Toward a Rejuvenation Factory for Software Landscapes

Herwig Mannaert

Normalized Systems Institute
University of Antwerp, Belgium
Email: herwig.mannaert@uantwerp.be

Tim Van Waes and Frédéric Hannes

Research and Development
NSX bv, Belgium
Email: tim.van.waes@nsx.normalizedsystems.org

***Abstract*—The agile paradigm has become the default methodology for the delivery of software-based products. While there is a widespread belief that this methodology has numerous benefits, including improved and timely delivery of software projects, it can be argued that the lack of an overall architecture to which developers must adhere can result in increased technical debt. Through its normative structure of software application skeletons, NST (Normalized Systems Theory) provides a possible mechanism to manage the delicate balance between intentional architecture and emerging design. Moreover, the systematic rejuvenation of application skeletons, featuring harvesting and re-injection of custom code, enables to accommodate not only changes in the functional model, but also in the software skeletons, including the technology frameworks that are used. In this contribution, we describe the setup and operations of an NST rejuvenation factory, where dozens of software applications are being developed using agile methodologies, and rejuvenated on an approximately weekly basis. Both the size of the application models, codebase, and technologies, and their evolution in time, are presented. The achieved levels of agility, and the realized abilities to change are discussed, as well as the current limitations and some future work to address them.**

***Index Terms*—Software Evolvability; Software Factories; Normalized Systems Theory; Case Study.**

I. INTRODUCTION

The *agile paradigm* has become the default methodology for software development. While there is a widespread belief that this methodology has numerous benefits, including improved and timely delivery of software projects, it can be argued that the lack of an overall architecture may result in increased technical debt and reduced evolvability. Normalized Systems Theory (NST) aims to provide higher levels of evolvability through its normative structure of software application skeletons. This underlying architecture could serve as a mechanism to manage the delicate balance between evolvable architecture and agile design. In this paper, we conduct a case study to investigate this potential by studying the evolvability behavior of a software factory that operates in an agile way, while adhering to the NST architecture to realize evolvability.

The remainder of this paper is structured as follows. In Section II, we briefly discuss some related work and the methodology. In Section III, we describe the issues related to software evolvability, and the way NST aims to provide higher levels of evolvability. We present the structure, operations, and possibilities of a software factory based on NST in Section IV. In Section V, we present the case study analyzing the realized software evolvability in a specific NST software factory. Finally, we discuss some conclusions and future work in Section VI.

II. RELATED WORK AND METHODOLOGY

In this paper, we investigate whether NST is able to realize the substantial improvement in evolvability that it proposes, by studying its application at scale in a state-of-the-art software factory. Section III gives an overview of related work on the deep issues regarding software maintenance and evolution, and on the way that NST aims to address some of these issues in a structured way. In Section IV, we go through some related work on current state-of-the-art software factories.

The methodology of this paper is based on *Design Science Research* [1]. The artifact that we consider is the NST methodology aimed at the development of software applications that exhibit higher levels of evolvability. We conduct an observational case study to investigate whether the application of this methodology at scale in a software factory is able to realize the envisaged evolvability. While the operations of the NST software factory contribute to the relevance cycle by applying NST to the appropriate environment, this case study aims to contribute to the rigor cycle by extending the knowledge base.

III. THE PREMISE OF NORMALIZED SYSTEMS THEORY

In this section, we introduce NST as a theoretical basis to obtain higher levels of evolvability in information systems, and the approach to realize its promise through a code generation or *expansion* framework.

A. On Software Maintenance and Evolvability

Software maintenance is not merely about fixing defects. While originally three categories of maintenance were defined, i.e., *corrective*, *adaptive*, and *perfective* maintenance [2], modern standards also include *preventive* maintenance. Studies have indicated that about eighty percent of maintenance effort is used for non-corrective actions and functionality enhancements [3] [4]. This means that software maintenance is intimately related to software evolution, even though users often perpetuate its reduction to bug fixing by submitting enhancements as problem reports.

