This evolvability can be achieved by adhering to a limited set of theorems that result in a very specific and evolvable software architecture. The NS theory has been extended for several years now, up to a point that it has become fully theoretically founded [8] and implemented in several software projects. Although the theoretical contributions of NS have been widely documented in previous research (e.g., [11,12,13]), few reports are available on real-life cases in which NS was employed. Nevertheless, NS offers advantages in both theory and practice. In this paper, we document such a development project to (1) show the feasibility of the NS approach for building evolvable software in practice and (2) to highlight the benefits of a real-life NS development project.

As the case description requires an understanding of the NS theory, its foundations are discussed in Section 2. The practical implications of these foundations will be explained in Section 3, by describing the NS conforming development of a budget management application for a local Belgian government. In Section 4, we will discuss one specific advantage of NS development, which is the inclusion of NS knowledge into new applications. In the next Section we discuss some observations, contributions and future research. We end the paper with a conclusion in Section 6.

978-1-4799-2504-9/14 $31.00 © 2014 IEEE
DOI 10.1109/HICSS.2014.585

2. Normalized Systems

The Normalized Systems theory postulates that software architectures should exhibit *evolvability* due to ever changing business requirements [8,11,12,13]. In the theory, evolvability is operationalized by the absence of *combinatorial effects*. Such an effect is defined as a change of which the impact is not solely related to the kind of the change, but also to the size of the system it is applied on. As the NS theory assumes that over time software is subject to an unlimited evolution (i.e., both additional and changing requirements), combinatorial effects have a highly undesirable effect on software evolvability. Indeed, if changes to a system depend on the size of the ever-growing system, these changes become ever more difficult to cope with (i.e., requiring more effort) and hence reduce the evolvability of the system.

The theoretical foundation of NS reasoning is the concept of systems stability from systems theory [11], which states that a bounded input (i.e., changing requirements) should result in a bounded output (i.e., changes in the software). Additionally, significant progress has recently been made in establishing the theoretical concept of entropy as a second foundation for the NS theory [10].

Normalized Systems theory proposes a set of four theorems and five expandable elements that constitute the foundation for developing evolvable software through pattern expansion of the elements. The theorems are formally proven principles (cf. [12]) which offer a set of necessary conditions that should be strictly adhered to, in order to avoid combinatorial effects. The NS theorems have been implemented in NS elements. These elements provide a set of predefined higher-level structures, patterns or "building blocks" offering an unambiguous blueprint for the implementation of the core functionalities of realistic information systems, adhering to the four stated theorems [13].

2.1 Theorems

NS theory proposes four theorems, which have been proven to lead to combinatorial effects if not adhered to [11]:

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

These theorems are not new in themselves but relate to well-known and often tacit design heuristics of software developers, as mentioned explicitly in [12]. For example, well-known concepts such as an integration bus, a separated external workflow or the use of multiple tiers can all be seen as manifestations of the Separation of Concerns theorem [12]. The value of the four NS theorems can however be found in the fact that they (1) make certain aspects of that heuristic design knowledge explicit, (2) offer this knowledge in an unambiguous way (i.e., violations against the theorems can be proven), (3) are unified based on one single postulate (i.e., the need for evolvable software architectures having no combinatorial effects) and (4) have all been proven in a formal way in [11].

2.2 Normalized Systems Elements

Consistently adhering to the four NS theorems is very challenging for developers due to two reasons. First each violation of the NS theorems during any stage of the development process results in a combinatorial effect. Secondly, the systematic application of these theorems results in very fine-grained structures. Therefore five expandable elements were proposed which make the realization of NS applications more feasible. These elements are encapsulated high-level patterns that comply with the four NS theorems:

- *data element*, being the structured composition of software constructs to encapsulate a data construct into an isolated module (including get- and set-methods, persistency, exhibiting version transparency,...);
- *action elements*, being the structured composition of software constructs to encapsulate an action construct into an isolated module;
- *workflow element*, being the structured composition of software constructs describing the sequence in which a set of action elements should be performed in order to fulfill a flow into an isolated module;
- *connector element*, being the structured composition of software constructs into an isolated module allowing external systems to interact with

the NS system without calling components in a stateless way;

- *trigger element*, being the structured composition of software constructs into an isolated module which controls the states of the system and checks whether any action element should be triggered accordingly.

