FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ADAPTING ENTERPRISE ENGINEERING AND NORMALISED SYSTEMS THEORIES TO DEVELOP A METHODICAL FRAMEWORK SUPPORTING TECHNOLOGY TRANSITIONS

by

Mgr. ONDŘEJ DVOŘÁK

A dissertation thesis submitted to
the Faculty of Information Technology, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

Dissertation degree study programme: Informatics

Department of Software Engineering

Prague, August 2021

Supervisor:
doc. Ing. ROBERT PERGL, Ph.D.
Department of Software Engineering
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Prague 6
Czech Republic

Abstract and contributions

Moore's law states that the number of transistors on a chip will double every two years. A similar force appears to drive the progress of information technology. Companies tend to struggle to keep up with the latest technological developments, and software solutions are becoming increasingly outdated. The ability for software to change easily is defined as evolvability. Consequently, software evolvability influences the ability of organisations to compete and thrive in the digital age by quickly responding to market changes and emerging opportunities. This capability is known as business agility.

One of the major fields researching evolvability is Enterprise Engineering (EE). The EE research paradigm applies theories from other fields to the evolvability of organisations. We argue that such theories can be applied to Software Engineering (SE) as well, which can contribute to the construction of software with a clear separation of dynamically changing technologies based on a relatively stable description of functions required for a specific user.

In this dissertation thesis, we present our research journey towards designing a methodical framework aiming at better technology transitions. This methodical framework is based on EE notions such as function, construction, and affordance. We reify them in terms of SE. Based on this reification, we propose affordance-driven assembling (ADA) as a software design approach that can aid in the construction of more evolvable software solutions. We exemplify the implementation of ADA in a case study of a commercial system and measure its effectiveness in terms of the impact of changes, as defined by the normalised systems theory.

We employed a design science research methodology. Therefore, we continuously developed, evaluated, and published design artefacts along the way. Although ADA is the result of our journey, it is not the only contribution of this dissertation thesis. The particular design artefact also contribute to an understanding of:

1. How EE theories may influence interactions between people and technology.

2. A phenomenon of flexibility-usability trade-off.

3. Bridging technology gap using robotic process automation.
4. Evolvability of financial models.
5. Architectural concepts limiting transitions between frameworks for graphical user interfaces.

Keywords:
evolvability, agility, technology, affordances.

Acknowledgements

This thesis is about transforming systems to cope with rapid technology changes. However, I strongly believe that doing a Ph.D. is also a transformation journey. Although I started this journey by focusing on technology, I realised its value in the context of something bigger – business agility – the ability of organisations to thrive with uncertainty of today's unprecedented market. This opinion was supported by observing how organisations fought for their existence during COVID crisis. I realized that the ability to adapt to new things (including technology) is much more crucial than being able to build things from scratch. Therefore, regardless how tough it was to finish the thesis during this COVID outbreak, I felt committed to completing my contribution to this problematics.

First of all, I would like to express my gratitude to my supervisor, doc. Ing. Robert Pergl, Ph.D. I would also like to thank to prof. Dr. Ing. Petr Kroha, CSc. for giving me valuable advises throughout my research, and co-authoring many of my articles.

Special thanks go to the staff of the Department of Software Engineering, who maintained a pleasant and flexible environment for my research. I would like to express special thanks to the department management for providing most of the funding for my research. My research has also been partially supported by the Ministry of Education, Youth, and Sport of the Czech Republic, by the Grant Agency of the Czech Technical University in Prague, SGS15 grant no. 118/OHK3/1T/18, SGS16 grant no. 120/OHK3/1T/18, and SGS17 grant no. 11/OHK3/3T/18.

I would like to express thanks to my colleagues from CCMi group, namely Ing. Marek Skotnica, Ing. David Šenkýř, and others, for their valuable comments and proofreading throughout my research. Also, I do really appreciate the outcomes of my students who significantly contributed to this dissertation thesis with their master and bachelor theses.

Next big thanks go to all my colleagues from company COPS who gave me a confidence and support while working on this dissertation thesis. Finally, there are no words I could possibly write to articulate my gratitude to my parents, Táňa and Přemek, and to my beloved wife Klára, and sons Kryštof and Marek. This thesis is dedicated to them, for everything they taught me.

Dedication

To my patient family.

Contents

List of Figures

Glossary

BETA theory Binding Essence, Technology and Architecture, EE theory. 30, 36, 65, 119, 156–158, 162

Camunda A vendor of BPM solutions. 73, 120

COPS A software development company specialised in financial management. 80, 84, 89, 129, 156, 167, 173, 177, 179, 182

Corima Treasury management system developed by company COPS GmbH. xiv, 79, 80, 85, 89, 93–99, 101–106, 108, 116, 122, 125, 127, 142, 156, 165, 167, 170, 171, 173, 175–177, 179–182, 185, 187–189

DevOps A set of practices combining software development and IT operations. 17, 22, 28, 186

FI theory Fact and Information, EE theory. 30–32

NSX A spin-off of the University of Antwerp developing NS. 75, 76, 158, 192

PSI theory Performance in Social Interaction, EE theory. 30, 31, 37, 39, 93, 96, 97, 108, 119, 155, 187

Rossum Artificial intelligence extracting data from invoices. xiv, 123–127

TAO theory Teleology Across Ontology, EE theory. xiii, 30, 33–36, 58, 62, 119, 156–158, 160–162

UiPath Market leader in RPA technology. 70–72, 120, 122–125, 127

Acronyms

ADA Affordance Driven Assembling. 83, 85, 87, 89, 90, 155, 156, 158, 159, 161, 162, 164, 165, 167, 170, 171, 173, 176–178, 180–182, 185, 187–193

AI Artificial Intelligence. 72, 123

AM Action Model. 40, 41, 96

API Application Programming Interface. 62, 120, 121, 123, 125, 127

BA Business Agility. 3–7, 81, 83, 87

BoM Bill of Materials. 36

BPM Business Process Management. 5–7, 43, 64, 65, 72–74, 80, 119–123, 125, 127, 186–188, 190

BPMN Business Process Model and Notation. 29, 73, 74, 120, 121, 127

BPMS Business Process Management System. 5, 6, 73, 74, 120–122, 125, 127

CBS Component-Based System. 10, 43–47, 74, 79, 84, 109, 110, 112–114, 118, 157–159, 162, 173, 186

CE Combinatorial Effect. 18–21, 44, 75, 129–131, 134–141, 144, 145, 148, 149, 152, 177, 178, 182, 186, 188, 189, 191

CEO Chief Executive Officer. 71, 150

CFO Chief Financial Officer. 66

CI/CD Continuous Integration and Delivery. 17, 22–24, 28, 186

CIRS Currency Interest Rate Swap. 137, 138, 140

CM Construction Model. 40

CMMI Capability Maturity Model Integration. 65

CoCoMo Constructive Cost Model. 114

CoR Chain of Responsibility. 50, 147

CRM Customer Relationship Management. 120

CRUD Create, Read, Update, and Delete. 182

DDD Domain-Driven Design. 27

DEMO Design Engineering Methodology for Organisations. 29, 39–41, 93–99, 101–103, 107, 108, 119

DMN Decision Model and Notation. 73, 74

DSL Domain-Specific Language. 162–164, 167, 170, 171, 173

DSR Design Science Research. 81, 82, 86, 89

DSRM Design Science Research Methodology. 9, 58, 79, 81–83, 85–87, 90, 162, 185

DSRP Design Science Research Process. 81, 86, 87, 90, 185

EA Evolutionary Architecture. 17, 21–23, 25, 28, 143, 147, 186, 188

EE Enterprise Engineering. 6–8, 15, 29–31, 39, 41, 75, 79, 83, 84, 87, 89, 93, 129, 155, 157, 162, 177, 185–187, 191

EQ EQuity. 135, 136

ERP Enterprise Resource Planning. 120, 139

F/C Function/Construction. 8, 36, 65, 117–119, 157, 159, 162

FaC Forms and Controls. 51–54, 59, 142, 143, 145

FCA Financial Conduct Authority. 4

FPA Function Points Analysis. 114

FX Foreign eXchange. 93, 94, 131, 134, 135, 137, 138, 167, 177, 178

GAO Government Accountability Office. 4

GDPR General Data Protection Regulation. 24

GoF Gang of Four. 48

GPL General-Purpose Language. 171

GUI Graphical User Interface. 7, 8, 10, 25, 27, 43, 44, 48–61, 63, 65, 74, 76, 84, 85, 87, 89, 98, 108, 109, 114, 116, 119, 124, 125, 127, 141–143, 145, 147–153, 155, 156, 158, 161–165, 173, 175, 179–182, 185–192

HR Human Resources. 5, 69

HTML HyperText Markup Language. 62, 161, 168

HTTP Hypertext Transfer Protocol. 125

IDE Integrated Development Environment. 143

IR Interest Rate. 135, 138, 178

IRRBB Interest Rate Risk in the Banking Book. 178

IS Information System. 3, 17, 18, 29, 81, 82, 86, 129, 130, 132, 134, 138, 139

IT Information Technology. 3–5, 7, 18, 36, 64, 66, 69, 70, 86, 110, 150, 191

LeSS Large Scale Scrum. 6

LINQ Language INtegrated Query. 182

LoC Lines of Code. 114, 180

MD Man-Day. 116, 117

MDA Model-Driven Architecture. 191

MDD Model-Driven Development. 191, 192

MVC Model View Controller. 48, 51, 53–55, 58, 59, 63, 191

MVI Model View Intent. 51, 56, 57

MVP Model View Presenter. 54, 55

MVVM Model View ViewModel. 56, 57, 61, 147–149, 191

NDA Non-Disclosure Agreement. 156

NPM Node Package Manager. 62, 63, 192

NS Normalised Systems. 75, 76, 182

NST Normalised Systems Theory. 10, 18–21, 28, 44, 60, 61, 75, 84, 89, 95, 129, 130, 133, 138, 139, 143, 144, 147, 156, 177–180, 182, 186, 188, 193

O-ADA Objectified Affordance Driven Assembling. 161, 167–171, 176

OCR Optical Character Recognition. 65, 67, 68

OOP Object-Oriented Programming. 46, 130, 143, 163

OP Observer Pattern. 49, 54–56, 58

PM Process Model. 40, 41

PRM PResentation Model. 55–57

REST REpresentational State Transfer. 74, 118, 120, 125

RFD Risk Factor Data. 134, 135

RPA Robotic Process Automation. 7, 8, 43, 64–74, 119–122, 125, 127, 155, 186–188, 190

SAFe Scaled Agile Framework. 6

SBVR Semantics of Business Vocabulary and Business Rules. 85

SD Semantic Description. 170, 171, 176

SDF Semantic DiFference. 112

SE Software Engineering. 6, 8, 10, 15, 30–32, 34–36, 41, 44, 46, 79, 109, 114, 155, 158, 167, 185, 186, 189, 190, 193

SM State Model. 40, 41

SoS Scrum of Scrums. 6

SP Structured Programming. 46, 130

TMS Treasury Management System. 80, 84, 156

UI/UX User Interface/User Experience. 48

UML Unified Modeling Language. 34, 35, 85, 114–116, 161, 162, 167, 169, 171

URL Uniform Resource Locator. 125

UWP Universal Windows Platform. 56

VaR Value-at-Risk. 131–137, 140, 178

WinForms Windows Forms. 59–61, 63, 142–145, 149, 153

WPF Windows Presentation Foundation. 56, 61, 63, 142, 145, 147–149, 153, 163, 167, 175, 179, 181

XAML eXtensible Application Markup Language. 56, 61, 145, 147, 148

XML eXtensible Markup Language. 56, 61, 74, 163, 164, 170, 171, 173

YAGNI You Ain't Gonna Need It. 25

Part I

Introduction

Chapter 1

Research Overview

The start of the 21st century is characterised by a digitalisation of every aspect of society. The number of computers, internet access, applications, and information increases exponentially and societal structures adapt to it [104, 139, 87]. At the same time, enabled by this digitalisation, the expectations of customers and regulators compel enterprises to become more agile. Therefore, many individuals, small businesses, big enterprises, organisations, states, and international institutions have shifted their attention to so-called Business Agility (BA) [10, 11] – 'a set of organisational capabilities, behaviors, and ways of working that affords their business freedom, flexibility, and resilience to achieve its purpose' [10]. However, often they are unable to gain these capabilities – 'to swiftly and easily change businesses and business processes beyond the normal level of flexibility to effectively manage unpredictable external and internal changes' [215]. Their current software and enterprise architecture may hold them back. Even though there are sophisticated modern technologies, they cannot use them since they are obliged to their current ageing technology. Even though there are modern approaches to manage people, e.g., with the help of agile methodologies, they cannot use them since they may again depend on the current ageing technology blocking an implementation of new ways of management.

For instance, as observed by Oosternout et al. [215], the existence of inflexible legacy systems is perceived to be a major disabler in achieving BA. They remind that the existence of inflexible Information Technology (IT) may increase the maintenance and support cost (e.g., in Finance, Logistics, and Utilities). On the other hand, according to Oosternout et al. [215], IT is also an enabler for BA. The right architecture of Information System (IS) may prevent agility gaps arising when the firm has difficulty in meeting the required level of BA. Therefore, IT can both inhibit agility, as well as be a means to achieve it [215]. Oosternout et al. [215] add that this stresses the need for organisations to implement an agile IT and process architecture in areas where BA is required. Such an agile IT architecture can be analysed on four different levels of the business network – from lower to top level: hardware and systems software infrastructure, application software, management of an individual business, and governance of the business network [217]. All of these levels must support integration and quick-connect and quick disconnect capabilities to external

partners [173].

In our research, we will solely focus on the level of application software. We will inspect how the application software may be designed to support the adaption of modern technologies as well as to support modern ways of agile management.

1.1 Loosing Pace with Modern Technology

It is obvious that in business networks, the application software can be an important enabler in achieving BA. Yet, many organisations are still dependant on ageing technologies. For instance, according to a report by the UK's Financial Conduct Authority (FCA), 'Nearly 50% of banks do not upgrade old IT systems as soon as they should, and 43% of US banks still use COBOL' [113], an antiquated programming language dating from 1959. House of Common Treasury committee adds in their report from 2019 that 'ageing architecture is often referred to as a cause of IT incidents' [106]. Another example comes from US Government Accountability Office (GAO). In their report from 2019 [211], they identified the ten most critical federal systems in need of modernisation. Some of which date back to the 1970s. Many of them also depend on COBOL.

This problem is even worsened by an accelerating pace in which new information technology is coming. The gap between new and obsolete technologies is widening. Moore's law focuses on the phenomenon of accelerated development in computing. This widely known exponential doubling of transistors on a chip has led to significant advances over the past five decades. A similar force appears to be driving the development of IT. Ray Kurzweil introduced the so-called Law of Accelerating Returns [128, 129], where he argued for extending Moore's law to describe the exponential growth of diverse forms of technological progress. He demonstrated that the rate of technological change is also exponential, with the overall rate of progress doubling every decade [128]. He made the following statement:

> *'We won't experience 100 years of progress in the 21st century – it will be more like 20,000 years of progress (at today's rate).'*
>
> *– Ray Kurzweil, 2004*

If Kurwzeil is right, we can expect that new technological innovations will be introduced at an increasingly rapid pace. This can lead to serious problems in terms of handling *legacy systems*, which are business-critical software systems that strongly resist modification. The failure of such systems can have a significant impact on business evolution [22, 23]. In this context, it is irrelevant whether the system was installed a few days ago or a decade ago. Based on the rapid pace of technological development, software systems can become legacy systems before reaching a production. As the pace of technological development increases, so does the pace of technology obsolescence [187]. Therefore, the software industry should address the problem of adapting to new technologies quickly. Otherwise, it may soon face challenges in terms of maintaining legacy systems, including a lack of IT resources, lack of

skilled human resources, lack of up-to-date documentation, and large costs associated with support and maintenance [199]. This problem is becoming increasingly crucial as evidence suggests that companies already spend most of their available budget on maintenance [3]. It has been stated that 'by some estimates, seventy-five percent of the IT budgets of banks and insurance companies are consumed maintaining existing systems' [44]. This practically means that the maintenance budgets increasing at the expanse of the budget for technology innovation to which the enterprises are obliged.

1.2 Loosing Pace with Modern Management

Many organisations follow a traditional structuring of departments – sales, marketing, IT, Human Resources (HR), etc. These organisations may be considered as siloed organisations structured into so-called functional silos. This term was coined in 1988 by a consultant on organisational development, Phil S. Ensor [73]. He described so-called Functional Silo Syndrome as an 'overall organisational mentality [that] is one of imposing control on people rather than eliciting commitment from them' [73]. In other words, in such an organisation, teams of employees are grouped by function that all operate separately from each other, without cross-collaboration. Ensor [73] adds that 'their goals are primarily functional. Communication is heavily top-down – on the vertical axis. Little is shared on the horizontal axis partly because each function develops its own special language and a set of buzzwords'.

However, siloed organisations face many challenges – coordination between people is hindered [19], a transparent strategic direction of the company is missing [8], the ability to learn is damaged [73], and many more. Because of that, in siloed organisations, the decision-making and prioritisation of work may be ad hoc without a broader context. The management may lack an overview of who does what and why, the communication within departments and in between multiple departments may be spread across different communication channels, etc.

Fisher [81] clarified that many organisations are therefore moving from inefficient functional silos. They are 'discovering the possibilities for enhanced performance based on a movement toward a process-driven approach to business'. Curtice et al. [45] add that when functional silos or roadblocks disappear, the company is better positioned to satisfy all stakeholders – customers, owners, and employees. Bruin and Gaby [50] describe that in practice this move towards process-driven organisation can be tackled by Business Process Management (BPM). They claim that BPM may be an important enabler for a company to successfully align business practices with strategic objectives and increase business performance. Therefore, many organisations are revealing their end-to-end business processes. Along this initiative, they implement information systems dealing with the definition, administration, customisation, and evaluation of tasks evolving from these processes. These information systems are called Business Process Management Systems (BPMSs) [121]. They are able to delegate business tasks to the right people at the right time using the right information resources [121].

Such an environment is a great basis to improve the aforementioned BA. Depending on

the size of an organisation, frameworks like Scrum, Scrum of Scrums (SoS), Scaled Agile Framework (SAFe), Large Scale Scrum (LeSS), etc, are implemented. This enables the organisation to better align its business strategy with the actual tasks to be done by each department, team, or individual. However, BPMSs typically come with an application software. Therefore, they become obsolete similarly to another application software. It is obvious that they must be designed with respect to their move to modern technology regardless the changes of business processes.

1.3 Motivation

As we described above, it is visible that companies must thrive with uncertainty of today's unforeseeable market. A capability to thrive is referred to as BA. Unfortunately, the ageing application software disables organisations to improve this capability. Even though they may use agile frameworks, they are unable to keep up with the latest technology developments, not with their application software, nor with BPMS.

We believe that this challenge of ageing technology may be tackled by an approach that can aid in software better suited for technology transition. In other words, software that can evolve better with respect to modern technologies. One of the major fields researching evolvability is Enterprise Engineering (EE). The EE research paradigm applies theories from other fields to the evolvability of organisations. We argue that such theories can be applied to Software Engineering (SE) as well. It can contribute to the construction of the aforementioned software that clearly separates dynamically changing technologies from a relatively stable description of functions required for a specific user.

With our long-lasting industrial experience with SE, we believe that we can reify the concepts of EE in terms of SE. Therefore, with this reification, we can contribute to an industry producing more evolvable software. This may inherently help to keep the application software up with the modern technology. Since BPMS include an application software, our contribution should help with its further adaption to modern technology as well. Thus, it should support organisations to improve their BA using technologically matured BPM solutions.

1.4 Problem Statement

With respect to the provided research background, it is obvious that the latest technological development supports companies to innovate in different fields on different levels. However, organisations of different sizes are often lacking the capacity to adapt to, create, and leverage changes to use these latest technologies for their customer's benefit. Their software solutions are becoming increasingly outdated. Accordingly, the problem in this research can be formally formulated as follows:

> *'Despite the potential in modern technologies, SE do not offer guidance and architectural pattern on adapting software artefacts into the latest technologies*

in a more efficient and manageable manner.'

The problem above can be an impediment for IT firms lacking an architectural pattern that might help them to respond quicker to their customer's requirements, and to adapt their products into modern technologies. In the same time, the above problem is an obstacle for any company that wants to improve its BA with the help of modern digitalisation solutions and trending agile management techniques.

A variety of technological developments following a range of design or architecture patterns have been observed in the literature. Their thorough review and evaluation is in Chapter 4. It reveals the findings of component-based systems, the strengths and weaknesses of design patterns and the corresponding Graphical User Interface (GUI) technologies implementing them, and it inspects BPM and Robotic Process Automation (RPA) to bridge the gap between the modern and obsoleting solutions. However, in Chapter 2, we also reviewed formal roots of EE and the theories on how systems can evolve. We argue that the corresponding concepts of EE and evolvability can be reified to a software design approach that aids in the construction of more evolvable software solutions.

1.5 Goals of the Dissertation Thesis

According to the defined problem, the goal of this research is to:

'Design and develop a new methodical framework that aids in the construction of software solutions enabling controlled technology transition.'

1.5.1 Research Scope

While our research goal is aiming at technology transitions in general, our scope is narrowed down to GUIs where the problem of technological transitions is generally perceived as being cumbersome in terms of the proliferation and dynamics of GUI frameworks. Furthermore, this research focuses on conceptual rather than detailed solutions. By so doing, attention is drawn to the conceptual aspects of enterprise engineering and evolvability rather than to the particular implementation in a specific technology. Therefore, the activities like user or technical testing are out of the scope of this work. This research scope is visually outlined later in Fig. 6.1.

Additionally, in this study, many examples are exemplified in a treasury management system. Therefore, even though the proposed methodical framework is generally applicable, we solely focus on evaluating it in the context of financial applications rather than in other contexts. Finally, since we work primarily in a research environment where .NET and JavaScript dominate, we will mostly exemplify our ideas on that technology stack.

1.5.2 Research Questions and Objectives

To address the research problem and to achieve the goal of this research, we have to answer two research questions:

Research Question 1. What is the current practice and research status of approaches to build evolvable SW? What are their limitations?

Research Question 2. Can a methodical framework for building software better suited for technology transition be grounded in enterprise engineering research? How this framework can be designed?

A number of objectives must be defined to achieve the research goal and answer the research questions in our research scope. Each of the objectives may help answering one or the other.

Research Objective 1. To investigate and understand the state-of-the-art of the

RO 1.1 research on SW evolvability;

RO 1.2 EE theories from which SE may potentially benefit;

RO 1.3 technological practices aiming at better technology transition;

RO 1.4 related research aiming at better technology transition.

Research Objective 2. To design a methodical framework aiming at controlled technology transition.

RO 2.1 To investigate how user-interactions can be captured in SW development.

RO 2.2 To describe trade-off between flexibility and usability of SW components when having their Function/Construction (F/C) devised.

RO 2.3 To show how RPA technologies can bridge the gap between legacy SW and SW built with technology transition.

RO 2.4 To demonstrate how to measure the evolvability of systems.

RO 2.5 To describe architecture concepts that limit GUI transitions.

Research Objective 3. To design & develop a prototype of the designed framework in an industrial-scale system.

Research Objective 4. To demonstrate the framework and evaluate its effectiveness for constructing SW that aid controlled technology transition

RO 4.1 in the industrial-scale environment;

RO 4.2 from a theoretical point of view.

Next, we present our research approach to achieve the objectives and reach the research goal.

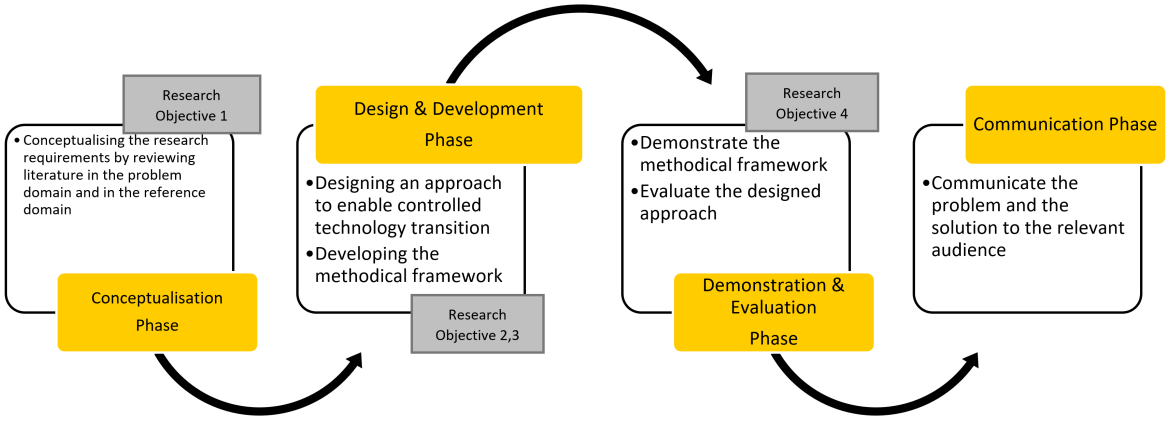

Figure 1.1: Research approach and its link to research objectives

1.5.3 Research Approach

The methodological tool used to examine the research objective of this thesis is Design Science Research Methodology (DSRM) (Hevner [102]). It addresses the research problem put forward earlier in Section 1.4. According to the selected methodology, the adopted research approach here consists of four main phases namely conceptualisation of the research, design & development, demonstration & evaluation, and communication. They are presented schematically in Fig. 1.1.

Conceptualisation Phase. The theoretical background for this research is grounded in the first phase. It aims to acquire a knowledge necessary to build an awareness of the problem. A wide range of journal articles, conference papers, books, and other relevant resources were reviewed in the 'problem domain' – construction of software solutions enabling controlled technology transition. This thorough literature review is undertaken to address RO 1.1 and RO 1.2 in Chapter 2 and Chapter 3 respectively.

To understand the problem and reveal the gap in the technology stack, we inspect practices to build software suited for technology transition. Technical reports, technical literature as well as the Internet were utilised to address RO 1.3 in Chapter 4.

DSRM encourages researches to also review other perspectives and insights to the problem. Therefore, in Chapter 5, we address a last RO 1.4 by reviewing similar research aiming at technology transition.

Design & Development Phase. After the conceptualisation, we design a methodical framework aiming at controlled technology transition. First, we address a range of objectives behind RO 2. This helps us to understand the perspectives we need to keep in mind when designing the methodical framework. Next, by addressing RO 3, we reveal the constituents of the framework itself. This helps us to understand the components of a possible software architecture following the framework.

Demonstration & Evalutation Phase. To showcase the framework in practice, we employ a range of convenient technologies to implement the designed framework in a prototype system. This development results in a subsystem of an industrial application in the area of finance. This helps us to tackle RO 4.

Communication Phase. In the final phase, we retrace the problem of the research, its solution, and novelty together with all the findings we learned along the way. Here, we reveal a number of academic publications as well as posters that we presented to other researchers and relevant audiences.

1.6 Thesis Roadmap

The thesis is structured into seven parts namely *Introduction*, *Background & State-of-the-art*, *Our approach*, *Main results*, *Conclusion*, *Publications*, and *References* each of which may contain one or more chapters as illustrated in Fig. 1.2. Below, we briefly explain the order and content of each chapter. For a convenience of the reader, we also show the mapping of our research approach into the corresponding chapters. We distinguish the corresponding phases by colour.

The thesis starts with Part I. Its only chapter – Chapter 1 provides a motivation to our research with respect to software industry loosing a track with modern technology as well as with modern management techniques. The chapter formulates a research problem, its goal and corresponding objectives. Finally, it gives an overview of our research approach and the selected methodology. Finally, the roadmap of our thesis is presented together with the mapping of our research approach and objectives into the corresponding chapters.

Part II starts with the breakdown of the background and state-of-the-art. In Chapter 2, the research on evolvability is critically reviewed. Chapter 3 introduces enterprise engineering theories that are relevant to our research. In Chapter 4, numerous technological developments in the area of SE are inspected. The historical development of Component-Based Systems (CBSs) is presented at first. Next, it reviews typical GUI patterns and corresponding GUI technology. Finally, in Chapter 5, we investigate related work to our research.

Part III contains one an only chapter – Chapter 6. It grounds the research methodology used in our research. It explains the phases of the selected research process and employs it to our research.

