PATTERNS 2024

The Sixteenth International Conferences on Pervasive Patterns and Applications

ISBN: 978-1-68558-161-9

April 14 - 18, 2024

Venice, Italy

PATTERNS 2024 Editors

Geert Haerens, Faculty of Applied Economics Antwerp University, Belgium

Herwig Mannaert, University of Antwerp, Belgium

PATTERNS 2024

Forward

The Sixteenth International Conferences on Pervasive Patterns and Applications (PATTERNS 2024), held on April 14 – 18, 2024, continued a series of events targeting the application of advanced patterns, at-large. In addition to support for patterns and pattern processing, special categories of patterns covering ubiquity, software, security, communications, discovery and decision were considered. It is believed that patterns play an important role on cognition, automation, and service computation and orchestration areas. Antipatterns come as a normal output as needed lessons learned.

The conference had the following tracks:

- Patterns basics
- Patterns at work
- Discovery and decision patterns
- Medical and facial image patterns
- Tracking human patterns

Similar to the previous edition, this event attracted excellent contributions and active participation from all over the world. We were very pleased to receive top quality contributions.

We take here the opportunity to warmly thank all the members of the PATTERNS 2024 technical program committee, as well as the numerous reviewers. The creation of a high quality conference program would not have been possible without their involvement. We also kindly thank all the authors that dedicated much of their time and effort to contribute to PATTERNS 2024. We truly believe that, thanks to all these efforts, the final conference program consisted of top quality contributions.

Also, this event could not have been a reality without the support of many individuals, organizations and sponsors. We also gratefully thank the members of the PATTERNS 2024 organizing committee for their help in handling the logistics and for their work that made this professional meeting a success.

We hope PATTERNS 2024 was a successful international forum for the exchange of ideas and results between academia and industry and to promote further progress in the area of pervasive patterns and applications. We also hope that Venice provided a pleasant environment during the conference and everyone saved some time to enjoy this beautiful city.

PATTERNS 2024 Steering Committee

Herwig Manaert, University of Antwerp, Belgium
Wladyslaw Homenda, Warsaw University of Technology, Poland
Yuji Iwahori, Chubu University, Japan
Alexander Mirnig, University of Salzburg, Austria
George A. Papakostas, International Hellenic University – Kavala, Greece

PATTERNS 2024 Publicity Chair

José Miguel Jiménez, Universitat Politecnica de Valencia, Spain
Sandra Viciano Tudela, Universitat Politecnica de Valencia, Spain

PATTERNS 2024

Committee

PATTERNS 2024 Steering Committee

Herwig Manaert, University of Antwerp, Belgium
Wladyslaw Homenda, Warsaw University of Technology, Poland
Yuji Iwahori, Chubu University, Japan
Alexander Mirnig, University of Salzburg, Austria
George A. Papakostas, International Hellenic University – Kavala, Greece

PATTERNS 2024 Publicity Chair

José Miguel Jiménez, Universitat Politecnica de Valencia, Spain
Sandra Viciano Tudela, Universitat Politecnica de Valencia, Spain

PATTERNS 2024 Technical Program Committee

Andrea F. Abate, University of Salerno, Italy
Akshay Agarwal, IIIT Delhi, India
Carlos Alexandre Ferreira, INESC TEC, Portugal
Vijayan K. Asari, University of Dayton, USA
Danilo Avola, Sapienza University of Rome, Italy
Johanna Barzen, University of Stuttgart, Germany
Frederik Simon Bäumer, Bielefeld University of Applied Sciences, Germany
Martin Beisel, University of Stuttgart, Germany
Nadjia Benblidia, Saad Dahlab University - Blida1, Algeria
Anna Berlino, Consultant in Tourism Sciences and Valorization of Cultural and Tourism Systems, Italy
Fatma Bouhlel, University of Sfax, Tunisia
Uwe Breitenbücher, IAAS - University of Stuttgart, Germany
Alceu S. Britto, Pontifical Catholic University of Paranā (PUCPR), Brazil
Jean-Christophe Burie, L3i laboratory | La Rochelle University, France
Eliot Bytyçi, University of Prishtina "Hasan Prishtina", Kosovo
Isaac Caicedo-Castro, University of Córdoba, Colombia
Simone Cammarasana, CNR-IMATI, Genova, Italy
David Cárdenas-Peña, Universidad Tecnológica de Pereira, Colombia
Bidyut B. Chaudhuri, Indian Statistical Institute, India
Sneha Chaudhari, AI Organization | LinkedIn, USA
Diego Collazos, Universidad Nacional de Colombia sede Manizales, Colombia
Sergio Cruces, University of Seville, Spain
Mohamed Daoudi, Institut Mines-Telecom / Telecom Lille, France
Jacqueline Daykin, King's College London, UK / Aberystwyth University, Wales & Mauritius
Moussa Diaf, Mouloud Mammeri University, Algeria

Copyright Information

For your reference, this is the text governing the copyright release for material published by IARIA.

The copyright release is a transfer of publication rights, which allows IARIA and its partners to drive the dissemination of the published material. This allows IARIA to give articles increased visibility via distribution, inclusion in libraries, and arrangements for submission to indexes.

I, the undersigned, declare that the article is original, and that I represent the authors of this article in the copyright release matters. If this work has been done as work-for-hire, I have obtained all necessary clearances to execute a copyright release. I hereby irrevocably transfer exclusive copyright for this material to IARIA. I give IARIA permission or reproduce the work in any media format such as, but not limited to, print, digital, or electronic. I give IARIA permission to distribute the materials without restriction to any institutions or individuals. I give IARIA permission to submit the work for inclusion in article repositories as IARIA sees fit.

I, the undersigned, declare that to the best of my knowledge, the article is does not contain libelous or otherwise unlawful contents or invading the right of privacy or infringing on a proprietary right.

Following the copyright release, any circulated version of the article must bear the copyright notice and any header and footer information that IARIA applies to the published article.

IARIA grants royalty-free permission to the authors to disseminate the work, under the above provisions, for any academic, commercial, or industrial use. IARIA grants royalty-free permission to any individuals or institutions to make the article available electronically, online, or in print.

IARIA acknowledges that rights to any algorithm, process, procedure, apparatus, or articles of manufacture remain with the authors and their employers.

I, the undersigned, understand that IARIA will not be liable, in contract, tort (including, without limitation, negligence), pre-contract or other representations (other than fraudulent misrepresentations) or otherwise in connection with the publication of my work.

Exception to the above is made for work-for-hire performed while employed by the government. In that case, copyright to the material remains with the said government. The rightful owners (authors and government entity) grant unlimited and unrestricted permission to IARIA, IARIA's contractors, and IARIA's partners to further distribute the work.

Table of Contents

Systematic Rejuvenation of a Budgeting Application over 10 years: A Case Study

Chetak Kandaswamy, Jan Verelst

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

***Abstract*—Normalized Systems (NS) theory has recently been proposed as a means of increasing software agility. NS theory posits that software evolvability, or the ease with which software can be changed, can be achieved by adhering to a set of theorems that result in a specific and evolvable software architecture, based on the use of NS-specific code generators called expanders. While the theoretical contributions of NS theory have been well-documented in previous research, there are few reports on real-life cases where NS theory has been employed. This paper documents a development project that demonstrates the feasibility of the NS approach for building evolvable software and highlights the benefits of a real-life NS development project over a period of more than 10 years, in which the system was built and afterwards regenerated using the NS code generators. The results confirm the feasibility of systematically regenerating information systems in Java over time with limited resources, eliminating or drastically reducing the need for rebuilds from scratch, in order to deal with structure degradation of information systems, more specifically for information systems of limited size and complexity, which are commonplace in today's digital economy.**

Keywords-Normalized Systems; Evolvability; Agility; Software Rejuvenation

I. INTRODUCTION

In recent years, there has been a growing body of research on agile software development. While this research has yielded valuable insights into improving agile development processes, there has been comparatively less focus on enhancing the agility of the software itself. Important Agile frameworks, such as the Scaled Agile Framework (SAFe), define Agile Architecture as a set of values and principles that support the active evolution of the design and architecture of a system while implementing new capabilities. This definition points more in the direction of a process than it does in assuring that the system itself will be agile. In that respect, Agile Architecting is a better term to refer to an agile way of doing architecture, and Agile Architecture could point to the intentionality of creating an evolving system.

Normalized Systems (NS) theory has recently been proposed as a means of increasing software agility. NS posits that software evolvability, or the ease with which software can be changed, can be achieved by adhering to a set of theorems that result in a specific and evolvable software architecture. This architecture offers systems theoretical stable responses to changing business and/or technical requirements. NS theory has been refined and extended over the years and has been implemented in several software projects. While the theoretical contributions of NS theory have been well-documented in previous research, there are few reports on real-life cases where NS theory has been employed.

This paper documents a development project that demonstrates the feasibility of the NS approach for building a Budgeting application and maintaining it over a period of 10 years.

The paper is structured as follows. In Section II, we review the concepts behind NS theory and software rejuvenation. Section III will provide information about the Budgeting application and an overview of the different changes applied to the application over a period of 10 years. In Section IV, we will discuss the Budgeting application from the perspective of the owner of the application, the Province of Antwerp, and report their reflections on the past 10 years. Section V presents our conclusions.

II. FUNDAMENTALS OF NS THEORY

Software architectures should be able to evolve as business and technical requirements change over time. In NS theory, evolvability is measured by a lack of Combinatorial Effects (CE) in software architectures. Combinatorial Effects constitute a specific kind of ripple effects: when the impact of a change, measured in the number of impacted modules, depends not only on the type of change but also on the size of the software system, a Combinatorial Effect occurs. NS theory assumes that software undergoes unlimited evolution (i.e., that both new and changed requirements will make a software system increase in size over time), which makes Combinatorial Effects very harmful to software evolvability. Indeed, if changes to a system depend on the size of the growing system, these changes become harder to handle (i.e., requiring more work and therefore lowering the evolvability of the system).

NS theory is built on principles from systems theory (stability) and statistical thermodynamics (entropy). In systems theory, a system is stable if it has bounded input leading to bounded output (BIBO). NS theory applies this idea to software design as a bounded change in functionality should only cause a bounded change in the software. In systems theory, stability is measured at infinity. NS theory considers systems that grow infinitely large over time and will go through infinitely many changes. According to NS theory, a system is stable towards changes, if it does not have CE,

 ISBN: 978-1-68558-161-9

meaning that the effect of a change only depends on the type of change and not on the system size.

NS theory suggests four theorems and five elements as the basis for creating evolvable software through pattern expansion of the elements. The theorems have been proven formally, and provide a set of design guidelines that must be followed strictly in order to avoid Combinatorial Effects. The NS elements offer a set of predefined higher-level structures, patterns, or "building blocks", that provide functionality while conforming to all NS theorems. Therefore, they constitute a blueprint for implementing the core functionalities of realistic information systems.

A. NS Theorems

NS theory proposes four theorems that describe the necessary conditions for software to be free of Combinatorial Effects:

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

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

B. NS Elements

Consistently adhering to the four NS theorems seems very challenging for developers because of several reasons. First, following the NS theorems leads to a fine-grained software structure as concerns and states are separated, which does introduce some development overhead that may slow down the development process. Second, the theorems must be followed all the time, which is problematic in a context where human programmers work under varying project conditions, including (occasionally) limited time and budgets. Third, the accidental introduction of Combinatorial Effects results in an exponential increase of rework that needs to be done at a later time.

Five elements were therefore proposed which make the realization of NS applications more feasible, as they can be instantiated by code generators called expanders. These elements are carefully engineered patterns that comply with the four NS theorems and that can be used as essential building blocks for a wide variety of applications. The elements are named according to the elementary functionality they offer: data element, action element, workflow element, connector element, and trigger element.

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

The element not only provides core functionality (such as persistency of data, execution of an action, etc.) but also addresses the cross-cutting concerns that each of these core functionalities require to function properly. As cross-cutting concerns cut through every element, they require careful separation from other concerns in order not to introduce Combinatorial Effects.

C. Element Expansion

An application is mainly composed of a set of data, action, workflow, connector, and trigger elements that realize its requirements. An NS expander instantiates the software elements into source code for the specific application. The expanded code will provide functionalities specified in the application definition and constitutes a fine-grained modular structure that follows the NS theorems (see Figure 1) and is therefore free from combinatorial effects. This generated part of an application is also called the skeleton of the application.

Next, remaining functionality, such as the business logic for the application, is manually programmed or customized inside the expanded modules, at pre-defined locations. This functionality is called a customization or crafting. The presence of combinatorial effects in this manually programmed part of the application, depends on the adherence of the individual programmer to the NS theorems. However, a strength of this approach is that the only location where Combinatorial Effects can be introduced, is in the customized code.