More extensive descriptions of these elements are for example available in [11,12,13]. Each of the five elements discussed are in fact design patterns, as they represent a recurring set of constructs encapsulated in the element. Each element contains the intended core construct and a set of relevant cross-cutting concerns (such as remote access, logging, access control, etc.). This construction entails that the elements facilitate a set of anticipated changes that ensure the elements are evolvable, as more thoroughly described in [12]. As discussed in [2], the definition and identification of the NS elements is based on the implications of the set of NS theorems. As an example we can quote how the theorems Separation of Concerns (SoC) and Separation of States (SoS) indicate the need to formulate a workflow element. Such a workflow element allows the stateful invocation of action elements in a (workflow) construct. Indeed the SoS theorem requires this kind of stateful invocation and the SoC theorem demands that the concern of invocation is handled by a separate construct.

The implementation of a data element in a Java Enterprise Edition (JEE) implementation (a widely used platform for the development of distributed systems [14]) has also been described in previous work. In [12] it is discussed how a data element Obj is associated with a bean class ObjBean, interfaces ObjLocal and ObjRemote, home interfaces ObjHomeLocal and ObjHomeRemote, transport classes ObjDetails and ObjInfo, deployment descriptors and EJB-QL for finder methods. Additionally, methods to manipulate a data element's bean class (create, delete, etc.) and to retrieve the two serializable transport classes are incorporated. Finally, an agent class ObjAgen provides the remote access. Combined, these elements provide the main concerns and cross-cutting concerns of the data element instance. Similarly, the functionality of other NS element instances is provided by about 10 classes per instance. Comparing this to for example the observer design pattern defined by [5], the complex architecture of NS conforming applications becomes clear. Whereas the observer pattern of Gamma requires two classes and two interfaces, the NS implementations requires seven NS elements and thus about 70 classes. Consequently, it is clear that in order to prevent combinatorial effects, a very fine-grained modular structure needs to be adhered to. How the complexity of the large amount of classes is coped with will be discussed in the next section.

Moreover, the complete set of elements covers the core functionality of an information system. Consequently, as such detailed description is provided for each of the five elements, an NS application can be considered as an aggregation of a set of instantiations of the NS elements. This is shown in Figure 1. The top level of this figure shows the five NS elements. Based on these elements, the functional analyst will formulate instantiations that are the foundations of a NS application. Figure 1 shows how the application discussed in this paper includes amongst others Budget, Budget change and Product instances of the NS data element. At run time, these instances are instantiated once more (i.e., form a double instantiation) to form specific occurrences of, for example, a budget.

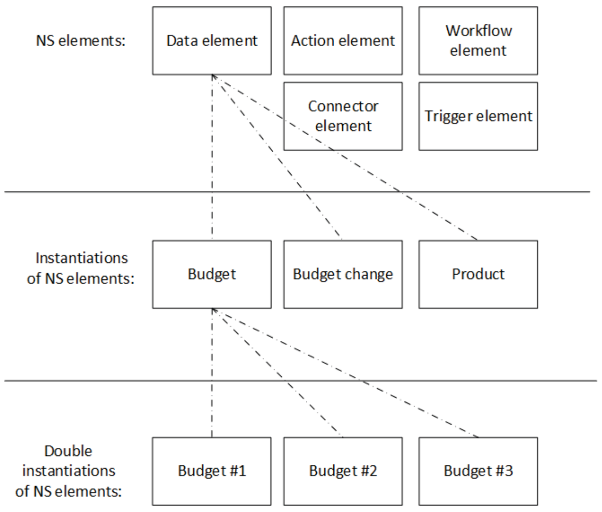


Figure 1. Principle of double instantiation in Normalized Systems

2.3 Pattern expansion

In practice it seems very unlikely to arrive at the very fine-grained modular structure implied by the NS theorems without the use of higher-level primitives or patterns. The process of defining these patterns and transforming them into code is shown in Figure 2, which will be discussed in Section 3. As NS proposes a set of five elements that serve as patterns, this figure shows how the actual software architecture of NS conforming software applications can be generated in a relatively straightforward way by the use of NS expansion. This expansion mechanism is an essential part of making the NS theory applicable in practice.

3. The NS Budget Application Case

Because of the fundamental new insights the discussed Normalized Systems theory offers in the development of evolvable software, there was a need for a new software development process that supports the NS theorems. In this section we will explain and discuss this development process by using a completed real-life case as an exemplar.

Over the last few years, several evolvable software applications have been built according to the NS theory. Although these applications have been quoted as examples of the feasibility [12] and theoretical soundness [8] of NS theory, none of them have been extensively elaborated on in academic literature. In this paper we will therefore discuss one of the finished NS projects at length. As this is the first time we describe a NS case in this way, we have specifically chosen a project with limited complexity. This allows us to fully explain the application at hand while also explaining the development process and discussing some interesting observations.