Part IV intends to design the methodical framework what is a goal of our thesis. In this part, we integrate an output from our long-lasting research initiatives. These initiatives were triggered with respect to address various research objectives emerging over time. Chapter 7 presents a confirmation engine improving interactions between people. Chapter 8 investigates a trade-off between flexibility and usability of a design. Chapter 9 shows how robotic process automation may support technology transitions. Chapter 10 describes how Normalised Systems Theory (NST) may be used to evaluate financial models. Chapter 11 inspects current GUI technology in lens of design patterns. Finally Chapter 12 describes

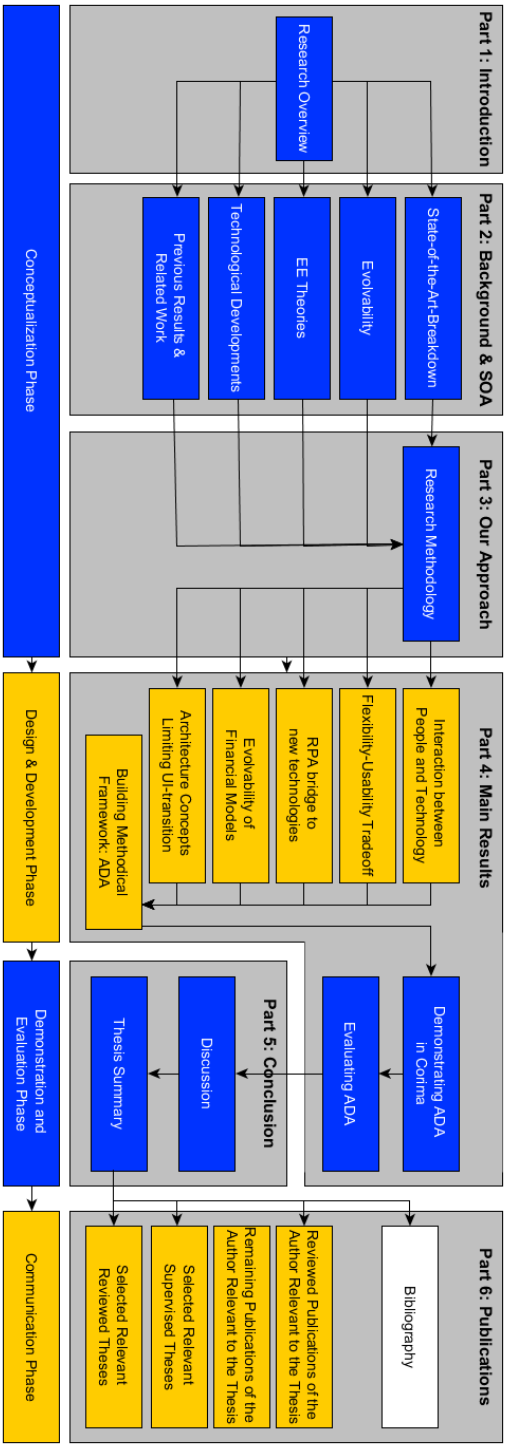

Figure 1.2: Roadmap and a its linkage to our research approach and text of this thesis

the design of our methodical framework, and it explains its different components. The framework is demonstrated in Chapter 13. Its evaluation is further provided in Chapter 14.

The part Part V concludes the whole dissertation thesis. In Chapter 15, it discusses how specific objectives where achieved. Finally, in Chapter 16, we summarise the whole dissertation thesis.

The last part – Part VI shows the achievements in the presented research. Each publication comes with quotations of our work and the information where it was published.

1.7 Chapter Summary

In this chapter, we laid down foundations of the research presented in this thesis. We formulated the research problem and goal, and we formalised the essential research questions that must be answered. The research objectives and its scope were also put forward in this chapter.

We provided a brief overview of the adopted research approach and its phases. We delineated the roadmap of this thesis, and we structured it into parts. To guide the reader throughout the whole document, we provided a linkage between each chapter and the corresponding research phase.

Part II

Background and State-of-the-Art

Part II – Breakdown

In the following chapters, we provide a state-of-the-art literature review to create an understanding of the existing knowledge on evolvability and rapid technology change. This background is necessary to set a body of knowledge (knowledge base) and environment, further described in Section 6.4, with which we are working in the latter stages of this research. We use this background as an input for designing the artefacts produced throughout this research.

First, in Chapter 2, we address the objective RO 1.1. We inspect different perspectives on evolvability, its connection to business agility, and we review its academic as well as industrial state-of-the-art. Second, in Chapter 3, we address the objective RO 1.2 to understand the formal grounds in EE. The paradigms, trends, and SE practices tackling challenges of rapid technology change are discussed in Chapter 4 to address RO 1.3. Last but not least, to address RO 1.4, Chapter 5 includes a knowledge of neighbouring research fields and research fields which are comparable – residing in the same problem space.

Chapter 2

Evolvability

To tackle the objective RO 1.1, we need to get an understanding of the research on evolvability and practices focused on building evolvable SW. To understand evolvability, we need to start with agility. Therefore, in Section 2.1, we will explain the linkage between them. Next, in Section 2.2, we will present a theory researching evolvability. Finally, in Section 2.3, we will connect it to evolutionary architectures dealing with evolvability with the help of DevOps and Continuous Integration and Delivery (CI/CD) practices.

2.1 The Linkage between Agility and Evolvability

As explained by Oorts et al. [161], under the current volatile and competitive economic conditions, organisations attempt to be agile at all levels. Las Mathiassen and Jan Pries-Heje [140] narrowed this topic to the aforementioned property called business agility. They explain it as a novel paradigm presented as a solution for maintaining competitive advantage during times of uncertainty and turbulence in the business environment. They clarified that 'the agile mind is defined as having a quick, resourceful, and adaptable character. Therefore, agile organisations respond quickly, they are resourceful, and they are able to adapt to their environment' [140].

A similar trend is also recognised in the Evolutionary Architecture (EA) coined by Ford et al. [83]. They explained that a shift to a large and complicated software system can be facilitated by standalone business-oriented teams using technological advances such as DevOps, CI/CD, and containers such as Docker. However, Oorts et al. [161] added that simply having an agile organisation is not sufficient. It is also necessary to develop SW that is capable of changes. They said that 'this can be considered as an important step or precondition for establishing an agile organisation' [161].

For ISs, this can be interpreted as the capability to adapt to new functional requirements. These new functional requirements arise from changes in the environment and processes that surround the system as well as the user's experience [32]. This phenomenon is captured by Lehman's law of continuing change. At the same time, Lehman posits the law of increasing complexity. It states that unless something is done to prevent it, the

structure of an IS deteriorates over time [131]. The challenges linked to that are often explained as *technical debt* [20]. At the end, this results in the compulsory replacement of the existing system. Op't Land [164] adds that a consequence of the deteriorating structure is an annual growth in budgets for development and maintenance. He clarifies that 'Enterprises that decrease – or (even) keep constant – the IT budget will be faced with less satisfactory IT, decreased support of organisational changes, decreased business IT alignment and decreased situational awareness' [164]. Based on this statement, he identifies a challenge in a software development area – an approach that could help IT companies to develop software that exhibits high quality and is quickly continuously changeable over time. This challenge is captured in a term *evolvability*.

There are many definitions of evolvability. For example, Cook et al. [39] defined it as 'the capability of a software product to be evolved to continue to serve its customer in a cost-effective way'. Liguo Yu and Ramaswamy [233] elaborated on evolvability in the context of biological systems and refined Cook's definition to 'the ability of a system to support self-organisation such that it can effectively evolve to serve a new requirement without appreciable degradation of, and perhaps much better suited to support its current capabilities' [233]. Another definition comes from NST [137] what is one of the leading theories for handling software evolvability. NST focuses on how modular structures of a system influence its evolvability. This theory explains software evolvability as 'the ability of software to be changed easily' [161]. In other words, it defines an evolvable system as an information system to which a set of anticipated changes can be applied easily [134, 136]. As clarified by Vanhoof [216], NST specifies the meaning of 'easy' by using the notion of Combinatorial Effect (CE) discussed in Section 2.2.

Finally, a modular decomposition of a large system into smaller subsystems has long been identified as a way to improve evolvability and facilitate changes (e.g., [192], [184], [12]). To obtain its benefits, the subsystems should be partitioned in a precise, unambiguous, and complete way, and they should interact through standardised interfaces [12]. However, it is recognised that no universal measure or composition of a product's modularity exists ([94, 183, 27]). Different decompositions are possible and a firm should choose a decomposition that aligns with its objectives [27]. In their highly referenced review paper [27], Campagnolo and Camuffo emphasise a lack of research on product design modularity that addresses market or industry-specific factors possibly affecting product design modularity itself [27].

Section Takeaway

We share the opinion of Oorts et al. [161], and we agree that SW evolvability is a precondition for an organisation to achieve business agility. Therefore, regardless the need for agility at different levels, we solely focus on the SW evolvability that contributes to achieve it. We understand evolvability in terms of NST and its perspective on modularity.

2.2 Normalised Systems Theory

The sections above describe the difficulty linked to the demand for evolvable systems. NST offers an answer to this challenge. It offers a systematic methodology for modular design with the goal of creating evolvable systems [137]. It uses the formal foundations of system theoretic stability to study the transformation of (basic) functional requirements to the software primitives of a stable system [137]. This stable system is defined as 'bounded input/bounded output' – if the system receives bounded input, it should create a bounded output. It means that for a set of anticipated changes (i.e., changes in basic functional requirements), the impact on the system should only depend on the change itself and not on the size of the system [135].

2.2.1 Combinatorial Effects

As mentioned earlier, NST works with so-called CEs. CEs are defined by Mannaert et al. [137] as follows:

Definition 1. ... Functional changes causing impacts that are dependent on the size of a system and the nature of the changes... [are called] CEs. [135, p.5].

The fewer the CEs that are caused by a change, the easier a system is to change and the more evolvable the company is [134, 136]. Vanhoof [216] added that 'the definition of changing easily then becomes that it is easy to impose changes on the organisational design of a company when a change does not induce CEs'.

Additionaly, CEs stem from improper division in modules or an incorrect encapsulation of modules [137]. They may lead to large costs as changes will need to be implemented in multiple modules. This is what Mannaert et al. [137] call *the law of exponential ripple costs*. However, modularity combinatorics might as well induce flexibility following the *law of exponential variation gains* [137]. It states that if an overall system consists of independent modules, the development and maintenance cost of those modules is the sum of all modules, whereas the number of variations is the product of all modules. In systems with multiple variants of the same unit of work, this leads to an exponential increase of possible combinations [137]. Modules following the NST theorems both leverage the opportunity described by the law of exponential variation gains while avoiding the unwanted CEs.

To summarise, in our research, technological change is considered as the NST *change driver*. We measure the CEs that are defined in Definition 1. Therefore, the main criterion used for evaluating the quality of a specific software architecture is the *bounded* impact of technological transition. As Mannaert et al. [135] clarified, 'bounded is a term from system stability defined in system theory, which states that a bounded input (i.e., changing requirements) should result in a bounded output (i.e., changes in the software)'. More specifically, NST uses the formal foundations of system theoretical stability to study the transformation of (basic) functional requirements into the software primitives of a stable system [137]. A stable system is defined as a system that creates bounded outputs when receiving bounded inputs. In other words, for a set of anticipated changes, the impact on

the system should depend only on the changes, and not on the size of the system [135]. When this is not true and the impact of a change depends on the size of the system, a CE occurs.

2.2.2 Theorems of Normalised Systems Theory

NST formulates four software (formally proven) design theorems. It argues that following these principles is a necessary condition in order to avoid CEs in a software system [53, 135, 137]. However, the violation of any of these design theorems incurs in the occurrence of undesirable CEs.

The following postulate is defined as the ultimate goal [137]: The *separation of concerns* principle states that in order to isolate change drivers, an entity may only execute one task. Those tasks should furthermore exhibit *action version transparency*, meaning that a change in the task may not impact other tasks that call on the first task. Data used in tasks need to exhibit *data version transparency*. If data is modified, this may not have an impact on the tasks that use the data. The final requirement is the call for *separation of states*, meaning that the status of every task should be kept.

NST argues that a software system following these principles is guaranteed to be CE-free. However, the violation of any of these design theorems results in the occurrence of undesirable CEs. The following postulate is defined as the ultimate goal [137]:

> *'An evolving information system should not have instabilities (CEs). A bounded amount of additional functional requirements cannot lead to an unbounded amount of additional (versions of) software primitives.'*
>
> *– Herwig Mannaert, Jan Verelst, Peter De Bruyn, 2016*

The aforementioned theorems were defined by Mannaert et al. [137] as follows:

Theorem 2.2.1. *Separation of Concerns*: A processing function can only contain a single task in order to achieve stability.

The separation of concerns is broadly recognised *best practice* in software architecture design. It demands that each function should implement a single task and be impacted by a single change driver. In reality, this theorem requires the implementation of single-purpose functions and the avoidance of code duplication to prevent CEs [137]. It also requires the strict separation of technologies because every technology is defined in NST as a change driver or concern.

Theorem 2.2.2. *Data Version Transparency*: A structure that is passed through the interface of a processing function needs to exhibit version transparency in order to achieve stability.

Data version transparency is a principle used to handle additions or removals of data fields in entities. This implies the encapsulation of data fields to facilitate the coexistence of various versions of an entity. This theorem highlights the need for encapsulation to prevent CEs [137].

Theorem 2.2.3. *Action Version Transparency*: A processing function that is called by another processing function, needs to exhibit version transparency in order to achieve stability.

This theorem focuses on the ability to upgrade the implementation of processing functions and tasks. The new version of a task implementation must not break the existing system. Calling the new version of a function should be seamless and should not require additional changes, facilitating the avoidance of CEs [137].

Theorem 2.2.4. *Separation of States*: Calling a processing function within another processing function, needs to exhibit state keeping in order to achieve stability.

This is a formalisation of avoiding the transition to an undefined state. When a state is maintained for every call of a processing function, the whole system behaves as a deterministic state machine, eliminating the need for complicated recovery from undefined error states. This combats the problem of CEs emerging from synchronous calling pipelines that are natural in object-oriented systems [137].

Section Takeaway

Let us summarise, NST grounds the evolvability in formally proven theorems. It builds on a concept of CE that should be avoided in an evolving information system. The violation of any of the design theorems incurs in the occurrence of undesirable CEs.

2.3 Evolutionary Architectures

In this section, we will continue addressing the research objective RO 1.1. Together with Mareš, we inspected state-of-the-art of EA. In this section, we provide its excerpt.

EA is a term coined in the book *Building Evolutionary Architectures, support constant change* by Neal Ford et al. [83]. Their motivation comes from the observation that:

> *'Despite our best efforts, software becomes harder to change over time. For a variety of reasons, the parts that comprise software systems defy easy modification, becoming more brittle and intractable over time. Changes in software projects are usually driven by a reevaluation of functionality and/or scope ... though architects like to be able to strategically plan for the future, the constantly changing software development ecosystem makes that difficult. Since we can't avoid change, we need to exploit it.'*
>
> *– Neal Ford, Rebecca Parsona, Patrick Kua, 2017*

The idea was first sparked at O'Reilly conference. Many speakers talked about microservices and the disruption it caused. Ford et al. [83] add that in many companies building architectures such as microservices, their teams are structured around service boundaries rather than technical positions in siloed organisations discussed in Section 1.2. People work in teams that impact myriad dimensions of software development and reflect the problem size and scope. Ford et al. [83] conclude that 'companies typically structure teams that resemble the architecture by cutting across functional silos and including team members who cover every angle of the business and technical aspects of the architecture'. They clarify that this shift in big complicated software systems is only possible due to advances like DevOps, CI/CD, and containers like Docker. Deployments could be made small and rapid. This also changed the notion of 'Architectural change is hard'. It enabled an architecture designed to accommodate the change where replacing one microservice for another should be as easy as switching LEGO bricks. Ford et al. [83] define it as follows:

Definition 2. An evolutionary architecture supports guided, incremental change across multiple dimensions.

- *Incremental change* describes the level to which teams may build and deploy SW incrementally. For development, incremental change refers to granularity of a change, because incremental changes are easier if the scope of change is small. For deployment, incremental change refers to the level of modularity and decoupling for business features and their mapping to architecture.
- *Guided change* helps to protect characteristics the architect chooses to be important, e.g., with the help of a fitness function that encompasses a variety of mechanisms to ensure architecture doesn't change in undesirable ways.
- *Multiple architectural dimensions* refers to the parts of architecture that fit together in often orthogonal ways. For example, technical, data security, and operational/system dimension.

To be in practice in line with this definition, Ford et al. [83] propose several useful characteristics. They describe principles directing us in a way towards those characteristics. All of that is based on heuristics distilled from successful industrial projects and assurance by experts. Now, we broadly introduce these characteristics and guiding principles.

2.3.1 Concepts of Evolutionary Architectures

Let us talk about the guiding principles that voice throughout EA [83] research.

Conway's Law. The first guiding principle was introduced by Melvin Conway [38] in 1968. He codified what has become known as Conway's Law:

> *'Organisations which design systems are constrained to produce designs which are copies of the communication structures of these organisations.'*

—Melvin E. Conway, 1967

Ford et al. [83] explains that Conway describes that in the very early stages of a design, one must come up with a high-level understanding of the system. It is made to realise how to break down areas of responsibility into different patterns. The way the problem is broken down affects the choices that can be made later. To bring EA into the real world, we need to keep Conway's Law in mind. As Conway notes, we introduce coordination problems when technologists break down the problems into smaller chunks.

This problem is mostly visible in siloed organisations having teams separated by technical functions, e.g., front-end, back-end, business logic, database, and so on. Typical problems that cut vertically across these layers may increase the coordination overhead. Later, we will show that microservice architecture is one of the types of EA. As Ford et al. [83] exemplify, in such an architecture, one of the good examples of Conway's Law in action might be trying to change the contract between two services. This might be difficult if the change of a service owned by one team requires a coordinated and agreed-upon effort of another.

Therefore, in the context of EA, Conway's Law is a warning to software architects. To make EA happen, we cannot have a company divided along the knowledge expertise. Therefore, not just the architecture and design of the SW, SW architects should also pay attention to the delegation, assignment, and coordination of the work between teams.

The lesson learned here is to inverse this law. If we want to build standalone independent services, we need to disperse the experience and build teams around projects and business functionality. This aligns with an agile approach to build teams of diverse members and makes a lot of sense given the context of CI/CD, etc. mentioned above.

Fitness Function. Another guiding principle of EA is so-called *fitness function*. In order to achieve EA defined in Definition 2, the changes must be guided – they must protect the various architectural characteristics required for the system. Fitness function is the means embodying this protection mechanism. EA borrowed that term from evolutionary computation techniques like genetic algorithms where it is commonly used to define success.

The concept comes from the character of evolutionary computing where changes emerge gradually via small changes in each generation. At each generation, we want to assess the current state and evaluate whether it is closer or further from the desired goal. Ford et al. [83] define the architectural fitness function as follows:

Definition 3. An architectural fitness function provides an objective integrity assessment of some architectural characteristic(s). [83]

The idea is that each system has a list of essential '-ilities' like usability, security, accessibility, traceability, fault tolerance, low latency, testability and many many more. The authors of EA separate these into different categories. However, the key message is that architects and developers should pay attention to them. They should be identified as soon as possible. Each should be provided with a rating based on how important it is for a given project.

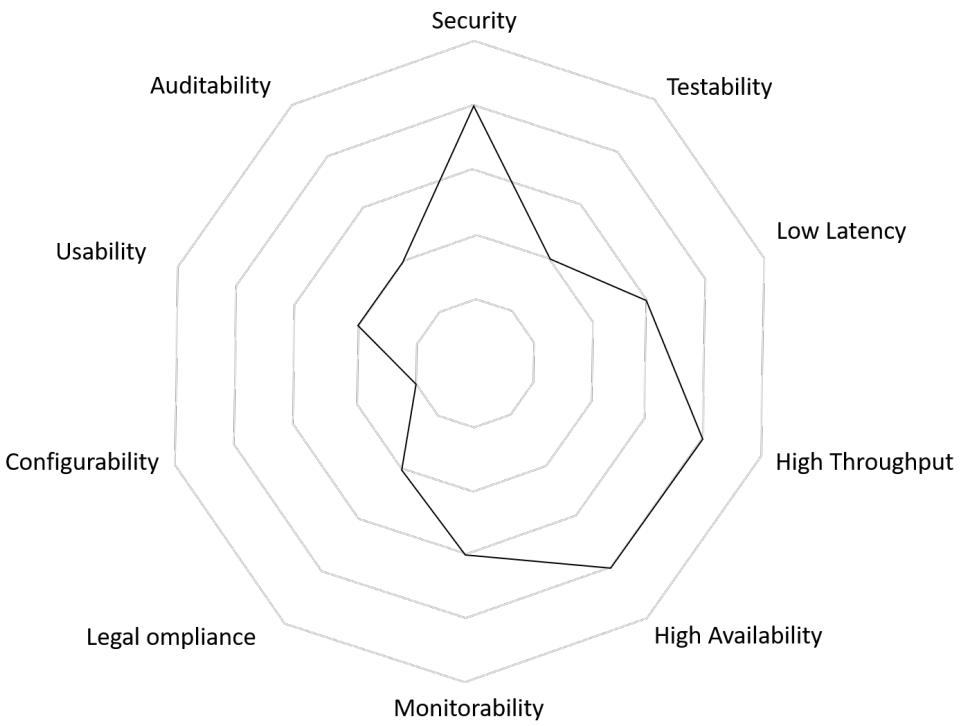

Figure 2.1: Example of EA fitness function fit

An example of such a fitness function is depicted in Section 2.3.1. It exemplifies gatekeepers implemented into a production pipeline. It extends the CI/CD principles with additional checks that are placed on systems and modules. These checks may refer to, e.g., code coverage over 90% resulting from static code analysis, load testing demanding to serve all web request within 10 seconds regardless the network latency, General Data Protection Regulation (GDPR)[1] compliance showing logs of how personal data are handled and stored, and so on. These observations make it possible to keep an eye on the state of the architecture and make informed decisions for future changes.

2.3.2 Principles of Evolutionary Architectures

Bring the pain forward. This principle promotes the idea that things having a potential to cause pain must be done often and earlier. It observes that technical debt does not behave linearly. Instead, as projects grow, it increases exponentially. CI/CD offers a solution. It is a 'SW development practice that enables organisations to frequently and reliably release new features and products' [189]. It encourages us to automate development steps that are complicated, time consuming, and thus barely executed. Things requiring close attention, database migration, or code refactoring should be prioritised. The authors advise to identify these issues and remove the pain early before interest accumulates.

[1]The GDPR is a regulation in EU law on data protection and privacy for all individuals within EU.

Last Responsible Moment. This principle is used to counter the project's hazard of buying complexity too early [83]. It may be considered as an extension to the well-know You Ain't Gonna Need It (YAGNI) heuristic. In traditional architectures, many subsystems, technology stacks, and tools are chosen very early or even before coding entirely.

EA weights the cost of incorrect early decisions against delayed decisions. It emphasises that delayed decisions might benefit from additional information gained during the time difference, and it may argue for the later. Arguably, this decision to delay has its own price – a potential re-work that can be soften by some abstraction. However, YAGNI strikes again. The benefit is that this cost ought to be significantly smaller than, e.g., an inappropriate messaging system, which could slow down the development in many other areas and eventually be marked as tech debt replaced much later in the life of the project.

The essential question is when is the last responsible moment for a certain decision. The fitness function may provide help. Decisions that have a bigger impact on the whole system or are of significant importance should be made earlier. The core of the idea is to wait as much as possible, but do not stall.

2.3.3 Architectural Styles

Big Ball of Mud. In SW architecture, 'Big Ball of Mud' [82] is colloquially known as an antipattern when frameworks and libraries are typically in place, yet not built on purpose. As Ford et al. [83] clarified, these systems suffer from the following:

- They are highly coupled and they lead to rippling side effects when changes occur.
- They contain highly coupled classes with poor modularity.
- Database schemes snaked into the GUI and other parts of the system what effectively insults them against change.

This type of architecture can be seen as the least evolvable architecture. Ford et al. [83] exemplifies it in Fig. 2.2. It depicts a class coupling diagram where each node represents a class and the line represents inward or outward coupling. The boldness of the line indicates the number of connections. As the authors clarify, changing any part of the application shown in Fig. 2.2 presents intense challenges. Due to the exuberant coupling, the impact of a change in one part is unforeseeable.

Layered Monoliths. To score better from the standpoint of evolvability, let us review layered monolithic architectures. These may be illustrated in Fig. 2.3. As Ford et al. [83] explain, each layer represents a technical capability to allow developers swapping out functionality easily. The main advantage of this architecture are isolation and separation of concern discussed in Theorem 2.2.1. The layers are isolated. They can only be accessed via well-defined interfaces. Therefore, implementation changes in one layer can be made without impacting the other layer. To judge the monolithic architecture, it is worth understanding so-called *quantum*. In physics, the quantum is the minimum amount of any

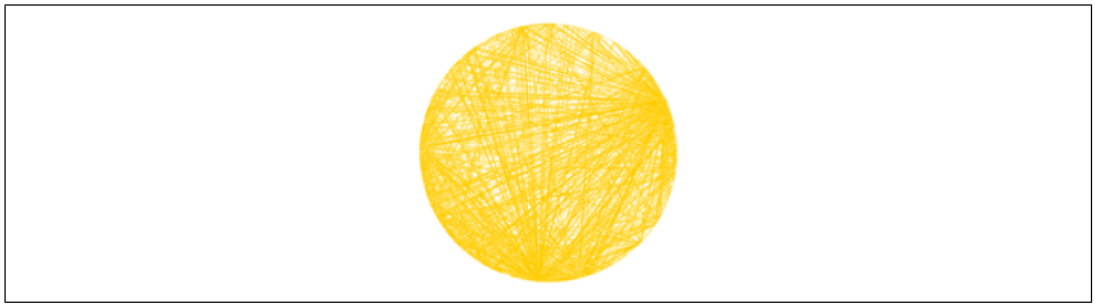

Figure 2.2: Afferent and efferent coupling for a dysfunctional architecture [83]

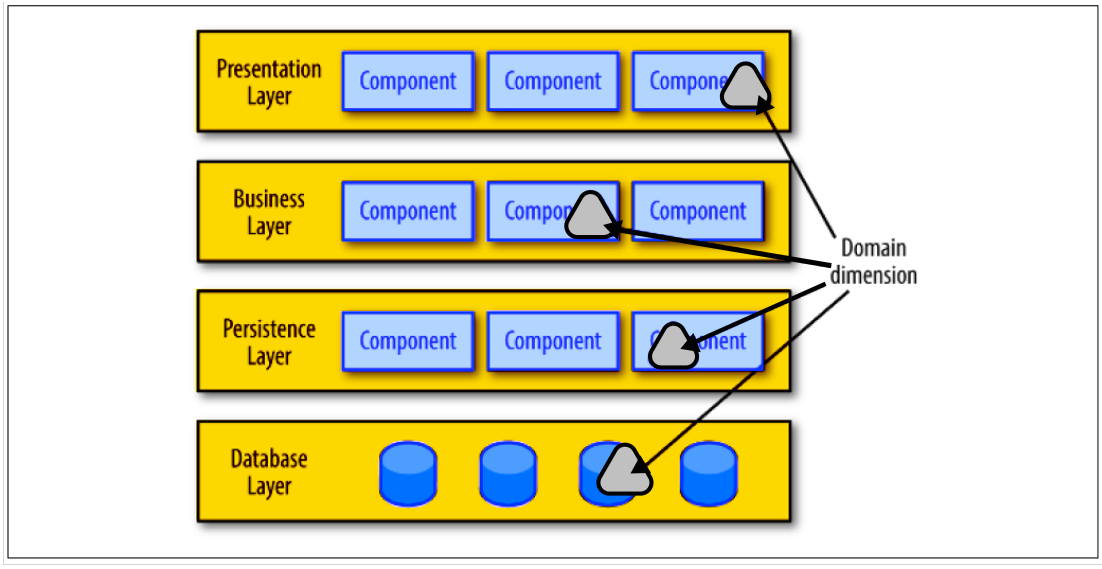

Figure 2.3: Layered monolithic architecture and the domain dimension embedded in it [83]

physical entity involved in an interactions. Ford et al. [83] similarly defines *architectural quantum* as:

Definition 4. Architectural quantum is an independently deployable component with high functional cohesion, which includes all structural elements required for the system to function properly [83, p.4].

As Ford et al. [83] clarify, in a monolithic architecture, the quantum is indeed the entire system. Everything is highly coupled, thus it must be deployed in a mass. For example, a persistence layer typically encapsulates all implementation details of how data is saved. In layered monolithic architecture, the other layers can ignore those details. Still, evolving systems with large quantum size is difficult since cross-layer changes can cause coordination challenges between teams, Ford et al. [83] conclude.

Nevertheless, Ford et al. [83] explain that layered monolithic architectures are a common choice when starting a project. Developers understand the structure easily. However,

because of decreasing performance, size of the code base, and other factors, many monoliths reach the end of life and must be replaced. Architects often start this innovation by using other architectures, yet with improved modularity – microkernel architecture [179], event-driven architectures [143], or service-oriented architectures [74].