Copyright (c) IARIA, 2024. ISBN: 978-1-68558-161-9

Software evolution and evolvability were studied in depth by Manny Lehman over a long period of time, leading to the insight that maintenance is really an evolutionary development, and to the formulation of Lehman's Laws. One of these laws, the *Law of increasing complexity* [5], states that systems, as they evolve, grow more complex and more difficult to maintain, unless some action such as code refactoring is taken to reduce the complexity. Though never formally proven, this empirical law is widely accepted by software developers.

While the evolvability of information systems (IS) is considered as an important attribute determining the agility and therefore the survival chances of organizations, it has traditionally not received much attention within the IS research area [6]. More recently, software maintenance and evolution have attracted more attention through the introduction of concepts like *technical debt*, representing the need for refactoring to reduce structure degradation, and *maintenance debt*, corresponding to maintenance needs generated by dependencies on external IT factors such as libraries, platforms and tools, that have become obsolescent [7].

B. Normalized Systems Software Applications

Normalized Systems Theory (NST) was developed by applying the concept of *stability* from systems theory to the evolution of engineering artifacts such as software systems. It operationalizes the concept of systems theoretic stability, i.e., a bounded input should result in a bounded output, in the context of information systems development and maintenance, by demanding that a bounded set of changes should only result in a bounded impact to the software, or, that the impact of changes to an information system should not be dependent on the size of the system to which they are applied, but only on the size of the changes to be performed [8] [9]. Changes causing an impact dependent on the size of the system are called *combinatorial effects*. Being a major factor limiting the evolvability of information systems, these combinatorial effects are considered to be one of the mechanisms causing the structure degradation described by Manny Lehman.

The theory derives four theorems and formally proves that any violation of these *theorems* will result in combinatorial effects, thereby hampering evolvability [8] [9] [10]:

- *Separation of Concerns*: no two concerns or change drivers should be combined in a software construct.
- *Action Version Transparency*: invoking new versions of processing functions should not demand changes.
- *Data Version Transparency*: exchanging new versions of data objects should not demand changes.
- *Separation of States*: no two processing functions should be sequenced without keeping state.

The application of these theorems to software applications results in very fine-grained modular structures within these applications. The theory also proposes a set of design patterns, and presents a constructive proof that these patterns are free of combinatorial effects with respect to a number of basic changes. Specifically, NST proposes five *elements*, i.e., detailed design patterns, and argues that instantiations of these elements are sufficient to build the main functionality of information systems [9] [10] [11]:

- *data elements* to store and retrieve data entities.
- *action elements* to perform operations on data entities.
- *workflow elements* to orchestrate the operations on data.
- *connector elements* to interface with users and systems.
- *trigger elements* to drive and activate operations.

Implementing and enforcing detailed design patterns of fine-grained modular structures is, in general, difficult to achieve by manual programming. Therefore, an implementation of modular code generators, called *expanders*, was made to generate information systems based on NST. The development of such a *Normalized Systems (NS)* information system starts by defining a set of data, task and workflow elements. Based on the detailed design patterns, the expanders generate source code for the various elements that are defined. The code generation mechanism, called *expansion*, is quite straightforward, i.e., simply instantiating parametrized copies of a set of coding templates. The generated code makes up the evolvable skeleton of the information system. It is in general complemented with custom code or *craftings* to add non-standard functionality not provided by the skeletons. These craftings may reside in separate classes, or placed at well specified places identified by anchors within the generated boiler plate code.

C. Variability Dimensions and Evolvability

Information Systems generated by an NS expansion process consist of application skeletons that are free of combinatorial effects with respect to a set of basic changes [9]. This entails a number of evolvability characteristics, essentially based on the separation of four variability dimensions as schematically visualized in Figure 1. While we have discussed elsewhere [12] [13] in more detail how such an application with separate variability dimensions can evolve throughout time, we briefly describe here these dimensions.

First, as represented at the upper left side of the figure, the skeletons are based on the *models* or *mirrors* of the required information system such as data models and workflows. As the model can have multiple versions throughout time (e.g., being updated or complemented), it constitutes a first dimension of variability or evolvability.

Second, the *expanders* (represented by the big blue icon in the figure) generate application *skeletons* by instantiating the various class templates, taking the specifications of the model as *parameters*. As these expanders, or rather template skeletons, can have multiple versions throughout time (e.g., solving bugs or offering additional features), they represent a second dimension of variability or evolvability.