The case we have chosen is the development of an application to manage the budgets of a local Belgian government. The administration of this government does intensive tracking of its budgets. The overall available budget is divided into very fine-grained sub-budgets, complicating the budget assignment, reservation, fixations, changes, etc. This was traditionally done using the flexibility offered by pivot tables within Microsoft Excel. These tables allowed for the selection of subsets of a specific budget and for quick calculation of the available budget for a department, activity, etc. In an effort to enable the integration of these budget management functionalities with project management, budget reporting and budget simulations functionalities, a project was initiated to capture the budget management functionalities in a stand-alone application.

However, the development of this application presented some challenges. The first one was that the new application needed to satisfy the flexibility and versatility the users got used to in managing the budgets in Microsoft Excel pivot tables. To cope with this challenge, it was decided to focus the initial application solely on budget management and its user-friendliness. This application would then be a sound basis for further extending the application to include the other requirements of budget reporting, simulation and project management. These incremental expansions of the application will be supported by the fact that NS applications can be changed and expanded without the needed effort increasing due to the size of the application.

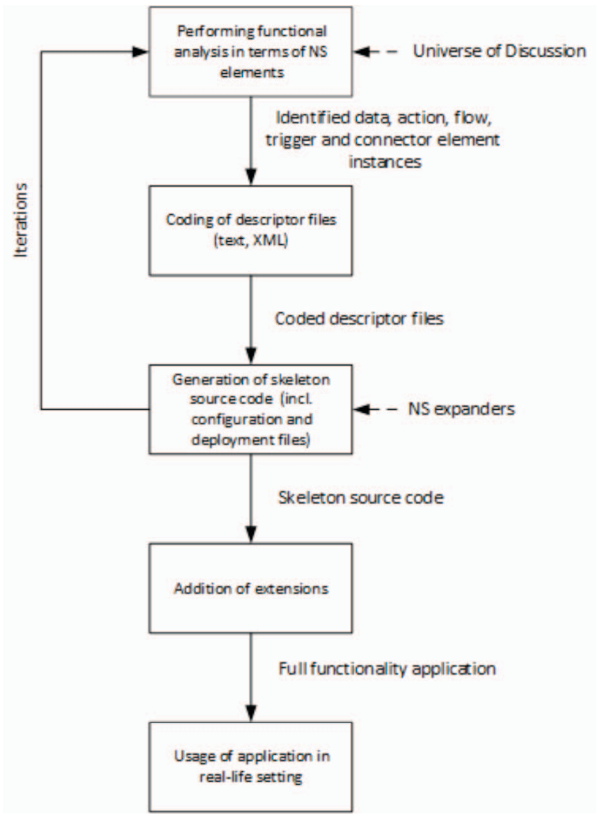
Another challenge was that the budget management tool is very context-specific. Budgets are defined at different levels of the specific government. Therefore budgets can be managed on both very general and very fine-grained levels, but the application needs to include the composition of budgets at all levels. More specifically, budgets are defined by a combination of the following six parameters: department, activity, article, domain, product and budget year. The unique combination of these six parameters is the key of a budget in its most specific manifestation. However, budgets also need to be consulted as combinations of these parameters. For instance, the aggregated budget of a specific department or the combined budget of a product in a specific department also need to be retrieved. This specific composition of budgets could not be realized in common ERP-systems and therefore a custom application had to be built.

These challenges were however all successfully coped with in the development process, and this in no small part due to the NS development process. Five sequential steps, which are shown in Figure 2, characterize this development process. These steps will be discussed in-depth in the following sections, together with their interpretation in the budget application development process.

3.1. Functional analysis

As in most software projects, the first stage of the budget application development process is the functional analysis. Similar to other development methodologies (such as the object-oriented approach), this analysis is advised to be done in terms of the constructs defined by the approach itself. In NS development, this means that real-world requirements in any form (e.g., use cases, natural language description, domain class diagrams, Business Process Modeling Notation (BPMN)-diagrams, etc.) are translated into instantiations of the five NS elements discussed earlier.

For the budget application, a manageable set of requirements was extracted together with end users. System analysis happened in two sessions in which an Entity Relationship Diagram (ERD) and a table with data elements were drawn up. The ERD diagram that resulted from these sessions is shown in Figure 3. This figure shows the identified NS element instantiations of the application. As the application is very data-intensive, the application could be built only using NS data element instantiations.