Ford et al. clarify that in a layered architecture, there is no clear concept of the domain dimension. Although in most projects, the common unit of change revolves around the domain concepts, they are not accommodated in the layered architecture. The focus is on a technical dimension, or how the mechanics of the application work: persistence, GUI, business rules, etc. Therefore, as visible in Fig. 2.3, domain concepts are segregated via many technical layers. Some portion is in GUI, some lives in the business rules, and another is handled in the persistence layer. Therefore, in highly coupled architectures, the change is difficult because of the high coupling between the corresponding parts. In the aforementioned siloed organisation, development teams are resembled around each layer, therefore a change requires coordination across many teams.

Modular Monoliths – Microservices. Microservices-style architectures are usually the common final target of migration from layered monolithic architectures. These architectures are defined as follows:

Definition 5. A microservice is an independently deployable component of bounded scope that supports interoperability through message-based communication. Microservice architecture is a style of engineering highly automated, evolvable software systems made up of capability-aligned microservices. [153]

They are more complex than monolithic architectures in many areas – service, data granularity, operationalisation, coordination, transaction, and so on. However, according to the authors, regardless the complexity of microservice architectures, from the perspective of evolvability, things can be finally truly exploited. A migration of layered architecture to such a microservice architecture may be an expensive architecture restructuring exercise, Ford et al. [83] admit.

In contrast to the monolithic architecture, the microservice architecture partion across domain lines. Ford et al. illustrate it in Fig. 2.4. Each service is defined around domain concepts that come from so-called Domain-Driven Design (DDD) coined in Eric Evan's book of the same title [75]. These concepts include so-called *bounded context*, where everything related to the domain is visible internally but opaque to other bounded context. With that in mind, each service encapsulates the technical architecture and all other dependent components into a bounded context. It communicates with other bounded contexts via messages as defined in Definition 5.

Ford et al. [83] concludes that the operational goal of this architecture is to replace one service with another without disrupting the other service. Microservice architecture is an exemplar of an evolutionary architecture, thus it is not a surprise that it scores well from the evolutionary standpoint.

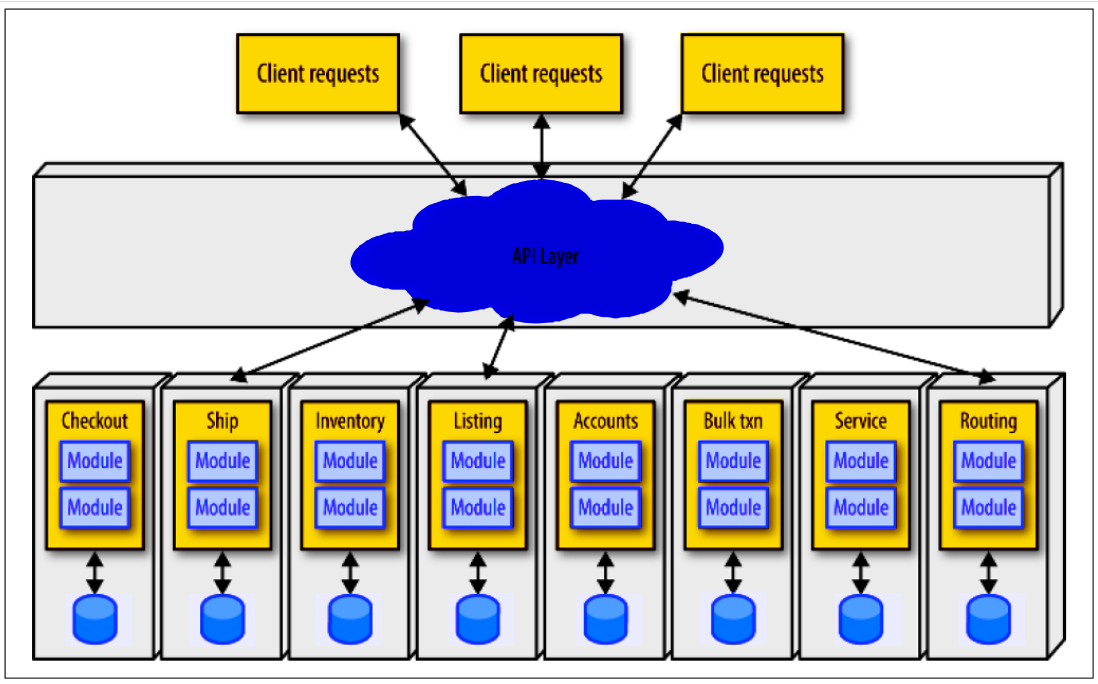

Figure 2.4: Microservices architectures partion across domain lines, embedding the technical architecture [83]

2.4 Chapter Summary

In this chapter, we addressed the objective RO 1.1. We presented two views on evolvability – NST and EA. Each of them sees the topic from a slightly different angle.

NST draws on parallels from different engineering areas. It grounds a solid theoretical foundation and then presents proven theorems that must be followed to achieve evolvable software product. It build from the bottom up talking about small structures, classes and single functions in its examples.

EAs build on advances of DevOps and CI/CD. They advise architects not to dwell on static diagrams of the current architecture. They encourage them to build software products with evolution and openness to change in mind. The authors also remind that the architecture is abstract until operationalised. It means that we cannot judge architecture as a diagram, not even after its first implementation. To consider the architecture success, the corresponding system must go through several upgrades and breakthroughs in some premises that were used to build it in the first place. The EAs come from top-down perspective. They are based on knowledge gained through experience and time-proven heuristics. They also draw on the largest concepts of microservices. They introduce the characteristics that should be present in an evolvable system and provide principles on how to achieve them.

Even though both NST and EA come from completely different directions, they reach a similar conclusion. As to our understanding, they are applicable simultaneously.

Chapter 3

Enterprise Engineering Theories

When migrating IS from one technology to another, we must handle the incompatibilities between them. In practice, we typically handle the problem of moving to another framework or library by using the same technology stack (e.g., .NET, Java, C++, JavaScript). Therefore, we typically identify components that can be left untouched while other components are adapted to the new technology. This problem raises the question of which concepts can easily be reused and which cannot.

According to Giachetti's [95] research, software ISs are just one group of systems. Next to them, there is an interesting group of systems – enterprises. They can be seen as big systems comprising interdependent resources of people, information, and technology. Let us take the role of a human in such a system into account, and let us consider their uniqueness from the communication point of view. Under these circumstances, each enterprise must be unique and substantially complex, too. Because of our limited cognitive faculties, the complexity of an enterprise usually exceeds our ability to deal with that directly. Instead, we need to handle the complexity by abstracting and modelling. A range of methods and notations like Business Process Model and Notation (BPMN) [191], FlowChart [178], or Archimate [118] were developed for that purpose. Unfortunately, not all of them do really abstract from the complexity of an enterprise effectively. On the other hand, Design Engineering Methodology for Organisations (DEMO) [65] was invented to put emphasis on an intellectual manageability of enterprises while concentrating on their essence – the social interactions. It has a strong theoretical background represented by EE theories.

Our hypothesis is that big software information systems can be seen as enterprises. Thereby, the aforementioned EE theories and methodologies can contribute to their understanding and might be applicable to their engineering. Beside others, the complexity of an enterprise is given by the diversity of options people can use to interact and by the expressive abilities of a language with which they communicate. Analogically, the complexity of a software system is influenced by the offer of various languages, libraries, and frameworks. This gives a considerable architectural freedom to all developers. They can use their skills and express thoughts in a form of a source code. As a business process architect can reduce the complexity of an enterprise by introducing guidelines for a human

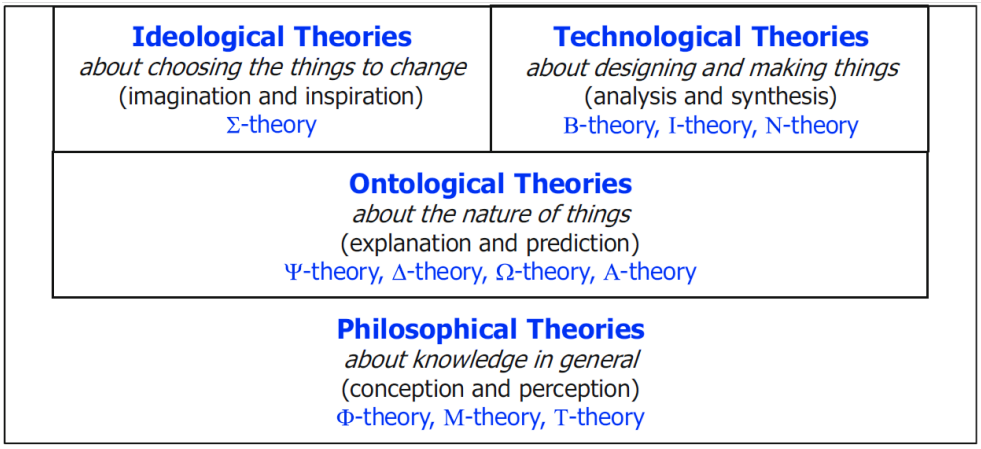

Figure 3.1: The EE theories [65]

interactions, a software architect can set up guidelines for a software implementation to reduce inconsistencies in a source code. A common mission of an enterprise is to successfully deliver and maintain its products while reducing costs on the resources with which it operates. Equally, a common mission of a software system is to provide features expected by their users while reducing costs on the code typing and on its maintenance.

Our experience regarding the connection between SE and EE supports the hypothesis that some EE theories can aid in the construction of software aimed at technology transitions. This hypothesis is supported by Huysmans et al. [108], who discussed an EE research paradigm in the context of applying relevant theories from other fields to the evolvability of organisations. Therefore, we argue that the theories can be used to underpin our efforts theoretically, thereby satisfying our research objective RO 1.2.

By 2020, the EE theories are organised into ideological, technological, ontological, and philosophical. We depict them in Fig. 3.1 as they were presented by Dietz and Mulder [65]. The theories at the bottom influence the theories on the top. For example, the philosophical theories represent the foundation of all the others. In our work, we do not concentrate on an influence of ideological theories to SE. Indeed, e.g., political philosophy of J. Habermas laid down the origins of EE. However, we do not dare to elaborate it in our research. Moreover, we only focus on the theories that, after our initial research, might be applicable to SE practices.

We start with philosophical EE theories [65] – FI theory and TAO theory. These theories address the broad understanding of the core notions of organisations [64]. They introduce the concepts of function and construction. We continue with technological BETA theory [62] that discuss the relationships between function and construction thoroughly. Finally, we discuss PSI theory [67]. It is a theory about the operation of organisations. The theory declares that the commitments of subjects (human beings) are raised in patterns, so-called transactions.

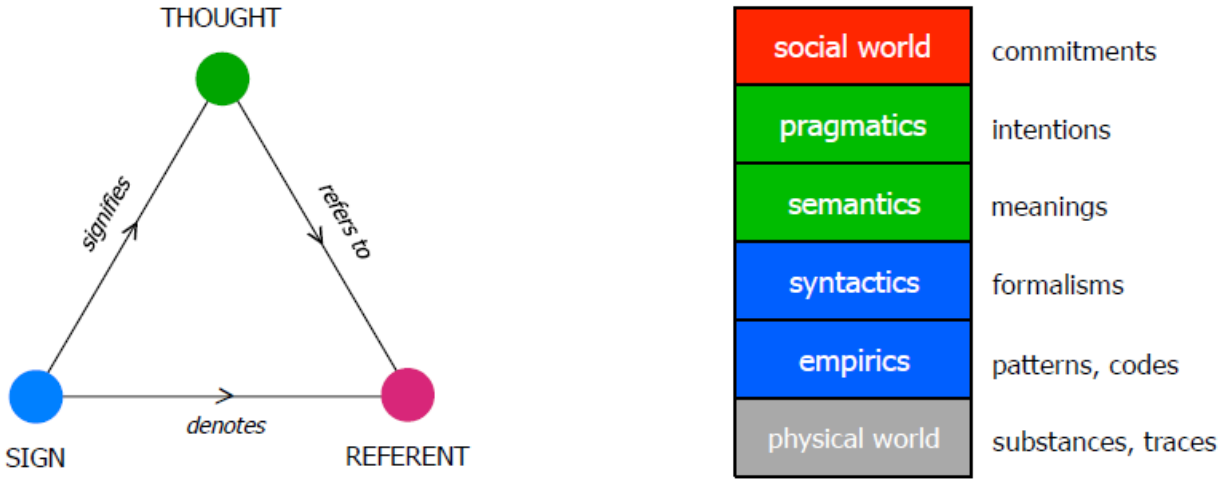

Figure 3.2: Adapted semiotic triangle (left) and semiotic ladder (right) [60]

3.1 FI theory

The FI theory (ϕ-theory) (Fact and Information) is a theory about knowledge in general. It sees the fact as a part of an elementary thought, which in addition consists of an intention. As Dietz clarifies, 'every concrete system (e.g., organisation) has an associated world... A world consists of things and someone's knowledge about these things consists of facts' [60]. Information is an intermediary for exchanging thoughts between humans.

The FI theory is rooted in a *semiotic triangle* presented in Figure 3.2 (left). It was introduced by Ogden and Richards. Dietz presents its slightly adapted version [60]. In our minds, there are *thoughts*. Thoughts refer to concrete things called *referents*, and they are expressed in *signs*.

Nevertheless, the semiotic triangle is a simplified representation of the three core concepts in FI theory: thought, sign, and referent. A more sophisticated framework for studying thought (information) was proposed by Ronald Stamper. It is called *semiotic ladder*, and it is shown in Figure 3.2 (right). Its main focus is on the thought and the sign. On the top of that ladder, there is a *social world* which is dealt in a PSI theory discussed in the next section. It investigates how intentions are related to the commitments of social individuals. The semiotic triangle studies the content of thoughts and is divided into semantics and pragmatics. *Semantics* inspects a meaning of a sign (or a sentence) in some language. *Pragmatics* is about an intention of sharing the thought among subjects. A form of a thought is also divided into two parts: syntactics and empirics. *Syntactics* deals with a form of a sentence that must respect formalisms and rules given by a grammar. *Empirics* studies how to express parts of sentences (e.g., words can be written in Roman letters or in Morse code) in a form of codes and patterns. Finally, a *physical world* does not belong to the field of semiotics. It contains traces and substances inscribing patterns, and the codes above.

Although FI theory belongs to EE theories, we believe it can be easily seen as SE theory.

The SE is about transforming objects (*concrete objects*) from a concrete world to conceptual objects. These are represented by symbols understandable for software. Software operates in a world of symbols and semantics. For example, a given programming language has its world of language elements (variables, expressions, exceptions, flow constructs, etc.). Their composition into grammatically correct sentences has a specific meaning for a computer.

The challenge of SE is to map the concepts introduced by FI theory into symbols and concepts of a computer. The final behaviour should match. In Figure 3.3 (left), a semiotic triangle is shown from an ontological point of view. The referents are considered as concrete objects. They are represented in our minds as conceptual objects. Names are signs by which humans signify the conceptual objects. Conceptual objects in a human world are mapped into conceptual objects in a context of a computer. Names in a human world are mapped into symbols of a computer. It is built using a range of paradigms, programming languages, libraries and various technologies supporting software development. Needless to clarify that the mapping is not all the time one-to-one relationship.

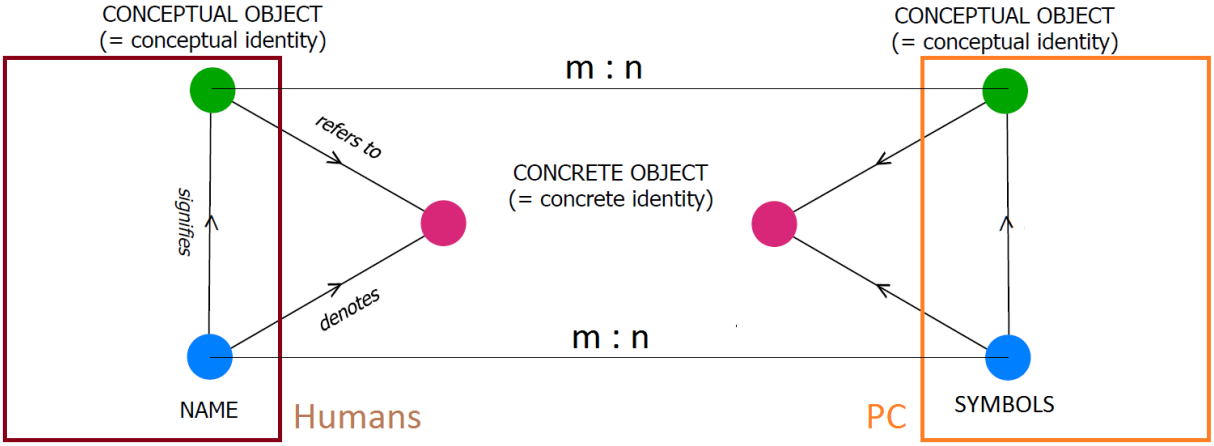

Figure 3.3: Mapping of a semiotic triangle [60]

3.2 TAO Theory and Affordances

In 1728, philosopher Christian Wolff brought a term *teleology*. Wolf [221] defined it as a natural philosophical branch that explains the ends and purposes of things. In other words, teleology studies objects as purposive, or goal-directed, entities. It inspects them with regard to which purpose the subjects can use them. The objects are viewed as complex machines in which each part is minutely adopted to others, and each contributes to the purpose of a whole by performing a specific function. As Ayala [9] explains, the objects made by people are teleological: chair is made for sitting, a hammer is made for hitting, a knife is made for cutting, a thermometer is made for telling a temperature, etc. Similarly, features of organisms are teleological, too: eyes are for seeing, a heart is for providing organs with an oxygen, lungs are for breathing, etc. On a top of teleology principles,

Dietz created TAO theory. He categorised it as an ontological theory, as its concern is an understanding the nature of things and the way we use them.

3.3 Affordances

The TAO theory (τ-theory) is a theory about a relation between subjects with purposes, and objects with properties. The theory addresses this relationship as an *affordance*. Merriam-Webster defines affordances as 'the qualities or properties of an object that define its possible uses or make clear how it can or should be used' [141]. However, a more formal definition is offered by the TAO theory. Among other topics, this theory elucidates the relationships between subjects with purposes and objects with properties. An affordance is a term bridging teleological and ontological perspectives [65]. Although a teleology studies objects as purposive entities, ontology concentrates on objects as they are, regardless of the purpose for which the subjects (human beings) can use them. From the theory of affordances presented by Gibson [115], subjects observe and manipulate objects to satisfy their needs and desires. However, objects with properties cannot satisfy their needs and desires alone. Instead, they are satisfied through affordance relationships. We illustrate the core concept behind affordances in TAO theory in Fig. 3.4, and we attach Dietz's [63] quote explaining the notion of affordances below.

> *'If you (subject) want to sit (purpose), you may perceive that you can sit (affordance) on a tree-stump (object), because the height of its surface (property) fits your purpose. So, whereas the purposes of subjects are purely subjective, and the properties of objects are purely objective, an affordance is a subject-object relationship. Because of the unlimited imagination of the human mind, the number of affordances of an object is basically unlimited. Note that the being subjective of an affordance implies the abilities of the subject: for a 2-year old child, the above mentioned tree-stump is not sit-on-able, and for a physically disabled person, a ladder is not climb-able.'*
>
> *– Jan L.G. Dietz, Jan Hoogervorst*

Let us review the idea behind the affordance. We take the definition directly from Dietz [63] and we put it in a more mathematical manner:

Definition 6. Affordance is a subject-object relationship that can be represented by the formula below. The symbol '$*$' denotes the concept 'is in relation'.

affordance: (subject * purpose) * (object * properties)

An example following Dietz's quote could be:

sitting: (José * want to sit) * (this tree stump * 50 cm)

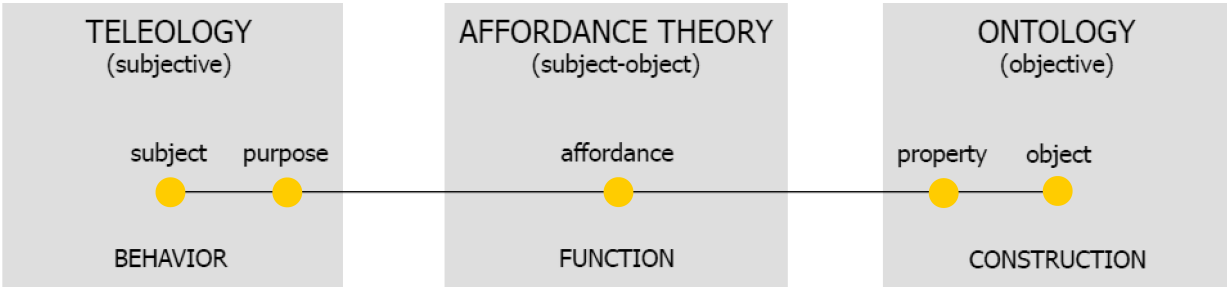

Figure 3.4: Core objects of study in TAO theory presented by Dietz [63]

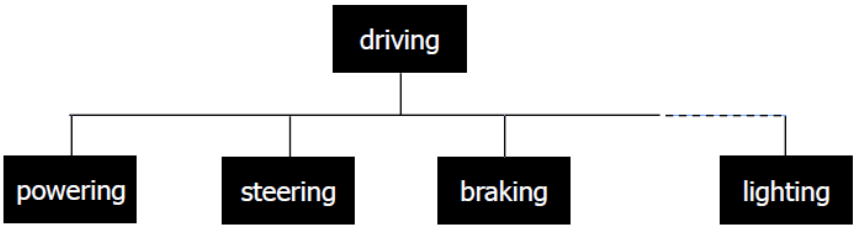

Figure 3.5: Black-box model for the function decomposition of a car [65]

As Dietz and Hoogervorst [63] conclude, 'the basic idea in the theory of affordances is that subjects, in their pursuit of satisfying needs, do not primarily perceive objects and their properties but the affordances that the objects offer them'.

3.4 Function

In TAO theory [65], subjects also create objects. These newly created objects are called *artefacts*. They are typically designed and created with some affordances in mind to provide corresponding functions. Dietz and Hoogervorst [65] explained that 'a chair may offer an affordance sit-on-able while providing a function sit-on to a subject'. According to this reasoning, a hammer is hit-able (function hit), a knife is cut-able (function cut), a thermometer is measure-temperature-able (function measure temperature), etc.

It is clear that a number of affordances (and functions) can be assigned to a single artefact. This assignment depends on the purposes for which subjects wish to use objects. Formally, all functions of a given artefact can be decomposed into a hierarchically organised structure called a *functional decomposition*, which is a black-box model that captures how a system can be used.

In SE, there are several well-established methods for performing functional analysis to construct a black-box model, including Unified Modeling Language (UML), use-case model [93], extreme-programming user stories [18], or COCOMO II Object Points [24]. An example of such a black-box model can be found in Fig. 3.5, which illustrates the functional decomposition of a car. By using this example, Dietz and Hoogervorst illustrated functional decomposition in the TAO theory [65].

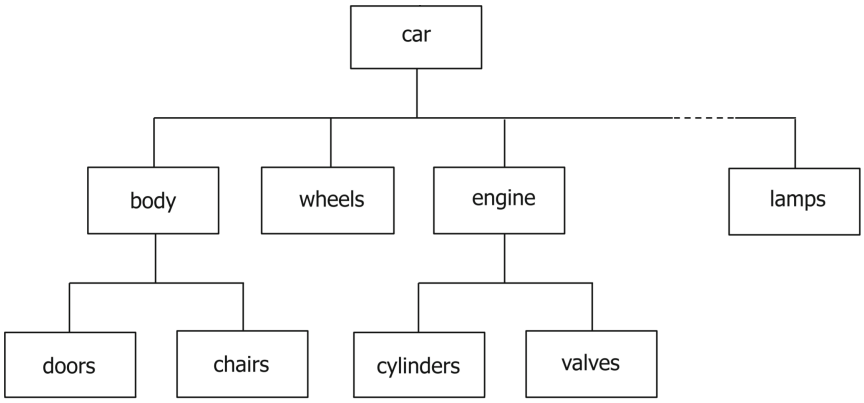

Figure 3.6: White-box model of the construction decomposition of a car [65]

In summary, they stated that [65] 'artefacts are created with an affordance in mind, which is commonly called a function of the artefact'. Functions can be organised into a structure called a functional decomposition. Dietz and Hoogervorst [65] also remarked that 'because of the unlimited imagination of the human mind, the number of affordances for an object is unlimited'.

3.5 Construction

There is a notion of construction discussed in the TAO theory. Dietz and Hoogervorst [65] clarified that 'the function of an artefact is made possible by its construction'. They described the construction as 'the parts an artefact is composed of, their interconnections, and the substances the parts are made of'. Next, the notion of a system was defined in [65] as follows.

Definition 7. Something is a system if and only if it has the following properties:

- *Composition*: A set of elements that are atomic in some category (physical, social, etc.).
- *Environment*: A set of elements of the same category; the composition and the environment are disjoint.
- *Structure*: A set of interaction bonds among the elements in the composition, and between them and the elements in the environment.

The construction of an artefact can be decomposed into a hierarchically organised structure called a *constructional decomposition*, which is a white-box model. Dietz [58] clarified this concept in terms of Definition 7 as follows: 'it is, in fact, a technique to compose a system as a construction of parts (elements or subsystems)'. Typically, a white-box view of a system is captured in SE using a UML component/package diagram [177].

Dietz and Mulder [65] add that a well-known example of a construction decomposition is the Bill of Materials (BoM) in manufacturing. They demonstrate it in Fig. 3.6 that exhibits a part of the BoM of a car. They explain that 'going up the tree is called composition and going down the tree is called decomposition' [65]. However, importantly, one needs to recognise that a composite thing has its own identity, independent of the identities of its components. Moreover, such a decomposition may hold only for some cars, thus not necessarily for all cars.

3.6 BETA Theory and F/C Relationship

To conceive the relationship between a function and its construction, we follow the TAO theory put forward in Section 3.2, and BETA theory. The TAO theory perceives relationships between subjects with purposes and objects with properties, as described above. BETA theory focuses on the design of systems as defined by TAO theory.

The BETA theory (β-theory) stands for Binding Essence, Technology and Architecture [62]. Dietz [62] explains BETA theory as follows. By *Essence*, it is meant a functional and constructional essence. The constructional essence is addressing technical systems. By *Technology*, it is meant all applicable means (e.g., IT or humans) that can be used to implement a system. Finally, *Architecture* is seen as a collective name for functional and constructional principles. However, the notion of Architecture is not as straightforward as it may look like. In TAO theory [63], Dietz points out that a range of engineering disciplines uses the term widely, while defining it just in a vague and ambiguous way. Software engineers and scientists employed the term in a context of (re)engineering business processes and IT systems. For building architects, the term 'architecture' hides a design freedom as a creative flexibility when drawing an artefact.

Here, the notion of architecture is grounded in the context of so-called *generic system development process* illustrated in Fig. 3.7. According to this process, 'architecture is a collective name for functional and constructional principles' [62].

The BETA theory covers the important notions of distinguishing F/C design. A given set of functional requirements typically has multiple constructions to satisfy the requirements. For example, there can be several houses with different appearances that satisfy the same set of functional requirements. These differences can be explained by the substantial amount of design freedom that architects can exercise. When designing a house, an architect can choose from various technologies and materials. The same principle is valid for SE. A software architect can generally choose from a substantial number of technologies and components (as outlined in Fig. 3.7).

All-in-all, BETA theory strongly separates the function and the construction. Moreover, it counts on the design freedom which explains that one function can have multiple constructions. However, based on the subjectivity of functional decomposition and the potential for multiple different constructions for each function, Dietz [58] warned that 'it is a misunderstanding that one can choose the components in a functional decomposition such that they coincide with constructional components, because this is impossible. Black-box

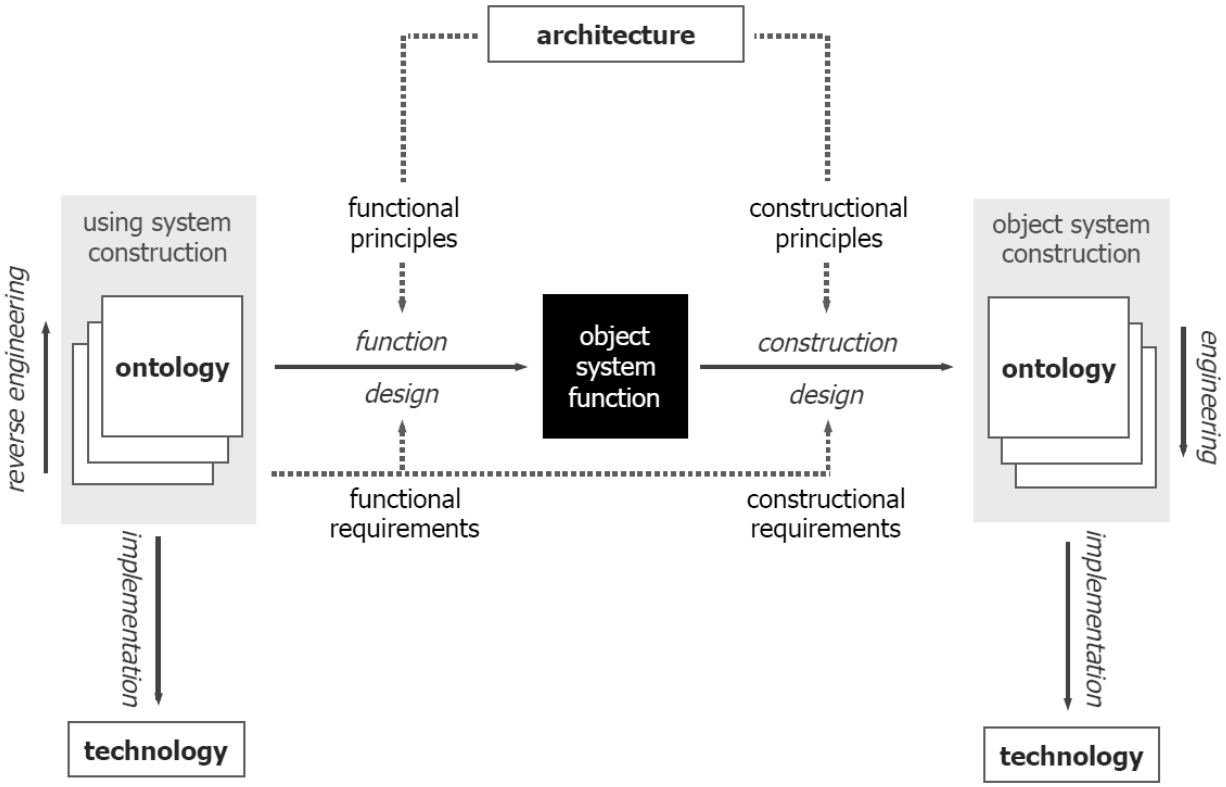

Figure 3.7: Generic system development process [62]

models and white-box models are fundamentally different types of models. There is no way of simply mapping one to the other'. He concluded that 'constructional designers must bridge the mental gap between function and construction' [58].