Third, as represented in the upper right side of the figure, the skeletons use a number of frameworks or *utilities* to take care of several so-called *cross-cutting concerns*. As these frameworks and the generated adapter code, specified through

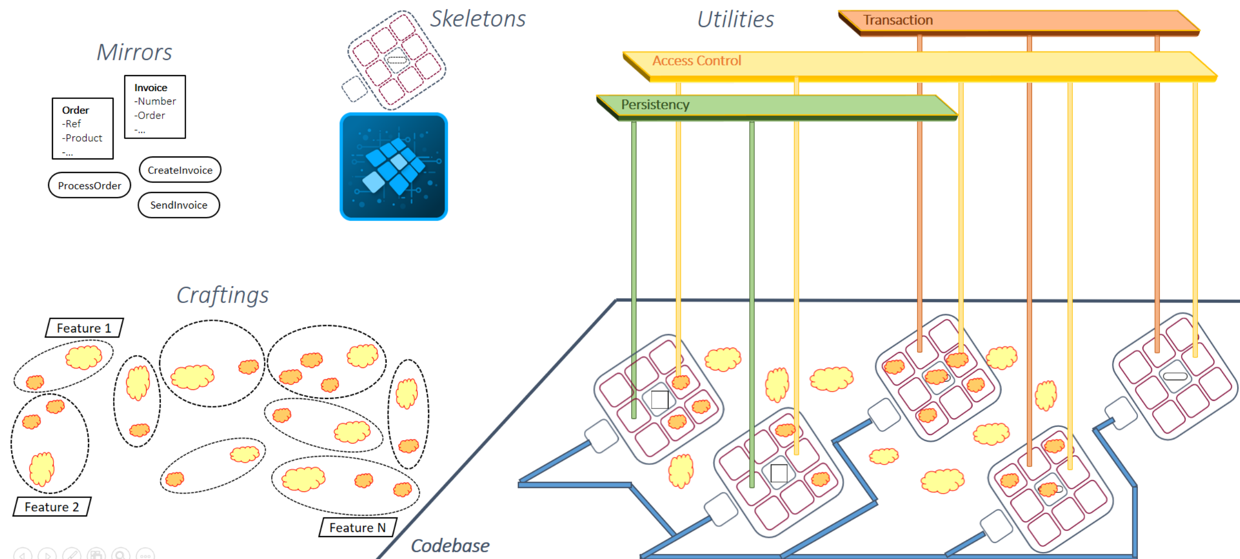

Figure 1. A graphical representation of four variability dimensions within a Normalized Systems application codebase.

infrastructure settings, can have multiple versions throughout time (e.g., new versions of existing frameworks or alternative frameworks), these settings or frameworks represent a third dimension of variability or evolvability.

Fourth, as represented in the lower left of the figure, *custom code* or *craftings* are used to enrich the generated skeletons. These craftings are harvested into a separate repository to enable their re-injection into a newly generated application skeleton. As these craftings can have multiple versions throughout time (e.g., improvements or additional features), they represent a fourth dimension of variability or evolvability.

To summarize, NS software applications as represented in Figure 1, exhibit four different and independent variability dimensions. This means that the concept of *the "version"* of an NS application is more refined, as the version of an application codebase corresponds to a specific combination of four different versions representing the four variability dimensions [13]. Given certain constraints, e.g., certain versions of the expanders do not (yet) support certain versions of the frameworks, the versions of the different dimensions are independent and can be used in various combinations. Conceptually, with M, E, I and C referring to the number of available model versions, the number of expander versions, the number of infrastructure settings, and crafting versions respectively, the total set of possible versions V of a particular NST application could become equal to:

$$V = M \times E \times I \times C$$

This is an example of a quite fundamental principle stating that the thorough decoupling of concerns can realize exponential gains in their recombination potential, leading to higher levels of evolvability and variability [9].

IV. A Normalized Systems Software Factory

In this section, we describe how expansion and rejuvenation are integrated into the Normalized Systems software factory, and discuss the different rejuvenation modes.