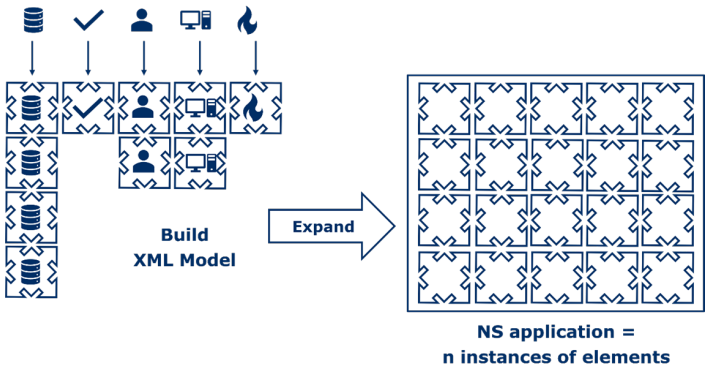


Figure 1. Requirements expressed in an XML description file, used as input for element expansion

D. Harvesting and Software Rejuvenation

The expanded skeleton has some pre-defined places where customizations can be made. To keep these customizations from being lost when the application is re-expanded at a

later time, these customizations are gathered and put back when the application is re-expanded. This process of gathering and putting back the customizations is called harvesting and injection.

The application can be re-expanded for different reasons. For example, the code templates of the elements are improved (bug fixes, performance improvements, new versions of supporting technologies, or changes in the technology, such as a new persistence framework, etc.).

The purpose of software rejuvenation is to carry out the harvesting and injection process routinely to ensure that the improvements of the 5 element code templates are incorporated into the skeleton of the application.

In our experience, in a Java environment, expansion produces more than 80% of the code of a production-ready 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 such as persistency, remote access, logging and security at an advanced level. The manual production of such code often is time consuming. Using NS expansion, this time can now be spent on, e.g., the constant improvement of the element code templates, the development of new code templates that make the elements compatible with new technologies, and on meticulous coding of the business logic. The changes in the elements can be applied to all expanded applications, giving the concept of code reuse a new meaning. A modification on a code template by one developer can be used by all developers on all their applications, with minimal impact, thanks to the rejuvenation process. Figure 2 summarizes the NS development process.

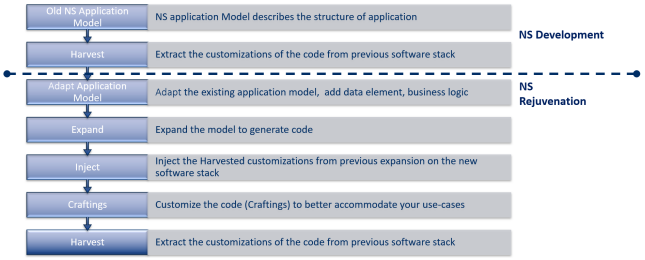


Figure 2. The NS development process

III. THE USE CASE: PROVINCE OF ANTWERP BUDGETING APPLICATION

In this section, we first describe the Budgeting Application at a functional level, and then describe the evolution and rejuvenation process that took place over the course of about 10 years. This case study is based on interviews with the Head of IT Projects of the Province of Antwerp as well as the programmers who were involved in development and maintenance.

A. The Application

As a case study for software rejuvenation, we selected a Budgeting application of a local Belgian government. The application was built because the existing financial ERP package was difficult to adapt to the specifics of Belgian government budgeting regulations. The application was first built using NS technology in 2012 and is currently still in use. The functional requirements of the application are budget creation and management, expense tracking and control, managing different revenue streams, forecasting and planning, reporting and analysis, compliance and audit trail, integration with financial systems, data security, and privacy.

This Budgeting application has played a crucial role in enhancing transparency, accountability, and efficiency in the budgeting process. It has enabled the government to monitor and manage its financial resources effectively, ensure compliance with fiscal policies, and make data-driven decisions to allocate resources efficiently. This application has integrated well with the existing financial systems used by government entities, such as accounting software or Enterprise Resource Planning (ERP) systems. This integration has ensured data consistency and has reduced manual data entry.

The functional requirements are easily explained using the Entity Relationship Diagram (ERD) shown in Figure 3. The diagram shows the Budget as the central data element instance of the application. The Budget element is defined by a combination of the following 11 data elements: Article, Budget type, Budget change, Budget year, Cell, Domain, Product, Recording, Service, Supplier, and Team. The unique combination of these 11 parameters is the key to the budget in its most basic manifestation.

The current budget is an aggregation of many sub-budgets over time along with the combination of the above parameters in real-time for data integrity reasons. The calculated current budget is not stored in the database to avoid error propagation which may lead to faulty data. The most granular budget is calculated based on the following data elements: Article, Budget type, Budget change, Budget year, Department, Domain, and Product instances as visualized on the left of the figure. The specific budget belongs to a single department, activity, etc. The activity is grouped with the Economic groups, which in turn makes the Budget estimate.

B. Application evolution and rejuvenation

Over the past 10 years, the Budgeting application has been subject to many changes. Although the business logic of the application, mainly driven by legislation, required only one major update over this period, the number of changes in user functionalities were more frequent. Both changes in legislation and user functionalities required new code customizations. Also, there have been many changes to the element code templates. They have been updated based on feedback from customers (bug reports, performance issues, etc.) and the changing technological landscape (new operating system versions, database updates, programming language evolutions, application server changes and even the switch in deployment methods from onsite to cloud).

Figure 4 summarizes the software rejuvenation of the Budgeting application over the past 10 years. The efforts

 ISBN: 978-1-68558-161-9

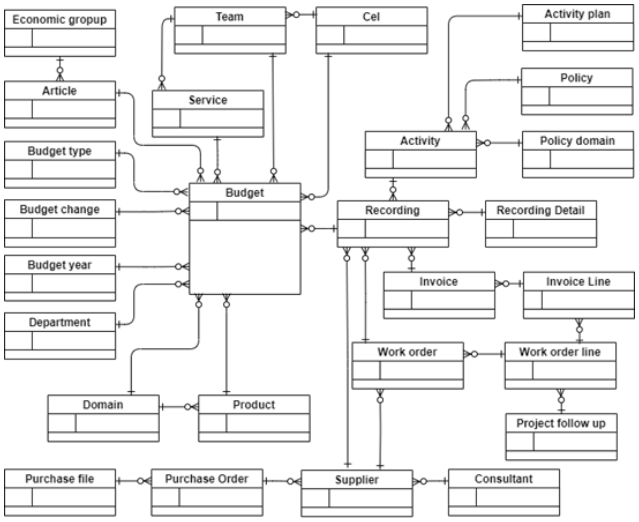


Figure 3. ERD model of the Budgeting application

required to perform technology updates using rejuvenation add up to a total of 5 days over 10 years. Between 2012 and 2019, the technologies used by the application have endured, which contributes to the relatively limited effort required. The rejuvenation mainly included updates of the element code templates, benefiting from the continuous evolution of element code template improvements done by other developers. In 2019, a significant update at the technology side happened: a change in the programming language version, application server, and frontend technology. The total effort was 2 man-days to accomodate these changes. In 2023, the changes in technology were even more profound as the programming language version, frontend, database, application server, and deployment method (container instead of server-based) changed. The effort was only 1 man-day. In summary, the skeleton of the Budgeting application was rejuvenated several times over the past 10 years, each time requiring an effort in terms of one to several man-days, which can be considered a limited investment to incorporate all the benefits of a rejuvenation described above.

The total time invested in changes to customizations or craftings adds up to 50 man-days from application conception (2012) to the current state (2023). The effort of implementing new customizations (new legislation in 2014) and user functionalities (2014, 2015 and 2019) can be considered similar to whatever development method and/or technology was used in the industry at that time, which is unsurprising as this essentially manually written code in an NS application.

In summary, over a period of 10 years, the total effort of change has been 28 man-days, of which 23 have been purely functional changes and 5 due to rejuvenation. These figures confirm and even outperform estimations that were made about the development effort of this very same application in 2014 (see Figure 5) [4].

IV. VOICE OF THE CUSTOMER

This section is based on interviews with the Head of IT Projects and Solutions at the Province of Antwerp.

1) On the advantages of Rejuvenation using NS framework: "The main advantage for us was the speed that can be gained with the rejuvenation of the application. Because the process of expansion and re-injection is fully automated and fast, a new version can be put in place and the actual functionality can be tested instead of also having to validate and test the boiler-plate code."

2) On developing with or without NS: "We have no real data concerning the effective difference between development with or without NS. In my opinion, if we did not use NS, the first change of the application in 2014 (new budgeting legislation), would have resulted in building a new application, instead of just rejuvenating the existing one. Such a rebuild would have probably taken 50 man-days. While with rejuvenation, we only had a few days of functional testing to do.".

3) On Maintenance cost: The maintenance of 6 different applications at the Province of Antwerp built using the NS methods (including the Budgeting application) required only 4 man-days of maintenance operations both in 2021 and in 2022 (across all 6 applications).

4) On NS vs. Low Code: As the proprietary budgeting tool was to be used by only 20 users, low-code and no-code platforms could also have been considered as development platforms, as they allow users to create applications using a minimal amount of coding. At the time of development (2012), such platforms were not considered by the Province of Antwerp. Revisiting the NS vs. low-code decision at this point in time, can be done based on a number of criteria. First, it is important to note that stakeholders from the Province of Antwerp required specific customizations for the Budgeting application, potentially causing low-code platforms to be challenged in terms of customization, scalability, and flexibility. Second, if an organization builds applications heavily reliant on the low-code platform's proprietary features or architecture, migrating to a different platform or transitioning away from the platform can be difficult and time-consuming.

5) On NS vs. Shadow IT: The Budgeting application is not a challenging and complex application, and one might be tempted to turn to the usage of a MS Excel or MS Access-based application, completely created and maintained by the business, instead of IT. The Province of Antwerp did not go down this path as they already had some years of experience in doing their budgeting work in MS Excel and noticed important drawbacks such as the fragility of the solution, dependence on a few people who master the implementation of the business logic in MS Excel and high maintenance cost.

V. CONCLUSION

In this paper, we discussed how the NS theory can be applied to rejuvenating a Budgeting application, which essentially is a small CRUDS application, which was built using the NS expanders. Over this period, this application has undergone multiple functional and non-functional, technical

 ISBN: 978-1-68558-161-9

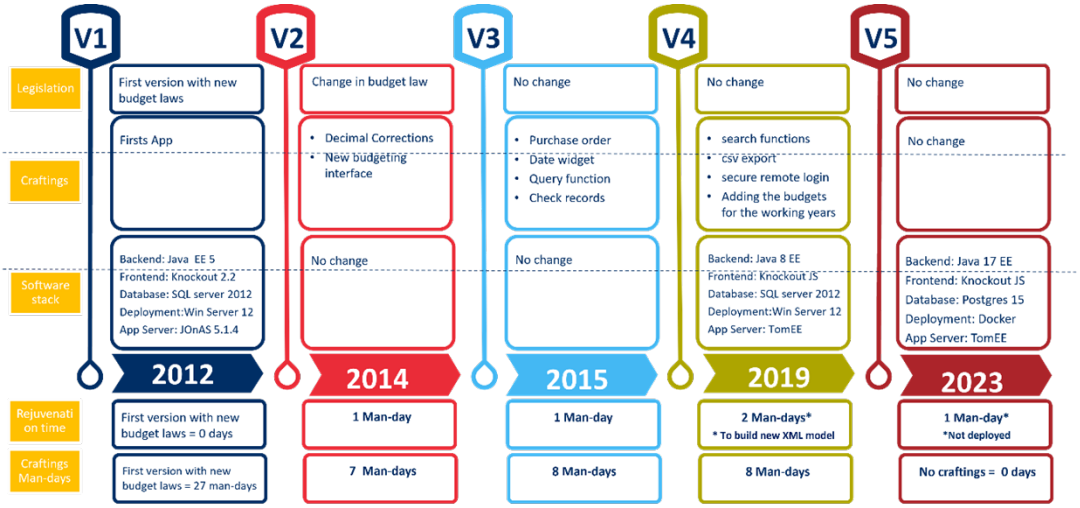


Figure 4. Summary of Software Rejuvenation for 10 Years

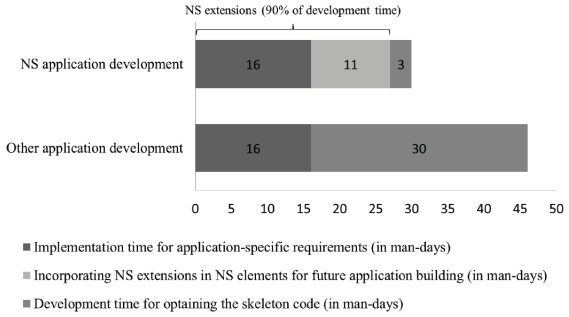


Figure 5. Comparison of estimated development time [4]

changes. The technical changes were limited in the sense that updates from technologies were required, but no major shifts to other technologies. Nonetheless, in a time where many applications are rebuilt after 5-10 years, it is interesting to see that it is feasible to see that rejuvenation is feasible over a period of 10 years, with the skeleton of the application being updated to the most recent version of the underlying technologies. This suggests that the increased use of code generators holds significant promise for the future.

REFERENCES

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

[2] 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.

[3] 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, Springer, ISSN 1865-1348-142, pp. 43-63, 2013.

[4] G. Oorts, et al., "Building Evolvable Software Using Normalized Systems Theory: A Case Study", Proceedings of the annual Hawaii international conference on system sciences, ISBN 978-1-4799-2504-9, pp. 4760-4769, 2014.