3.7 PSI Theory and Interactions

PSI theory (ψ-theory) deals with a communication and an interaction between subjects (human beings). It serves to investigate the operational essence of organisations. Diets and Mulder [65] clarify that the word 'organisation' indicates the construction perspective on enterprises. They categorise organisations as social systems, meaning that its system elements are social individuals, called actors. They add that 'the operating principle is that actors enter into and comply with commitments towards each other' [65]. The *actor* is defined as a subject (human being) in an actor role. The actor role, on the other hand, determines the authority that the actor may take and the responsibility for doing so.

The generalised version of PSI theory, so-called General PSI theory, coins a term *transaction*, and it defines a transaction axiom. Therefore, the aforementioned actors interact by performing coordination acts which are raised within patterns called (business) transactions. As Dietz [61] says: 'carrying through a transaction is a 'game' of entering into and complying with commitments'. A result of these commitments are the products/services

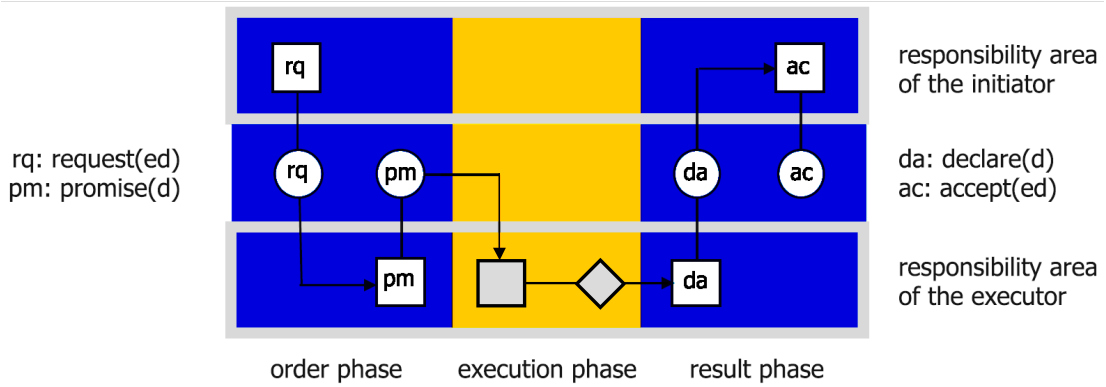

Figure 3.8: The basic transaction pattern [67]

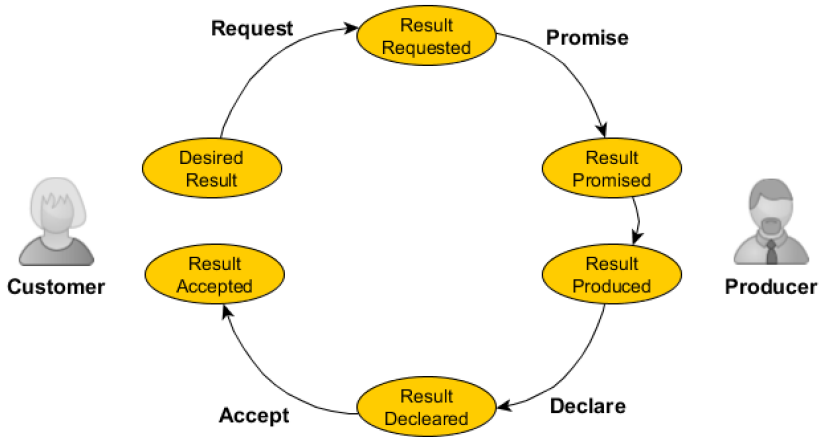

Figure 3.9: Happy flow of basic transaction pattern [58]

the subjects bring about in a coordination. This is again the operating principle mentioned above. Diets and Mulder [65] specify that these coordination acts are communicative acts coming from the category of regulativa in Habermas's *theory of communication action* [99]. They clarify that the result of performing a coordination act is so-called coordination fact.

Both coordination acts/facts are the atomic building blocks of organisations. As mentioned, they occur in patterns called (business) transactions. In industry, they are typically called (business) processes. In these transactions, the interactions happen between subjects who are either in an initiator role or in an executor role. In every transaction, this interaction typically starts by a *request* from the initiator, follows by the *promise* and *declare* by the executor, and it finishes by the *accept* of the initiator. This is known as a *basic transaction pattern* in Fig. 3.8. Its happy flow is depicted in Fig. 3.9.

However, in practice, we do not say yes to all acts of the counterpart. Therefore, so-called *standard transaction pattern* in Fig. 3.10 was invented. Apart from the happy flow, *decline* may happen instead of *promise*, and *reject* may happen instead of *accept*. Then,

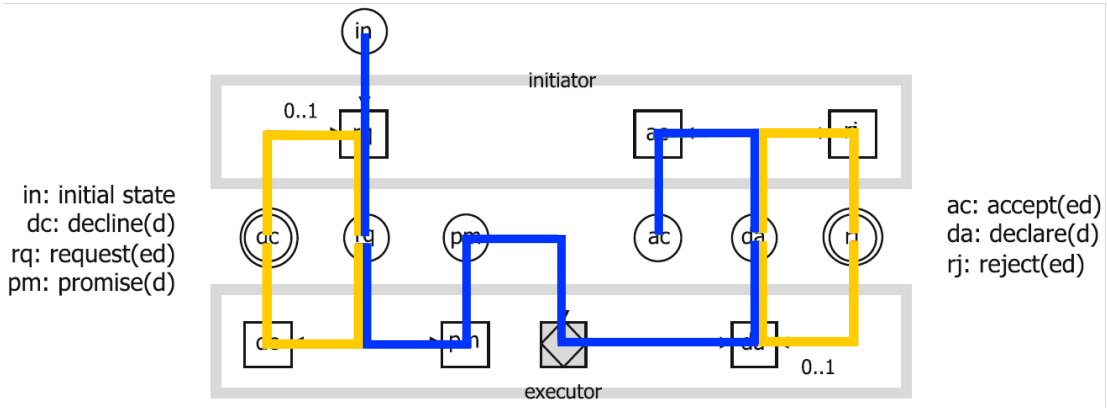

Figure 3.10: The standard transaction pattern [67]

a new attempt may be made, or *quit*, resp. *stop* may end the transaction unsuccessfully. This logic is automatically included in all transactions. It is one of the reasons why the models are so compact – it would need a lot more diagram elements to express the transaction pattern of every transaction kind using a flowchart-like notation.

Nevertheless, real situations may become even more complicated. Thus, such a happy flow in Fig. 3.9 may be rare. Often, the actor may want to revoke ('take back') the act done before. It is addressed by the *complete transaction pattern*, which is considered to be the universal patter in all organisations. It extends the standard transaction pattern of four revocations patterns. In Fig. 3.11, we show the standard pattern (middle part) and the four revocation patterns. As we may see, in such a transaction, the actors my *decline* their promise, or they may *reject* their acceptance. Them. a new attempt may be made, or *quit*, resp. *stop* may end the transaction unsuccessfully.

To conclude, PSI theory is the basis to construct a methodology providing an ontological model of an organisation. This methodology is called DEMO and we introduce it in the next section.

3.8 DEMO Methodology

DEMO is one of the leading methodologies of EE discipline. It has been introduced by Dietz [58] in 1980s. Its main motivation is to provide a comprehensive, consistent, concise, and coherent conceptual model of an enterprise, so that it may cope with current and future challenges [66].

It is a methodology for (re)designing and (re)engineering of organisations [66]. The methodology facilitates an understanding of the communication and interaction principles of subjects (human beings) across the processes of an enterprise. The methodology is grounded in well-founded theories about the construction and operation of enterprises, mainly in the system ontology of Bunge [26], teleology, and the theory of communicative act of Habermas [99]. At the same time, its benefits for practical use have been proven, as

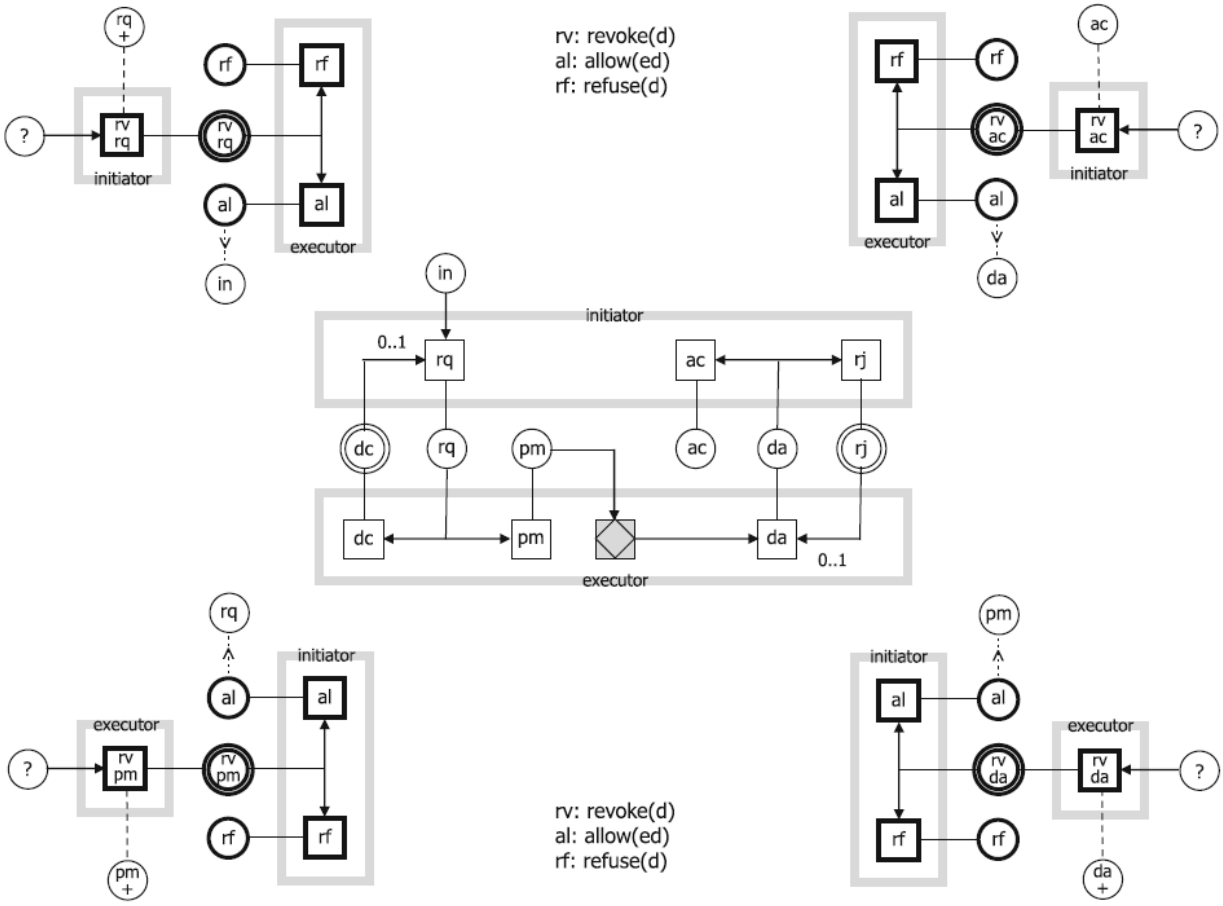

Figure 3.11: The complete transaction pattern [65]

documented, e.g., by Op't Land et al. [163] or by Décosse et al. [54].

DEMO sees an enterprise as a system of actors in a social interaction. All commitments of actors are raised in patterns, so-called *transactions* described in Section 3.7. This concept helps to build a conceptual level of an organisation. DEMO is a methodology that performs a well-defined conceptualisation of a given domain. By identifying all operations of an enterprise, and by describing them in transactions, we can create a proper conceptual model. The model is completely independent to the implementation.

Let us take Op't Land and Dietz [163] to help describing the essential concepts of DEMO.

> *'...A complete, so-called essential model of an organisation consists of four aspect models: Construction Model (CM), Process Model (PM), Action Model (AM), and State Model (SM). The CM specifies the composition, the environment and the structure of the organisation. It contains the identified* transaction kinds*, the associated* actor roles *as well as the information links between actor roles and transaction banks (the conceptual containers of the process his-*

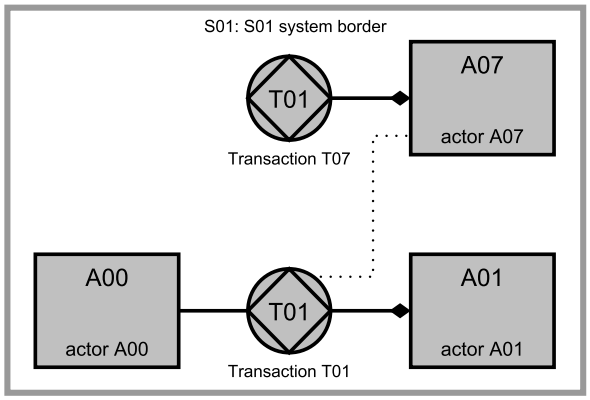

Figure 3.12: Typical constructs of a DEMO construction model [163]

> *tory). The PM details each transaction kind according to the* complete transaction pattern*. In addition, it shows the structure of the identified business processes that are trees of transactions. The AM specifies the imperatively formulated* business rules *that serve as guidelines for the actors in dealing with their agenda. The SM specifies the* object classes*, the* fact types *and the declarative formulations of the* business rules*...'*
>
> *– Martin Op't Land and Jan Dietz, 2012*

Construction Model shows a network of identified transaction kinds and the corresponding actor roles. For example, transaction kind `T01` (Fig. 3.12) delivers a business service to actor role `A00`. `A00` is called the *initiator* (consumer) and `A01` the *executor* (producer). The executor of a transaction is marked by a small black diamond on the actor's role box. The solid line between `A00` and `T01` is the initiator link; the solid line between `A01` and `T01` is the executor link. Fig. 3.12 also shows that another actor role (`A07`) needs to have access to the history of transactions `T01` (production facts as well as coordination facts (e.g., status 'requested', 'promised', 'declared', 'accepted')). This is represented by the dashed line between `A07` and `T01`. However, the diagram notation is not important for our purpose, we will just utilise the underlying concepts.

While PM, AM, and SM are also crucial for DEMO, we do not consider them needed in the scope of our review.

3.9 Chapter Summary

In this chapter, we addressed RO 1.2. We introduced EE theories and outlined their mapping into SE practices. We analysed each theory separately and elaborated on the key concepts of function, construction, and their relationship. Additionally, we introduced philosophical theories that concerns the operation of an enterprise and by some research might be applicable to SE as well.

Chapter 4

Technological Developments

To tackle the research objective RO 1.3, we should broadly understand the origins and latest trends aiming at adapting software to new technology. In Section 2.3, we discussed evolutionary architectures. We took the microservice architecture as one of its good examples. We discovered the importance of being able to replace one microservice for another as easy as switching LEGO bricks.

In Section 1.5.1, we grounded our research scope. We decided to focus to GUI and its technological transitions. Already decades ago, GUI development envisioned a future of LEGO-like design. It typically builds on notions like reusability, modularity, and components, and it often follows certain design patterns. Therefore, we dedicate this chapter to the historical evolution, research, and industrial practices targeting GUI development.

In Section 4.1, we start with an overview of how the vision of CBS evolved. Next, in Section 4.2, we will distinguish design and architecture patterns commonly used in GUI development. These patterns are used by few GUI frameworks that we refer to in this dissertation thesis. Therefore, in the Section 4.3, we introduce these frameworks briefly. In Section 4.4, we will complete the overview of GUI development by providing a rough intro into component libraries, and their strengths and weaknesses in terms of the vision of CBS. Finally, we will close this chapter in Section 4.5. We will provide an introduction to RPA and BPM as they also play a role in a topic of technology transitions.

4.1 Evolution of Component-based systems

As the roof tiles, crossbeams, or door frames are building blocks for setting up a house, components are building blocks for assembling software. Kaisler [119] reminds that one of the best analogies can be found in a famous Danish building kit LEGO. This legendary LEGO kit saw the light of the day back in 1949. It consists of a set of bricks. Each plastic brick has a standardised interface with sockets on one side and plugs on the other. The interface allows the brick to be connected to another one. From such a set of bricks of whatever shape, we can build rather large systems like castles, crafts, or even entire cities. However, although these LEGO constructions look complex, they are actually quite

simple since we can change them rather easily. For instance, we can unplug an entire tower and plug it somewhere else. Unfortunately, the software components in CBSs can have more functions than the LEGO brick. It influences their standard interface which can vary depending on their functions. Moreover, if we replace a single LEGO brick with another one differing only in colour, it is highly unlikely that the overall system (e.g., castle, aircraft) behaviour will change. In software, such a replacement might influence the overall system due to the coupling with other components. It means that in software, we have to identify all possibly affected parts of the system. NST discusses this in a context of CEs. We have to pay special attention to verify whether the whole system behaviour has not been affected by that change.

With respect to that, software solutions can exhibit certain qualities and characteristics. They concern the design of a given application. Many of them are worth to be mentioned because we refer to them throughout the dissertation thesis. To be complete, we list them together with their brief description.

1. *Reusability* is the extent to which certain building-blocks can be shared across a software product.
2. *Flexibility* concerns 'the ability of a resource to be used for more than one end product [69].
3. *Usability* specifies how easy it is to use a given piece of software or component.
4. *Composability* concerns the ability to meet user requirements by assembling a system out of pre-tested components.
5. *Comprehensibility* is defined as a capability of being comprehend or understand. In SE, we usually speak about a comprehensibility of a source code. We see comprehensibility as a measure on how easy the developer can understand a given piece of code.
6. *Interoperability* is an ability of a system to work with other systems regardless their architecture, operating system, or other specifics. In terms of CBSs, we see the interoperability as a characteristic that enables a composition of the system out of heterogeneous, reusable components. These components might have been developed by different teams or they might have been intended to other platforms. However, we expect that the components were created with a similar usage in mind.
7. *DRYness* is a principle in software development that stands for 'Don't Repeat Yourself'. Its overall goal is to reduce a repetition of information of any kind.

Regardless the importance of all those characteristics, we primarily focus on the reusability that is most relevant for our research. Therefore, in Section 4.1.3, we footprint the concepts of reusability, as it has turned out to be an important pattern in software development. Additionally, in Chapter 8, we specifically inspect the trade-off between usability and flexibility that plays a key role when designing technology agnostic GUI.

4.1.1 McIlroy's Dream of Component Library

The tipping point for CBS was NATO Symposium in 1968 where McIlroy [68] presented his dream about a component library:

> *'...I would like to see components become a dignified branch of software engineering. I would like to see standard catalogues of routines, classified by precision, robustness, time-space performance, size limits, and binding time of parameters. I would like to apply routines in the catalogue to any one of a large class of often quite different machines, without too much pain...'*
>
> *– Doughlas McIlroy, 1968*

He dreamed about components organised in standard libraries. One could have purchased the component tailored to his exact needs and reused it in a system without inspecting its internal details. For example, McIlroy [68] anticipated a lowly sine routine to exist. The catalogue would offer this routine in a range of modifications varying in precision (e.g., different approximating functions), floating vs fixed computation, argument range (e.g., 0-pi, 0-2pi, ...), robustness (ranging from no argument validation through signalling of complete loss of significance). These combinations would lead to an inventory of, e.g., 300 sine routines. This is actually what McIlroy called 'mass production' in the title of his paper. He didn't address a multiplicity of replications of each routine, rather he meant a multiplicity in something that the manufacturing industry commonly refers to as 'models' or 'sizes' [68].

McIlroy's vision grounded a principle of a black box reuse of 'routines', respectively, building blocks. It is a concept where the interface becomes a first-class citizen. Implementation details are never revealed. The functionality can only be derived from the exposed interface.

4.1.2 Bemer's Call for Software Factory

In the same year, when McIlroy foresaw his dream about a component library, reusability has been discussed in the context of 'software factories'. Cusumano [142] mentioned that this trademark referred to an approach highlighting the importance of a standardisation of development methods and tools, disciplined project management, quality control, design, and finally a structured collection of related software assets. Although McIlroy focused on the reusability aspect of factory-like approaches and introduced a 'component factory', Bemer [180] reinforced the need for standardised tools and controls. He argued that a software factory should be a programming environment residing upon a computer. The program construction, checkout, and usage should be done entirely within this environment with the help of build-in compilers for machine-independent languages, various simulators, support for test cases, documentation tools, and many more. As Bemer later mentioned on his blog, there seems to be no component more important to the software factory than interchangeable and reusable piece parts [21].

Toshiba embedded that in practice in 1977 [142]. Its R&D group established a highly disciplined factory grounded in four policies. One of the policies regarded 'reuse of existing designs and code when building new systems'. As nailed by Cusumano [142], rather than writing all software from scratch, Toshiba delivered 'semi-customised' programs that combined reusable designs with newly written modules. To gain the reuse objectives, Toshiba's methodology required programmers to deposit a certain number of modules in a library each month. Managers asked the developers to register their reusable modules in a database and awarded those who registered frequently reused modules. Nevertheless, by these investments in applying reusability in practice (e.g., bigger budgets for documenting reusable parts, awarding personnel, etc.), Toshiba gained better productivity and quality improvements.

4.1.3 Reusability

As mentioned earlier, the important property of CBS is reusability. This notion is known since humans have been around. We face the problems and learn their solutions. Once we encounter a new problem, we try to adopt the same or similar solution of already existing and solved problems. As put by Prieto-Díaz [182]:

> *'Proven solutions, used over and over to solve the same type of problem, become accepted, generalised and standardised.'*
>
> *– Rubén Prieto-Diaz, 1993*

To manifest his thoughts, he developed a faceted taxonomy of eleven types of reuse. Keisler [119] captured it similarly to Table 4.1

The process of reuse seems to originate already in 1950 with macro-processors replacement systems [206]. Back than, a first subroutine library was created on Electronic Delay Storage Automatic Calculator (EDSAC) [227]. By 1951, the library contained a number of subroutines organised in categories like floating point arithmetic, arithmetic operations on complex numbers, printing, division, quadrature, double-length numbers, etc. [228]. The reuse was just about a replacement of macro definitions that were contained in the text. The text was then 'compiled' to produce a final program to execute [228].

Hooper [114] explained that software reuse is a goal. The reusability is necessary to achieve the goal. He defines reusability as 'the extent to which a software component can be used (with or without adaptation) in multiple problem solutions'.

The best examples of a successful reuse can be found in mathematical models. In our research, we are mostly concerned about a reuse of artefacts. In SE, we can identify several levels of such reuse. Historically, the first attempt at reuse was achieved on a level of functions in Structured Programming (SP). The second attempt was introduced with Object-Oriented Programming (OOP) where classes became reusable artefacts. Naturally, libraries and packages emerged to collect classes as reusable code constructs. Finally, CBS

Type of Reuse	Description
Ideas	The reuse of formal concepts such as general solutions to a class of problems
Artefacts	The reuse of software components (such as the Booch Ada components)
Procedures	The reuse of software processes and procedures
Vertical	The reuse of software within the same domain and even within the same application suite
Horizontal	The reuse of generic parts across multiple applications
Planned	The reuse of guidelines, development and testing processes, and metrics across multiple projects
Ad hoc	An informal practice in which components are selected from general libraries in an opportunistic fashion
Compositional	The reuse of existing components as the building blocks of new applications
Generative	The reuse of specifications and requirements to develop application or code generators
Black-box	The reuse of software components without any modification; usually linked together with glue code
White-box	The reuse of components by modification and adaptation

Table 4.1: Prieto-Diaz's taxonomy of reuse. Described by Keisler [119]

development was proposed. It perceived a component as a modular layer on top of objects in object-oriented languages [69]. Szyperski [34] defined a component as follows:

Definition 8. A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties'.

However, after decades of various approaches to reuse artefacts in software, we still seem to be far from the original McIlroy's dream. The modular architecture must bear other challenges, e.g., high coupling and low cohesion. As again Prieto-Díaz [182] explains in his status report: 'the problem is not a lack of reuse, but a lack of widespread, systematic reuse ... the programmers are used to reuse, but they do all this informally'.

Section Takeaway

The notion of CBS dates back to the 70's when McIlroy [68] envisioned that components should become a dignified branch of software engineering. Shortly after that, Cusumano [142] triggered discussions about reusability as an important approach to standardise software development methods and tools. The types of reuse were further organised by Keisler [119] into a taxonomy. Nowadays, reusability is considered one of the characteristics of software solutions next to flexibility, usability, etc. When referring to a reusable component, we stick on its definition by Szyperski [34].

4.2 GUI Architectural and Design Patterns

GUIs became an indispensable part of many software applications. Nowadays, they fill the screens of our mobile phones, tablets, desktops, smart watches, e-book readers, and other devices. Lately, they even replaced areas that were traditionally managed manually. For example, Tesla's cars do not have any physical dials behind the steering wheel (or anywhere else), thereby the LCD dashboard is largely used to drive the car.

As such, the requirements on User Interface/User Experience (UI/UX) have changed over the years. Multiple different approaches and architectures were introduced to tackle problems encountered on this journey. With respect to the RO 1.3, in this section we will expand our knowledge base with widely known architectures and design patterns that were used for GUI development throughout the history. We will compare them and summarise their pros and cons. This may help us to classify GUI frameworks – the set of classes and interfaces defining the elements and behaviour of a window-based GUI subsystem.

4.2.1 Architectural Versus Design Patterns

GUI frameworks typically use certain design or architectural patterns. There is a widespread confusion between these patterns. To avoid it, let us describe their difference. The design patterns are known from the book *Design Patterns, elements of reusable Object-Oriented software* by the Gang of Four (GoF) [89]. The authors defined them as:

> *'Design patterns are descriptions of communicating objects and classes that are customised to solve a general design problem in a particular context.'*
>
> *– Erich Gamma, Richard Help, Ralph Johnson, and John Vlissides, 1995*

In other words, the design pattern is an abstract way of solving recurring problems. They can be used on different levels of abstraction in source code classes as well as in big SW solutions.

On the other hand, the architectural patterns have a broader scope. They focus on describing an organisation at its highest abstraction level. Although they also serve solving problems, these problems typically reside in keeping a mental picture of a big system or its subsystem. Architectural patterns often, if not always, use many instances of design patterns. This is visible from the GoF book [89]. Here, for instance, Model View Controller (MVC) architectural pattern is used as an example of many different design patterns that collaborate [89, p. 4]. The opposite is not true, design patterns do not use architectural patterns in their description.

The literature recognises many design patterns. The patterns relevant to our research were deeply inspected together with Mareš [A.13]. Next, we provide its brief excerpt.