A. Integrating Expansion in a Software Factory

The production and/or assembly of software in a more industrial way has been pursued for many decades. It dates back at least to 1968 with the work of Doug McIlroy on *mass produced software components* [14], and is currently associated with terms like *Software Product Lines (SPLs)* and *Software Factories*. The term *Software Factory* is for instance defined by Greenfield et al. as a software product line that configures extensive tools, processes, and content using a template based on a schema to automate the development and maintenance of variants of an archetypical product by adapting, assembling, and configuring framework-based components [15]. However, the systematic reuse of software artifacts is not a trivial task facing may different issues, as was for instance recently argued by Saeed [16].

These issues become even more challenging when integrating a code generation environment into such a software factory. Many existing code generation technologies, identified with terms like *Model-Driven Engineering (MDE)*, *Model-Driven Architecture (MDA)*, *Low-Code Development Platforms (LCDP)*, amd *No-Code Development Platforms (NCDP)*, enable programmers to create software applications by interactively defining domain models that drive code generation. However, in general these technologies do not support the harvesting of custom code and their re-injection into newer regenerated versions of the software. A software factory based on NST on the other hand, has to support this harvesting and re-injection of custom code in order to enable the proper separation of the various dimensions of variability.

Copyright (c) IARIA, 2024. ISBN: 978-1-68558-161-9

B. From CI/CD Toward Continuous Rejuvenation

The current mainstream approach to organize and control the operations of so-called software factories is a methodology called *DevOps* to integrate and automate the work of software development *(Dev)* and IT operations *(Ops)*. As stated by Ravi Yarlagadda, *Through DevOps, there is an assumption that all functions can be carried out, controlled, and managed in a central place using a simple code* [17]. In accordance with the main purpose of such a DevOps environment, it is often called a *Continuous Integration, Continuous Delivery (CI/CD)*, or *Continuous Integration, Continuous Deployment* infrastructure. The various tools used in such an infrastructure, being both commercial and open source, are in general quite numerous and versatile. While the technical community often focusses on these tools, it needs to be stressed that DevOps is essentially a methodology striving to improve the collaboration and integration between development and operations teams.

In an NS software factory, the CI/CD infrastructure needs to contain an *expansion* cycle before the *build* phase. The control structure of such an NS CI/CD infrastructure is schematically represented in Figure 2, and described in more detail in [18]. Of course, the modular code generators or expanders are being built, integrated and tested themselves in a CI/CD infrastructure. As the CI/CD pipelines of expanders and information systems are integrated, rejuvenation, i.e., application skeletons that are regenerated with new versions of expander templates, becomes part of the CI/CD infrastructure. In that sense, we obtain a *Continuous Integration, Continuous Deployment, Continuous Rejuvenation (CI/CD/CR)* infrastructure.

C. Normalized Systems Rejuvenation Modes

Having an infrastructure that includes rejuvenation of the application skeletons, we are now able to distinguish different modes of structural rejuvenation. Conceptually, this corresponds to evolutions and improvements in the variability dimensions of expander templates and external frameworks, while allowing respectively modelers and programmers to further improve and extend the model and the custom code.

First, various external frameworks can be upgraded to new versions. This includes both minor version upgrades, or even patches to address vulnerabilities, and more major version upgrades. While this kind of 'rejuvenation' is also available and even standard practice in traditional applications, an NS approach aims at making this more straightforward by embedding the code to interface with these frameworks in the expanded skeletons. In this way, the expanded boiler plate code should cope with changes in the interfaces of the frameworks. Recently, solutions like *OpenRewrite* [19] have become available to enable traditional applications to deal in a more productive way with such interface changes.

Second, new versions of expanders and the corresponding templates can be used in the expand phase. This includes possible bug fixes, minor improvements in functionality or coding style, and new features that may have become available.

TABLE I. DOMAIN, LIFESPAN, MODEL AND CUSTOM CODE SIZE OF VARIOUS APPLICATIONS.