 ISBN: 978-1-68558-161-9

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

Christophe De Clercq

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

Jan Verelst

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

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

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

I. Introduction

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

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

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

II. Fundamentals of NS theory

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

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

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

 ISBN: 978-1-68558-161-9

core functionalities of realistic information systems, following the four theorems.

A. NS Theorems

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

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

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

B. NS Elements

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

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

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

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

C. Element Expansion

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

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

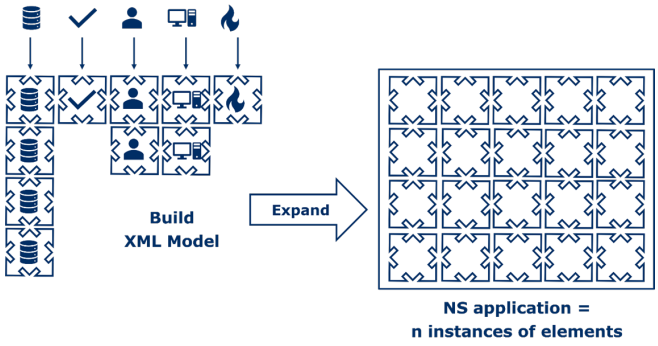


Fig. 1. Requirements expressed in an XML description file, used as input for element expansion.

D. Harvesting and Software Rejuvenation

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

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

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

Code expansion produces more than 80% of the code of the application. The expanded code can be called boiler-plate-code, but it is more complex than what is usually meant by that term because it deals with Cross-Cutting Concerns. Manually

 ISBN: 978-1-68558-161-9

producing this code takes a lot of time. Using NS expansion, this time can now be spent on the constant improvement of the code templates, the development of new code templates that make the elements compatible with new technologies, and on meticulous coding of the business logic. The changes in the elements can be applied to all expanded applications, giving the concept of code reuse a new meaning. A modification on a code template by one developer can be used by all developers on all their applications with minimal impact, thanks to the rejuvenation process.

III. Fundamentals of Software Migration Strategies

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

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

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

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

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

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

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

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

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

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

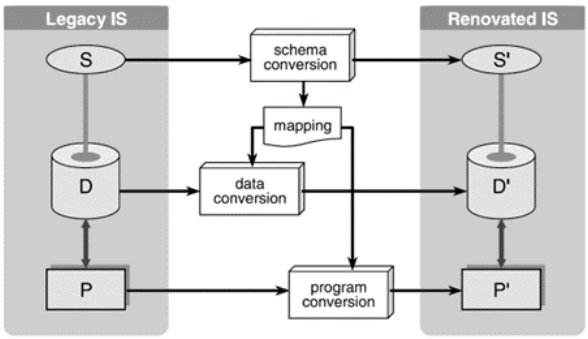


Fig. 2. Conceptual schema conversion strategy (from [5])

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

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

IV. Use Case: Connecting-Expertise

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

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

A. Functional perspective

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

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

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

 ISBN: 978-1-68558-161-9

B. Technical perspective

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

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

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

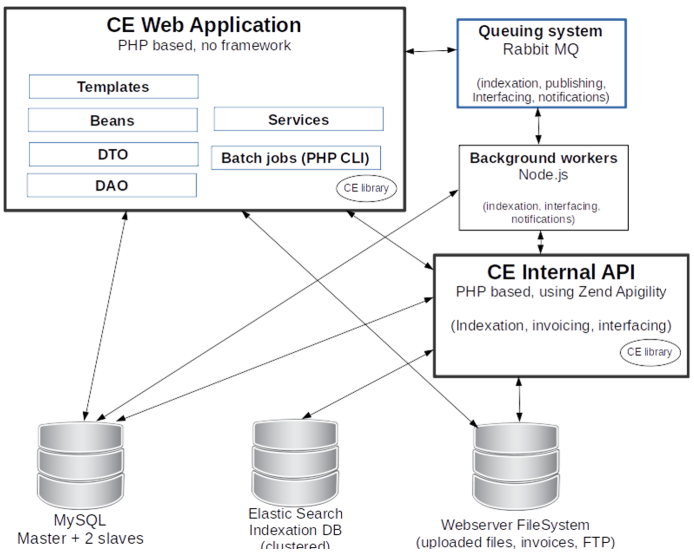


Fig. 3. CE2 VMS technical architecture.

C. Maintainability and evolvability issues

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

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

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

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

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

5) Functionality: The system is complicated to set up for new clients. They frequently need new application settings,

 ISBN: 978-1-68558-161-9

reports, or even application functions. This makes it hard to expand the application to more customers (for example, in a new country). The system also has a limitation on the currency: some system modules only support the Euro.

D. The Need for Change

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

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

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

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

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

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

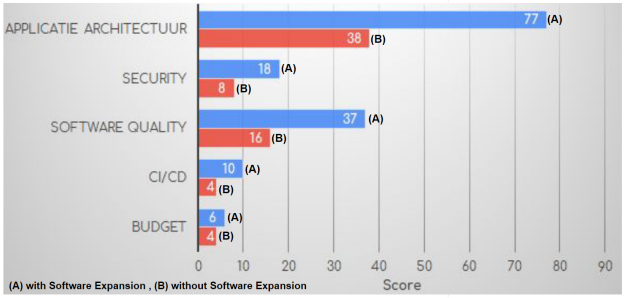


Fig. 4. Implementing CE2 with or without Software Expansion.

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

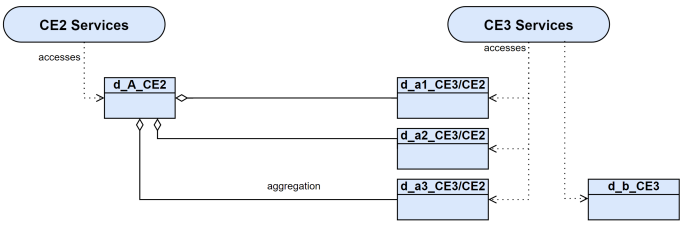


Fig. 5. Transformation of data objects between CE2 and CE3.

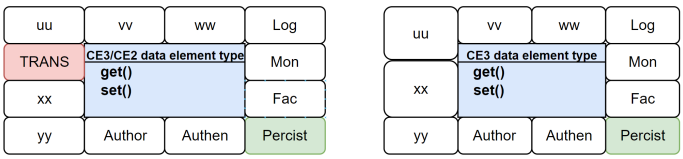


Fig. 6. Transformer as a Cross-Cutting Concern of the CE3/CE2 data element type.

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

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

 ISBN: 978-1-68558-161-9

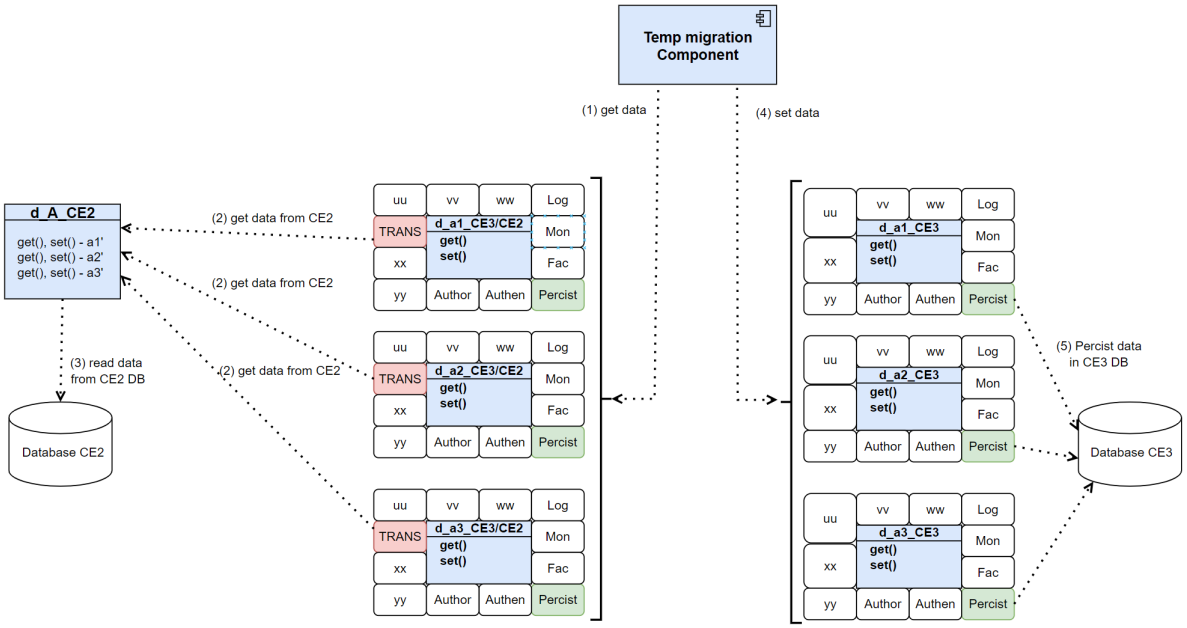


Fig. 7. Migration of data from CE2 VMS to CE3 VMS.

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

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

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

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

V. Discussion

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

A. The choice for NS Expansion

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

B. Migration Approach

The usage of the transformers plays an essential role in the migration from CE2 VMS toward CE3 VMS. The idea of gradually shifting functionalities from one system to another while keeping both live is referred to as the Chicken Little approach (see [6]). The main drawback of using this approach is the need for gateways between the source and target system. These gateways must be meticulously designed and consistently implemented, which can be daunting. NS Expansion mitigates the downsides of doing Chicken Little dramatically. The gateways are implemented using the transformer classes that are part of the data elements. Using NS Expansion ensures that each gateway/transformer is identical in structure and

 ISBN: 978-1-68558-161-9

usage. The transformers can evolve, and all modifications and improvements can be quickly and easily redeployed using re-expansion/rejuvenation. When functionality is fully migrated from the source to the target system, there is no longer the need to keep the gateways in place. With classic coding practices, the manual removal of the gateways comes with risks. Accidental removal of too much could result in broken functionalities. Insufficient removal results in traces of legacy code in a brand-new system. With NS Expansion, it suffices to perform a rejuvenation cycle to replace the code templates that contain transformers with code templates without transformers. All traces of legacy are removed in a consistent and precise way.

C. Phased migration

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

VI. Conclusion

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

Acknowledgment

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

References

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

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

[3] H. Mannaert, J. Verelst, and K. Ven, “The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability”, Science of Computer Programming, Volume 76, Issue 12, pp. 1210-1222, 2011

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

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

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

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

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

 ISBN: 978-1-68558-161-9

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.

 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

 ISBN: 978-1-68558-161-9

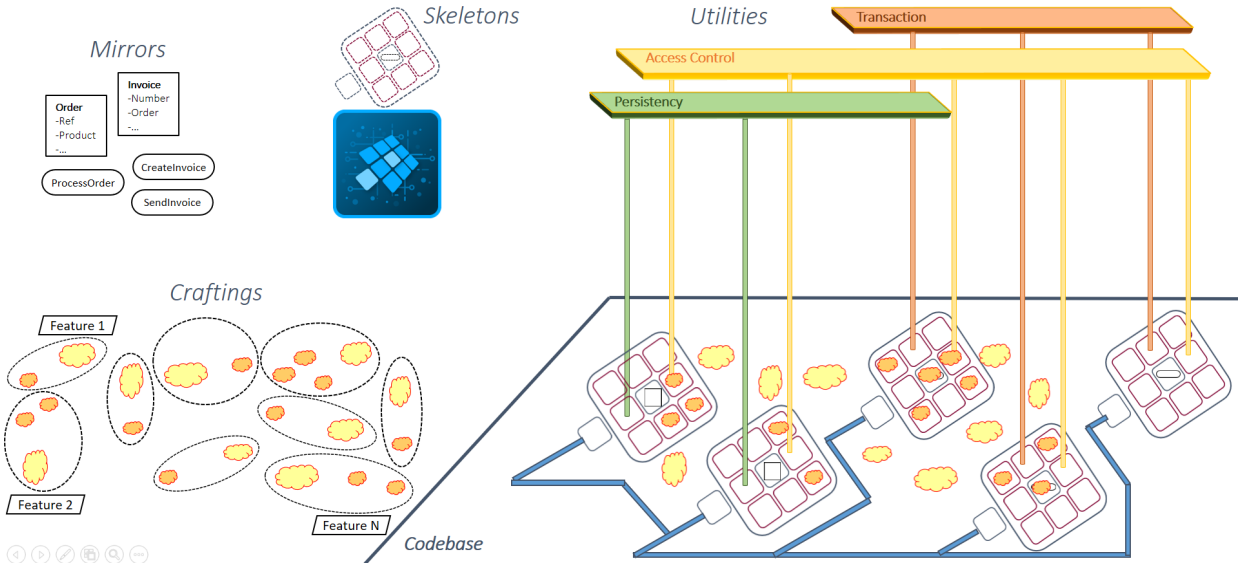


Figure 1. A graphical representation of four variability dimensions within a Normalized Systems application codebase.

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.