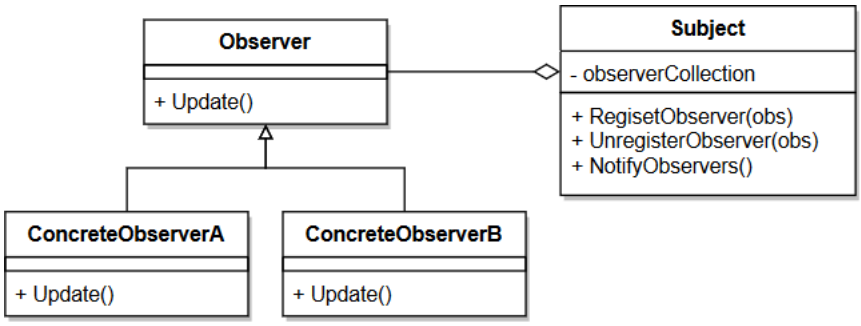

Figure 4.1: Observer pattern [89, p. 293]

4.2.2 Design Patterns

Observer Pattern. The Observer Pattern (OP) [89, p. 293], also known as Publish-Subscribe Pattern, comes from the need of establishing a relationship between a subject and its observers.

GUI frameworks often separate the presentations aspects of the GUI from the underlying application data [226, 127, 133]. Data structures defining these application data and presentations can work together, or they may be reused independently. For instance, an application depicting information in the same application data object may present them in the form of a spreadsheet or a pie chart. The spreadsheet is not aware of the pie chart and vice versa. Therefore, each can be replaced without letting the other know. However, when the user updates the information in a chart, the spreadsheet reflects the change immediately.

The observer pattern describes these types of relationships. It is illustrated in Fig. 4.1. Observers are notified whenever the subject undergoes a change in a state. The observers then retrieve the subject's state and carry on their business. The subject does not need to know who the observers are nor how many there are. Therefore, it can be useful to announce changes in a loosely coupled way without assumptions about the observers.

Composite Pattern. The Composite pattern [89, p. 163] is a partitioning design pattern. It comes from the need of representing part-whole hierarchies of objects. Such a representation is typical for graphical applications, e.g., drawing editors, that let us orchestrate complex diagrams from simple components. These components can be composed to a bigger component, which in turn can be composed to an even bigger component. The implementation of this kind of application typically includes graphical primitives such as curves, lines, labels, and similar. Therefore, it is an abstraction to treat individual objects and compositions of objects uniformly. We depict it in Fig. 4.2.

The key to this pattern is the abstraction class that represents both primitives and their containers allowed to use their common functionality. This helps us to manipulate the hierarchies of objects without special treatment to view them all as Components.

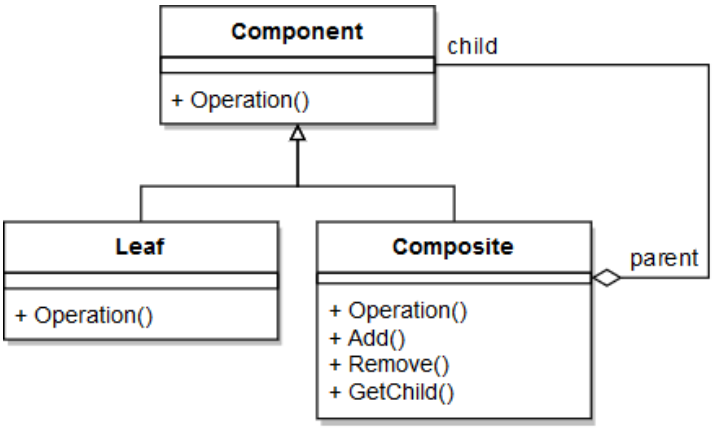

Figure 4.2: Composite pattern [89, p. 163]

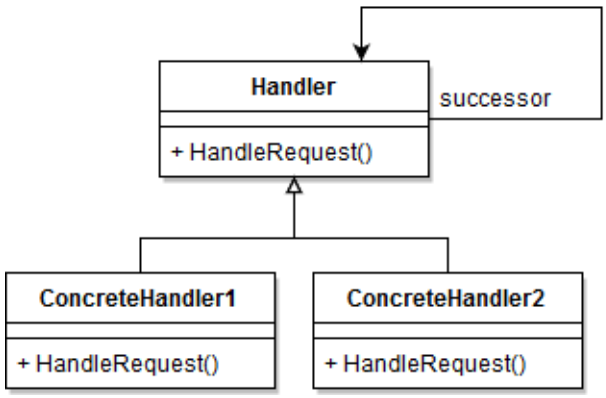

Figure 4.3: Chain of Responsibility pattern [89, p. 223]

Chain of Responsibility Pattern. The Chain of Responsibility (CoR) pattern [89, p. 223] describes a way to avoid coupling between sender of a request and its receiver. According to Gamma et al. [90], the pattern can be illustrated in a problem of context-sensitive help facility for GUI. The users can obtain help on any part of the GUI by clicking on it. The provided help depends on the context and on the part of the GUI that is selected. Gamma et al. [90] add that 'for example, a button widget in a dialog box might have different help information than a similar button in the main window. If no specific help information exists for that part of the interface, then the help system should display a more general help message about the immediate context–the dialog box as a whole, for example'.

According to Vinoski [219], 'in object-oriented terms, the CoR pattern aims to decouple a caller from its target object, and it accomplishes this by interposing a chain of objects between them. This arrangement lets each object in the chain act on a request as it flows from the caller to the target'.

The idea builds on handlers privileged to react to a request. These handlers also have a reference to its successor. Thus, if they see the fit, they can forward the request. The

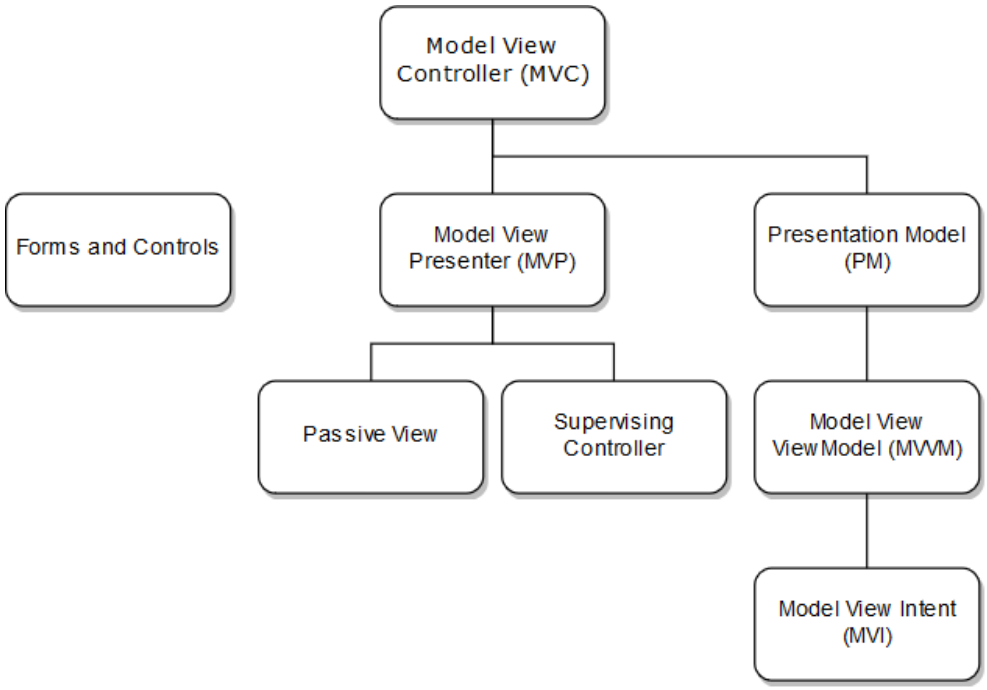

Figure 4.4: Presentation patterns

specific handlers are derived from a common class, see Fig. 4.3. This pattern might be useful in similar GUI situations as described by Gamma et al. [90]. Thus, when we register a user's click action while we let different entities react to this input sequentially. This is also known as event routing.

4.2.3 Architectural Patterns

As explained in Section 4.2.1, architectural patterns are often based on the aforementioned design patterns. With Mareš [A.13], we concluded that some of the architectural patterns are enormously difficult to get the grasp of what they are supposed to present in their pure form. There are dozens of sources describing MVC pattern, yet they often do not present the same principles and ideas. Some of them are even hard to be considered an adaptation of MVC at all. Martin Fowler [84] is often used as a reference.

Before diving into individual patterns, with Mareš [A.13], we present a higher-level overview in Fig. 4.4. The approaches are organised loosely by chronological order MVC being the oldest and Model View Intent (MVI) the latest enhancement. Their linkage is a take on showing their line of evolution as it is not exactly easy to always pinpoint the origins.

Forms and Controls. In 90s, the development of server-client application in Visual Basic and Delphi was trending. This encouraged an architecture pattern called Forms and

Controls (FaC). Although by that time, it was not labelled with this term, we stick on what Fowler [84] came up with as FaC later.

This pattern builds on basic building blocks – custom-made forms out of generic reusable controls. These controls represent the elements of the GUI – text-box, button, label, etc. In practice, a wast majority of GUI frameworks offer corresponding ready-made controls that could populate a specific form. Custom controls may be implemented when necessary, yet they remain generic and reusable for different forms and applications.

The form fulfills two main roles:

- *Screen layout* – the arrangement of the controls and the hierarchical structuring between them;
- *Form logic* – behaviour that is difficult to get out of control alone, usually some form of state or shared metadata.

Practically, the corresponding GUI frameworks often come with a graphical editor. This allows developers to drag & drop controls to a convenient place in a form. Such an approach delivers a well-known WYSIWYG[1] experience. However, it also has drawbacks, e.g., when dealing with increasing demand on responsive design.

This pattern typically manipulates with different types of data:

- *Record state data* – the data residing directly in the SQL database that is possibly shared with multiple users and applications simultaneously;
- *Session state data* – a copy of the record state inside in-memory record set. It is exclusively used by the running application and must be published by 'save and commit' to get the corresponding record state updated;
- *Screen state data* – the copy of the data in GUI elements, this data is displayed on the screen.

Synchronisation of the session state and screen state data is a crucial behaviour of every GUI. The *Data Binding* is the easiest way to do that. The idea is that any change to the screen data is propagated to the underlying record set and vice versa. Thus, a user modifying the text in a text-box updates the corresponding cell in the Record Set. We face two issues with data binding – cyclic updates, and record set-specific logic. Cyclic updates happen when a change to the control changes the record set, which updates the control, which updates the record set... The other issue relates to the parametrisation. Data binding typically requires a type of parametrisation that often will not fit the required logic. The variance must be calculated by the GUI itself, e.g., in so-called event handler. In such cases, the logic is placed directly in the form which is application specific, thus it may bring further issues, e.g., with maintenance, or with migrating the logic to a new form or application.

[1] What you see is what you get. Approach used not only for designing user interfaces. The author can see the result of his work directly. For example MS Office Word uses this approach with documents.

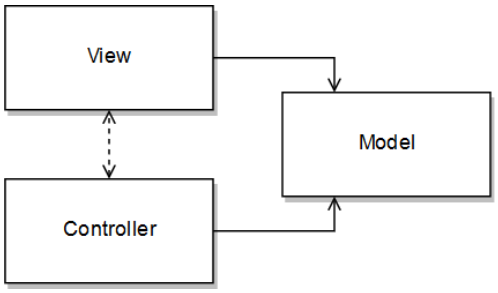

Figure 4.5: MVC pattern

FaC pattern is the simplest to grasp and is very straightforward. The developer orchestrates application-specific forms from generic controls. The forms define the layout and structure of the GUI, and they respond to specific events triggered by the user. Simple data edits are usually handled by data binding. Complex logic, on the other hand, is implemented using form or application-specific event handlers.

Model View Controller. One of the very first attempts to invent any sort of GUI architecture was made in Smalltalk-80 [201]. Here lies the origins of MVC pattern. Although it dates to the era of monochromatic graphical systems built in the 70s, its concepts are still well used today.

It is grounded in the idea of *Separated Presentation*. In SW development, we may recognise two types of objects – domain objects and presentation objects. Domain objects model our real world, these are also recognised as business logic objects. Presentation objects are solely used as GUI elements on the screen, e.g., text-boxes, drop-down menus, charts, etc. The notion of separated presentation isolates these domain objects from the presentation objects. This already came in handy in Unix culture allowing for one underlying program to have GUI and command-line interface.

In MVC, the domain objects are referred to as *Model*. Model is completely independent from GUI. The MVC is also assuming actual domain model objects not a record set. This simply reflects the fact that unlike Forms and Controls, that were intended to manipulate records in a database, MVC was initially intended for Samlltalk purely object oriented environment.

The presentation part of MVC constitutes of two elements – *View* and *Controller*. The Controller responds to the user's input and figures out what to do. The View's job is to present the Model to the user. View and Controller have a direct reference to the Model. They also have references to each other, but on purpose, this connection is used occasionally. Typically, there are many Controller-View pairs. Each control on the screen has its pair, and the screen itself has a pair too. Therefore, the very first step in responding to the user's input is to decide what controller to execute.

Controller-View pairs may also be reused. They are plugged into an application specific behaviour. There would also be a higher level View representing the whole screen and describing the layout of the lower level controls, similarly to form in Forms and Controls

pattern. Unlike the form, however, there are no events raised by controls and no event handlers in the higher level View. All information is conveyed through the Model.

MVC works slightly else comparing to FaC. There is no interaction between View or Controller and another View or Controller. There are no events and no handling of the visual logic. When the Controller changes the value in the Model, it does not update its View directly. These are what M. Fowler calls *Flow Synchronisation* and *Observer Synchronisation*.

- *Flow Synchronisation* – the element that is changing directly updates all those that need to be updated. This is a heavy handed approach for a rich GUIs. For instance, when omitting data binding, all interactions of Session state and Screen state data (described in FaC) must be done manually by the developer. Typically, this means opening a screen, hitting the save button and other interesting points in the application flow.

- *Observer Synchronisation* – the essence of this synchronisation is that each screen, with its associated screen state, acts as an Observer on a common area of session data. The Controllers are completely oblivious to any other widget needs. This is very useful especially in GUIs with multiple screens showing the same data, like graphs and tables. Imagine dealing with forms synchronisation that would need to check what other forms are open to propagate changes. So in this case the OP is like a blessing, when it is not a blessing at all is when you want to read the code and find out what is going on. The inherent obfuscation of the OP functionality means that what is really going on can be only seen during a debugging time.

MVC can be credited for the idea of *Separated Presentation*. This means that we have isolated presentation layer, Controller-View pair, and the domain Model. Each Control has its own pair – the Controller handles the user's input, the View presents information. The communication is mainly done through the Model. Finally, we have the great contribution of OP to the observer synchronisation that indirectly updates controls.

Model View Presenter. The term Model View Presenter (MVP) was coined by Potel [148] when it appeared in his paper dated back to 90s. It is a pattern lifting the best of what we already know from FaC and MVC. Reusable widgets are directly taken from FaC. These are combined with the Separated Presentation and isolated domain model presented in MVC. Additionally, it adds a requirement on GUI testing.

Potel [148] describes View as a structure of widgets similar to controls on a form. However, he removes the pairing. We do not use Controllers in the sense we had in MVC. The user's input is handled by a Presenter that decides what to do. Although the View has an initial entry point for actions of users, it just delegates the control to Presenter. Potel [148] outlines the scenario when Presenter interacts with Model using commands and selections. This idea is useful as it enhances testability and enables undo & redo functionality. Although the Model is updated by the Presenter, the View is ideally

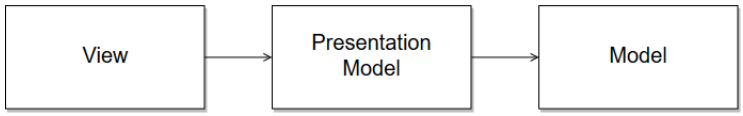

Figure 4.6: Presentation Model

updated using the OP that we know from MVC. In complex actions, the Presenter gets involved and sets the View directly. This is what become known as *Supervising Controller*.

Potel kept the term Presenter clean in his paper. Unfortunately, the later adaptations did not. Some MVP literature confuses the meaning of Controller with the meaning of Presenter. There is a solid case for calling it Controller as it handles user input. Unfortunately, some terms like *Supervising Controller* or even some frameworks like ASP.NET MVC do not strictly follow the original terminology.

Presentation Model. As we have seen with MVC and MVP, rich GUIs brings challenges with presentation layer. The first issue is the placement of the View logic. The second one concerns the placement of the View's state caused by the user interaction.

Unfortunately, most MVC or MVP frameworks encourage developers to put the presentation logic directly into the View. The PResentation Model (PRM), presented by M. Fowler [85], strives to remedy this. The PRM is essentially a self-contained class representing all what any GUI framework needs to know or use in order to render controls. Multiple views can utilise a single PRM, but each View should refer to a single one. Composition is possible and a PRM may contain several child PRMs, but each control will again refer to a specific one.

To do this PRM has data fields for all information necessary for the view. This concerns the content of the controls, and it also includes information about their visual – if they are visible, enabled or disabled, highlighted, etc. It does require PRM having these fields for every control. Unused properties can be omitted.

PRM comes with a drawback linked to a tight synchronisation. Suddenly, synchronisation does not only happen on the level of screens, it also happens on a lower level – field or key level synchronisation. This opens the possibility for fine-grained synchronisation. Fowler [85] discourages from it as it brings a lot of difficulties, especially when things do not work as intended. However, it depends on the nature of a specific project. Coarse-grained synchronisation in the form of syncing the whole state of View with Presentation Model is definitely simpler.

There is also a question of where to put the synchronisation code. PRM enables us to test the synchronisation, which should already be a fairly simple code (coarse-grained sync for sure). On the other hand we can choose the View, this is a natural place for it as the PRM can be oblivious to the View completely. If we ever feel the need to write tests for anything in the View objects, it might signal that we need to rethink how this synchronisation works and what code lives where.

PRM steps into providing a place for visual logic and View state. Widgets do not observe domain Model instead they observe PRM. This fits well to the architecture of rich and complex GUIs. On the other hand, it calls for tighter synchronisation with View making heavy use of OP that could be alleviated by frameworks.

Model View ViewModel. Model View ViewModel (MVVM) architecture pattern was first presented by Gossman [146] in 2005, when he shared this idea in Microsoft's blog. Essentially, it is grounded in the PRM. This pattern was primarily developed for Windows Presentation Foundation (WPF) and then used by Silverlight and Universal Windows Platform (UWP).

Its idea is identical to Fowler's PRM [85]. However, it brings more to the table. The View described declaratively using a modified EXtensible Markup Language (XML), so-called EXtensible Application Markup Language (XAML). XAML determines the visual appearance. It is expected that XAML is maintained by a designer, not necessarily a developer.

MVVM builds on a strong Data Binding between View and ViewModel. This is handled by the framework and the problem of tight synchronisation is solved under the hood for anyone using this technology. MVVM promotes using ViewModel commands that are triggered by GUI events. Again, this is beneficial for reusability and testing.

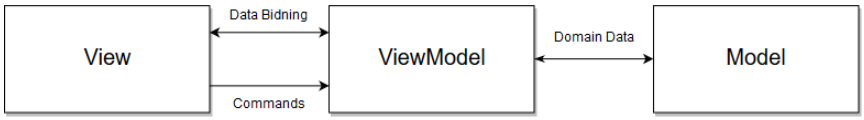

Figure 4.7: MVVM Pattern

MVVM grew from .NET ecosystem and thereby is linked to Microsoft's implementation, the idea is also referred to as Model View Binder. Correspondingly, Java has its own implementation, ZK framework [235] that does not uses Microsoft's XAML, but ZK User Interface Markup Language (ZUML). Similar, fairly popular implementation is in JavaScript – KnockoutJS. MVVM does not bring anything completely new in terms of architecture it is more of an extended implementation of PRM.

Model View Intent. MVI is the latest evolution on the MVVM pattern. It was introduced by Medeiros [46] in his JavaScript framework Cycle.js. Further, it was readily adopted by Android developers and Kotlin environment where it tackles few cumbersome problems in mobile GUIs.

It mainly focuses on the mutability of ViewModel which gets often misused by developers. Additionally, it addresses the coupling between View and ViewModel in the form of tight synchronisation, and it concerns asynchronous events that are more present in web and mobile applications rather than on desktop. To answer this problems MVI is building on the concept of *Reactive programming* from functional programming. The term is spread to reactive applications and reactive frameworks. Adding ideas of states together

with related immutability and unidirectional flow. The core principle can be described in terms of a mathematical formula as follows:

$$view(model(intent()))$$

User acts on the GUI and exhibits intents. These intents are processed by a Model. They become a basis for the View to render the results. The Fig. 4.8 illustrates the essentials of the MVI pattern.

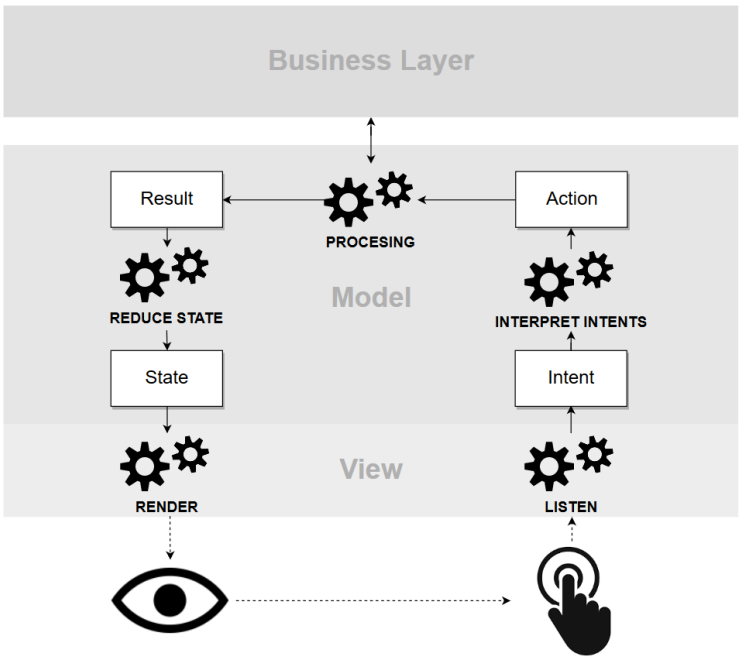

Figure 4.8: MVI Pattern

The View listens to the user actions. These actions are further handed over to the Model in the form of intents. The Model is created for the purpose of GUI. Thus, it plays the same role as ViewModel or Presenter in MVVM or PRM patterns. In this pattern, the Model consumes intents. As an intermediate product it creates a result caused by user intent. Finally, it goes through state reduction. In this step, it combines the current State of the GUI with results, and it produces a new instance of State. It is passed to View and represents a complete description for rendering. The loop is closed when View renders this State and awaits the next user's action. The flow of information goes only this way. There are also no side effects, the only place where information is held and exchanged is during processing in business layer.

This is a pretty complex setup, but it allows for multiple asynchronous actions ultimately affecting the View without the aforementioned problems. This is achieved thanks to the immutable State, that is created in the state reduction step. The problem of tight coupling of View and Model is minimised as the contracts are very simple. The View is only capable of rendering State and producing Intents. The Model is ultimately capable

of consuming Intents and producing a new State. The connection here is realised just by OP.

Overall, this is the most complex pattern for a presentation layer. However, it addresses problems in web and mobile GUIs that are hard to solve in other architectural patterns. It makes it a powerful pattern, if we consider that it works smoothly with asynchronous actions and asynchronous streams where data trickle by pieces. Ultimately, developer has to decide if the trade-off for this complexity is worth it.

Section Takeaway

Software developers distinguish between design and architecture patterns. Design patterns are abstract ways of solving recurring problems. Architecture patterns focus on describing the highest level of abstraction and are often orchestrated from different design patterns. Although they are often discussed and implemented by GUI frameworks, there is no consensus in their definition, nor their implementation. However, they are continuously developed from 90s to reflect the needs of current HW and SW requirements.

4.3 GUI Frameworks

In the previous section, we discovered design and architecture patterns that are common to many GUI technologies. It is not realistic to elaborate on all available GUI frameworks and libraries in different technology stacks. Therefore, with DSRM, we limited our research environment to .NET and JavaScript. In this section, we will introduce a few that have something in common with our research goal and objectives, or they were determined by the technology stack of our industrial case study.

4.3.1 ASP.NET MVC

The framework ASP.NET MVC is Microsoft's edge technology. It intends to develop web pages following a general MVC pattern introduced in Section 4.2.3. This pattern is based on separation of concerns principle mentioned in Section 2.2.2. In terms of TAO theory, ASP.NET MVC separates a function from its construction. Let us review it now.

In our research, we are concerned with the creation of convenient GUI that corresponds to a functional decomposition of an application and that may continuously move from one technology to another. In Section 12.3.1, we will show that the functional decomposition can be obtained from an information about the purpose for which the component can be used. ASP.NET MVC tends to describe this purpose by so-called annotations.

The annotation is a set of attributes extending the GUI components with an information about what they can be used for. In ASP.NET MVC, it is a domain model that is annotated, not the GUI itself. Therefore, the GUI elements understand their purpose out of the related annotated model.

For instance, we can annotate a string property[2] with [Email] attribute to state that value of this property should be a valid email address. Consequently, the text field intended to edit the email automatically restricts the user from entering grammatically invalid email addresses. The attribute [StringLength] defines a maximal allowed length of a string property. The attribute [IsRequired] forces the user to enter a value of a mandatory property. Property annotated with a [Regexp] represents a custom validation of strings against a defined regular expression. Moreover, we can use an attribute [UIHint] to advice the GUI to generate required GUI element. For instance, if a property represents a gender, we can ask for a drop-down menu with two options, male and female.

To summarize, ASP.NET MVC is a flexible framework. It can derive some GUI characteristics from a domain model and adapt the GUI components accordingly. It automatically finds a convenient editor if a developer encrypts the purpose into the annotation, e.g., an [Email] attribute. However, out of the box, it can not cope with domain-specific attributes and GUI elements. The ASP.NET MVC attributes only cover general annotations. Moreover, it does not consider a notion of the affordance (see Section 3.2) when building the GUI. Thus, it hardly adapts GUI to a user that is, e.g., physically challenged. In the scope of our research, we must cover both a broader domain of possible annotations, and a notion of affordance to tailor the GUI with respect to the purpose of the user.

Nevertheless, ASP.NET MVC shows that not just a purpose of required GUI components matters. The domain model is equally important to find a suitable construction of GUI.

4.3.2 Windows Forms (WinForms)

Let us now discuss the GUI framework that is often referred to in this dissertation thesis.

Windows Forms (WinForms) [144] is the GUI framework shipped by Microsoft together with the first version of .NET in 2002. It was meant to support the GUI development of .NET applications. At that point of time, it was a completely new concept. It builds on a previous Microsoft Foundation Class Library written in C++. Back then, Windows XP operating system was trending on the market. Nearly all applications were desktop-oriented as the internet was booming.

Even though it is an almost two-decades old framework, it is by no means not a dead platform. WinForms is still supported today, like high DPI[3] scaling coming in with .NET Framework 4.8. It is also supported by the latest version of .NET Core 5.0. These decisions might be indicators that the framework is present in many business critical applications that are hard to replace. Microsoft does not want to let its corporate customers down.

The architecture of WinForms. Let us take a closer look at the architecture of WinForms. As suggested by its name, it is based on FaC architectural pattern put forward in

[2]Property is a language construct in C#. It is a class member providing a flexible mechanism to access private class fields.

[3]Dots per inch (DPI) is a measure of spatial printing, video or image scanner dot density.

Section 4.2.3. Certainly, the documentation presents it as visual surface on which we display information to the user. As already explained in Section 4.2.3, WinForms works with so-called *controls*. They represent the elements of the GUI – text-boxes, buttons, labels, etc. They display data to users and they trigger events when interacting with them. Many controls are available out of the box. However, it is possible to bring new custom controls as well. They help WinForms to be extended with new shareable GUI elements.

Another concept introduced by WinForms is called *component*. Technically speaking, it is any class that either represents the business logic or it performs a background work. It does not have a GUI representation. A special type of a component is the mentioned control. It comes with visual representation and can be rendered on the screen. Microsoft provides several traditional controls with the framework itself.

Developers have two alternatives to adjust them. They either define a custom control with brand new visuals and possibly enhanced capabilities over the provided controls, or they create a composite control called *user control*. User control is built out of already made controls and the visuals cannot be redefined here. However, it is possible is to introduce more complex behavior. For example the add/remove control is commonly used by many applications even today. We depict it in Fig. 4.9. Finally, the *form* in WinForms is a representation of any window displayed in the application. Its class can be used to create standard, tool, borderless, and floating windows.