Performing functional analysis in terms of NS elements
Universe of Discussion
Identified data, action, flow, trigger and connector element instances
Iterations
Coding of descriptor files (text, XML)
Coded descriptor files
Generation of skeleton source code (incl. configuration and deployment files)
NS expanders
Skeleton source code
Addition of extensions
Full functionality application
Usage of application in real-life setting

Figure 2. The NS development process (adapted from [2])

The functional requirements can be easily explained using the ERD. As visually represented in the ERD, the *Budget* is the central data element instance of the application. The current budget is defined by the aggregation of changes to that budget over time. The consultation of the current budget is therefore done in real-time, meaning that the application calculates the current budget based on all previous *Budget changes*. This is done for data integrity reasons, as one single error in the calculation can lead to erroneous data stored in a database. By calculating all current budgets in real-time, no budgets are saved to a database and errors cannot get stored permanently. The *Department*, *Activity*, *Article*, *Domain*, *Product* and *Budget year* instances on the left of the figure are used to define the most granular budgets. A combination of these instances can be used as a key for defining a budget, as a specific budget belongs to a single department, activity, etc. Furthermore the application allows for the grouping of articles in *Economic groups*, which in turn make up a *Budget estimate*. This estimate is used to draw up a target budget at the beginning of a budget year. The management of budgets is controlled by the data element instances on the right side of the ERD. Fragments of a budget can be reserved (i.e., a *Budget fixation*) for a specific cause and the fixations are allocated to a specific supplier. Over time, these fixations can be called in *Budget calls*, so the budgets can be partially spent when needed. For these budget calls, *Invoices* and *Work orders* need to be made so the calls can be successfully supported with the necessary paperwork.

3.2. Descriptor files

Once the requirements have been formulated as NS element instances, the instantiations need to be coded. This is done in descriptor files, which are text- or XML-based files describing the inputs for the expanders. For example, in case of a data element instance, the pattern expansion mechanism would need a set of parameters including the basic name of the data element instance (e.g., Budget), context information (e.g., component and package name), data field information (e.g., data type) and its relationships with other element instances. This shows that with a minimum of input, descriptor files can be used to expand code into large applications. Through the process of expansion, this minimum of information can be transformed in a full application, as discussed in the next section. For the budget application, all 15 element instances were defined in descriptor files.

3.3. Code expansion

In the next phase, the descriptor files get expanded into the code of a functional application. This is done by software (called NS expanders) developed especially for this purpose by the Normalized Systems eXpanders factory (NSX). The NS expanders expand the descriptor files into skeleton source code for all the identified instantiations, together with all deployment and configuration files required to construct a working application on one of several supported technology stacks. The classes of the skeleton code represent the modular structure of the defined NS elements. Moreover, the required boilerplate code is included as well. For the budget example, this would be the set of classes and data fields: the bean class BudgetBean, interfaces BudgetLocal and BudgetRemote, etc.. Because the code expansion process is typically very fast, the NS development process allows for iterative and interactive sessions with end users. In these sessions, changes to functional requirements and data models can immediately be made in the descriptor files, followed by a re-expansion into a new version of the application.