 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

 ISBN: 978-1-68558-161-9

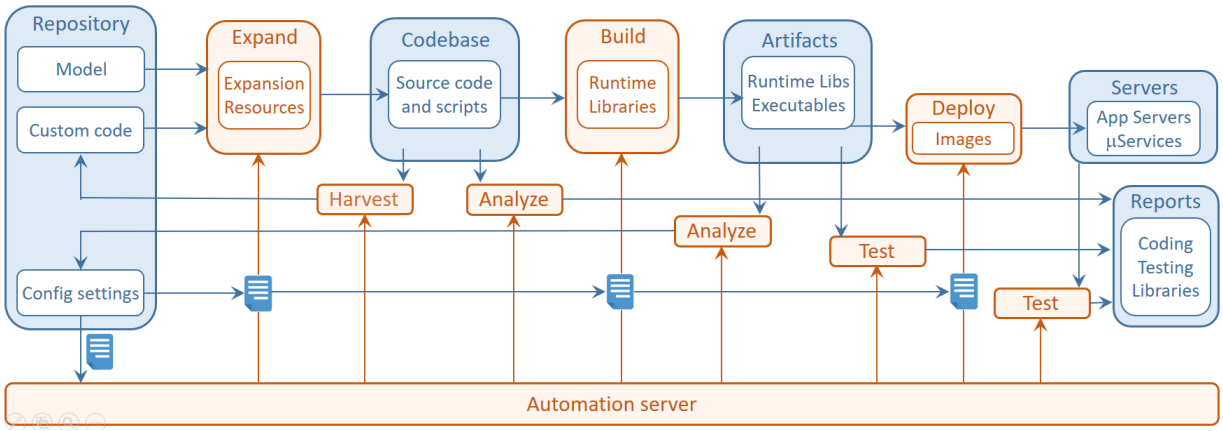


Figure 2. A traditional representation of a typical DevOps infrastructure.

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

 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].

 ISBN: 978-1-68558-161-9

Converging Clean Architecture with Normalized Systems

Gerco Koks
Antwerpen Management School, Alumini
Zundert, Netherlands
email:gerco.koks@outlook.com

***Abstract*—This paper explores the convergence between Clean Architecture and Normalized Systems principles and design elements, highlighting their synergistic potential to enhance software design and evolvability. The paper draws upon the research described in the thesis of "On the Convergence of Clean Architecture with the Normalized Systems Theorems" from G. Koks through a comparative analysis. It demonstrates how each paradigm contributes to modular, maintainable, and evolvable software design and how integrating both approaches can lead to a more widely spread adoption and an improved software design.**

***Keywords*-Software; Architecture; Evolvability; Modularity; Stability.**

I. Introduction

In the evolving landscape of software architecture, the software development paradigms of Clean Architecture (CA) and Normalized Systems (NS) have emerged as pivotal in addressing the multifaceted challenges of software design, particularly in managing stability, modularity, and evolvability to achieve resiliency in software. This paper delves into the synergy between these two paradigms, each contributing significantly to the contemporary discourse on software architectural complexity.

Tracing the historical underpinnings of these concepts reveals the works of pioneers like D. McIlroy [2], who championed modular programming, and Lehman [3], who underscored the importance of software evolution. Contributions from Dijkstra [4] on structured programming and Parnas [5] on software modularity further cemented the foundation for CA and NS. These historical insights contextualize the evolution of software engineering principles and underscore the relevance of fostering maintainable and evolvable software systems.

The foundation of this paper is an exploration of findings from extensive research on the convergence of CA and NS [1]. This research provides a nuanced perspective on integrating these distinct yet harmonious frameworks to enhance software design. It meticulously examines the core principles and elements of both CA and NS, presenting a scientifically robust synthesis that addresses critical challenges in software architecture.

This paper outlines the insights from G. Koks' research, exploring the significant benefits and practical implications of integrating the strengths of CA and NS within the dynamic field of software development.

The introduction is intended to set the stage and articulate the goal of this paper. Section 2 lays out the theoretical background, zooming in on the specific principles and elements of each Software Design Paradigm while also highlighting their unified concepts. In Section 3, we analyze the similarities and differences of their principles and elements and their effect on the evolvability of software constructs. The paper summarizes the conclusions in Section 4.

II. Theoretical Background

This Section explores the theoretical background of both CA and NS frameworks in software engineering. It focuses on the synergetic concepts, underlying principles, and architectural building blocks of both approaches and paradigms, providing the foundation for the comparative analysis.

A. Unified concepts

In this Section, we will examine concepts related to both CA and NS. Understanding these concepts is crucial for executing the research and interpreting its results.

1) Modularity: The original material of Martin [6, p. 82] describes a module as a piece of code encapsulated in a source file with a cohesive set of functions and data structures. According to Mannaert *et al.* [7, p. 22], modularity is a hierarchical or recursive concept that should exhibit high cohesion. While both design approaches agree on the cohesiveness of a module's internal parts, there is a slight difference in granularity in their definitions.

2) Cohesion: Mannaert *et al.* [7, p. 22] consider cohesion as modules that exist out of connected or interrelated parts of a hierarchical structure. On the other hand, Martin [6, p. 118] discusses cohesion in the context of components. He attributes the three component cohesion principles as crucial to grouping classes or functions into cohesive components. Cohesion is a complex and dynamic process, as the level of cohesiveness might evolve as requirements change over time.

3) Coupling: Coupling is an essential concept in software engineering that is related to the degree of interdependence among various software constructs. High coupling between components indicates the strength of their relationship, creating an interdependent relationship between them. Conversely, low coupling signifies a weaker relationship, where modifications in one part are less likely to impact others. Although not always possible, the level of coupling between the various modules of the system should be kept to a bare minimum. Both Mannaert *et al.* [7, p. 23] and Martin [6, p. 130] agree to achieve as much decoupling as possible.

B. Normalized Systems

NS in software engineering revolves around stable and evolvable information systems, drawing from System Theory

 ISBN: 978-1-68558-161-9

and Statistical Entropy from Thermodynamics. NS is rooted in software engineering but applies to other domains, such as Enterprise Engineering [8], Hardware configurations like TCP-IP firewalls [9], and Business Process Modeling [10].

The NS theory emphasizes stability as a crucial property derived from the concept of Bounded Input leading to Bounded Output (BIBO). Stability in NS means that a bounded functional change must result in a bounded amount of work, regardless of the system's size. Instabilities, also referred to as combinatorial effects, occur when the number of changes depends on the system size, negatively impacting its evolvability.
In the following list, we will describe the design Theorems of NS, first presented by Mannaert and Verelst [11].

- ***Separation Of Concerns (SoC)***: A processing function containing only a single task to achieve stability.
- ***Data Version Transparency (DvT)***: A data structure passed through a processing function's interface must exhibit version transparency to achieve stability.
- ***Action Version Transparency (AvT)***: A processing function that is called by another processing function needs to exhibit version transparency to achieve stability.
- ***Separation of State (SoS)***: Calling a processing function within another processing function must exhibit state-keeping to achieve stability.

NS aims to design evolvable software independent of the underlying technology. Nevertheless, a particular technology must be chosen when implementing the software and its components. For object-oriented programming languages, the following normalized elements have been proposed [7, pp. 363–398]. It is essential to recognize that different programming languages may necessitate alternative constructs [7, p. 364].

The following list describes each element using the definition from Mannaert *et al.* [12, p. 102]

- ***Data Element***: Based on DvT, data elements have "get" and "set" methods for wide-sense data version transparency or marshal -and parse- methods for strict-sense DvT. Supporting tasks can be added in a way that is consistent with the principles of SoC and DvT.
- ***Task Element***: Based on SoC, the core action entity can only contain a single functional task, not multiple tasks. Based on AvT, arguments and parameters must be encapsulated data entities. Based on SoC and SoS, workflows need to be separated from action entities and will therefore be encapsulated in a workflow element. Based on AvT, tasks need to be encapsulated so that a separate action entity wraps the action entities representing task versions. Supporting tasks can be added in a way that is consistent with SoC and AvT.
- ***Workflow Element***: Based on SoC, workflow elements cannot contain other functional tasks, as they are generally considered a separate change driver, often implemented in an external technology. Based on SoS, workflow elements must be stateful. This state is required for every instance of use of the action element and, therefore, needs to be part of, or linked to, the instance of the data element that serves as an argument.
- ***Connector Element***: Based on Theorem SoS, connector elements must ensure that external systems can interact with data elements, but that they cannot call an action element in a stateless way. Supporting tasks can be added in a way that consistent with SoC and AvT.
- ***Trigger Element***: Based on SoC, trigger elements need to control the separated —both error and non-errorstates, and check whether an action element has to be triggered. Supporting tasks can be added in a way that is consistent with SoC and AvT.

C. Clean Architecture

CA is a software design approach emphasizing code organization into independent, modular layers with distinct responsibilities. This approach aims to create a more flexible, maintainable, and testable software system by enforcing the separation of concerns and minimizing dependencies between components. CA aims to provide a solid foundation for software development, allowing developers to build applications that can adapt to changing requirements, scale effectively, and remain resilient against the introduction of bugs [6].

CA organizes its components into distinct layers. This architecture promotes the separation of concerns, maintainability, testability, and adaptability. The following list briefly describes each layer [6]. By organizing code into these layers and adhering to the principles of CA, developers can create more flexible, maintainable, and testable software with well-defined boundaries and a separation of concerns.

- ***Domain Layer***: This layer contains the application's core business objects, rules, and domain logic. Entities represent the fundamental concepts and relationships in the problem domain and are independent of any specific technology or framework. The domain layer focuses on encapsulating the essential complexity of the system and should be kept as pure as possible.
- ***Application Layer***: This layer contains the use cases or application-specific business rules orchestrating the interaction between entities and external systems. Use cases define the application's behavior regarding the actions users can perform and the expected outcomes. This layer coordinates the data flow between the domain layer and the presentation or infrastructure layers while remaining agnostic to the specifics of the user interface or external dependencies.
- ***Presentation Layer***: This layer translates data and interactions between the use cases and external actors, such as users or external systems. Interface adapters include controllers, view models, presenters, and data mappers, which handle user input, format data for display, and convert data between internal and external representations. The presentation layer should be as thin as possible, focusing on the mechanics of user interaction and deferring application logic to the use cases.

 ISBN: 978-1-68558-161-9

- ***Infrastructure Layer***: This layer contains the technical implementations of external systems and dependencies, such as databases, web services, file systems, or third party libraries. The infrastructure layer provides concrete implementations of the interfaces and abstractions defined in the other layers, allowing the core application to remain decoupled from specific technologies or frameworks. This layer is also responsible for configuration or initialization code to set up the system's runtime environment.

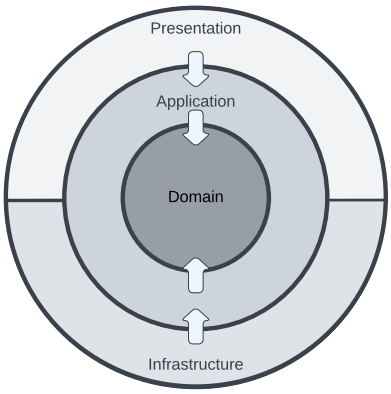


Figure 1. Flow of control

An essential aspect is described as the dependency rule. The rule states that *source code dependencies must point only inward toward higher-level policies* (Robert C. Martin, 2018, p. 206). This 'flow of control' is designed following the Dependency Inversion Principle (DIP) and can be represented schematically as concentric circles containing all the described components. The arrows in Figure 1 clearly show that the dependencies flow from the outer layers to the inner layers. Most outer layers are historically subjected to large-scale refactorings due to technological changes and innovation. Separating the layers and adhering to the dependency rule ensures that the domain logic can evolve independently from external dependencies or certain specific technologies.

Martin [6, p. 78] argues that software can quickly become a well-intended mess of bricks and building blocks without rigorous design principles. So, from the early 1980s, he began to assemble a set of software design principles as guidelines to create software structures that tolerate change and are easy to understand. The principles are intended to promote modular and component-level software structure [6, p. 79]. In 2004, the principles were established to form the acronym SOLID.

The following list will provide an overview of each of the SOLID principles.