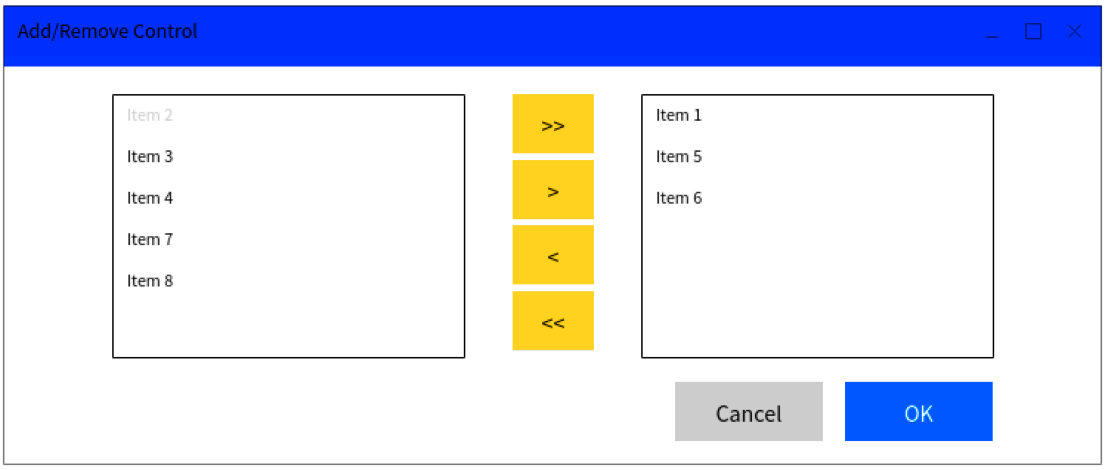

Figure 4.9: Example of Add/Remove User Control in WinForms

Certainly, since WinForms is quite matured technology, there are many implementation details and approaches to achieve specific goals. This is reflected in the current state of its documentation, different fragments of the original approaches are peaking between new layers and functionality that was added in later versions of .NET. It is impossible to uncover how the framework was originally released, but the origins have big ramifications for the current state. It is not our intention to go into detail. However, in Chapter 11, we will elaborate on Section 4.3.2 with respect to its evolvability in terms of NST.

4.3.3 Windows Presentation Foundation (WPF)

The WPF [145] is another GUI framework frequently referred to in this dissertation thesis. It was released in 2006 with .NET Framework version 3.0. It was formerly known as Avalon. Although it was introduced as a new take on GUI development, it is not a direct replacement of WinForms. It allows similar concepts and code practices. However, initially, it was a mean providing support for rich content such as animations, media, documents, etc. Nowadays, it is an extensive GUI framework for building desktop applications with rich contents, controls, dynamic layouts, data binding, and more. Through the years, WPF evolved to a very versatile framework supporting modern GUI approaches. Currently, the WPF project is hosted on GitHub pages [231] and turned to open source under the MIT license[4].

The architecture of WPF. It is based on the MVVM architectural pattern discussed in Section 4.2.3. We explained that it builds on the idea of a declarative approach to define GUI. It goes even as far as having a graphical designer doing this job. Microsoft supports this approach with standalone product called Blend, where one can fully engage with the design part of GUI. It uses XAML to describe the GUI. XAML is a markup language based on XML. Therefore, it describes a tree-like structure. For instance, the root element represents a window that encapsulates layout elements such as grid views, stack panels, etc., each of which hosts other elements like buttons, text-boxes, labels, etc. Although Microsoft provides an option to use XAML, it is not enforced by WPF. It does not depend on that. The GUI elements can be also defined as procedures. However, as it also resonates throughout the WPF community, XAML looks much cleaner than that.

Similarly to WinForms, there is a concept of controls such as drop-down menu, text-box, etc. However, their visual is rendered depending on its template, what is usually a XAML file. This template can be overridden to change the visuals of any given control. It also supports the concept of user controls that we know from WinForms.

The WPF ecosystem is large. It is out of scope of this dissertation thesis to describe it in detail. However, since our focus is on evolvability we will touch certain WPF concepts in Chapter 11 where we review it in terms of NST.

Section Takeaway

.NET GUI frameworks are often based on architecture patterns. Although some were invented more than two decades ago, they are still actively used and developed. Some manifest concepts that may support technology transitions, e.g., annotations, data binding, etc.

[4]The MIT License (X11 License) is a permissive free software license originating at the Massachusetts Institute of Technology (MIT) in the late 1980s. [1]

4.4 GUI Component Libraries

To gain an understanding of the current trends in sharing reusable components, and the fulfilment of McIlroy's dream put forward in Section 4.1.1, we reviewed trending component repositories. Few will be briefly introduced in the next sections.

4.4.1 NPM – NodeJS

Node Package Manager (NPM) [186] is a package manager for JavaScript. It helps to share and reuse controls created by different developers. NPM offers a browsing functionality. One can explore the portfolio of thousands of controls. By entering a filter condition, the relevant controls are found.

For example, the keyword 'filter' results in 2323 results. A more restrictive 'filter table' ends up in 79 options. From our perspective, the entered keyword is a sort of purpose in TAO theory put forward in Section 3.2. It indicates for what the component may be used, e.g., to filter data. This keyword helps us to find a construction that TAO theory talks about. Such a construction is capable of satisfying the purpose one searches for. Let us pick a random result, e.g., *tablefilter* component labelled with 'JavaScript library making HyperText Markup Language (HTML) tables filterable and a bit more' [186]. From this rough description, we can easily find that it is a library extending any HTML table with a 'filter by column' feature to easily manipulate with long tables. To use this component, one must understand the documented Application Programming Interface (API) and consider its technical dependency on other JavaScript packages. In this particular case, there is none.

However, we are mostly concerned with the purpose for which the component can be used. Besides the verbose documentation, it is represented by the following keywords: 'pagination', 'sort', 'datagrid', 'grid', 'filterable', 'javascript', 'table', 'filter'. The question is whether we can rely on them. Can we smoothly replace the component with another one exhibiting similar keywords? For example, another component called 'listfilter' declares nearly the same keywords. It does not have a technical dependency either. Is it possible to replace 'tablefilter' with 'listfilter' while keeping the functionality?

In NPM, these keywords are assigned manually. The documentation of NPM advise: 'Put keywords in it. It is an array of strings. This helps people discover your package as it is listed in NPM search' [186] . Therefore, even if strict rules for using the keywords would have existed, they might be violated. This answers the question whether we can rely on the keywords when replacing a component with another one – there is no guarantee that the components are functionally equivalent although they list the same set of keywords.

To summarise, NPM is an interesting platform to establish a component library along McIlroy's dream. However, the purpose of the components is vague – captured only by keywords. It helps to browse thousands of components, yet it does not guarantee their functionality.

4.4.2 Syncfusion

The next example of GUI component library is Syncfusion [160]. It is a company delivering a range of web, mobile, and desktop controls for enterprise technologies. There are more vendors with the similar business strategy, e.g., DevExpress, or Telerik. All of them sell a portfolio of reusable components. Our goal is not to benchmark them against each other. We decided to pick one to analyse the drawbacks and benefits of such a technology.

For an overview, Syncfusion was founded already in 2001, and they claim to gain already 23000 customers by 2021. Syncfusion has grown up into a company, we could easily call a 'GUI components factory'. They generate controls for a number of technologies. Beside components for Microsoft's edge technologies like ASP.NET Core, ASP.NET MVC (see Section 4.3.1), Xamarin, or UWP[5], they support a market with components for JavaScript, Angular, AngularJS, and PHP, or even ageing Silverlight and Windows Phone parts.

Control panel is a catalogue of controls. All in all, it seems that Syncfusion really follows the vision of McIlroy. They create a large portfolio of controls. The purchaser just picks a package that suits demanded technology, e.g., WPF, WinForms JavaScript, AngularJS, or PHP. The best fitting control can be browsed in the catalogue. One can easily filter the components by their name. Moreover, it is possible to play around with a live prototype showing the integration of a control in a broader context of use. Fig. 4.10 depicts the control panel. In this particular case, the developer filtered the *Line Chart* control. Its sample can be explored, or its code can be viewed.

Undoubtedly, Syncfusion (similarly to other vendors) provides a vast portfolio of controls. They seem to be aligned with the McIlroy's dream rather well. However, their controls suffer from the same problem as the aforementioned NPM. Although one can scrutinise the construction of each control deeply, its functional side is vague. The components do not exhibit much information about the purpose the component can be used for. Therefore, the elaborated changes of their controls and their replacement with functionally similar controls is still an issue.

Section Takeaway

GUI component libraries are quite often used by developers. However, it does not seem to be a common practice to focus on describing the purpose for which the components can be used. Therefore, although the components can typically be searched by a keyword and by the desired technology, this information is typically insufficient to understand the higher purpose for what the component can be used, or to possibly automatically replace one component by another matching the keyword.

[5]Universal Windows Platform

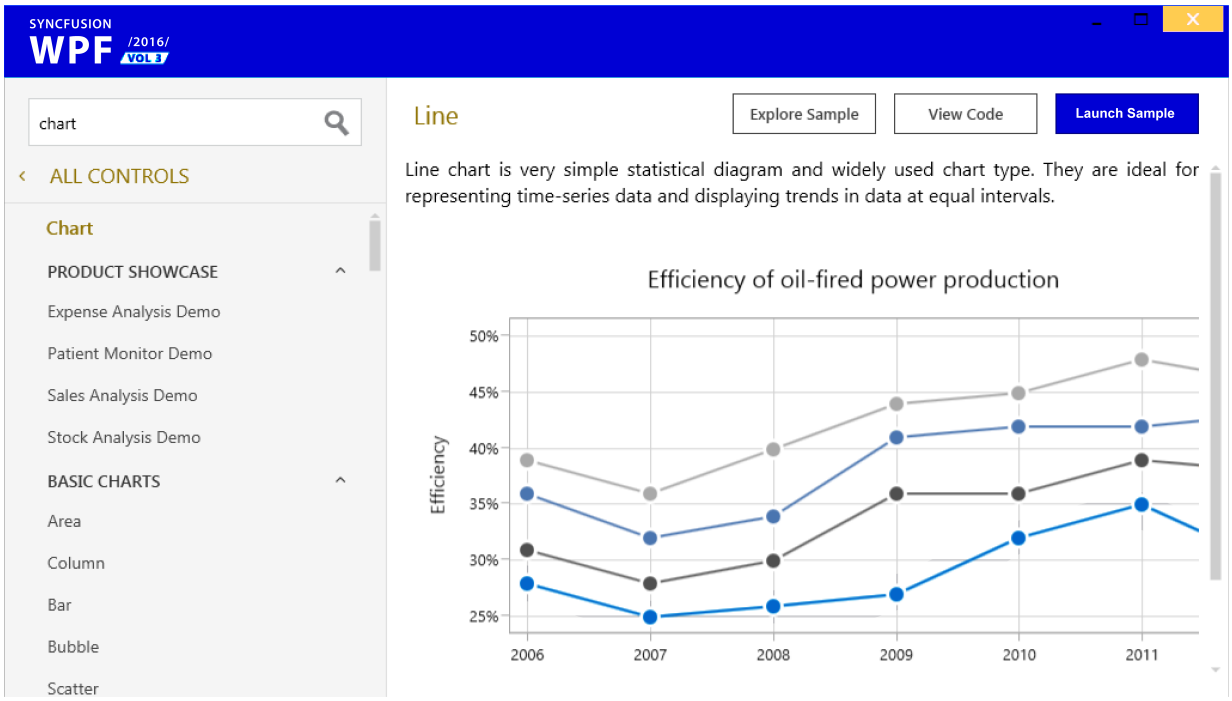

Figure 4.10: Control panel of Syncfusion catalogue

4.5 Robotic/Business Process Automation

In recent years, there has been a growing trend of digitalisation where digital technology is often used to change the business model to provide new value opportunities. In Chapter 1, we explained that many businesses are urged towards this trend to improve traditional working practices. According to KPMG consulting company, in the next 10 to 20 years, 47% of the jobs will be automated or replaced by robot labour [126].

RPA in a combination with well-known BPM come in handy under this situation. RPA is defined by Gartner [91] as follows:

Definition 9. RPA is a productivity tool that allows a user to configure one or more scripts known as 'robots' or 'bots', to activate specific keystrokes in an automated fashion. The result is that the bots can be used to mimic or emulate selected tasks within an overall business or IT process. These may include manipulating data, passing data to and from different applications, triggering responses, or executing transactions by combining user interface interaction and descriptor technologies. [91].

However, not just RPA helps with digitalisation, it seems to also support technology transitions. Let us add RPA to our knowledge base and thereby respond to RO 2.3.

4.5.1 RPA and Technology Innovation

RPA helps organisations to quickly accelerate their digital transformation initiatives and cut the cost of repetitive work. This allows them to reinvest the resources in more strategic and analytical activities. These activities can in turn help their business to grow.

Since RPAs often automate the routine work of employees working in GUI of legacy SW, our assumption is that they may also contribute to transform legacy SW products into those aimed at better technology transition. Thus, RPAs should not only be seen as a tool to automate tedious work and as a means to digital transformation initiatives. They may serve as entry points to discover business processes.

According to OMG [158], RPAs are often independent of any particular implementation environment. This may help the organisation to distinguish between its operations and its technology. When recalling BETA theory explained in Section 3.6, RPAs may help them to distinguish between F/C of organisations. It is argued that properly captured business processes may help them to increase their Capability Maturity Model Integration (CMMI)[6] level. We argue that this in turn may support their evolvability. Thereby, organisations may replace ageing technologies better if their business processes are known and correctly managed.

From this perspective, RPA may be seen as a bridge that organisations can cross to become manageable by BPM systems. At that point, the organisations gain all related benefits that help them to evolve their technology better, what is our research goal.

Together with Nacevska [A.14], we did a broad inspection of different RPA vendors. Next, with Vahalik [A.15], we used it to automate certain academic processes. Finally, with Woola [A.16], we are evaluating it in terms of its integration with Optical Character Recognition (OCR) and Neural Networks. An excerpt of our findings is presented next.

4.5.2 Challenges of RPA in Finance

The current market sees many opportunities to use RPA in Finance. We also reviewed them together with Nacevska [A.14]. In this section, we shortly present the background we revealed.

International companies such as Deloitte, PwC, Capgemini, KPMG, EY, and others, have already implemented RPA solutions and use it for their day to day tasks [126, 57, 223, 196]. These companies benefit from using RPA as it provides accurate, reliable, consistent output with high productivity rates [126, 57, 223, 196]. The implementation of RPA is typically compatible with systems that already exist in the business. Therefore, RPA finds application in various sectors such as healthcare, accounting, financial and customer services, and human resources by making work easier in a reliable manner.

Our research scope anchored in Section 1.5.1 mainly concentrates on financial services and digital treasury. Therefore, we analyse the usability of RPA in this area. Further,

[6]CMMI is a set of practices that help organisations to improve processes and increase productivity. It was developed by the Software Engineering Institute at Carnegie Mellon University as a process improvement tool for projects, divisions or organisations [35].

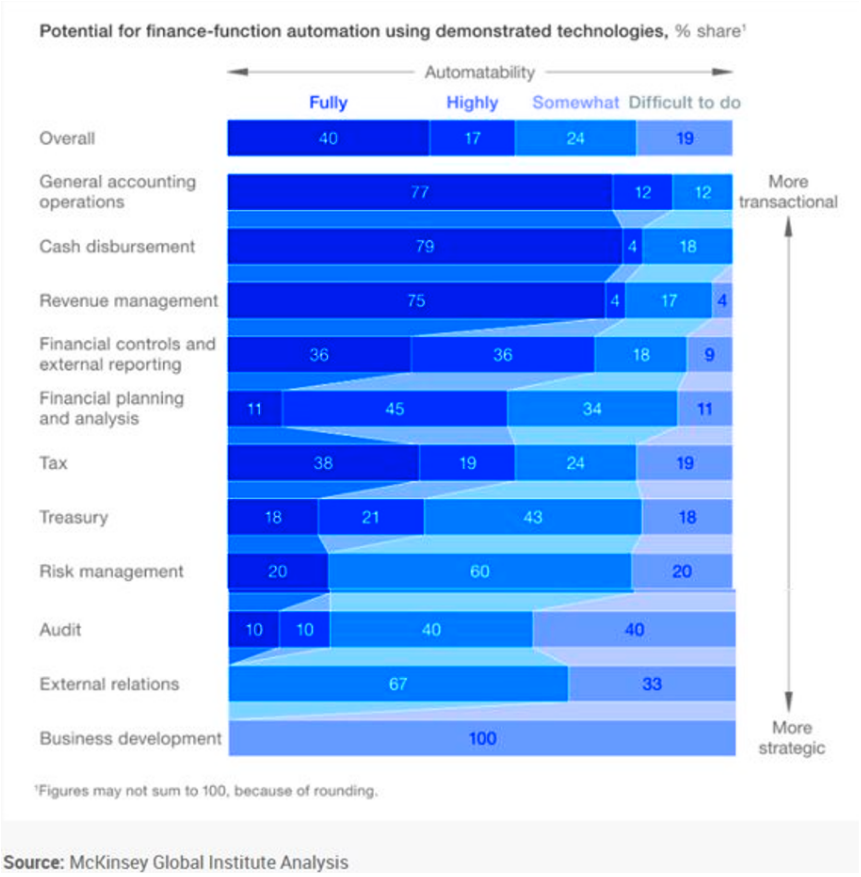

Figure 4.11: McKinsey Global Institute Analysis: Potential for automation in Finance [120]

we investigate the challenges companies are facing when implementing it. Since RPA has been trending in the last couple of years, many consulting companies conducted surveys and studies to present the development, application, implementation, and the benefits it brings. Most of the results are in favour of using RPA automating processes in financial services.

For example, Grand Thornton conducted a survey '2019 CFO Survey report' [223] among 378 senior finance executives from companies with revenues between 100 million and over 20 billion US dollars. 87% of executives agree that the technology will impact the way their finance function operates. According to their report, 40% of financial activities can be fully automated with the use of RPA. It is mentioned that 25% of the questioned Chief Financial Officers (CFOs) have already implemented RPA in 2019. This shows significant growth compared to 2018 when only 7% had it while 23% plan to invest in RPA during the next 12 months. The technical report from McKinsey outlines the extent to which certain financial functions may be automated. The results are depicted in Fig. 4.11.

According to another report, '2020 Hot Topics IT Internal Audit in Financial Services' [196] from Deloitte, the opportunities of automation in the area of finance are: auto-

mating of processes, automating controls testing and transforming metrics, and reporting. They point out that by using RPA tools the key benefits will be:

- *Enhancing the quality* – removing error-prone manually intensive activities.
- *Standardisation* – codifying activities to reduce inconsistent performance.
- *More timely and frequent insights* – rapidly integrate data from multiple systems to provide a more real-time view.
- *Increase time for value-added activities* – more time spent on high-value activities that increase the development and skills of the team.
- *Reducing cost* – reduce manual effort and reduce the cost of operating, testing, monitoring and reporting.

2019 Global Treasury Benchmark survey conducted by PwC [57] shows that more than 60% of the respondents see the potential of RPA in the next 3 years. This should increase productivity and allow insights that were once unattainable, also in cash flow forecasting and financial risk management. The most relevant areas in treasury where RPA finds application are:

- Payment execution
- Deal confirmation
- Accounting
- Monitoring of payments
- Deal settlements
- Management reporting
- Financial reporting
- Deal execution
- Exposure capture and exposure analysis

The same survey also explores the challenges that companies generally face in the process of implementing RPA. One of the biggest obstacles is considered having no clear business use case and no long-term strategy. Missing standardisation of the processes as well digitalisation of the same are another challenge that might need to be tackled. Once these challenges are faced and RPA is implemented successfully, companies can use it daily.

Even though many processes can be automated, it is crucial to keep in mind that there are finance tasks which are not very suitable for RPA. For example, performing due diligence on acquisition targets, finding the lowest cost of funds or allocating an investment portfolio, anything that requires significant decision-making, or tasks that do not need to be executed frequently (e.g., annually) or with uncertain frequency. Processes with too bigger complexity are also not recommended to be automated.

Additionally, together with Woola [A.16], RPA may find many use-cases in a combination with tools for OCR. This term refers to software which can electronically extract text from visual stimuli such as images and documents. It is often linked to the 'robot's eyes' [130]. It may be used for many types of documents including common office files,

contracts, pictures, invoices, bills, and reports. When combined with RPA, OCR brings numerous capabilities. It is especially popular in financial institutions where stacks of unstructured data in paper format are common.

4.5.3 Robots in RPA

There are currently more than 80 RPA vendors globally with most of Fortune 500 companies using RPA software [151]. Their RPA tools can be used to configure and automate manual and repetitive tasks. Before we compare the market leaders, let us rephrase RPA definition set in Definition 9. In other words, RPA is the technology that allows automation of business processes by creating software robots ('bots') to reduce human intervention within digital systems. The term *Robots* [208] refers to entities that mimic human actions in order to perform variety of repetitive tasks, while *Process* [112] is a set of activities that transforms the input into output.

The RPA robots are agents running on client computers and executing assigned workflows. They can be categorised as either *front and back office* robots [197], also correspondingly known as *attended* and *unattended* robots. Below, we list some of the main differences between them.

Attended robots (Front office robots). They usually perform only repetitive, easy to define steps in a larger process (e.g., copying data from one system to another or to put some data into an Excel table). The employee can then use the result of the robot and decide what to do next in the process. Their essential characteristics are:

- User intervention is needed
- Standalone
- Triggered by the user, runs only under human supervision
- Mainly used in service desk, call centres, key accounts/control towers
- Drives customer satisfaction

Unattended robots (Back office robots). Unattended robots, on the other hand, are fully independent and do not require any human interaction at all. They are started automatically, either by some event or at a predefined time. Unattended robots are used to fully replace human employees. They are available 24/7, not only if someone tells them to run. Their essential characteristics are:

- They do not require human intervention.
- Initiation of robots is on server-side.

- Triggered by a schedule or monitoring.
- Mainly used in Order Management, Finance, IT, HR, Shared service centres.
- Drives cost down and reduces errors.

Depending on the needs of automation, the user can choose a vendor that supports both of them or only front office robots.

RPA Processes As mentions before, not every process can be automated. Therefore, the question is 'What makes a process to be a good candidate for automation?'. To see whether a process can or cannot be automated, some rules need to be followed. One option is to use a methodical framework from Fingent [117]. This framework helps to distinguish the process based on *process fitness* and *automation complexity*. They use the process fitness to categorise tasks into different categories.

- *Rule–based* – decisions made (including data interpretation) in the process can be captured in a predefined logic. The exception rate is either low or can be included as well in the business logic.
- *Automatable and repetitive processes* - there are processes that have been already automated using other technologies than RPA.
- *Standard input* – input in the process should either be electronic and easily readable or readable using a technology that can be associated with RPA.
- *Stable* – processes that have been the same for a certain period of time and no changes are expected within the next months.

As for the automaton complexity, they name several factors upon which the process automation depends [117], such as:

- *Number of screens* – the higher the number of screens, the more elements have to be captured and configured prior to the process automation.
- *Type of applications* – some applications are more easily automated (such as the Office suite or browsers), others heavily increase the automation effort (Mainframe, for example).
- *Business logic scenarios* – an automation's complexity increases with the number of decision points in the business logic.
- *Type and number of inputs* – standard input is desirable. Non-standard inputs can be of different complexity grades, with free text being the most complex.

By using the above explained factors, Fingent [117] sorts the process into four categories:

- *No RPA* – process that cannot be automated with the help of RPA tools.
- *Semi-Automation* – not completely automated.
- *High-Cost RPA* – use complex tech or require programming skills.
- *Zero-Touch Automation* – processes that are digital and involve a highly static system and process environment, so that they can be easily broken into instructions and simple triggers can be defined.

4.5.4 RPA Vendors

IT Central Station [200] analysed RPA vendors based on key factors such as user reviews, pros and cons, etc. According to them, the top eight vendors are the following.

- UiPath
- Automation Anywhere
- Blue Prism
- Blue Prism Cloud,
- Kryon RPA
- Microsoft Power Automate,
- WorkFusion,
- VisualCron

Typically, the choice of the vendor depends on the type of administrative processes to be automated. Some of the major criteria are summarised below:

- *Technology* – in which technology is the RPA tool built. Has to be platform-independent and should support any application and platform.
- *Interface* – how easy is the tool to use. Complex interface will cause a delay in the process of implementation. (Prefer drag and drop, auto-capture, image recognition, etc.).
- *Management* – how effectively and easy the robots can be managed. Should provide a high level of visibility and control in terms of process monitoring, change, development, and reuse.
- *Security* – how safe are robots compared to humans.

According to Forrester Research 'The Forrester Wave: Robotic Process Automation, Q4 2019' [33], the leaders on the market are UiPath, Blue Prism and Automation Anywhere. Another report issued by PwC 'Digital masters' in November 2019 [174], reports that companies, based on the survey results, favour the same RPA vendors.

Although with Nacevska [A.14], we deeply reviewed a number of vendors, in the scope of this dissertation thesis, we only take a closer look at UiPath. First, it seems to be a market leader, and second, it is written in .NET where most of our case studies reside. Nevertheless, our results show that the other vendors do not differ much in terms of their functionality.

UiPath. UiPath is one of the fastest growing enterprise software companies in history. By 2019, it was named a leader in the Gartner Magic Quadrant for Robotic Process Automation Software [176]. By 2005, the company was founded in Romania under the name 'DeskOver'. Later, in 2013, they launched the desktop automation product and finally, in 2015 they introduced the enterprise platform along with the new name UiPath [209].

The main goal of the company, according to their Chief Executive Officer (CEO), is having a robot for every person [209].

> *'Bill Gates used to talk in Microsoft about a computer in every home. I want a robot for every person.'*
>
> *– Daniel Dines, CEO of UiPath*

UiPath is built on the Microsoft .NET platform. Thus, it offers a native integration with other Microsoft's products such as Microsoft Office, Office365, and Dynamics 365, as well as PowerBI. The tool is available for customers to deploy in Microsoft Azure and thereby has out-of–of-box integration with many Azure Services, including Azure Cognitive Services, enabling more intelligent RPA solutions to be created. UiPath product is only available for Windows OS [210]. Technologies that are covered within UiPath:

- Desktop automation
- Web automation
- GUI automation
- Screen scraping
- Citrix automation
- Mainframe automation
- SAP automation
- SAP S/4HANA migration
- Excel automation
- Macro recorder

UiPath Platform offers all components needed to design and develop automation projects, execute the instructions automatically and manage the robot workforce. The main components are UiPath Studio, Orchestrator, and UiPath Robot.

1. *UiPath Studio* helps to design and model automation workflows from prebuild activities. It is available for developers, Studio, or business users, StudioX Preview. UiPath Studio offers many predefined operations that allow interaction with a variety of desktop applications, web browsers, and OCR engines.

2. *UiPath Orchestrator* is a web application that provides an overview of the processes and allows the user to manage the robots. Its main uses are control, management, and monitoring.

3. *UiPath Robot* is used for executing workflows and instructions sent locally or via Orchestrator. There are two types of Robots:

- *Attended* – is triggered by events and needs human interaction. Always operates on the same workstation.
- *Unattended* – run unattended in virtual environments and can automate any number of processes.

UiPath platform offers many possibilities of automation. However, according to the study 'Ovum Decision Matrix: Selecting RPA' [190], the main reason why the customers choose this vendor is due to 'ease of use for less-skilled business users, computer vision, simplified automation development based on Microsoft Workflow foundation, orchestrator multitenancy, and support for high-density robotics'.

4.5.4.1 Embedding in Practice

RPA allows organisations to automate tasks across applications and systems. Its main focus is on replacing repetitive tasks performed by humans with a virtual workforce. The companies employing RPA mainly benefit from a cost reduction, saving precious time and resources, increasing scalability, providing real-time visibility and discovering of bugs. The majority of the RPA tools available on the market are drag-and-drop. Almost no programming skills are needed to configure a software robot. Thus, many nontechnical staff can set up a 'bot' or even record their steps to automate the process.

RPA also comes with disadvantages such as the limitation of robot to the speed of the application. Small changes made in the automation application will require robots to be reconfigured. RPA generally targets large companies, though small to medium-sized organisations can deploy RPA to automate their business with very high initial costing.

The selection of RPA vendor should be based on the decision whether the tool allows easy reading and writing business data into multiple systems, which type of tasks are being mainly performed (rule-based or knowledge-based processes), whether the tools work across multiple applications, does it provide built-in Artificial Intelligence (AI) support to mimic human users, etc.

4.5.5 Business Process Management

Unlike RPA, BPM is end-to-end decision-making workflow. It is defined as follows:

Definition 10. BPM is a discipline involving any combination of modeling, automation, execution, control, measurement, and optimisation of business activity flows, in support of enterprise goals, spanning systems, employees, customers and partners within and beyond the enterprise boundaries [168].

To better understand the definition, Palmer [168] clarifies that:

- *Modeling* a process means defining and representing a process so that it is supported in every step of the communication.

- *Automation* in most of the cases means providing a software solution that executes the process.
- *Execution* means that instances of a process are performed, by following the BPM model.
- *Control*, either strict or loose, means that there is some aspect of making sure that the process follows the designed course.
- *Measurement* is understood the effort taken to determine how well the process is working.
- *Optimisation* means that the discipline of BPM is an ongoing activity that builds over time to steadily improve the measures of the process.

Therefore, the aforementioned RPA is more system-to-system interface dealing with discrete repetitive tasks typically occurring at the beginning of the process. Unlike RPA, BPM should be done in the context of a whole enterprise and not only a part of it. Thus, it provides business-level automation.