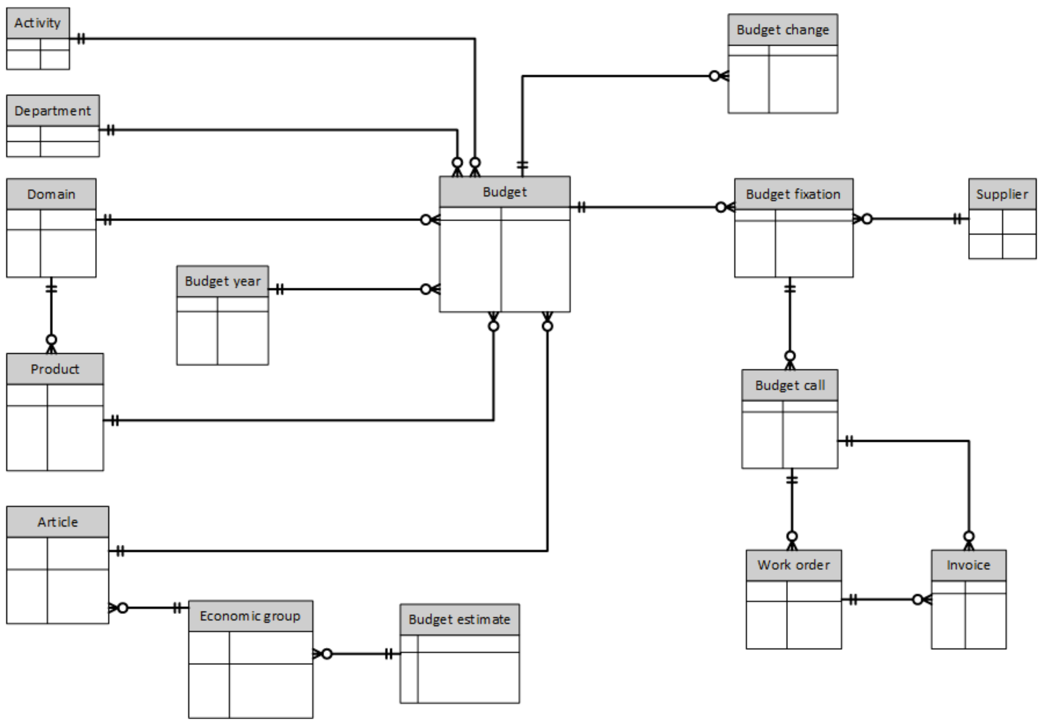


Figure 3. Entity Relationship Diagram of Budget application

Therefore, the correctness of requirements, data model and descriptor files can be validated within a single or very few sessions. This way, the first three steps on the NS development process -analysis, creation of descriptor files and expansion- are in fact an iterative loop that is repeated as long as needed. Because the descriptor files can be easily changed and re-expanded, this loop can be gone through very fast, leading to short development cycles. Once the end user expectations have been fully verified by the iterative development cycles, the basic functionalities of the application are fixed, which significantly reduces the risk of scope creep in the remainder of the project.

For the budget application, the expanded code base consisted of 379 Java files and 586 Strut files. The fact that these files are all part of the 15 NS elements defined in the functional analysis, shows how the meticulous adherence to the theorem of Separation of Concerns impacts the granularity of modules in the codebase.

3.4. Extensions

Although the process of expansion delivers a fully working application that includes all defined NS element instances, the functionalities of the application most likely still need to be extended in the fourth phase of the development cycle. This is because the NS expanders are carefully designed to only expand code that fully complies to the NS theory. However, not all code can already be expanded in this way, implying that two types of requirements may still need to be added to the code of an expanded application: (1) requirements that are very specific to the application and (2) generic requirements that have not yet been “Normalized” and therefore not have been included in the NS expanders. The first type of requirements will always have to be implemented by developers, as it concerns extensions for specific customer requests. These context-specific extensions are kept out of the NS expanders, as these should only contain (general) architectural deductions from the four NS theorems. The latter manual additions are due to the fact that extending the NS expanders to include new features in a normalized way is a difficult process: any addition to the expanders needs to be in full accordance with the NS theory. Therefore, not every feature of NS applications can yet be expanded and manually coded extensions are needed to enhance the functionalities of the expanded application. When building applications, the developers however constantly look for possibilities to include extended features in the NS elements, as has been done with some features of the budget application.

Adding extensions to the expanded code needs to happen in a controlled way, as experience shows that combinatorial effects can be injected in software when they are not included in the right way. This has however been resolved by only allowing extensions to be added in two controlled ways: (1) by adding it in a separate class or (2) by adding extensions within pre-specified anchors in the expanded code. An automated harvesting mechanism then allows for the extensions to be extracted from the expanded code and stored separately. After a re-expansion, the extensions can then be re-injected in the expanded code without any impact on the process of expansion. When a new version of the expanders is built (for example with new frameworks in the web tier or in the persistence tier, or with minor upgrades), the application is re-generated by first expanding the skeleton code and then injecting the extensions. This re-expansion process is highly automated and can be performed quickly, but can still result in conflicts between the extensions and the skeleton code. That is one reason why NS applications should regularly be regenerated; another reason is that with this minimal regeneration effort, all new features in the expander code (for example, new user interface widgets, value types or validation rules) are made available in all regenerated applications.

As the functional analysis and code expansion can be done fast, most of the effort in building the budget application was invested in programming the extensions. Of the total development time of 30 man-days, about 90% of the effort was spent on developing the extensions. Of these 27 man-days of development time, approximately 60% of the effort was spent on actually incorporating the extensions and the other 40% was spent on incorporating the extensions of the budget case into the NS elements in a way that the same extensions can be expanded in a fully evolvable way in future NS applications.