- ***Single Responsibility Principle (SRP)***: This principle has undergone several iterations of the formal definition. The final definition of the Single Responsibility Principle (SRP) is: "a module should be responsible to one, and only one, actor" Martin [6, p. 82]. The word 'actor' in this statement refers to all the users and stakeholders represented by the (functional) requirements. The modularity concept in this definition is described by Martin [6, p. 82] as a cohesive set of functions and data structures. In conclusion, this principle allows for modules with multiple tasks as long as they cohesively belong together. Martin [6, p. 81] acknowledges the slightly inappropriate name of the principle, as many interpreted it, that a module should do just one thing.
- ***Open/Closed Principle (OCP)***: Meyer [13] first mentioned the OCP and formulated the following definition: *A module should be open for extension but closed for modification.* The software architecture should be designed such that the behavior of a module can be extended without modifying existing source code. The OCP promotes the use of abstraction and polymorphism to achieve this goal. The OCP is one of the driving forces behind the software architecture of systems, making it relatively easy to apply new requirements. [6, p. 94].
- ***Liskov Substitution Principle (LSP)***: The LSP is named after Barbara Liskov, who first introduced the principle in a paper she co-authored in 1987. Barbara Liskov wrote the following statement to define subtypes (Robert C. Martin, 2018, p. 95). *If for each object o1 of type S, there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.1.* Or in simpler terms: To build software from interchangeable parts, those parts must adhere to a contract that allows those parts to be substituted for another (Robert C. Martin, 2018, p. 80)
- ***Interface Segregation Principle (ISP)***: The ISP suggests that software components should have narrow, specific interfaces rather than broad, general-purpose ones. In addition, the ISP states that consumer code should not be allowed to depend on methods it does not use. In other words, interfaces should be designed to be as small and focused as possible, containing only the methods relevant to the consumer code using them. This allows the consumer code to use only the needed methods without being forced to implement or depend on unnecessary methods [6, p. 104].
- ***DIP***: The DIP prescribes that high-level modules should not depend on low-level modules, and both should depend on abstractions. The principle emphasizes that the architecture should be designed so that the flow of control between the different objects, layers, and components is always from higher-level implementations to lower-level details. In other words, high-level implementations, like business rules, should not be concerned about low-level implementations, such as how the data is stored or presented to the end user. Additionally, high-level and low-level implementations should only depend on abstractions or interfaces defining a contract for how they should interact [6, p. 91]. This approach allows for great flexibility and a modular architecture. Modifications in the low-level implementations will not affect the high-level implementations as long as they still adhere to the

 ISBN: 978-1-68558-161-9

contract defined by the abstractions and interfaces. Similarly, changes to the high-level modules will not affect the low-level modules as long as they still fulfill the contract. This reduces coupling and ensures the evolvability of the system over time, as changes can be made to specific modules without affecting the rest of the system.

Martin [6] proposes the following elements to achieve the goal of "Clean Architecture."

- ***Entities***: Entities are the core business objects, representing the domain's fundamental data.
- ***Interactor***: Interactors encapsulate business logic and represent specific actions that the system can perform.
- ***RequestModels***: RequestModels represent the input data required by a specific interactor.
- ***ResponseModel***: ResponseModel represents the output data required by a specific interactor.
- ***ViewModels***: ViewModels are responsible for managing the data and behavior of the user interface.
- ***Controllers***: Controllers are responsible for handling requests from the user interface and routing them to the appropriate Interactor.
- ***Presenters***: Presenters are responsible for formatting and the data for the user interface.
- ***Gateways***: A Gateway provides an abstraction layer between the application and its external dependencies, such as databases, web services, or other external systems.
- ***Boundary***: Boundaries are used to separate the different layers of the component.

III. The Analysis

This Section delves into the convergence of CA and NS, exploring their convergence and application in software design. The discussion is anchored in the results of the research "On the Convergence of Clean Architecture with the Normalized Systems Theorems" [1], which meticulously examines the principles and design elements of both CA and NS mentioned in previous chapters. By aligning the theoretical constructs of both paradigms, the thesis provides a perspective on achieving modular, evolvable, and stable software architectures. This convergence reinforces the robustness of software systems and enhances their evolvability and longevity in the face of future requirements. The subsequent sections will summarize the key components of their convergence by highlighting the practical implications and the potential for evolvable software design.

A. The converging principles

The main goal of both the SRP and SoC is to promote and encourage modularity, low coupling, and high cohesion. While their definitions have minor nuances, the two principles are practically interchangeable. Even though SRP does not implicitly guarantee DvT or AvT, it supports those theorems by directing design choices in a certain way. One example lies in separating data models for requests, responses, and views and respective versions of these models.

The OCP and its relation to NS theory emphasize the importance of designing software entities that are open for extension but closed for modification. This principle aligns with the NS approach to evolvability, advocating for structures that can adapt to new requirements without altering existing code, thus minimizing the impact of changes. An example of this synergy can be seen in the use of expanders within NS, which allow for introducing new functionality or data elements without disrupting the core architecture, cohesively supporting the OCP principle goal of extendibility and maintainability.

The LSP emphasizes that objects of a superclass should be replaceable with objects of a subclass without altering the correctness of the program. This principle strongly aligns with the emphasis on modular and replaceable components in NS, advocating for flexibility and the seamless integration of new functionalities. Applying this principle within NS is evident in designing tailored interfaces specific to a particular version. This ensures system evolution without compromising existing functionality, thereby upholding the LSP directive for substitutability and system integrity.

The ISP advocates for creating specific consumer interfaces rather than one general-purpose interface, aligning with NS principles to enhance system evolvability and maintainability. This alignment is evident in the modular and decoupled design strategies advocated by both NS and ISP, where the focus is on minimizing unnecessary dependencies and promoting high cohesion within systems. By applying ISP, developers can ensure that system components only depend on the interfaces they use, which mirrors the approach in NS to create evolvable systems by reducing the impact of changes across modules.

The DIP and its alignment with NS are centered on inverting the conventional dependency structure to reduce rigidity and fragility in software systems. DIP promotes high-level module independence from low-level modules by introducing abstractions that both can depend on, thereby facilitating a more modular and evolvable design. This principle mirrors the emphasis on minimizing dependencies to enhance system evolvability in the NS paradigm. Examples from the thesis demonstrate how leveraging DIP in conjunction with NS principles leads to systems that are more adaptable to change, showcasing the practical application of these combined approaches in achieving resilient software architectures. Designers should also be aware of the potential pitfalls of using DIP as faulty implementations can increase combinatorial effects.

In the following table, we summarize the analysis in a tabular overview using the following denotation:

- ***Strong convergence*** (✚✚): This indicates that the principles of CA and NS are highly converged. Both have a similar impact on the design and implementation.
- ***Supports convergence*** (✚): The CA principle supports implementing the NS principle through specific design choices. However, applying the CA principle does not inherently ensure adherence to the corresponding NS principle.
- ***Weak or no convergence*** (━): The principles have no significant similarities in terms of their purpose, goals, or architectural supports.

 ISBN: 978-1-68558-161-9

TABLE I
THE CONVERGENCE BETWEEN CA AND NS PRINCIPLES.

Clean Architecture	Normalized Systems	Separation Of Concerns	Data Version Transparency	Action Version Transparency	Separation of State
Single Responsibility Principle		++	+	+	−
Open/Closed Principle		++	−	++	−
Liskov Substitution Principle		++	−	+	−
Interface Segregation Principle		++	−	+	−
Dependency Inversion Principle		++	−	+	−

B. The converging elements

The Data Element from NS and the Entity Element from CA represent data objects of the ontology or data schema, typically including attributes and relationship information. While both can contain a complete set of attributes and relationships, the Data Element of NS may also be tailored to serve a specific set of information required for a single task or use case. In CA, these types of Data Elements are explicitly specified as ViewModels, RequestModels, or Response Models.

The Interactor element of CA and the Task and WorkFlow elements of NS are all responsible for encapsulating business rules. NS has a more strict approach to encapsulating the execution of business rules in Task Elements, as it is only allowed to have a single execution of a business rule. Additionally, the WorkFlow element is responsible for executing multiple tasks statefully and is highly convergable with the Interactor element of CA.

The convergence of the Controller element from CA with NS is highlighted by its partial interchangeability with the Connector and Trigger elements in NS. The Controller Element is primarily responsible for interaction using protocols and technologies involving the user interface, while the Connector and Trigger elements are also intended to interact with other types of external systems.

The Gateway element of CA and the Connector element of NS communicate between components by providing Data Version Transparent interfaces to provide Action Version Transparency between these components.

The Presenter is responsible for preparing the ViewModel on the controller's behalf and can be considered a Task or Workflow Element in the theories of NS.

The Boundary element of CA strongly converges with the Connector element of NS, as both are involved in communication between components and help ensure loose coupling between these components. However, the Boundary element's scope seems more specific, as this element usually separates architectural boundaries within the application or component.

In the following table, we summarize the analysis in a tabular overview using the same denotation used in Section III-A.

TABLE II
THE CONVERGENCE BETWEEN CA AND NS ELEMENTS.

Clean Architecture	Normalized Systems	Data Elements	Task Element	Flow Element	Connector Element	Trigger Element
Entity Element		++	−	−	−	−
Interactor Element		−	++	++	−	−
RequestModel Element		++	−	−	−	−
ResponseModel Element		++	−	−	−	−
ViewModel Element		++	−	−	−	−
Controller Element		−	−	−	+	+
Gateway Element		−	−	−	++	−
Presenter Element		−	+	+	−	−
Boundary Element		−	−	−	++	−

IV. CONCLUSION

The primary objective of G. Koks was to study the convergence between CA and NS by analyzing their principles and design elements through theory and practice. This Section will summarize the findings into a research conclusion.

Stability and evolvability are concepts not directly referenced in the literature on CA, but this design approach aligns with the goal of NS. The attentive reader can observe the shared emphasis on modularity and the separation of concerns, as all SOLID principles strongly converge with SoC. Both approaches attempt to achieve low coupling and high cohesion. In addition, CA adds the dimensions of dependency management as useful measures to improve maintainability by rigorously managing dependencies in the Software Architecture.

The DvT appears to be underrepresented in the SOLID principles of CA. DvT is primarily supported by the SRP of CA, as evidenced by ViewModels, RequestModels, ResponseModels, and Entities as software elements. It is worth noting that this application of Data Version Transparency is an integral part of the design elements of CA. While CA does address DvT through the SRP, a more comprehensive representation of the underlying idea of DvT within the principles of CA will likely improve the convergence of CA with NS.

CA Lacks a strong foundation for receiving external triggers in its design philosophy. This is partially represented by the Controller element. However, this element is described as being used for web-enabled environments and might result in a less comprehensive approach to receiving external triggers across various technologies or systems.

 ISBN: 978-1-68558-161-9

The most notable difference between CA and NS is their approach to handling state. CA does not explicitly address state management in its principles or design elements. NS Provides the principle of SoS, ensuring that state changes within a software system are stable and evolvable. This principle can be crucial in developing scalable and high-performance systems, as it isolates state changes from the rest of the system, reducing the impact of state-related dependencies and side effects.

The findings can only lead to the conclusion that the convergence between CA and NS is incomplete. Consequently, CA cannot fully ensure stable and evolvable software artifacts as NS has defined them.

While it has been demonstrated that the convergence between these two approaches is incomplete, combining both methodologies is highly beneficial for NS and CA for various reasons. The primary advantage of synergizing them lies in the complementary nature of both paradigms, where each approach provides strengths that can be leveraged to address a robust architectural design.

CA offers a well-defined, practical, and modular structure for software development. Its principles, such as SOLID, guide developers in creating maintainable, testable, and scalable systems. This architectural design approach is highly suitable for various applications and can be easily integrated with the theoretical foundations provided by NS. Conversely, the NS approach offers a more comprehensive theoretical understanding of achieving stable and evolvable systems.

To conclude, the popularity and widespread adoption of CA in the software development community can benefit NS. As more developers adopt CA, they become more familiar with NS and recognize their value to software design. Synergizing both approaches will likely lead to increased adoption of NS.

BIBLIOGRAPHY

[1] G. Koks, "On the Convergence of Clean Architecture with the Normalized Systems Theorems," en, Ph.D. dissertation, 2023-06. [Online]. Available: https://zenodo.org/record/8029971 (visited on 2024-03-24).

[2] D. McIlroy, "NATO Software Engineering Conference," en, 1968.

[3] M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980, ISSN: 0018-9219. DOI: 10.1109/PROC.1980.11805.

[4] E. Dijkstra, "Letters to the editor: Go to statement considered harmful," en, *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, 1968-03, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/362929.362947.

[5] D. Parnas, "On the criteria to be used in decomposing systems into modules," en, *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972-12, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/361598.361623. (visited on 2023-03-19).

[6] R. C. Martin, *Clean architecture: a craftsman's guide to software structure and design* (Robert C. Martin series). London, England: Prentice Hall, 2018, OCLC: on1004983973, ISBN: 978-0-13-449416-6.

[7] H. Mannaert, J. Verelst, and P. De Bruyn, *Normalized systems theory: from foundations for evolvable software toward a general theory for evolvable design*, eng. Kermt: nsi-Press powered bei Koppa, 2016, ISBN: 978-90-77160-09-1.

[8] P. Huysmans and J. Verelst, "Towards an Engineering-Based Research Approach for Enterprise Architecture: Lessons Learned from Normalized Systems Theory," en, in *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, vol. 8827, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2013, pp. 58–72, ISBN: 978-3-319-12567-1 978-3-319-12568-8. DOI: 10.1007/978-3-642-38490-5_5.

[9] G. Haerens, "On the evolvability of the TCP-IP based network firewall rule base," eng, PhD Thesis, Antwerp University, 2021. [Online]. Available: https://hdl.handle.net/10067/1834610151162165141 (visited on 2024-03-24).

[10] D. van Nuffel, *Towards designing modular and evolvable business processes*. University of Antwerp, 2011, ISBN: 90-8994-040-5.

[11] H. Mannaert and J. Verelst, *Normalized systems re-creating information technology based on laws for software evolvability*, English. Kermt: Koppa, 2009, OCLC: 1073467550, ISBN: 978-90-77160-00-8.

[12] H. Mannaert, J. Verelst, and K. Ven, "Towards evolvable software architectures based on systems theoretic stability," en, *Software: Practice and Experience*, vol. 42, no. 1, pp. 89–116, 2012-01, ISSN: 00380644. DOI: 10.1002/spe.1051.