One of the most used conventions for BPM is BPMN. It is a standard that provides a graphical representation of the business process. The BPMN models are typically managed by BPMS what is a software used to automate, analyse, organise and improve existing business processes [166]. BPMS lets the user map existing processes, ensure communication efficiency, and look for further improvements over time. These systems are not supposed to be a one-time solution, rather they are meant to be used for continuous use.

4.5.6 BPMS Vendors

Nowadays, we can find a number of BPMS. Their choice depends on the customers' needs and on the size of a company. With Nacevska [A.14], we reviewed the BPMS below. They seems to be the most relevant in our research scope. However, in the scope of this dissertation thesis we only provide an excerpt of Camunda that is further referred to in Chapter 9.

- Camunda BPM
- IBM BPM
- Bizagi BPM

Camunda. Camunda is an open-source BPM platform that provides tools for creating and modelling workflows, deploying and executing process models, and completing of a workflow task assigned to specific user [149]. Initially, it was formed as a consulting company and gradually switched to software development [149]. The platform consists of tools such as:

- *Modeler* – a desktop application used for designing BPMN diagrams as well as Decision Model and Notation (DMN) decision tables [28].

- *Cockpit* – tool for monitoring of processes, analysing and solving technical problems.
- *Tasklist* – application allowing end users to work on assigned tasks.
- *Workflow engine* - tool for deployment of workflows and their execution. Can be used as an orchestration service, event handler or for human task management. By using BPMN parser, the engine is able to translate BPMN 2.0 XML files to Java Objects [28].
- *Decision engine* – used for executing DMN decision tables. Can be used as a standalone application via REpresentational State Transfer (REST)[7] or Java application or as pre-integrated with the Workflow engine [28].

The implementation of BPMS is a complex process. It requires and integration across the whole organisation. This type of implementation might bring significant challenges. However, by 2018, RedHat [100] conducted a survey showing that that more than 50% of companies are implementing BPM, because it provides cost reduction. Participants in business processes experience improvements in productivity, understanding, and communication [80].

Section Takeaway

RPA and BPM are trending approaches supporting the automation. While RPA typically concentrates on automating discrete repetitive system-to-system tasks, BPM targets the business-level automation of the whole enterprise. The majority of BPMSs offers some kind of integration with RPAs. Therefore, a variety of use-cases can be covered by combining them.

4.6 Chapter Summary

This chapter was dedicated to the state-of-the-art of the technological developments in our research scope. First, in Section 4.1.1, we introduced a brief history of CBSs as means to achieve reusability in GUI. Second, in Section 4.2, we deeply inspected the architecture and design patterns that many GUI technologies are built on. Some particular GUI frameworks were presented in Section 4.3. In Section 4.4, we also briefly revisited component libraries that are traditionally used when building GUI. Finally, in Section 4.5, we analysed RPA and BPM as means that may help to transition from the legacy SW solution to those aimed at better technology transitions.

[7] REST defines an architectural style for distributed hypermedia systems [78].

Chapter 5

Previous Results and Related Work

In this chapter, we describe the work that we find related or complementary to our research. First, in Section 5.1, we introduce Normalised Systems (NS) expanders. Second, we mention low-code platforms in Section 5.2. Last, in Section 5.3, we relate our research to feature-rich descriptions based on event calculus.

5.1 Normalised Systems (NSX)

In Section 2.2, we introduced NST. Although the theory was formulated, its principles can be violated. It may be explained by the design freedom offered to developers. NST emphasises that 'any developer violating any theorem at any time during development or maintenance will generate CEs' [137]. They attach that 'applying all theorems leads to a software architecture being a very fine-grained modular structure of separated concerns which are version transparent towards one another and executed statefully' [137].

Such a CE-free system is difficult to create manually. Systems complied with NST are typically generated. This ensures the alignment with all their grounding principles.

NSX, a spin-off company of the University of Antwerp, developed so-called *NS expanders*. These expanders can generate a CE-free system from *NS descriptors*, which are conceptual models of the structure and behaviour of a system. In typical applications, only a small part of the code must be designed manually in the form of customisation [136]. The expanders has already been applied in practice on a large scale. In [165], Op't Land reminds that by 2011, NS approach and tools have already been used in 12 real-life projects.

Compared to our research, NS expanders guarantee high evolvability from the perspective of change impact limits (magnitude of only a few classes). However, in our research, we want to answer RQ 2. We want to develop a methodical framework that aids in the construction of SW suited for technology transitions. Moreover, we want to ground it in EE findings and possibly use NST to evaluate it. Thereby, our focus is on involving higher-level concepts of EE theories such as the notion of affordances described in Section 3.3.

From that perspective, NS descriptors are technical in nature. They operate on a technical level of typical relational and finite-state machine modelling. In our case, we want

to work with a higher level system description facilitating new ways of thinking about and automating technology transitions, such as distributed repositories of components and (semi-)automated upgrades with guaranteed parameters or the (semi-)automated generation of different types of applications (desktop, web, and mobile).

Nevertheless, we argue that our methodical framework and NSX might not necessary result in disjoint architectures. We can develop NS expanders for a system aligned with our methodical framework, thereby combining the powerful aspects of each method.

5.2 Low-code Platforms

According to Waszowski [224], 'the low-code platform enables quick generation and delivery of business applications with minimum effort to write in a coding language'. Such a development approach is typically based on drag-and-drop modules, and other user-friendly manipulations meant for building the application and configuring them. These platforms are typically focused on design and development of databases, business processes, or GUI. Although the coding is reduced, certain uncommon, or personalised situations may require it. According to Gartner [218], Mendix low-code is one of the market leaders popular among both professional developers as well as business staff. Nevertheless, other solutions are also trending on the today's market – K2, ServiceNow, Appian.

However, similarly to NS expanders, most low-code platforms are typically focused on a specific type of applications, in a specific technology stack. Our approach and low-code approach are not disjoint. We target technology-independent approach in general, the existing low-code solutions may benefit from it. For instance, low-code platforms may choose to adapt it to enable building the final application and help better transition from one technology to another.

5.3 Feature-Rich Descriptions based on Event Calculus

Similar to our research, Tun et al. [207] addressed the need for 'creating mappings between requirements and features, and between problem and solution structures to support the evolution of a feature-rich software system'. They derived an engineering approach for a clear separation between requirements for new features, requirements for incremental features, requirements for their composition, and their correspondences with specifications [207]. They demonstrated how to describe the requirements in a modular manner, how to map them to features specified using natural language, and how to apply a form of temporal logic called event calculus. However, they primarily focused on enabling 'developers to capture and reuse knowledge for solving similar problems' [207]. Our goals are similar, but our methodical framework is supposed to offer a more detailed conceptualisation of users and purposes. It should involve mapping functional requirements to technical constructions.

Part III

Our Approach

Chapter 6

Research Methodology

Klein and Myers [124] remind that it is essential to understand what guided us in making decisions through our research journey. Specifically, what guided us from an initial research problem and goals to a suitable solution answering them. According to them, this contributes to the perceived validity of the research process. Therefore, while in the previous chapters, we discussed the foundation of our research, its background, state-of-the-art, possible theories, methodologies, and technologies to address the research problem, now we provide details of how we employed DSRM and how we designed the research to achieve its objectives put forward in Section 1.5. We respectfully follow the approach and argumentation introduced by Amirebrahimi [4] in his dissertation thesis.

We structure this chapter into five parts. In the first section Section 6.1, we present a conceptual design framework meant for achieving our research goal put forward in Section 1.5. Throughout the research, a number of research outcomes were tested in a commercial system Corima. Since we often refer to Corima in the entire thesis, we provide its brief overview in Section 6.2. Hence DSRM was selected as our methodical tool, we introduce and justify this choice in Section 6.3. Next, in Section 6.4 we elaborate the details of how we adapted DSRM in our research. Finally, the overall research design and its process is presented in Section 6.5.

6.1 Research Design

In Part II, the theory and practice behind evolvability, EE, and CBS development were reviewed. In this thorough review, a number of gaps and opportunities were revealed. However, many concepts and technical developments may seek solutions to our research problem put forward in Section 1.4 – the lack of guidance and architectural pattern on adapting software artefacts into the latest technologies in a more efficient and manageable manner. Since our goal is to build a methodical framework solving it, a combination of EE-thories applied to SE and the evaluation in terms of evolvability may facilitate this methodical framework. While our first research question concerned the state-of-the-art, the other RQ 2 remain unanswered.

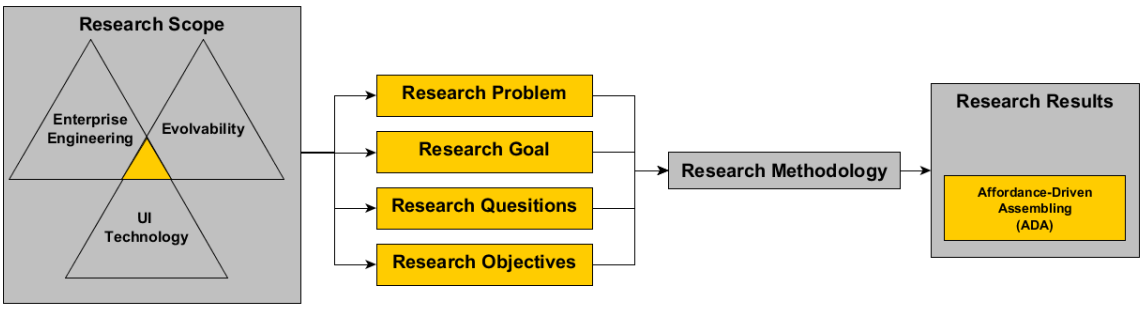

Figure 6.1: Conceptual design framework of our research

To address them, we designed a conceptual design framework to bring together the necessary concepts, theories, and methodologies to develop a new methodical framework for better technology transitions. It is a guideline to this research to identify the basic concepts and outline of the design flow or design solution. This conceptual design framework is depicted in Fig. 6.1. The research problem, scope, questions, and objectives were already discussed in Chapter 1. However, the research methodology to achieve the research goal (the methodical framework) is yet to be explained now.

6.2 Conjunction with COPS on the Research

Throughout our research, a number of research outputs were done in conjunction with company COPS [40]. We used its knowledge and products to design & develop artefacts connected to our research goal formulated in Section 1.5.

COPS is a group of SMEs[1] in Czech Republic, Austria, and Germany. They are on the market since 1979. In the last few decades, their main focus is to the financial sector, in particular to the area of corporate and banking treasury management. By 2021, they mainly target DACH[2] market.

6.2.1 Treasury Management System – Corima

The company implemented an application framework called Corima. It is a software development platform capable of hosting applications in various business domains[3]. In 2021, the most significant application suite is focused on supporting the banking and corporate industry. It is a Treasury Management System (TMS) and is contained in the *Corima.cfs* suite. However, in the last years, COPS extends this suite of BPM subsystem. Thereby supports treasury teams with a step by step automation of business processes related to finance.

[1]SME is an acronym used to describe small and medium-sized enterprises.
[2]DACH is an acronym used to describe Germany (D), Austria (A), and Switzerland (CH).
[3]Business domains refer to different areas of business, such as finance or healthcare

6.3 Design Science Research

Nowadays, we may recognise two different research styles – *Explanatory Science Research* and *Design Science Research*.

The explanatory research is mostly common for social and natural sciences such as biology, chemistry, physics, and astronomy. Here, the emphasis is to understand the reality [138]. According to Van Aken [213], the goal of explanatory science is to develop knowledge to describe, explain, and predict. Gerts [92] puts it in other words: 'Natural science research papers typically adhere to a structure that consists of the following steps: problem definition, literature review, hypothesis development, data collection, analysis, results, and discussion'. According to Peffers et al. [170], the resulting research outputs are mostly explanatory. Often, these outputs are arguably not applicable to other disciplines such as engineering or information systems. Peffers [171] explains it by the artificial nature of the so-called human-constructed domains. These are focused on creating and improving artificial solutions and products, rather than on answering 'how things work, and why they work the way they do?'.

Design Science Research (DSR) emerged in a response to the limitations above. March et al. [138] clarify that the focus of natural science is on understanding reality. The DSR, on the other hand, attempts to create things that serve human purposes. Walls et al. [222] add that it became an important methodology for discipline-oriented design or an improvement of solutions for human-related problems. Therefore, contrary to explanatory research, DSR puts an emphasis on 'how things ought to be in order to attain goals, and to function?' [193]. The key principle of DSR is to achieve understanding by building so-called 'artefacts' that satisfy a set of functional requirements. Peffers et al. [171] define the artefact as 'any designed object with an embedded solution to an understood research problem'. According to Hevner [103], the artefact can be a social innovation, information resource, construct, method, model, or an instantiation. However, the DSR artefact in IS is much broader. It may include methods, models, or frameworks on different levels of abstraction.

In our research, the artefact is a methodical framework that can aid in the construction of more evolvable software solutions. Such a framework may additionally contribute to BA of an organisation. Hence DSR is a good fit for its design, it is further adapted by this research. In particular, we employ Design Science Research Process (DSRP) from Peffers et al. [170] that builds on DSRM introduced by Hevner. This employment is discussed in Section 6.4.

6.4 Employing Design Science Research Methodology

In this research, the aim is to design a method for constructing software systems that are better suited to technological transitions. It contrasts with the aforementioned explanatory research attempting to understand the truth. Here, we design an artefact and investigate its feasibility for a defined problem. Therefore, it '...should not only try to understand how

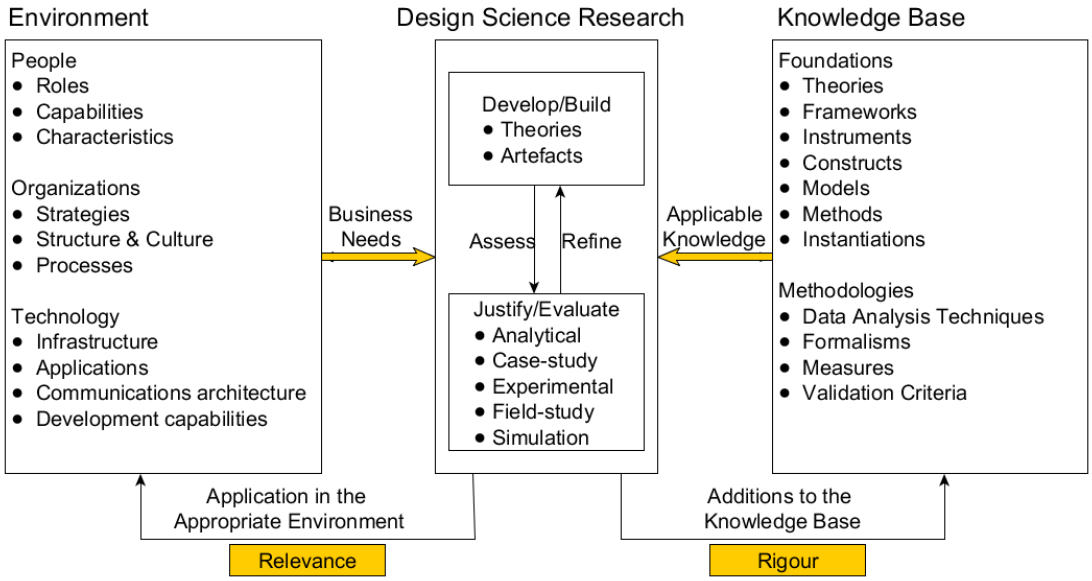

Figure 6.2: Selected Design Science model (adapted from Hevner [103])

the world is, but also how to change it' [29]. This makes our research suitable to employ DSR.

The selected DSR model in here is the DSRM originally presented by Hevner et al. [103]. Recently, Cater-Steel et al. [30] conducted a research across doctoral projects applying DSR. They positioned DSRM among the most matured and well-accepted methodologies in the research community. That is the reason we selected this particular model for our research.

Hevner et al. [103] explain it as 'a conceptual framework for understanding, executing, and evaluating DSR combining behavioral-science and design-science paradigms'. In DSRM, the artefacts are produced by enhancing an existing body of knowledge (knowledge base) to address a problem (business needs) emerged in an organisation (Environment). We present it in Fig. 6.2 with slight visual modifications. However, after DSRM was published in 2004, it got an attention of the research community. Iivari [110] contributed to a clearer understanding of the key properties of the design science research paradigm–ontology, epistemology, methods, and ethics. With a reference to Iivari, Hevner [102] enhanced the picture of what it means to do high quality design science research in IS. He borrowed the original model, and embodied three closely related concepts: relevance, rigour, and design shown in Fig. 6.3.

Hevner [102] explained these concepts as follows: 'The relevance cycle inputs requirements from a contextual environment into research and introduces research artefacts into environmental field testing. The rigour cycle provides grounding theories and methods along with domain experience and expertise from the foundations knowledge base into the research and adds the new knowledge generated by the research to the growing knowledge base. The central design cycle supports a tighter loop of research activity for the con-

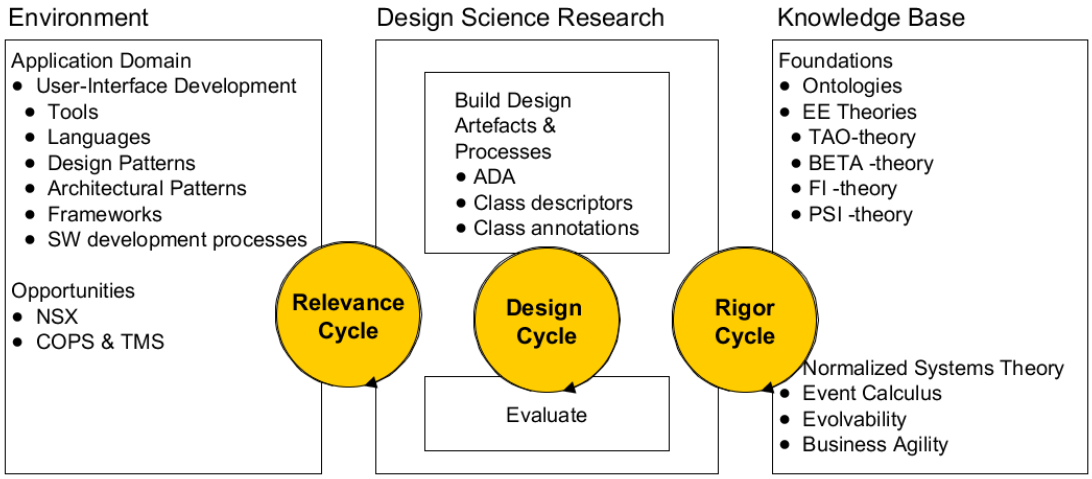

Figure 6.3: DSR cycles, environment, and the final knowledge base for our research

struction and evaluation of design artefacts and processes.' In other words, Hevner [102] reinforced the need to maintain a balance between academic rigour and industry relevance while representing the artefact as a major outcome of any design science project.

The three-cycle activities of DSRM were iteratively applied to address our research goal for over seven years, and we are now in a position to present our design cycles in terms of artefact development and evaluation backed by the rigour cycle (grounding of the scientific methods and related work) and the relevance cycle (alignment with the industry and best practices). In our context, the relevance cycle incorporates requirements from relevant software development practices and related industries (such as banking and corporate finance management) into the research and introduces the research artefacts into real-world applications. Among these artefacts, we can include prototype applications of EE theories to SW, flexibility-usability analysis, methodical framework, and others described in Part IV. The rigour cycle develops the theoretical concepts of our research artefacts along with resources and expertise from the body of knowledge (EE theories, ontology, theories of evolvability, concepts of BA, etc.) for the research. The design cycle supports the loop of research activities that provide the development, evaluation, and improvement of research artefacts.

The three research cycles that demonstrate the evolution of Affordance Driven Assembling (ADA) are discussed and illustrated in Fig. 6.3.

6.4.1 Environment and Relevance Cycle

Hevner [102] explains that design science research is motivated by the desire to improve the environment by introducing new and innovative artefacts and processes required for building these artefacts [193]. He stated that 'an application domain consists of the people, organisational systems, and technical systems that interact to work toward a goal'.

In our research, we focus on the application domain of GUI development. Therefore, in terms of relevance cycles, we focus on an environment that enables us to better understand the tools, languages, and frameworks used for GUI development, where we can learn the processes of the software development cycle, including requirements specification.

In addition, we have strengthened our connections with companies that may help us assess the challenges related to evolvable software development in practice, which can be beneficial for the relevance cycle of this research.

- *NSX Normalised Systems*[155] – a spin-off company of the University of Antwerp focused on applying NST on an industrial scale.
- COPS [40] – a company focused on providing TMSs for banking and corporate industries, where we can apply our research on an enterprise scale.
- *Design and Architectural Patterns* – to understand to understand the historical and latest development of patterns that are commonly used by GUI technology.
- *Software Engineering Practices* – to understand the state-of-the-art based on diverse areas of software engineering, e.g., the development of CBS, and to adapt relevant principles and formalism in the artefacts of a methodical framework.

6.4.2 Knowledge Base and Rigour Cycle

Hevner [102] reported that 'design science draws from a vast knowledge base of scientific theories and engineering methods that provide the foundations for rigorous design science research'. Our knowledge base has been extended by the following resources:

- *Ontologies* – to describe our concepts in a platform-agnostic manner and study objects in GUI development 'as they are, how they are composed of parts, of which substances the parts are, or how the parts are connected, etc., completely disregarding the purposes(s) subjects could use them for' [63].
- EE theories – to elaborate artefacts from the function-construction perspective based on theories rooted in teleology and ontology.
- *Evolvability* theories – to understand the state-of-the-art of the domain of systems evolvability.
- *Business Agility* – to understand the state-of-the-art and trending principles of business agility.
- *Normalised Systems* Theory – to evaluate our proposals in terms of mathematically proven theories regarding constructing evolvable systems from fine-grained, loosely coupled modules.

6.4.3 Design Science Research and Design Cycle

As discussed by Hevner [102], in the design cycle, 'the requirements are input from the relevance cycle, and the design and evaluation theories and methods are drawn from the rigour cycle'. Therefore, to propose a design approach that can help us construct more evolvable software solutions, we have been 'generating design alternatives and evaluating these alternatives against requirements until a satisfactory design is achieved' [102].

The initial requirements emerged from the aforementioned real-life application framework Corima. It was a rapidly growing ecosystem, which introduced new functionality into banking and corporate finance industries. However, the GUI layer was shown to be difficult to maintain. It was not easy to reuse certain elements without having an unforeseeable impact on the entire system. It was difficult to keep up with modern GUI technologies and to smoothly move GUI from one technology to another, e.g., from desktop to the web. As Corima grew, the core development team scaled up to multiple business-oriented teams across Austria, Germany, and the Czech Republic. Although the team used Scrum of Scrums and common development best practices to respond to changing requirements from businesses, it became difficult to preserve the GUI consistency across all applications they had developed. However, the observations showed that the requirements were similar in nature. They differed slightly depending on the user and his/her intention, regardless of the technology to be used in the future.

By assessing these requirements, the loops between our theoretical proposals and practical applications in industry led to the ADA approach described in Chapter 12. Throughout these improvement loops, we validated our research artefacts with mechanisms including expert reviews, experiments with early adopters, and real-life applications in Corima. In the research cycle, we analysed the theoretical concepts of evolvability, affordances, functions, constructions, and their relationships, and we have been reifying them in terms of concepts and approaches in software engineering (SE). At the same time, throughout these cycles, we improved so-called class descriptors and class annotations in Corima to evaluate ADA in practice. The implementation was continuously adjusted to reflect the challenges related to the new GUI technologies introduced in Corima. We also used insights into the formalisation of the whole-part relationship in UML from Barbier et al. [14], the notion of scope of interest in the DEMO methodology proposed by Dietz and Mulder [65], the orthographic software modelling concept proposed by Atkinson et al. [7], and Semantics of Business Vocabulary and Business Rules (SBVR) [159]. Finally, to evaluate ADA with respect to other related efforts in the SE industry, we positioned it in a related work that focused on reusability and evolvability.

6.5 Applying Design Science Research Process

Having the adoption of DSRM for this research explained and justified in the aforementioned section, now we present a corresponding process that guided us throughout the entire research.

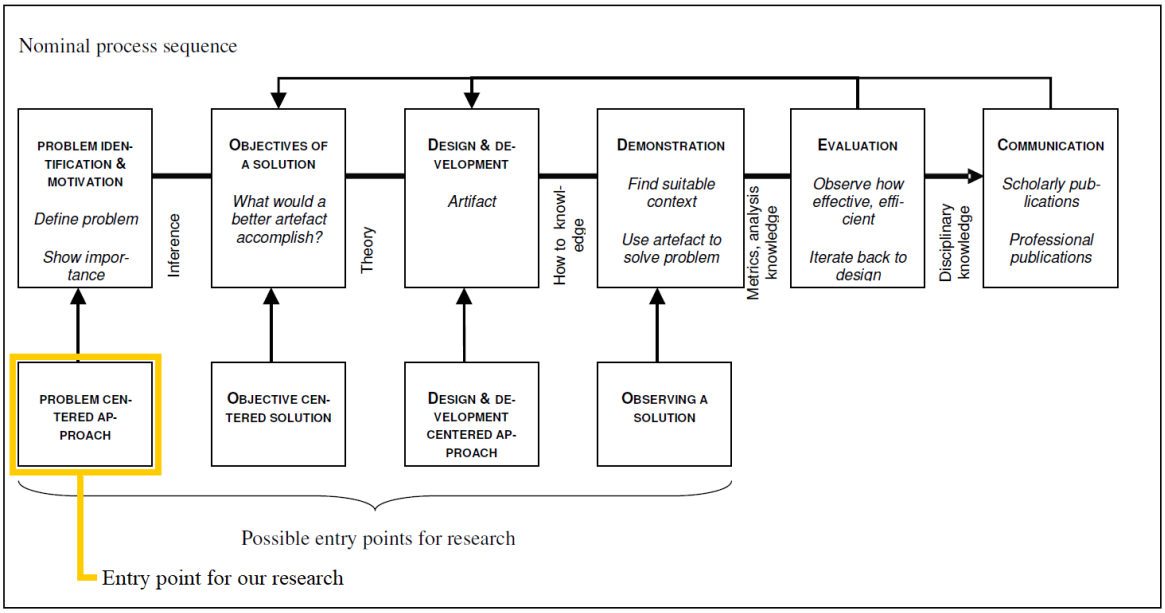

Figure 6.4: DSRP model [170] and the entry point for our research.

Although DSRM is broadly known among researchers, it does not provide generally accepted process to carry out the research. Peffers et al. [170] fill this gap by constructing DSRP model that is consistent with the prior literature about DSR, provides a nominal process model for doing DSR, and it grounds a mental model for processing and appreciating DSR in ISs. Fig. 6.4 illustrates the process.

It suggests that there might be multiple possible entry points for DSR. It distinguishes between the four entry points described by Peffers et al. [170]:

- *Problem centred approach* – if the research idea resulted from observing the problem or from suggested future research in a paper from a prior project.
- *Objective centred solution* – if the research idea resulted from by-product or consulting experiences whose results did not meet clients expectations.
- *Design & Development centred approach* – if the research idea resulted from the existence of an artefact that has not yet been formally thought through as a solution for the explicit problem domain.
- *Observing a solution* – if the researchers work backwards to apply rigour to the process retroactively.

Our research problem stated in Section 1.4 was observed by researchers and business in the IT domain. Therefore, the 'Problem centred approach' is the best fit for us.

Peffers et al. [170] explains that DSRP is structured in a nominally sequential order. However, the researchers are not expected to always proceed in a sequential order from

activity 1 through activity 6. It consists of six steps: i.e., *Problem Identification & Motivation, Objectives of a Solution, Design & Development, Demonstration, Evaluation, and Communication*. In our research, we group these steps into four phases: *Conceptualisation*, *Design & Development*, *Demonstration & Evaluation*, and *Communication*. We discuss them thoroughly below. In Fig. 1.2, we already outlined a roadmap of this dissertation thesis. Now, we can be more concrete in terms of DSRP. Therefore, in Fig. 6.5, we link the aforementioned research phases to the objectives and chapters in this thesis. We also briefly describe the deliverable of each of them.

6.5.1 Conceptualisation Phase

Step 1: Problem Identification & Motivation. In Section 1.3, we revealed our motivation to the long-lasting research. We presented the increasing need of organisations to improve their BA in order to thrive with uncertainty of today's and future unforeseeable market. We explained that such a capacity to adapt to, create, and leverage changes partially depends on the capacity to innovate technologies. Therefore, we decided to focus on the technology transitions on the technology level, in particular on the GUI level. This led to formulating a research problem put forward in Section 1.4.

Step 2: Objectives of a Solution. Our main objective is to develop a methodical framework aiming at better technology transition. This is represented by a research goal put forward in Section 1.5. The major challenges include the awareness of current GUI technology and the corresponding design & architecture patterns. Additionally, they address the understanding of theories on evolvability, and EE theories. Overall, the main challenge is to build a methodical framework that will combine aforementioned concepts, and that will be applicable to industry across a vast range of different GUI technology stacks. These objectives are formulated along the research questions RQ 1 and RQ 2 in Section 1.5.2.