For the budget application, two types of extensions were needed to satisfy the user's requirements: logic extensions and graphical extensions. The logic extensions included operations that are not included in the NS expanders because of their context-specific nature, such as the on-the-fly calculation of the current budget based on all previous budget changes, validation of uniqueness of budgets, validation of budgets calls not exceeding available budget, etc. These extensions only account for about 30% of the effort spent on extensions. The second type of extensions was responsible for much larger development efforts, amounting to 70% of the extension development time and therefore about 60% of the total development time of the application. The high costs of the graphical extensions were caused by the impossibility of expanding advanced graphical screens at the time of the start of the development of the budget case. As the application contains data regarding budgets at several different levels, the end users required different overviews of budgets (e.g., by department, activity, etc.) which were not included in the standard screens the NS. Therefore, a great deal of extensions was needed to provide this advanced screen functionality. In this way, these advanced screens incorporate a similar functionality as the Excel pivot tables that were used before. These advanced screens also allow the presentation of several NS data element instances within the same screen, so-called "composite screens".

Although developing these advanced screens was a time-intensive task, we need to stress that the effort to produce an important part of these composite screens can be re-used in future applications as well. Therefore approximately 40% of development time spent on the advanced screens is estimated to be in other applications. Additionally, while the initial attempts required 600 lines of code to correctly show a composite screen, it only takes approximately 60 lines of code in newer applications.

3.5. Launch and Use of application

Once the extensions have been added to the expanded application, an NS application needs to go through testing, verification and data input phases before it can be deployed. Because of the rapid expansions of applications, issues that are otherwise proportionally irrelevant, become some of the biggest issues during an NS project. This is the case since the resolution time of these issues cannot be shortened, even though the overall development time of NS applications is drastically shortened. Some of these issues are for example technical challenges such as data conversion and input. In the budget case, data from the replaced application and spreadsheets was fragmented and in different data standards. Therefore, data conversion and input were labor-intensive and provided one of the biggest challenges of the project. To handle these issues, import mechanisms have been developed to import existing data in new NS applications. This import is managed through either manual input screens or automatically generated import clients. The chosen method depends on the amount of instances that need to be imported. When there are a lot of instances to be imported, this is done by automated import clients. Sometimes these import clients need to be extended depending on the (type of) data to be imported, which means this will only be done if manual input is too cumbersome because of a large amount of instances.

The 500 existing budgets of the previous budget application were imported using clients. First, 500 instantiations of the budget element were created and their value was set by an initial budget change for every budget. Therefore this import process also included a verification whether each budget was unique (i.e., each budget should have its own unique key) and whether data for every empty budget field were present.

4. Transferring knowledge to new applications

An important aspect of the development of NS applications is that new insights obtained from practical application of the theory are constantly being added to the NS knowledge base. These insights range from newly normalized features in the NS elements that can be re-used in future applications to new general reflections on building Normalized software. The knowledge management processes that support capturing, storing, transferring and applying these knowledge have been discussed in previous work [2].

During the development of the budget application, several of such new insights have been gained which, since then, have been applied to other applications. Although there plenty examples of new knowledge and additions amassed from the budget case, we will discuss three of the most important types in the next paragraphs.

Through the repeated application of the NS theorems, software modules become more and more granular. This is mainly because of the Separation of Concerns theorem that requires concerns to be separated. For software to become truly evolvable, this separation should be applied very thoroughly. For the graphical screens, this for example means that an empty page is generated and all graphical items such as tables, figures, buttons, etc. are included as separate elements. This far-reaching separation of elements allows these elements to be changed without affecting other parts of the page (i.e., being evolvable) and it allows for the re-use of these elements (e.g., a table) in other pages. Another example of concerns that have been separated in the budget application is the presentation of a clickable button and the logic that determines what needs to happen when the button is clicked. Clicking the "view" button, for example, needs to trigger a selection model to determine which type of presentation will be used to present the requested data. Normally these selections are hard-coded in JavaScript for each button, meaning the selection model for the type of graphical representation for several pages cannot be changed in one single location in the code. However, by separating these concerns it becomes possible to change both concerns (i.e., the button click and graphic selection model) independently and at their own single location. But decoupling the code according to the NS theorems requires insight and well thought-out planning. For example, where does the graphic selection model belong: it is not related to the included table or the pagination, so where does it belong?