[13] B. Meyer, *Object-oriented software construction*, 1st ed. Upper Saddle River, N.J: Prentice Hall PTR, 1988, ISBN: 978-0-13-629155-8.

 ISBN: 978-1-68558-161-9

Warm-Starting Patterns for Quantum Algorithms

Felix Truger, Johanna Barzen, Martin Beisel, Frank Leymann, and Vladimir Yussupov
Institute of Architecture of Application Systems, University of Stuttgart
Universitätsstraße 38, 70569 Stuttgart, Germany
email: {*firstname.lastname*}*@iaas.uni-stuttgart.de*

***Abstract*—Quantum computing promises considerable advantages in efficiency and accuracy over classical computing for certain problems. However, today's Noisy Intermediate-Scale Quantum (NISQ) computers are error-prone and limited in the number of qubits, which complicates leveraging them in practice. To mitigate these issues, multiple warm-starting techniques are being introduced in the quantum computing domain to improve the efficiency and accuracy of quantum algorithms by utilizing known or efficiently generated results as a starting point for the quantum computation. However, heterogeneous warm-starting techniques are often tailored for specific algorithms and require expertise in multiple domains, such as quantum computing and machine learning, thus complicating the choice of technique. Well-structured patterns that abstractly document proven solutions to recurring problems can help quantum software engineers in this decision-making process. In this work, we extend the existing pattern language for quantum algorithms with four novel warm-starting patterns that refine a more abstract pattern introduced in previous work and document how recurring problems in the design and execution of quantum algorithms can be solved with warm-starts. Thereby, the underlying methods are made available for interested parties in a concise and easily digestible manner.**

Keywords*—*Quantum Computing; Hybrid Algorithms; Quantum Software Engineering; Warm-Start; Patterns.

I. Introduction

On quantum computers, information is represented by the states of quantum bits (qubits), which possess unique properties, such as *superposition* and *entanglement*. Due to these properties, quantum computing promises advantages over classical computing for certain problems [1]. For example, factorization of composite numbers is theoretically feasible with the help of quantum computers, but not known to be tractable with classical computers [2]. Moreover, it has been shown that a significant speed-up over classical machine learning is possible in certain cases when utilizing quantum computers [3].

In the current *Noisy Intermediate-Scale Quantum (NISQ)* era, quantum computers offer a limited number of qubits that are prone to errors [4][5]. Therefore, quantum algorithms are limited to quantum circuits acting on few qubits and requiring only few operations. Moreover, many algorithms are designed as hybrid quantum-classical algorithms with the intention to utilize both classical and quantum computation in a fruitful combination that mitigates these current limitations. The most prominent examples are *Variational Quantum Algorithms (VQAs)* consisting of parameterized quantum circuits and a classical optimizer employed to search for viable parameter values for these circuits to solve a problem at hand [6]. Quantum algorithms can be further improved using so-called warm-starting techniques that utilize known or efficiently generated results as a starting point instead of starting from scratch.

However, *warm-starting* is an umbrella term for a heterogeneous set of techniques that affect quantum algorithms in fundamentally different ways and exhibit a multitude of properties and potential benefits [7]. For example, warm-starts can be realized by encoding information into a quantum algorithm's initial quantum state or a sophisticated parameter initialization. Techniques proposed in the literature are often specialized for specific algorithms and problems, which complicates reusing them or even deciding about their suitability for a certain use case. Moreover, they often require expertise in different domains, including quantum computing and machine learning.

Patterns document abstract solutions to recurring problems [8] and can help engineers understand and apply these solutions for their specific use case. To support quantum software engineers in better understanding the concepts, applicability, and benefits of different warm-starting techniques, we present four novel warm-starting patterns. With these patterns, we capture recurring solution strategies for warm-starting quantum algorithms and refine the more abstract warm-starting pattern that exists in the pattern language for quantum algorithms.

The remainder of the paper is structured as follows: We discuss related work in Section II, before fundamentals and the pattern format are introduced in Section III. Section IV introduces the four new warm-starting patterns in detail. In Section V, we discuss aspects of the application of the patterns and the evaluation of the warm-starts. Finally, Section VI concludes the paper with a summary and outlook.

II. Related Work

Leymann [9] proposed and initiated a pattern language for quantum algorithms. This pattern language has been continuously extended, e.g., with refined patterns for state preparation, hybrid quantum algorithms, error handling, and execution semantics [10]–[16]. In this work, we further extend it by documenting four novel patterns capturing different solutions for warm-starting quantum algorithms, which refine the existing, abstract Warm-Start pattern. These new patterns were identified through an analysis of quantum-related warm-starting techniques encountered in the literature (cf. Section III). To the best of our knowledge, there exist no other works documenting patterns in the quantum computing domain and conforming to Alexander et al.'s notion of patterns [8].

Pattern languages, originally known from architecture [8], have been documented for various other domains, e.g., for software engineering [17], enterprise integration [18], and cloud

 ISBN: 978-1-68558-161-9

computing [19]. Leymann and Barzen [20] propose Pattern Atlas, a repository and tool to visualize and link patterns of different pattern languages. Moreover, Falkenthal and Leymann [21] propose the concept of solution languages that interconnect concrete solutions for patterns, i.e., implementation artifacts, to systematically collect implementation knowledge and reduce the manual efforts of (re)implementing existing solutions. Such solutions are linked to the corresponding patterns and other solutions as per the relations in the pattern language.

Warm-starting techniques were proposed and examined in various previous works. Mari et al. [22] discuss and evaluate forms of quantum transfer learning, particularly different directions in which quantum transfer learning can be utilized in the context of *Quantum Neural Networks (QNNs)*. Egger et al. [23] and Tate et al. [24], respectively, describe and evaluate how classical approximation algorithms can be utilized in the *Quantum Approximate Optimization Algorithm (QAOA)*, while Galda et al. [25] and Shaydulin et al. [26] focus on transferring parameters across problem instances. Truger et al. [7] explore and analyze warm-starting techniques in the quantum computing domain in a literature study, thereby summarizing categories of such techniques. Beisel et al. [27] propose a workflow modeling extension to facilitate the integration and orchestrations of VQAs in workflows. This includes modeling constructs for warm-starting VQAs with initial parameter values and approximations incorporated into a biased initial state. However, none of these works formally document the warm-starting techniques as solutions to recurring problems in the form of patterns.

III. Fundamentals and Pattern Format

In this section, we discuss fundamentals of quantum algorithms and VQAs in particular. Moreover, we present the pattern format and authoring method used in this work.

A. Fundamentals of Quantum Algorithms

Quantum algorithms are implemented as quantum circuits describing manipulations of qubits similar to classical logic circuits. Quantum circuits consist of wires representing the underlying qubits and gates representing operations on the qubits. The number of wires is called the *width* of the circuit and the number of gates acting on a qubit determines the circuit *depth*. Gates can act on a single qubit or multiple qubits, e.g., the Hadamard gate (H) creates a superposition on a single qubit and the two-qubit controlled-not gate (CNOT) can be used to entangle or disentangle qubits. Some gates, such as the rotation gates RX, RY, and RZ, are parametrized, i.e., the intensity of the manipulation depends on parameter values set at runtime. Therefore, circuits can be parameterized as well and their output upon measurement depends on the parameter values. Such parameterized quantum circuits are the basis for VQAs, such as the QAOA [28], *Variational Quantum Eigensolvers (VQEs)* [29], and QNNs [30]. To determine viable values for the circuit parameters of VQAs, classical optimizers are employed. Quantum and classical execution are then executed in a loop, in which the output of the circuit run on a quantum device is evaluated for the optimizer to steer parameter values in a favorable direction. Once a termination condition is met, e.g., when the result has converged or a set time limit has expired, the circuit can be executed with the final set of optimized parameter values to retrieve the result of the overall quantum-classical algorithm. This way, the aforementioned QAOA can be used to approximate solutions to combinatorial optimization problems or VQEs can be used to approximate eigenvalues with the help of a quantum computer. More generally, QNNs can be trained to compute arbitrary functions for various purposes, e.g., classification or regression [30].

B. Pattern Format and Authoring Method

We follow the pattern format from previous work on quantum computing patterns [9]–[16] and rely on best practices for pattern writing [17]–[19]. Each pattern is identified by its *Name* and an *Icon* that serves as a mnemonic. The *Problem* targeted by the pattern is highlighted with a brief question. Known alternative names are optionally listed as *Aliases*. Afterward, the *Context* in which the pattern is applicable, i.e., the situation in which the problem may arise, is explained. Next, *Forces* that need to be considered when solving the problem are described. Then, we elaborate on the high-level *Solution* that is additionally illustrated by a *Solution Sketch*. In the *Results* paragraph, we discuss the consequences of the solution. Afterward, we draw connections between the new pattern and *Related Patterns*, before we summarize *Known Uses* by listing implementations of the pattern.

As patterns are abstractions of existing solutions, the patterns in this work were identified by exploring warm-starting techniques proposed and used in the literature. In previous work, we conducted a systematic mapping study to survey scientific literature on warm-starting techniques in the quantum computing domain in general, thereby identifying different warm-starting techniques [7]. Recurring approaches that are regarded promising were further analyzed, and the underlying solutions were abstracted and documented as patterns.

IV. Warm-Starting Patterns for Quantum Algorithms

In this section, we first give an overview of the patterns introduced in this work and align them w.r.t. the existing patterns for quantum algorithms. Afterward, we document the four novel warm-starting patterns for quantum algorithms.

A. Pattern Language for Quantum Algorithms

Figure 1 provides an overview of the pattern language for quantum algorithms proposed and initialized by Leymann [9] with its essential patterns for quantum states, unitary transformations, and the program flow of quantum algorithms. It aims to support scientists and software developers in building quantum algorithms. Weigold et al. [10] [11] extended the pattern language with state preparation patterns for quantum algorithms focusing on how data can be encoded in quantum algorithms. Also, additional patterns for the program flow of hybrid algorithms were documented [12].

 ISBN: 978-1-68558-161-9

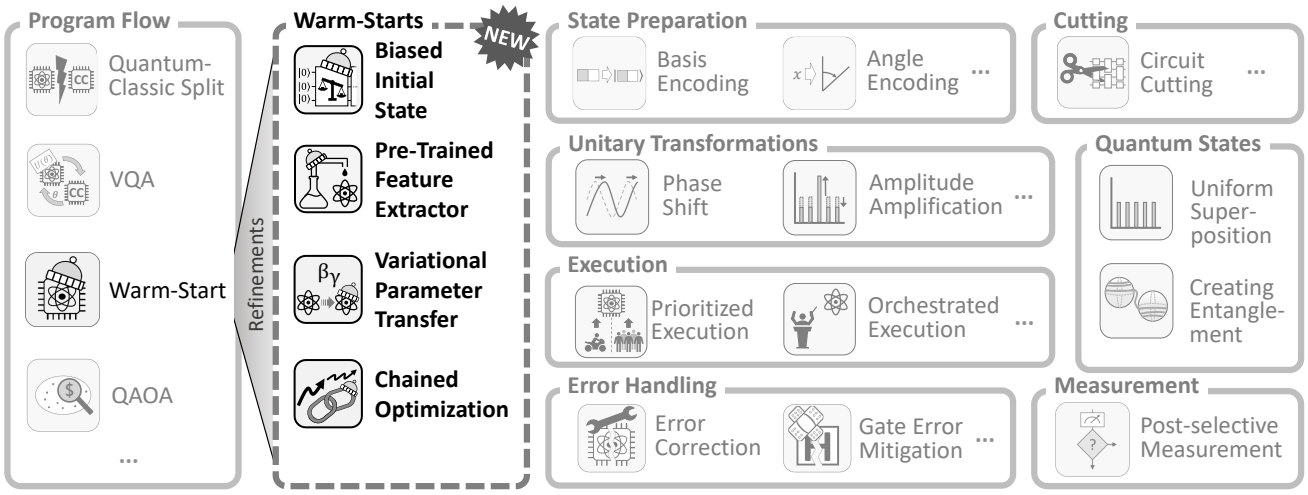


Figure 1. Overview of the pattern language for quantum algorithms, including the newly documented warm-starting patterns highlighted in bold.

Particularly, Weigold et al. [12] identified warm-starting as a general pattern applicable to quantum algorithms, which we aim to refine in this work with more concrete recurring solutions in that sense. Beisel et al. [13] describe patterns for quantum error handling and Georg et al. [14] document patterns for the execution of quantum applications. Moreover, patterns for the partitioning of quantum circuits, i.e., *circuit cutting*, have been introduced by Bechtold et al. [15].

B. Warm-Starting Techniques for Quantum Algorithms

The WARM-START pattern identified by Weigold et al. refines the more general QUANTUM-CLASSIC SPLIT pattern, which summarizes splitting of computational workload between quantum and classical computers [9][12]. In this sense, it suggests to use classical methods to approximate a solution to the problem at hand and utilize the approximation as a starting point. As shown in Figure 1, the new patterns presented in this work further refine the WARM-START pattern.