Application Domain	**Age** *(yrs)*	**Data Model** *(Nr. elem.)*	**Custom Code** *(Size kBytes)*
Energy Monitoring	> 10	116	6,352
	$3-5$	38	1,010
Power Grid Management	$1-3$	106	10,642
Human Resource Services	$3-5$	940	12,103
	$3-5$	59	1,433
Real Estate Services	> 10	491	70,449
	$1-3$	331	1,412
Unmanned Aviation	$5-10$	30	4,230
Traffic Management	$1-3$	134	2,896
Learning Management	$1-3$	133	1,794

This kind of rejuvenation, enabling a structural regeneration and modernization of application skeletons, is not available in a traditional development approach.

Third, the support of new infrastructure settings with corresponding templates to interface with these technologies, can conceptually enable the seamless migration of applications, or even entire application landscapes, to new and/or alternative frameworks. Indeed, as the code to interface with such new technologies should in general be embedded in the generated skeletons, both application skeletons and custom code should almost automatically support existing functionality through the new framework.

V. The Case of an NST Rejuvenation Factory

Since the publishing of NST, two development centers have been building and rejuvenating NS applications, one at the spin-off company to further develop NST, i.e., *NSX bv*, and one at the Dutch Tax Office. In this section, we study the developments and rejuvenations at the NSX development center after 12 years of existence. The development center operates in a realistic business environment, producing and maintaining operational applications for clients. During these years, the number of staff members, working on code generation tools and applications, increased almost linearly from 2 to 50.

Table I presents some overview data of some of the most prominent NS software applications that have been developed, and that are still being maintained and evolved at this point in time. While the functional domain of the application is identified in a first column, the second column lists the age in years, i.e., the number of years since the development of the software application started. To reflect the size of the model, we present the current total number of data elements, corresponding roughly to the number of database tables, in the third column. The total size of the craftings or custom code (in *kBytes*) is listed in the fourth column.

As stated in Section I, the main goal of the agile architecture is to rejuvenate the core structures of the various software applications, in a way that is independent and decoupled from

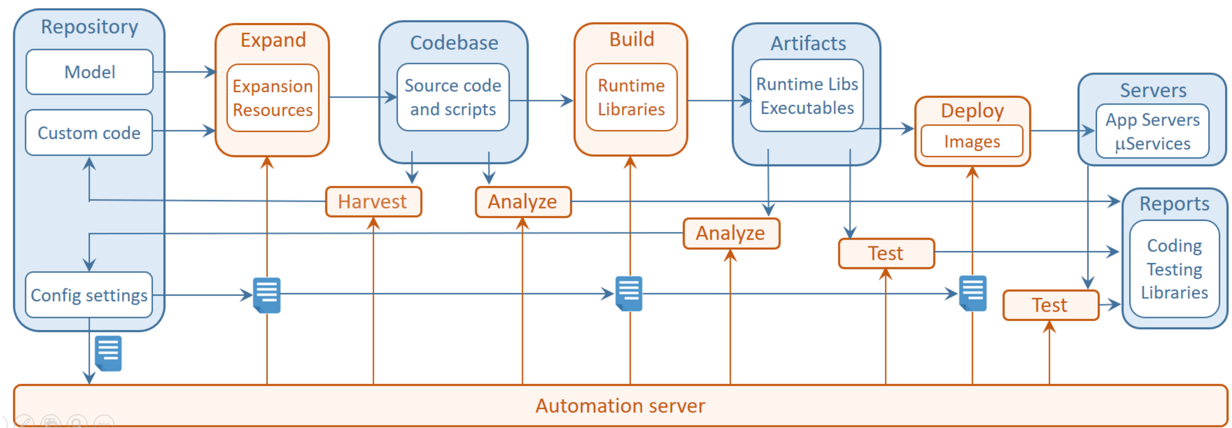

Figure 2. A traditional representation of a typical DevOps infrastructure.

the continuous evolution of the underlying model and custom features. We now discuss this structural rejuvenation according to the different modes that we have distinguished. Given the overall size of the applications, both in model and custom code size, we may consider this single observation to be significant. The detailed development resources spent are considered to be out of scope, as we want to observe the evolvability behavior *under normal market conditions*.