The second addition to the NS knowledge base gained from the development of the budget application is the advanced GUI screens (i.e., composite screens). Although these cannot yet be directly expanded (extensions are still necessary), the goal is to make this possible one day. However, thanks to the efforts made during the budget application development, some components of the composite GUI screens will be readily available to be (re-)used. To make complete GUI screens available out-of-the-box, some complexities needed to be overcome. According to the developers, the complexity resides in the fact that end users have different perspectives on a specific data element. Consider for example the data element "Contract". Some users are interested in the legal aspects of a contract, while others are interested in the financial aspects. These different interpretations of the same element should be translated in the presentation of the data element instances to the users, so that different data related to the same data element can be shown to users (i.e. different views/projections). For example, some possible views for the budget application are (1) which budget is available, (2) which budget is billed and (3) which budget is fixed. These distinctions are however not yet fully understood and implemented. But once they are, the more complex GUI screens can also be included in the expansion mechanism.

A third addition to the NS knowledge base was the use of an improved extension harvesting mechanism. As discussed earlier, extensions can be added either in separate classes or between pre-specified anchors in the implementation code. An automated harvesting mechanism can then harvest all extensions and re-inject them after the application has been expanded again. This process makes the extensions and expanded code independent. This mechanism has been developed during the budget case project. Before that, extensions were added by replacing existing implementation classes. This however led to duplicated code when these extensions needed to be applied in several double element instances. Additionally, the code extensions were not clearly identifiable which resulted in a loss of code (or manual retrieval) when a re-expansion was performed. Therefore the new harvesting mechanism was devised and has been used in the development of all NS applications since the budget application.

5. Discussion

Some interesting *observations* can be made from the description of the case in the previous section.

First, we can notice that -although both their discussion in this paper as the period of time spent on them in the project are rather lengthy- the NS extensions are anything but limitations of NS development projects. As the NS approach allows for a very fast way of developing the basic application (i.e., by descriptor files and expansion), the time spent on developing extensions becomes the only significant component of the complete development process, as shown by the 90% effort spent on the extensions (i.e., 27 man-days). Because of the expansion mechanism, time spent on programming skeleton code and "boilerplate code" is minimized. The overall effect of the expansion on the total development time is therefore positive, as the extensions would take up the same time when building an application according to a non-NS approach. This is shown by the equal development times (of 16 man-days) for application-specific requirements in both development processes in Figure 4. Furthermore, an audit shows that only 5% of the total code of the budget application was "touched" after expansion. That is, only 5% of the total application code is made up of extensions. However, the actual implementation of these extensions accounted for 54% (or 16 man-days) of the total development effort of the application (i.e. 90% of total development time was spent on extensions, of which only 60% was used for actually implementing them). As manually programming extensions is far more labor and cost-intensive than using automated code expansion, we can state that the NS development provides a great advantage over traditional development of software. This advantage is also shown in Figure 4. The time needed to program the skeleton code (including boilerplate code) is significantly reduced from approximately 30 man-days to only 3 man-days due to the expansion of NS elements. To make this reduction possible, the NS development process however requires an investment of development time in order to incorporate NS extensions of the application into the NS elements. For the budget management application 11 man-days needed to be invested in this phase. This additional effort is however needed to further extend the advantages of NS expansion so these extensions (e.g. advanced GUI items, logic extensions) can be rapidly expanded into future applications without the need for manual programming (as discussed in Section 4).

Second, we already mentioned that the functional analysis in the NS development process is done in terms of the constructs of the approach (i.e., instances

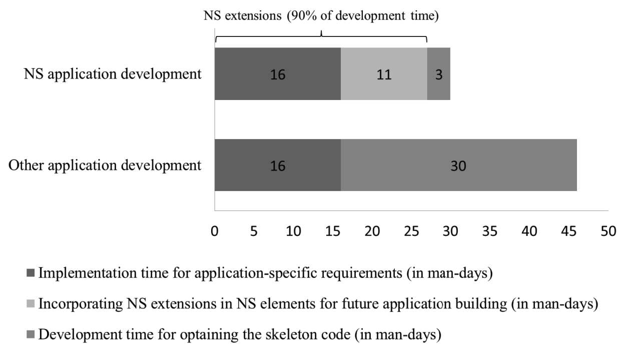


Figure 4. Comparison of estimated development times

of the 5 NS elements). We believe the NS development approach can actually completely fulfill this promise, which could be criticized for Object-Oriented (OO) analysis and design methods. In NS, contrary to "objects", the elements can truly be considered anthropomorphic. This is because the NS elements also include cross-cutting concerns such as persistency, security, etc. Therefore one can portray a complete application by only needing to describe its NS element instances. For the NS budget application presented in Figure 3, this means the presented ERD shows all aspects of the application, as all cross-cutting concerns are automatically included in the shown element instances. In contrast, a description of the anthropomorphic objects in object-oriented programming does not suffice for such a complete application description. Indeed, additional classes to provide persistency, security etc. need to be added manually to the model later on, thereby weakening the anthropomorphic character of the implemented class diagram.