In our previous work [7], we identified different properties of warm-starting techniques, e.g., warm-starts can be applied in different directions, i.e., classical-to-quantum (C2Q), quantum-to-quantum (Q2Q), and quantum-to-classical (Q2C) [22]. Since this work focuses on warm-starting patterns *for* quantum algorithms in line with the pattern language, only C2Q and Q2Q cases are considered in the following.

C. Biased Initial State

Problem: How to utilize efficient approximations in quantum algorithms to improve the solution quality or speed up the computation?

Context: For many computationally hard problems, efficient approximation algorithms exist. However, typical quantum algorithms neglect these approximations and valuable information remains unused as the quantum algorithm starts from a neutral position. As a result, deep quantum circuits may be required, which increases accumulative error rates, and more quantum resources may be required to solve a problem.
Forces: Moreover, current quantum devices are error-prone, thus, the depth of executable quantum circuits is limited. However, including approximations requires special care, as it can limit the quantum algorithm in an unintended way [31][32]. Also, changing the initial state may require additional adaptations of corresponding parts of the quantum circuit [23][33].
Solution: Encode approximations into the initial state of quantum circuits, thereby biasing the initial quantum state towards viable solutions. Hence, a chain of algorithm executions as depicted in Figure 2 is beneficial: First, an efficient algorithm is utilized to approximate a solution of a given problem instance. This can often be achieved at low cost on classical hardware. Then, the initial state $|\psi\rangle$ of the subsequent quantum algorithm is biased toward the approximation and the algorithm is

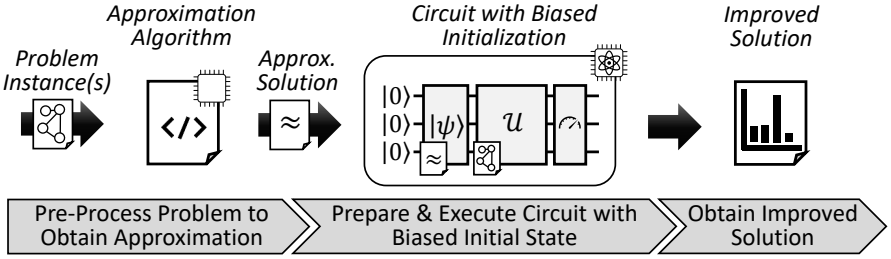


Figure 2. Solution sketch for the BIASED INITIAL STATE pattern

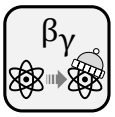
executed on a quantum device to obtain an improved solution.
Result: The quantum algorithm employed in the second step utilizes the approximation as a starting point to improve upon. Due to the biased initial state, optimal solutions can be explored quicker and the solution quality achievable in a set amount of time may therefore increase. Moreover, this way the workload of the overall computation can be distributed to multiple devices, e.g., classical and quantum devices.
Related Patterns: This pattern is a refinement of the WARM-START pattern and related to the state preparation patterns, e.g., ANGLE ENCODING, since different encodings may be applied to prepare and bias the initial state of a quantum algorithm [11][12]. Moreover, it can be applied with the VQA pattern and its refinements, such as the QAOA [12].
Known Uses: Egger et al. [23] introduce a biased initial state for QAOA and the Maximum Cut problem (MaxCut) utilizing the classical Goemans-Williamson approximation algorithm. Similarly, Tate et al. [24] adapt QAOA for MaxCut with a Burer-Monteiro relaxation of the problem. QAOA was also adapted for a biased initial state for the Knapsack problem [34]. Wang [35] proposes a "classically-boosted" quantum algorithm for the Maximum 3-Satisfiability and Maximum Bisection problems based on biased initial states. Beisel et al. [27] propose a workflow modeling construct facilitating the integration of warm-starts via biased initial states in VQAs.

D. Pre-Trained Feature Extractor

Problem: How to process large data items through QNNs when the number of available qubits is lower than the size of a data item?

Aliases: QUANTUM TRANSFER LEARNING [22]
Context: A QNN shall be trained for a specific task, that requires the processing of large data items, e.g., images or multi-dimensional data. However, the number of qubits required to load such data items into the QNN is larger than the number of qubits of the available quantum devices.
Forces: The width of circuits implementing QNNs is limited by the number of available qubits. In addition, quantum devices are scarce resources that should be utilized as efficiently as possible. However, naively reducing the original data items may result in the loss of information relevant for the computation. Large pre-trained classical models for various general tasks, such as object recognition for images, are widely available or can be created at low cost.
Solution: Use a pre-trained classical model to reduce the dimensions of the data items and train the QNN based on the reduced data. As shown in Figure 3, a pre-trained classical model for a wide range purpose, such as a neural network trained for object recognition, can be utilized for a hybrid QNN to be trained for a related special purpose task. Intermediate values of inputs processed through such models, e.g., those present at a condensed next-to-last neural network layer, can be seen as a compressed representation of the original data exhibiting its most significant features. Thus, the pre-trained model serves as a feature extractor. These features can be encoded into a quantum state to train the QNN for the target task.
Result: Due to the compressed representation obtained from the pre-trained feature extractor, fewer qubits are required to process data in the QNN. Furthermore, the compressed nature of the data may reduce the QNN's training time, as irrelevant information has already been omitted from the training data.
Related Patterns: This pattern refines the WARM-START pattern and is related to the state preparation patterns, e.g., ANGLE ENCODING, [11][12]. Different encodings may be applied to encode the extracted features into a quantum state. It is typically applied in conjunction with QNNs, a form of VQA [6]. Furthermore, the CIRCUIT CUTTING pattern solves a similar problem by partitioning the computation of a large quantum circuit into computations of multiple smaller circuits [15].
Known Uses: PRE-TRAINED FEATURE EXTRACTOR is frequently used when image processing, particularly image classification, shall be enhanced with QNNs [22] [36]–[41]. It was also applied for text classification [42]. Moreover, autoencoders [43] can be considered a special case of PRE-TRAINED FEATURE EXTRACTOR, that are designed and trained specifically for the purpose of data compression.

E. Variational Parameter Transfer



Problem: How to obtain a problem-aware parameter initialization for VQAs that reduces the optimization runtime?

Context: A VQA needs to be executed on a quantum device, which encompasses the optimization of its variational parameters. Parameter optimization requires repeated access to the quantum device, typically starting with random initial parameter values [44], to sample solutions and determine a direction for their optimization, e.g., through gradient descent.

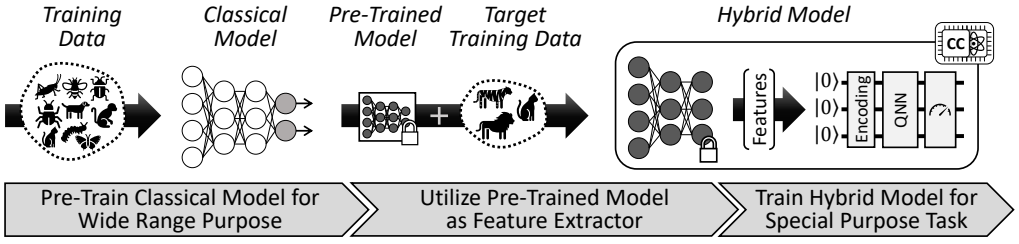


Figure 3. Solution sketch for the PRE-TRAINED FEATURE EXTRACTOR pattern

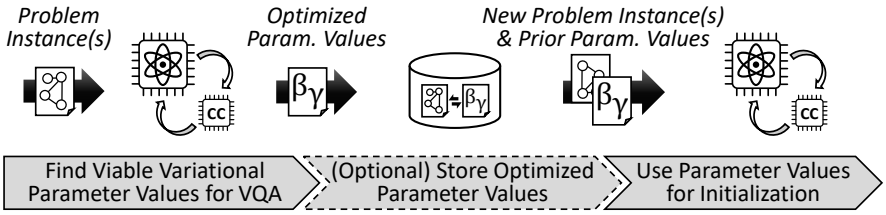


Figure 4. Solution sketch for the VARIATIONAL PARAMETER TRANSFER pattern

Forces: Obtaining viable parameter initializations for VQA is challenging due to large parameter spaces and effects, such as barren plateaus [45] and non-convex optimization landscapes [46]. Barren plateaus are areas with vanishing gradients in a cost function's parameter space that must be avoided, whereas local minima in non-convex optimization landscapes pose an additional challenge to efficient parameter initialization as they disturb the search for a global optimum.

Solution: Transfer viable variational parameter values from related problem instances. As shown in Figure 4, optimized parameter values may be stored or directly reused for new problem instances. In many cases, it can be expected that optimized parameter values for a solved problem instance are in proximity of viable parameter values for a related or similar new problem instance. Therefore, optimized parameter values from earlier executions may be utilized for a problem-aware parameter initializationinstead of a random initialization. Appropriate databases, toolkits, and provenance systems for quantum computing [47][48] facilitate the optional storage of optimized parameter values for their utilization in later executions.

Result: Parameter transfers can reduce the number of iterations of the optimization loop. A favorable parameter initialization can also increase the likelihood of finding globally optimal parameter values and thus increase the solution quality.

Related Patterns: This pattern is a refinement of the WARM-START pattern and can be applied in conjunction with VQA, including its refinements like QAOA, [12].

Known Uses: VARIATIONAL PARAMETER TRANSFER has been frequently proposed and applied for QAOA and Max-Cut [25][26][49][50]. Moreover, Shaydulin et al.'s repository of preoptimized parameters implements the storage option [47]. Beisel et al. [27] propose a modeling construct for workflows to integrate warm-starts via parameter initialization in VQAs.

F. Chained Optimization

Problem: How to avoid local optima and improve convergence when optimizing variational parameter values for VQAs?

Context: Optimal variational parameter values for a VQA need to be determined. The performance of the algorithm depends heavily on these values and a global optimum in the parameter space is needed to obtain optimal solutions.
Forces: Local minima in non-convex optimization landscapes and barren plateaus hinder the optimization, as the optimizer may be unable to reach a global optimum. Moreover, evaluating all possible parameter values is infeasibly expensive.
Solution: Chain different optimizers with different scopes or strengths together. As indicated in Figure 5, a global optimization strategy can be combined with a subsequent local optimizer. The former would determine a general area of interest in the overall optimization landscape. Afterward, the local optimizer is started from a point in this area of interest and searches on a smaller scale, aiming to find the global optimum.
Result: By chaining optimizers, the subsequent optimizers utilize previously obtained results as starting points to improve upon. Thereby, optimizers are combined to benefit from their respective strengths and achieve cost-efficient optimization.
Related Patterns: This pattern refines the WARM-START pattern and can be applied in conjunction with VQA, including QAOA, [12]. It is similar to the VARIATIONAL PARAMETER TRANSFER pattern documented above, with an unaltered problem instance, while the algorithm in use, specifically the optimization algorithm, is exchanged instead.
Known Uses: Rad et al. [51] use this method to avoid barren plateaus in VQAs. Tao et al. [52] apply it in a QNN optimization. Wauters et al. [53] supplement their Reinforcement Learning-based optimization approach for QAOA with subsequent gradient-based local optimization.

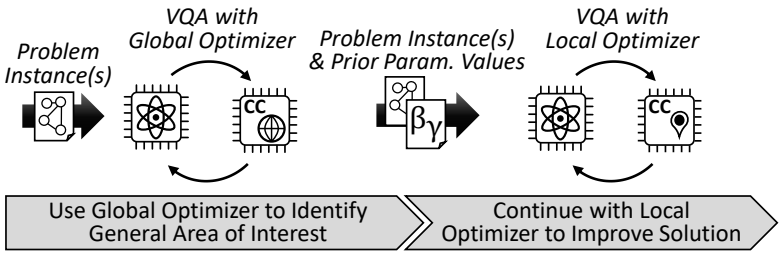


Figure 5. Solution sketch for the CHAINED OPTIMIZATION pattern

V. Discussion

We discuss known and potential challenges and limitations, and evaluation criteria for the application of the patterns above.

The dependency of concrete warm-starting solutions on different problem-specific factors, such as the nature of the quantum algorithm and problem at hand, available approximation algorithms, and feasible quantum state preparation procedures, can complicate the pattern application. In particular, the Biased Initial State and Pre-Trained Feature Extractor patterns require the determination of suitable techniques for obtaining and incorporating starting points on a case-by-case basis. Moreover, it was shown that the success of warm-starts through a biased initial state can depend on the careful selection of approach-specific hyperparameters [23][54]. In addition, such warm-starts can unintentionally prevent improvements as was shown, for example, for a warm-started variant of the QAOA where replacing the initial uniform superposition with the encoding of a good solution fails with little to no improvements [31]. Furthermore, applying biased initial states can impose restrictions on the parameterized quantum circuit in VQAs and some state preparations are not feasible on current NISQ hardware [32]. More specifically, some circuit designs complicate or prevent retaining the solution quality associated with the encoded biased initial state. These challenges and limitations likely also apply to the Pre-Trained Feature Extractor pattern, which likewise requires encoding information in the initial state. However, it was shown for the Biased Initial State that both problems can be avoided in some cases by transforming the initial state into a parameter transfer in VQAs [32]. Incorporating starting points for the parameter-focused Variational Parameter Transfer and Chained Optimization patterns is trivial since it reduces to a parameter initialization. Nonetheless, these warm-starts via parameter initializations could also potentially restrict the subsequent optimization in an undesired way when applied improperly, especially by limiting the optimization to an unfavourable area of the parameter space.