A. Continuous Development

The various applications summarized in Table I are in production, and either still in full development mode, or at least subject to extended development and/or perfective maintenance. The development teams, consisting of one to four people depending on the application, deliver bug fixes, minor improvements, and new features, that are implemented using modifications and extensions of both the model and the custom code. In several applications, this includes application-specific expanders or code generator modules that are being used and maintained. As part of the CI/CD infrastructure, applications are built and deployed in test on a daily basis, and new versions are typically deployed in production every two weeks.

B. Updating Dependencies

Updating frameworks to new versions is, similar to most software development environments, an integrated part of the CI/CD infrastructure. Besides urgent patches to address vulnerabilities, they follow the same cadence as the continuous development. When new versions are considered appropriate, they are included in the daily builds and test deployments, and the bi-weekly production deployments.

C. Rejuvenating Skeletons

The development of the NS expanders follow the same release rhythm, i.e., daily builds and testing and bi-weekly releases. As the pipelines of the expanders and the applications are part of the same integrated CI/CD infrastructure, they become available immediately upon release. As potential conflicts between the new skeletons and the existing custom code may lead to additional efforts, the various applications are only rejuvenated using a new version of the NS expanders every one or two months. Upon acceptance, they will proceed to the bi-weekly production deployments.

The systematic rejuvenation of the application skeletons, the CI/CD/CR environment, has only been realized the last 4 to 5 years of the development center. Reasons for this delay include learning effects and lack of critical mass in the NSX development center during the early years. Currently, the regular rejuvenation includes systematic improvements across the entire application landscape. These landscape-wide improvements include the cleanup of outdated coding constructs, performance enhancements in database queries, enhanced authorization and access control, additional options and features for generated screens, improved support for multitenancy and workflows, and additional options for parallel processing.

D. Replacing Technologies

The NST-based evolvable architecture of the applications also aims to facilitate the systematic replacement of external technology frameworks that handle the cross-cutting concerns of the multi-layer applications. Throughout the years, the NS expanders have introduced support for additional databases and persistency providers in the data layer. In the logic layer, improved JEE implementations have been introduced for transactions, timers and triggers. The entire application landscape has migrated seamlessly to these new technologies.

In the control and view layer, systematic migrations have been performed in the early days of the development center. First, from the *Cocoon* Model-View-Controller framework to *Struts2*, followed by migrating from *Struts2* to *Knockout* in the view layer, while *Struts2* remained the default technology in the control layer. More recently, new technologies were introduced without completely phasing out previous technologies. *JAX-RS* was introduced both in the control layer that supports the view layer, and in a separate integration layer to offer *REST* interfaces for third-party applications. *Angular* was introduced

Copyright (c) IARIA, 2024. ISBN: 978-1-68558-161-9

in the view layer, integrating with both the legacy *Struts2* control layer and the new *JAX-RS* implementation. The fact that custom code has been developed on quite a large scale over the last couple of years, often lacking discipline when calling into other layers, makes it nowadays less obvious to retire frameworks, stressing the need for coding discipline.

VI. Conclusion and Future Work

Software evolution has been facing many deep-seated issues for decades. While the current agile development paradigm has numerous benefits, it does not really solve these issues, and could potentially even worsen them. Normalized Systems Theory has proposed a software architecture that could provide software applications with higher levels of evolvability, while preserving the benefits of the agile development process. In this contribution, we have presented an observational case study to evaluate to what extent the envisioned evolvability characteristics have been realized in a state-of-the-art agile software factory based on NST.

Studying the evolvability characteristics of an NST-based agile software factory is believed to make some contributions. First, we have described in some detail how NST can be applied at a substantial scale in a modern agile software factory. Second, we have validated that some levels of evolvability envisoned by NST can indeed be operationalized in such an environment. Third, we have identified a concern that may hamper these evolvability features in a realistic environment.

Next to these contributions, it is clear that this observational case study is also subject to a number of limitations. First, the software development factory of the case study was set up in close collaboration with the authors of NST. It would be interesting to study how easily this could be reproduced in other development centers. Second, the software factory has only been operating at scale for a couple of years. Therefore, the number of significant evolutions across an entire application landscape is quite limited.

To increase significantly the time period during which the rejuvenation factory has been operating at scale, we plan to continue this observational case study for at least the next few years. We also intend to look into the added value of frameworks such as *Scaled Agile Framework (SAFe)*, that seek to guide enterprises in scaling agile practices [20] [21]. For instance, we could investigate the structured integration of techniques such as canary releases and feature toggles that are currently used on an ad hoc basis.