A final observation that can be made from the case is that the development of evolvable software does not need to entail exorbitant costs. Although building evolvable software is shown to be very complex, the NS expansion mechanism and the transfer of knowledge to new applications help to streamline this apparent impossible development. Moreover, the evolvable structure of an application will result in far lower adaptation and integration costs during later phases of the life cycle of the IT application. This is because new functional requirements can be implemented without the effort needed for a specific change to the system growing over time (i.e., when the application becomes larger and more complex). Presuming volatile environments that require regular changes to the organization and its information systems, this leads to an overall lower Total Cost of Ownership of the IT application.

This paper also has a number of *contributions*. First, the description of the development of an NS application shows the practical feasibility of the NS design theory [8]. In doing so, it addresses the requisite relevance cycle in design science research [7] and fulfills the "expository instantiation" component of the design theory anatomy defined by Gregor and Jones [6]. Therefore this paper has a theoretical contribution as well, as we once more demonstrated the completeness of the NS theory as a design science theory. Third, the case description shows the complexity involved with developing evolvable software. Considering the described application is rather small, the issues and difficulties cited in this paper show that developing evolvable software is surprisingly challenging. This paper however demonstrates how these challenges can be overcome thanks to the NS theorems, NS development process, pattern expansion, etc.

Several possibilities for *future research* can also be defined. As discussed in this paper, the included budget application is rather small and only includes data element instances. Therefore an evident first extension would be the discussion of more complex cases performed according to the NS theory and its development process. In such research, one could study whether additional challenges and problems arise because of the larger or more complex nature of the developed applications. A second possible route for future research is to do a quantitative study comparing the lead time and Total Cost of Ownership of NS applications to non NS-compliant software. However, it should be noted that such a study is very challenging and resource-intensive.

6. Conclusion

In this paper we discussed how the NS theory can be applied to develop evolvable software. This was shown by means of the extensive description of a budget management application developed according to the NS theory and its development process.

7. Acknowledgment

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

8. References

[1] Bieberstein, N., Bose, S., Walker, L., and Lynch, A. Impact of service-oriented architecture on enterprise systems, organizational structures, and individuals. *IBM systems journal 44*, 4 (2005), 691–708.

[2] De Bruyn, P., Huysmans, P., Oorts, G., et al. Incorporating Design Knowledge into Software development using Normalized Systems. *International Journal On Advances in Software 6*, 1 (2013).

[3] Dyba, T. and Dingsøyr, T. Empirical studies of agile software development: A systematic review. *Information and Software Technology 50*, 9-10 (2008), 833–859.

[4] Fitzgerald, B., Hartnett, G., and Conboy, K. Customising agile methods to software practices at Intel Shannon. *European Journal of Information Systems 15*, 2 (2006), 200–213.

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

[6] Gregor, S. and Jones, D. The anatomy of a design theory. *Journal of the Association for Information systems 8*, 5 (2007), 312–335.

[7] Hevner, A.R., March, S.T., Park, J., and Ram, S. Design Science in Information Systems Research. *MIS Quarterly 28*, 1 (2004), 75–105.

[8] Huysmans, P., Oorts, G., De Bruyn, P., Mannaert, H., and Verelst, J. Positioning the normalized systems theory in a design theory framework. *Lecture Notes in Business Information Processing 142*, (2013), 43–63.

[9] Layman, L., Williams, L., and Cunningham, L. Motivations and measurements in an agile case study. *Journal of Systems Architecture 52*, 1 (2006), 654–667.

[10] Mannaert, H., De Bruyn, P., and Verelst, J. Exploring entropy in software systems : towards a precise definition and design rules. (2012), 93–99.

[11] Mannaert, H., Verelst, J., and Ven, K. The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability. *Science of Computer Programming 76*, 12 (2011), 1210–1222.

[12] Mannaert, H., Verelst, J., and Ven, K. Towards evolvable software architectures based on systems theoretic stability. *Software: Practice and Experience 42*, (2011), 89–116.

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

[14] Oracle. Java platform, enterprise edition. http://www.oracle. com/technetwork/java/javaee/overview/index.html.