Also the evaluation of warm-starting techniques is challenging, as it is problem-specific and likewise dependent on different factors, such as available approximations and state preparation procedures. As different warm-starting methods aim to improve upon different behaviours, e.g., a reduced need for quantum computational resources, reduced runtime, or increased accuracy (cf. [7]), different approaches and metrics are required for analyzing and comparing them. Moreover, in the case of hybrid warm-starts, the trade-off between classical and quantum computational efforts may be ambiguous and dependent on the use case and concrete resources at hand.

The broad spectrum of potential applications of the warm-starting patterns introduced in this work may become even more extensive when considering warm-starts in other contexts outside of the quantum computing domain. It is conceivable that some techniques are analogously applicable in similar contexts of classical computing and, particularly, the classical domains of machine learning and optimization (cf. [7]).

VI. Conclusion

In this work, we elaborated on warm-starting techniques for quantum algorithms. We documented four novel patterns, Biased Initial State, Pre-Trained Feature Extractor, Variational Parameter Transfer, and Chained Optimization, thereby expanding the existing pattern language for quantum algorithms and refining the Warm-Start pattern. By documenting and making the knowledge on these solutions to recurring problems easily accessible for interested parties, we hope to assist quantum software engineers in utilizing warm-starting techniques in their applications.

In future work, we aim to analyze additional warm-starting techniques and their implementations to evaluate their compatibility with each other. Moreover, we will incorporate the warm-starting patterns presented in this work into the publicly available Pattern Atlas on the PlanQK platform [55], where also the other patterns of the pattern language for quantum algorithms have been incorporated. The accessibility for a broad audience enables refinement of the patterns based on community feedback. Moreover, the platform facilitates linking related patters together, even across different pattern languages.

ACKNOWLEDGEMENT

This work was partially funded by the BMWK projects *EniQmA* (01MQ22007B) and *SeQuenC* (01MQ22009B).

References

[1] J. Preskill, "Quantum computing and the entanglement frontier," *arXiv:1203.5813*, 2012.

[2] P. Shor, "Algorithms for Quantum Computation: Discrete Logarithms and Factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134.

[3] Y. Liu, S. Arunachalam, and K. Temme, "A rigorous and robust quantum speed-up in supervised machine learning," *Nature Physics*, vol. 17, no. 9, pp. 1013–1017, 2021.

[4] J. Preskill, "Quantum Computing in the NISQ era and beyond," *Quantum*, vol. 2, p. 79, 2018.

[5] F. Leymann and J. Barzen, "The bitter truth about gate-based quantum algorithms in the NISQ era," *Quantum Science and Technology*, vol. 5, no. 4, p. 044007, 2020.

[6] M. Cerezo *et al.*, "Variational quantum algorithms," *Nature Reviews Physics*, vol. 3, no. 9, pp. 625–644, 2021.

[7] F. Truger *et al.*, "Warm-Starting and Quantum Computing: A Systematic Mapping Study," *arXiv:2303.06133*, 2023.

[8] C. Alexander *et al.*, *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[9] F. Leymann, "Towards a Pattern Language for Quantum Algorithms," in *First International Workshop, QTOP 2019, Munich, Germany, March 18, 2019, Proceedings*. Springer, 2019, pp. 95–101.

[10] M. Weigold, J. Barzen, F. Leymann, and M. Salm, "Data Encoding Patterns For Quantum Algorithms," in *Proceedings of the 27th Conference on Pattern Languages of Programs (PLoP)*. HILLSIDE, 2020, pp. 1–11.

[11] ——, "Expanding Data Encoding Patterns For Quantum Algorithms," in *2021 IEEE 18th International Conference on Software Architecture Companion (ICSA-C)*, 2021, pp. 95–101.

[12] M. Weigold, J. Barzen, F. Leymann, and D. Vietz, "Patterns for Hybrid Quantum Algorithms," in *Proceedings of the 15th Symposium and Summer School on Service-Oriented Computing (SummerSOC)*. Springer International Publishing, 2021, pp. 34–51.

[13] M. Beisel *et al.*, "Patterns for Quantum Error Handling," in *Proceedings of the 14th International Conference on Pervasive Patterns and Applications (PATTERNS)*. Xpert Publishing Services (XPS), 2022, pp. 22–30.

[14] D. Georg *et al.*, "Execution Patterns for Quantum Applications," in *Proceedings of the 18th International Conference on Software Technologies (ICSOFT)*. SciTePress, 2023, pp. 258–268.

 ISBN: 978-1-68558-161-9

[15] M. Bechtold, J. Barzen, M. Beisel, F. Leymann, and B. Weder, "Patterns for Quantum Circuit Cutting," in *Proceedings of the 30th Conference on Pattern Languages of Programs (PLoP)*. HILLSIDE, 2023, [in press].

[16] F. Bühler *et al.*, "Patterns for Quantum Software Development," in *Proceedings of the 15th International Conference on Pervasive Patterns and Applications (PATTERNS)*. Xpert Publishing Services (XPS), 2023, pp. 30–39.

[17] J. O. Coplien, *Software Patterns*, ser. SIGS management briefings. SIGS Books & Multimedia, 1996.

[18] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2004.

[19] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014.

[20] F. Leymann and J. Barzen, "Pattern Atlas," in *Next-Gen Digital Services. A Retrospective and Roadmap for Service Computing of the Future: Essays Dedicated to Michael Papazoglou on the Occasion of His 65th Birthday and His Retirement*. Cham: Springer International Publishing, 2021, pp. 67–76.

[21] M. Falkenthal and F. Leymann, "Easing Pattern Application by Means of Solution Languages," in *Proceedings of the 9th International Conference on Pervasive Patterns and Applications (PATTERNS)*, 2017, pp. 58–64.

[22] A. Mari, T. R. Bromley, J. Izaac, M. Schuld, and N. Killoran, "Transfer learning in hybrid classical-quantum neural networks," *Quantum*, vol. 4, p. 340, 2020.

[23] D. J. Egger, J. Mareček, and S. Woerner, "Warm-starting quantum optimization," *Quantum*, vol. 5, p. 479, 2021.

[24] R. Tate, M. Farhadi, C. Herold, G. Mohler, and S. Gupta, "Bridging classical and quantum with SDP initialized warm-starts for QAOA," *ACM Transactions on Quantum Computing*, vol. 4, no. 2, pp. 1–39, 2023.

[25] A. Galda, X. Liu, D. Lykov, Y. Alexeev, and I. Safro, "Transferability of optimal QAOA parameters between random graphs," in *IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE, 2021, pp. 171–180.

[26] R. Shaydulin, P. C. Lotshaw, J. Larson, J. Ostrowski, and T. S. Humble, "Parameter Transfer for Quantum Approximate Optimization of Weighted MaxCut," *ACM Transactions on Quantum Computing*, vol. 4, no. 3, pp. 1–15, 2023.

[27] M. Beisel *et al.*, "QuantME4VQA: Modeling and Executing Variational Quantum Algorithms Using Workflows," in *Proceedings of the 13th International Conference on Cloud Computing and Services Science (CLOSER)*. SciTePress, 2023, pp. 306–315.

[28] E. Farhi, J. Goldstone, and S. Gutmann, "A Quantum Approximate Optimization Algorithm," *arXiv:1411.4028*, 2014.

[29] A. Peruzzo *et al.*, "A variational eigenvalue solver on a photonic quantum processor," *Nature communications*, vol. 5, no. 1, 2014.

[30] E. Farhi and H. Neven, "Classification with Quantum Neural Networks on Near Term Processors," *arXiv:1802.06002*, 2018.

[31] M. Cain, E. Farhi, S. Gutmann, D. Ranard, and E. Tang, "The QAOA gets stuck starting from a good classical string," *arXiv:2207.05089*, 2022.

[32] F. Truger, J. Barzen, F. Leymann, and J. Obst, "Warm-Starting the VQE with Approximate Complex Amplitude Encoding," *arXiv:2402.17378*, 2024.

[33] R. Tate, J. Moondra, B. Gard, G. Mohler, and S. Gupta, "Warm-Started QAOA with Custom Mixers Provably Converges and Computationally Beats Goemans-Williamson's Max-Cut at Low Circuit Depths," *Quantum*, vol. 7, p. 1121, 2023.

[34] W. van Dam, K. Eldefrawy, N. Genise, and N. Parham, "Quantum Optimization Heuristics with an Application to Knapsack Problems," in *IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE, 2021, pp. 160–170.

[35] G. Wang, "Classically-Boosted Quantum Optimization Algorithm," *arXiv:2203.13936*, 2022.

[36] S. Mittal and S. K. Dana, "Gender Recognition from Facial Images using Hybrid Classical-Quantum Neural Network," in *IEEE Students Conference on Engineering & Systems (SCES)*. IEEE, 2020, pp. 1–6.

[37] A. Gokhale, M. B. Pande, and D. Pramod, "Implementation of a quantum transfer learning approach to image splicing detection," *International Journal of Quantum Information*, vol. 18, no. 05, p. 2050024, 2020.

[38] V. Azevedo, C. Silva, and I. Dutra, "Quantum transfer learning for breast cancer detection," *Quantum Machine Intelligence*, vol. 4, no. 1, p. 5, 2022.

[39] M. J. Umer *et al.*, "An integrated framework for COVID-19 classification based on classical and quantum transfer learning from a chest radiograph," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 20, p. e6434, 2022.

[40] T. Kanimozhi, S. Sridevi, T. S. Manikumar, T. Dheeraj, and A. Sumanth, "Brain Tumor Recognition based on Classical to Quantum Transfer Learning," in *International Conference on Innovative Trends in Information Technology (ICITIIT)*. IEEE, 2022, pp. 1–5.

[41] A. Furutanpey *et al.*, "Architectural Vision for Quantum Computing in the Edge-Cloud Continuum," in *IEEE International Conference on Quantum Software (QSW)*. IEEE, 2023, pp. 88–103.

[42] C.-H. H. Yang, J. Qi, S. Y.-C. Chen, Y. Tsao, and P.-Y. Chen, "When BERT Meets Quantum Temporal Convolution Learning for Text Classification in Heterogeneous Computing," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2022, pp. 8602–8606.

[43] M. A. Kramer, "Nonlinear principal component analysis using autoassociative neural networks," *AIChE Journal*, vol. 37, no. 2, pp. 233–243, 1991.

[44] A. Kulshrestha and I. Safro, "BEINIT: Avoiding Barren Plateaus in Variational Quantum Algorithms," in *IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE, 2022, pp. 197–203.

[45] M. Cerezo, A. Sone, T. Volkoff, L. Cincio, and P. J. Coles, "Cost function dependent barren plateaus in shallow parametrized quantum circuits," *Nature communications*, vol. 12, no. 1, p. 1791, 2021.

[46] P. Huembeli and A. Dauphin, "Characterizing the loss landscape of variational quantum circuits," *Quantum Science and Technology*, vol. 6, no. 2, p. 025011, 2021.

[47] R. Shaydulin, K. Marwaha, J. Wurtz, and P. C. Lotshaw, "QAOAKit: A Toolkit for Reproducible Study, Application, and Verification of the QAOA," in *IEEE/ACM 2nd International Workshop on Quantum Computing Software (QCS)*. IEEE, 2021, pp. 64–71.

[48] B. Weder, J. Barzen, F. Leymann, M. Salm, and K. Wild, "QProv: A provenance system for quantum computing," *IET Quantum Communication*, vol. 2, no. 4, pp. 171–181, 2021.

[49] F. G. Brandao, M. Broughton, E. Farhi, S. Gutmann, and H. Neven, "For Fixed Control Parameters the Quantum Approximate Optimization Algorithm's Objective Function Value Concentrates for Typical Instances," *arXiv:1812.04170*, 2018.

[50] J. Wurtz and D. Lykov, "Fixed-angle conjectures for the quantum approximate optimization algorithm on regular MaxCut graphs," *Phys. Rev. A*, vol. 104, p. 052419, 2021.

[51] A. Rad, A. Seif, and N. M. Linke, "Surviving The Barren Plateau in Variational Quantum Circuits with Bayesian Learning Initialization," *arXiv:2203.02464*, 2022.

[52] Z. Tao, J. Wu, Q. Xia, and Q. Li, "LAWS: Look Around and Warm-Start Natural Gradient Descent for Quantum Neural Networks," in *IEEE International Conference on Quantum Software (QSW)*. IEEE, 2023, pp. 76–82.

[53] M. M. Wauters, E. Panizon, G. B. Mbeng, and G. E. Santoro, "Reinforcement-learning-assisted quantum optimization," *Physical Review Research*, vol. 2, no. 3, p. 033446, 2020.

[54] F. Truger, M. Beisel, J. Barzen, F. Leymann, and V. Yussupov, "Selection and Optimization of Hyperparameters in Warm-Started Quantum Optimization for the MaxCut Problem," *Electronics*, vol. 11, no. 7, 2022.

[55] PlanQK, "PlanQK - Pattern Atlas," https://patterns.platform.planqk.de, 2023, [accessed: 2024.02.28].

 ISBN: 978-1-68558-161-9