References

[1] A. R. Hevner, S. T. March, J. Park, and S. Ram, “Design science in information systems research,” MIS Quarterly, vol. 28, no. 1, 2004, pp. 75–105.

[2] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, “Characteristics of application software maintenance,” Communications of the ACM, vol. 26, no. 6, 1978, pp. 466—-471.

[3] T. M. Pigoski, Practical software maintenance: Best practices for managing your software investment. Wiley Computer Pub, 1997.

[4] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus, “Does code decay? assessing evidence from change management data,” IEEE Transactions on Software Engineering, vol. 27, no. 1, 2001, pp. 1—-12.

[5] M. Lehman, “Program, life-cycles and the laws of software evolution,” in Proceedings of the IEEE, vol. 68, 1980, pp. 1060–1076.

[6] R. Agarwal and A. Tiwana, “Editorial—evolvable systems: Through the looking glass of IS,” Information Systems Research, vol. 26, no. 3, 2015, pp. 473–479.

[7] J. Estdale, “Delaying maintenance can prove fatal,” in Proceedings of Software Quality Management XXVII: International Experiences and Initiatives in IT Quality Management, 2019, pp. 95—-106.

[8] H. Mannaert, J. Verelst, and K. Ven, “The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability,” Science of Computer Programming, vol. 76, no. 12, 2011, pp. 1210–1222, special Issue on Software Evolution, Adaptability and Variability.

[9] H. Mannaert, J. Verelst, and P. De Bruyn, Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design. Koppa, 2016.

[10] H. Mannaert, K. De Cock, P. Uhnak, and J. Verelst, “On the realization of meta-circular code generation and two-sided collaborative metaprogramming,” International Journal on Advances in Software, no. 13, 2020, pp. 149–159.

[11] H. Mannaert, J. Verelst, and K. Ven, “Towards evolvable software architectures based on systems theoretic stability,” Software: Practice and Experience, vol. 42, no. 1, 2012, pp. 89–116.

[12] P. De Bruyn, H. Mannaert, and P. Huysmans, “On the variability dimensions of normalized systems applications: Experiences from an educational case study,” in Proceedings of the Tenth International Conference on Pervasive Patterns and Applications (PATTERNS), 2018, pp. 45–50.

[13] ——, “On the variability dimensions of normalized systems applications : experiences from four case studies,” International Journal on Advances in Systems and Measurements, vol. 11, no. 3, 2018, pp. 306–314.

[14] M. D. McIlroy, “Mass produced software components,” in Proceedings of NATO Software Engineering Conference, Garmisch, Germany, October 1968, pp. 138–155.

[15] J. Greenfield, K. Short, and S. Cook, Steve; Kent, Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, 2004.

[16] T. Saeed, “Current issues in software re-usability: A critical review of the methodological & legal issues,” Journal of Software Engineering and Applications, vol. 13, no. 9, 2020, pp. 206–217.

[17] R. T. Yarlagadda, “Devops and its practices,” International Journal of Creative Research Thoughts (IJCRT), vol. 9, no. 3, 2021, pp. 111–119.

[18] H. Mannaert, K. De Cock, and J. Faes, “Exploring the creation and added value of manufacturing control systems for software factories,” in Proceedings of the Eighteenth International Conference on Software Engineering Advances (ICSEA 2023), 2023, pp. 14—-19.

[19] Moderne, “Introduction to OpenRewrite,” URL: https://docs.openrewrite.org/, 2023, [accessed: 2024-03-05].

[20] W. Hayes, M. A. Lapham, S. Miller, E. Wrubel, and P. Capell, “Scaling agile methods for department of defense programs,” Software Engineering Institute, Tech. Rep. CMU/SEI-2016-TN-005, 12 2016.

[21] D. Athrow, “Why Continuous Delivery is key to speeding up software development,” URL: https://www.techradar.com/news/software/why-continuous-delivery-is-key-to-speeding-up-software-development-1282498, 01 2015, [accessed: 2024-03-24].