6.5.2 Design & Development Phase

Step 3: Design & Development. Based on the acquired knowledge during the conceptualisation phase, now we design & develop different artefacts (methods, SW prototypes). They include a new methodical framework ADA that emerged from their integration.

6.5.3 Demonstration & Evaluation Phase

DSRM requires a demonstration of the designed artefact. This helps answering questions about whether the solution actually works or not. However, in applied DSRP, the demonstration of the methodical framework is formalised prior its evaluation. Therefore, we conceptually merge these steps together. The evaluation step of DSRP is redefined as a twofold process:

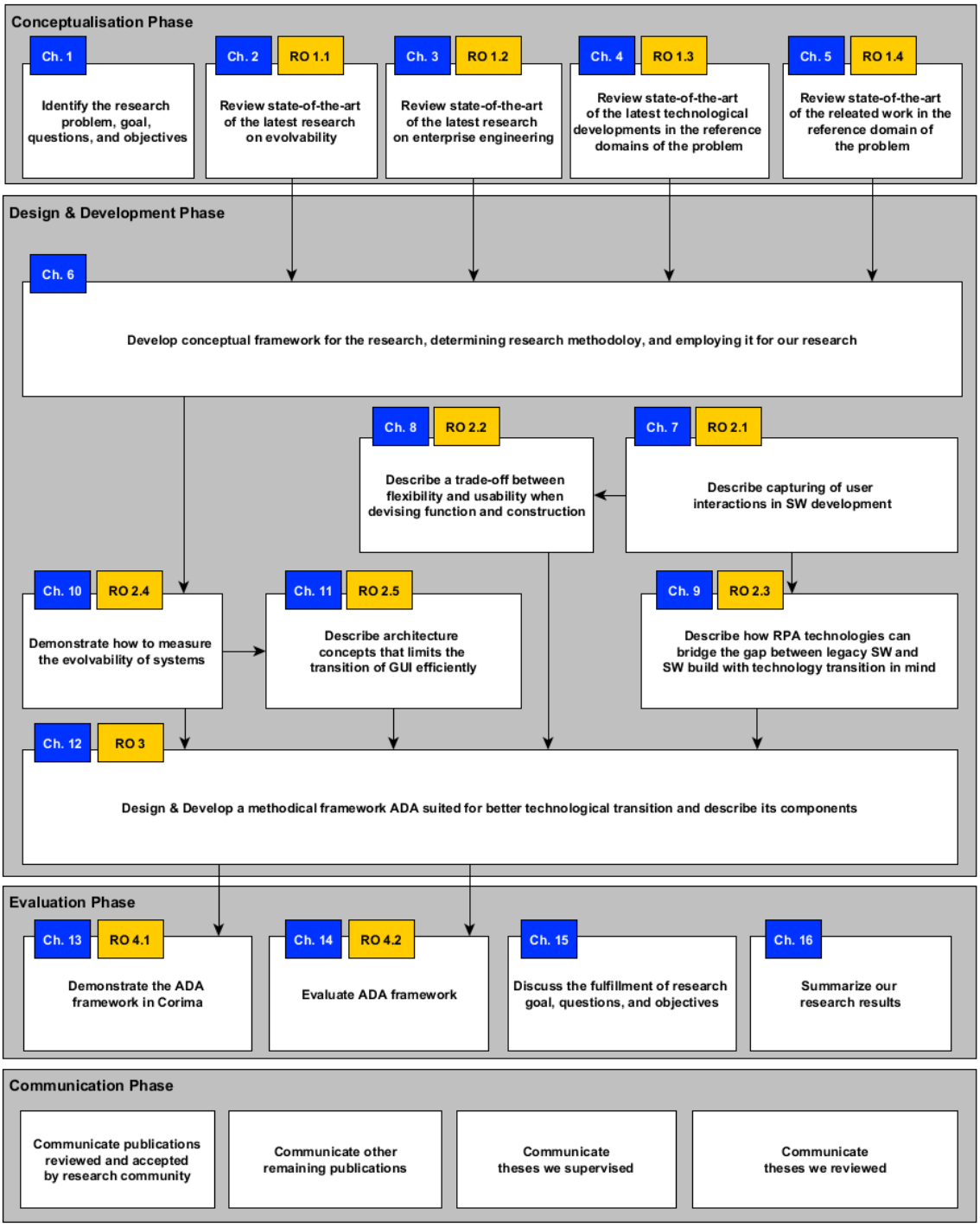

Figure 6.5: DSRP model of our research

- Demonstration of the methodical framework together with its prototype implementation in Corima.
- Formal validation and verification of the methodical framework, and the evaluation of its industrial feasibility.

Step 4: Demonstration. After developing proof-of-concept-level prototypes, the resulting ADA artefact was extensively adapted by COPS to production use in Corima. From the research point of view, we derived the software architecture of the implementation, and we extracted statistical data to understand the extent in which ADA is used. This helps us to further investigate its strengths and weaknesses in both industry scale as well as in the academic scale.

In this research, Corima becomes our case study. Since the prototypes are developed in that case study, other research methodologies like system development and prototyping were considered as part of the overarching DSR. As explained by Olfat [157], in a prototyping methodology the tasks flow from constructing conceptual framework and developing system architecture, to building prototype system, observing and evaluating it. Along the way, the prototype architecture is refined via simultaneous consideration for alternative design before building it using appropriate technologies. Although, testing the usability of the prototype should be included as a part of prototyping, it is not considered here. The prototype (in a case study) is only for demonstration of the feasibility of using ADA and not testing it. The formal evaluation of the ADA artefact is performed later in the evaluation phase. The details of the implementation of the prototype will be explained in Section 12.4.

Step 5: Evaluation. The ADA methodical framework has been used and refined for more than 7 years. It was included in shifting Corima across multiple GUI stacks, and thereby practically used by world-wide biggest corporate and banking customers of COPS. It helps us to further evaluate its strengths and weaknesses from the industrial as well as academic stand point.

6.5.4 Communication Phase.

Step 6: Communication. Manuscripts relating to all artefacts have been published in academic journals, academic conference proceedings, and posters. Mainly, the early-stage results were presented on Conference on Business Informatics in Lisbon in 2015 [A.1]. The same year, on the Enterprise Engineering Working Conference in Prague [A.2], the original idea of applying EE into SW was proposed. Later in 2017, we shared our progress on a Doctoral Consortium along Enterprise Engineering Working Conference in Antwerp [A.4]. On the same conference, in a conjunction with University of Antwerp Management School, we showed the results of applying NST to financial models [A.3]. Additionally, the same year, on World Conference on Information Systems and Technologies on Porto Santo [A.5], we revealed our perspective on the flexibility-usability trade-off. Finally, in 2018, we laid

down the roots of ADA and disclosed it in a paper on Enterprise Engineering Working Conference [A.6] in Luxembourg. The next year, 2019, the progress on ADA was presented on another Enterprise Engineering Working Conference [A.7], this time in Lisbon. By 2021, we submitted the paper to Journal of Science of Computer Programming, and we believe the results should be published later in 2021.

In addition, this research effort received attention in international journals and conferences. Namely, by 2018, it was referenced in IEEE International Conference on Research Challenges in Information Science [79]. The same year, it was referenced in a journal JMIR mHealth and uHealth ranking Q1 in the health informatics category. By 2019, it was recognised on International Symposium on Business Modeling and Software Design [42]. Finally, by 2021, it was spotted in two journals – Information Software and Technology journal ranking high in Q2 category, and Software and Systems Modeling journal, also ranked in Q2 category.

Moreover, along the way, a number of corresponding bachelor and master theses on Faculty of Information Technology along Czech Technical University as well as on Masaryk University in Brno were published, some of which we supervised [A.11, A.12, A.13, A.14, A.15, A.16], some got reviewed by us [A.17, A.18]. Finally, the very early stage idea of this research emerged from the diploma thesis about projectional editors, this thesis was also recently spotted on the Nordic conference NISK in 2020 [13].

6.6 Chapter Summary

In this chapter, we explained the overall research strategy and design employed in this research. We started with presenting a conceptual framework that helps us to answer the research questions formulated during the conceptualisation phase in Section 1.5.2. Furthermore, we discussed in detail DSRM, the selected research methodology. We outlined its brief history and justified its choice for our research. Because DSRM does not prescribe the research steps, we decided to use DSRP. We mapped different aspects of this process into the research objectives put forward in Section 1.5.2. The process goes through four phases – conceptualisation, design & development, demonstration & evaluation, and communication. Their details and approaches employed to complete them were described and visually presented.

Having the detailed understanding of state-of-the-art and the research path clarified, the next part of this dissertation thesis includes the remaining chapters of the design & development phase, and it also partially covers the evaluation phase.

Part IV

Main Results

Chapter 7

Interactions Between People and Technology

During design of Corima, a requirement for so-called *confirmation principle* came up. By 2015, we published a corresponding article on a Workshop on Cross-Organizational and Crosscompany BPM (XOC-BPM) co-located with the 17th IEEE Conference on Business Informatics in Lisbon [A.1]. In this chapter, we provide its version adjusted for this dissertation thesis and updated with respect to the latest version of DEMO.

In Corima, several users communicate their demands to a server connected to a risk management banking system. Corima processes various operations carrying out needed information (deals, Foreign eXchange (FX) rates, balances, etc.). This information is necessary for an underlying risk management system to do proper calculations. Since much of this information is critical for risk calculations, it cannot appear in the target risk management system unapproved by privileged users, so-called *confirmators*. This integral principle is called the confirmation principle.

While the confirmation principle includes actors that are in a social interaction, we argue it may benefit from General PSI theory described in Section 3.7. Therefore, in the context of this chapter, the term *transaction* refers to transactions defined in General PSI theory, and used in DEMO introduced in Section 3.8.

The transaction axiom of the General PSI theory declares that the acts are performed in patterns called transactions [61]. Within transactions, the commitments of subjects (human beings) are raised. Our hypothesis is that the confirmations in Corima are essentially these transactions. Users of Corima are mostly human beings acting in the roles of subjects. They enter into a commitment regarding a certain object affirmation. The affirmation is a specific outcome of the confirmator's decision. It may be a yes/no result or something more complex, like a scale value or even a free comment.

In this chapter, we sum up the key expectations from the confirmation principle in Corima, and we map them to the concepts described by General PSI theory and DEMO. We also outline a possible design and a basic implementation. This is our first attempt to map EE theory into SW what helps us to address our research objective RO 2.1.

This chapter contains a modified version of: *Ondřej Dvořák, Robert Pergl, and Petr Kroha. Confirmation engine design based on PSI theory.* Published in: *17th IEEE CBI, Workshop XOC-BPM. Lisbon, Portugal, 2015. [A.1]*

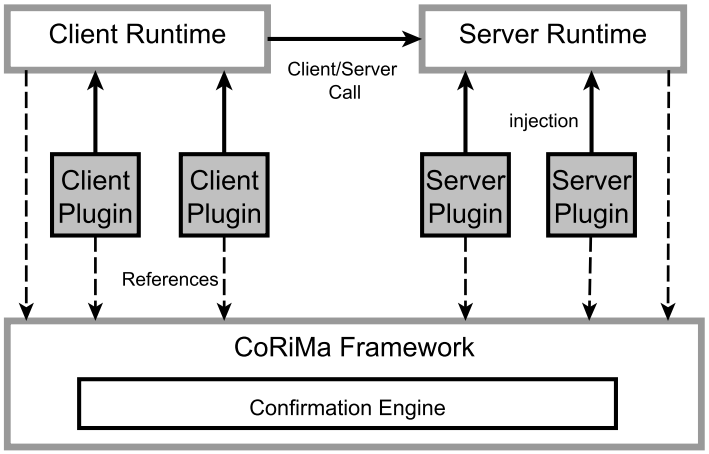

Figure 7.1: Corima architecture

This chapter is organised as follows. In Section 7.1, we describe the purpose of a confirmation engine in Corima. We map the confirmation principle to DEMO and bring a suitable naming of all its fundamentals in Section 7.2. On the top of the confirmation principle we introduce the confirmation engine in Section 7.3. We show how it fits the Corima architecture, and we present code snippets of its main components. Finally, we present an example of a deal confirmation in Section 7.4, we mention the related work in Section 7.5, and we conclude the paper in Section 7.6.

7.1 Corima and the Confirmation Principle

In Section 6.2.1, we put forward Corima. We explained that it is an application platform for the development and execution of financial-oriented applications. It consists of a runtime and a framework. Runtime is a client-server application that can host and execute plugins composed of modules and features offered by the framework. The plugins themselves implement a client and a server core that is executable within a corresponding part of the runtime. Each plugin maintains certain instruments from a financial market. Its client-core views the instruments, its server-core transfers the instruments into an underlying risk management system. For example, FX rates of currencies are displayed in the client-core, whereas the physical deposition of rates happens within the server-core.

Various instruments with which the plugins operate must undergo a specific confirmation process before landing in the risk management system. This process must go hand in hand with the so-called confirmation principle. The principle guides the process through predefined mandatory and optional steps. The use-cases for confirmation can significantly differ among the instruments. Some instruments can be confirmed automatically after a given time is elapsed. Some can be changed by a confirmator, while others cannot. Some instruments can even be cancelled and consequently result in a cancellation of the confirm-

ation. All in all, the demands for various kinds of confirmations can be custom-built.

The general idea is to design a unit in the Corima framework that would meet expectations regarding the confirmation principle. It should be clearly defined and easily applicable by a developer facing a request on confirming a given instrument. We call this unit *confirmation engine*, and we elaborate it in Section 7.3. The basic architecture of our approach is presented in Fig. 7.1.

7.2 DEMO and the Confirmation Principle

On top of the transactions, we can develop an infinite number of systems to maintain them. These systems can use various technologies, techniques, and platforms that are vital to the enterprise. However, they still have something in common – all are functionally equivalent. It means that regardless of the implementation, any such system must be updated when a single process of an enterprise changes. Clearly, as heavily discussed by NST (see Section 2.2), the process change then becomes a rather big cost.

The true power may come with a system that would not be sensitive to such a process change. Ideally, a system that would not even require any manual modification when a conceptual level evolves. The confirmation principle may be seen as such a process. Regardless the actual technology used by Corima, the confirmation process remains the same.

A range of similarities can be identified when comparing the confirmation principle and DEMO. Nevertheless, these can only be seen from a certain point of view. In this section, we analyse these similarities, and we evaluate a possibility of the confirmation principle benefiting from them. We follow up this section by outlining a DEMO-based design and an implementation of the confirmation engine as a software unit supporting the confirmation principle.

7.2.1 Requester, Confirmator, Confirmation Pattern, Confirmation, and Affirmation

In Section 3.7, we introduced DEMO transactions. It is a sequence of acts involving two actor roles, an initiator and an executor. The sequence must respect the transaction pattern defining legal acts during a transaction processing. A transaction is started by the initiator who performs a request to have a desired product declared. Such an act leads to the status `requested`. The executor reacts by promising the product, and the transaction advances to the status `promised`. This act represents a guarantee that the executor is going to produce the requested product within an execution phase of the transaction[1]. As soon as the executor finishes the production, it performs the `declare` act, thus bringing the transaction into the status `declared` meaning the product is ready. At this moment,

[1]Technically, it may happen that the promise is revoked later by the executor, however, we do not deal with this possibility at this place.

the initiator can accept the product and the transaction ends. However, by rejecting the product, the initiator could express its disagreement with the product, and the transaction goes back to the status `declared`. Such a decision would result in a negotiation to clarify mutual expectations on the product.

In Corima, the *confirmation* is a sequence of acts, too. It involves two subjects, a *requester* and a *confirmator*. As well as in DEMO, the sequence must respect a given *confirmation pattern*. The sequence is started by a requester performing a request to create a desired product. However, it is important to realise what the product actually is. The confirmation principle insists on preventing a subject to persist an object that has not been seen and subsequently confirmed by anybody else. Thus, the object itself is not a product, the sole *affirmation* pertaining to an object is. In case the confirmator has in mind to analyse the created object, it can give a promise and proceed to an execution phase. The production of an affirmation must result in a declare act performed by a confirmator. Again, it is highly important to realise what the declare practically means in a context of the confirmation principle. The confirmator must have at least an option to express its agreement or disagreement with an object, yet there can be much more. Nevertheless, this outcome must be a property of the affirmation. As soon as the confirmation is brought to the status `declared`, the initiator can proceed with either accepting or rejecting the declared affirmation.

Let us summarise the initial mapping between DEMO and the confirmation principle. The confirmation principle is generally a PSI theory from DEMO adapted to the needs of Corima. The notions of *initiator* and *executor* in DEMO are represented by the notions of *requester* and *confirmator* in the confirmation principle. The *transaction pattern* and the *confirmation pattern* both define appropriate steps of a process and relations among them. *Transaction* resp. *confirmation* is then a walkthrough in the corresponding pattern. *Product* and *affirmation* are the interests regarding of which subjects (requester and confirmator) enter into a commitment by initiating the transaction (resp. confirmation).

7.2.2 Confirmation Kind and Affirmation Kind

In DEMO, the transaction kinds imply a specific flow of allowed acts and the corresponding states of the transaction.

The same may be applied to a *confirmation kind*. The confirmation kind is based on a specific confirmation pattern. The main difference is that within the confirmation engine, the acts are performed explicitly (e.g., by calling a method `request()`). One can only define what should be an evidence that the act just happened (e.g., sending an email to its counterpart)[2]. If a certain act is senseless or not allowed for the given confirmation kind (e.g., if the requester is not allowed to disagree with a created affirmation), this must be declared. This is similar to specifying action rules in the AM of DEMO.

[2]This corresponds to facts in the PSI theory described above, however we do not go into such detail here.

A transaction in DEMO (hopefully) results in a product successfully delivered. Each transaction kind has a specific product kind as its result. Similarly, each confirmation kind has a specific affirmation kind as its result. An affirmation is the product of a confirmation. Thus, a confirmation (hopefully) results in an affirmation successfully produced. The affirmation itself is a set of properties and their corresponding values. All these properties represent the confirmator's notion regarding an object which is tasked to approve (e.g., a property called *result* can transfer an information whether the confirmator is fine with the object or not). Nevertheless, one has to introduce all the required properties before a confirmation process is started. This is done by an affirmation kind which defines it.

To sum up, a transaction kind in DEMO corresponds to a confirmation kind. The product kind is represented by an affirmation kind. The product is the affirmation itself. It expresses a decision of a confirmator whether the given object is approved or not.

7.2.3 Revocations

In DEMO, revocation is a situation when the initiator or the executor change their minds after performing a certain act. DEMO defines four different revocation patterns for `request`, `promise`, `declare` and `accept`, as we can see in Fig. 3.11. If allowed by the other side, each of them can lead to a transaction step in the standard transaction pattern, which constitutes the aforementioned complete transaction pattern shown in Fig. 3.11. It means that not all transaction kinds allow to revoke an already performed act. Revocations may be even technically impossible (e.g., shredding of a document).

Confirmations in Corima exhibit the same behaviour. For instance, shortly after a requester requests an affirmation of a given object, it can change its mind and want to take the request back. It depends on the confirmation kind if such a step is allowed, and how the confirmation engine should react. Since these situations are expectable, we want to support them, too. Thus, we have to introduce revocation patterns into the confirmation principle. To make it work, we have to define their specification for each confirmation kind individually.

7.2.4 The Confirmation Principle Summary

In this section, we clarified how DEMO and the underlying PSI theory can be mapped to the confirmation principle. We identified a mapping between a requester and an initiator, and a mapping between a confirmator and an executor. We showed that a confirmation kind maps to a transaction kind, and an affirmation maps to a product in DEMO. DEMO models successfully describe the whole enterprise process logic by transaction kinds derived from the complete transaction pattern. Thus, we argue that confirmations based on the same complete transaction pattern can handle all necessary situations of various confirmation processes. Table 7.1 presents the overview of the resulting mapping.

DEMO	Confirmation Principle
Transaction Pattern	Confirmation Pattern
Transaction Kind	Confirmation Kind
Transaction	Confirmation
Product Kind	Affirmation Kind
Product	Affirmation
Initiator	Requester
Executor	Confirmator

Table 7.1: Mapping between DEMO and the confirmation principle

7.3 The Confirmation Engine

Let us recall that the confirmation engine is supposed to be a unit in Corima supporting the confirmation principle. It should be universal, thus independent of the objects it is used for. It must allow an easy creation of a new confirmation kind for a given type of object. Finally, it must be able to integrate new confirmation kinds and thereby enable the confirmation principle for any corresponding object.

In this section, we elaborate a possible design of the confirmation engine. We show how it fits the architecture of Corima.

Since the implementation itself is out of the scope of this paper, we only present basic code snippets in C# of its most interesting parts. We follow a similar scenario as we did while describing the mapping of DEMO to the confirmation principle in Section 7.2. We start with an overall architecture by identifying the key components. We subsequently investigate these components deeper. We discuss a design of a requester, an executor, and a confirmation service. We show how the notions of a confirmation pattern, a confirmation, and an affirmation are embodied in these components. We continue with a confirmation and an affirmation kind, and we explain how they are integrated in the whole engine. We end up with revocations by determining their proper involvement.

7.3.1 The Overall Architecture (The Confirmation Clients and the Confirmation Service

Corima is a client-server application platform. Users interact only with a Corima client. In a suitable GUI, they observe and maintain financial-based data. All data changes are communicated to a server that stores them into an underlying risk management system. The confirmation is a process (a sequence of acts) between two users, the first one in a role of a requester and the second one in a role of a confirmator. The requester performs a data change, and a confirmator approves it before leaving it in the underlying risk management system.

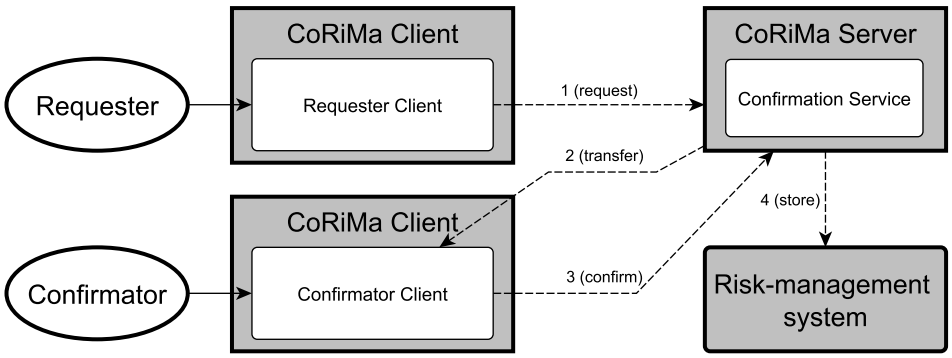

Figure 7.2: General Confirmation Process

Fig. 7.2 covers the usual confirmation process. A user in a requester role is connected to a Corima client. It performs a change of a certain data object. The object is sent to a server within a request call. The server just publishes a temporary created object for the corresponding confirmator. The object is transferred to the Corima client and displayed within an user interface designed for making confirmations.

A user in a role of a confirmator confirms the given object (i.e., it creates the affirmation) and sends it back to a server. Finally, the confirmed object is saved to the risk management system by the confirmation service.

It is obvious that a unit supporting such a confirmation process must take a part on both, client, and server side. On the client side, two components must be introduced. The first component comes from the requester, the second from the confirmator point of view. Let us call them *Requester Client* and *Confirmator Client*. These components represent actors in DEMO, the initiator and the executor. They provide methods for performing all valid acts. Technically, `RequesterClient` and `ConfirmatorClient` are classes that mediate all clients' demands to the server (see the C# code snippet below). For a simplicity, access modifiers and the detailed implementation are omitted. The *ConfirmatorClient* will be implemented analogously.

```
class RequesterClient<...>
 ...
 void Request(object obj) {
    // Client-server call requesting an
    // affirmation of an object
 };
 void Quit(object obj) { ... };
 void Accept(object obj) { ... };
 void Reject(object obj) { ... };
 ...
}
```

Listing 7.1: Snippet of the RequesterClent class

On the server, a *confirmation service* component will take a part. It is a component responsible for reacting on acts performed on the client. It basically contains two subcomponents, *Confirmation Provider* and a *Confirmation Handler*. The Confirmation Provider publishes end points to which the client-server calls are communicated. The calls are directly forwarded to a Confirmation Handler that maintains the whole confirmation process. Technically, these subcomponents are implemented by two classes, `ConfirmationProvider` and `ConfirmationHandler`. Again, we omit the implementation details and concentrate on the general structure only. We list the code snippets below.

```
class ConfirmationProvider
 ...
 void OnRequested(
             ...,
             object obj) {
    // Forward of a request to a Confirmation Handler
 };
 void OnQuit(...) { ... };
 void OnAccepted(...) { ... };
 void OnRejected(...) { ...};
 void OnPromised(... { ... };
 void OnDeclined(...) { ... };
 void OnDeclared(...) { ... };
 void OnStopped(...) { ... };
 ...
}
```

Listing 7.2: Snippet of the ConfirmationProvider class.

```
class ConfirmationHandler {
 ...
 void Register(...) {  }

 void Request(
          ...,
          object obj) {
    // Request confirmation of a given id
 };
 void Quit(...) { ... };
 void Accept(...) { ... };
 void Reject(...) { ... };
 void Promise(...) { ... };
 void Decline(...) { ... };
 void Declare(...) { ... };
 void Stop(...) { ... };
 ...
}
```

Listing 7.3: Snippet of the ConfirmationHandler class.

In this section, we clarified key components of the whole confirmation engine architecture. A requester and a confirmator roles, we identified in Section 7.2.1, are represented by `RequesterClient` and `ConfirmatorClient`. Both mediate demands to the server side. The confirmation principle itself is realised using two components – `ConfirmationProvider` and `ConfirmationHandler`. These constitute a Confirmation Service maintaining the entire confirmation logic. In the next section, we describe how the confirmation pattern, a confirmation, and an affirmation fits into the Confirmation Service.

7.3.2 The Confirmation Pattern in the Confirmation Service

The confirmation in Corima is a sequence of legit acts (see Section 7.2.2). Their legality is given by the current context of a confirmation (who is executing the act, which acts have already passed, and which object is a subject of the confirmation). Such a sequence must respect a certain confirmation pattern. We already pointed out that we suppose the standard transaction pattern in DEMO should satisfy all needs for confirming various objects. Thus, the valid subsequences of acts in a confirmation are driven by a standard transaction pattern.

In fact, the Confirmation Service, we built up in the previous section, is merely an implementation of this pattern. It contains a method for each possible act in the standard transaction pattern. It behaves in accordance to that pattern and evaluates if a given act (method call) is legal in the current situation. If so, it moves the confirmation into another state.

Each confirmation in Corima is of a specific kind, a confirmation kind. The purpose of each confirmation is a creation of an affirmation. Technically, an affirmation is just a set of properties and their values. It is an instance of a class specifying these properties. Confirmation Service only have to know how to deal with such an instance. When a confirmator performs a `declare` act, a new value of this instance is delivered within a client-server call. In case the Confirmation Service persists this value, a proper way of persisting it must take place. We already explained in Section 7.2.2) that an affirmation kind specifies the information contained in an affirmation (being its product). It means that if a Confirmation Service knows beforehand the affirmation kind, it can persist the affirmation appropriately.

Now, we are ready to discuss which information is actually needed by a Confirmation Service to serve the acts performed on the client (e.g. `request` or `declare`). It must at least know the confirmation kind, and the affirmation kind, and the object being confirmed. To identify this information smoothly, we introduce a concept of registrations. The confirmation kind is paired with the corresponding affirmation kind and registered with a confirmation handler. All client-server calls use the identifier of a registration to manifest in which confirmation kind they are interested in. The methods of `ConfirmationHandler` are enhanced as follows:

```
...
void Register(
       int registrationId,
       IConfirmationKind ck,
       IAffirmationKind ak);

void Request(
       int registrationId,
       object obj);
...
```

Listing 7.4: Snippet of the methods in ConfirmationHandler.

In this section, we outlined that the Confirmation Service implements the complete transaction pattern from DEMO. It registers pairs of confirmation kinds and affirmation kinds, and it uses these registrations for handling the client-server calls. To recognise the registered pair, `registrationId` is placed in each client-server call. In the next section, we show the purpose of a confirmation kind and an affirmation kind, and we describe how the Confirmation Service uses them.

7.3.3 The Role of a Confirmation Kind and an Affirmation Kind

The primary responsibility of a confirmation kind is to manage the process flow, namely to:

- decide if the act is allowed or not,
- specify whether the act is tacit[3] or not.

Next, in Corima, it is necessary to support *custom actions* tied to different acts in different confirmation kinds. For example, sending an email after giving a `promise` is a custom action that is applied only for some confirmation kinds. Thus, the second responsibility of a confirmation kind is to specify these custom actions.

Let us consider an interface of a confirmation kind in the code-snippet below. If each method is implemented, the Confirmation Service can take it into account. For instance, if the `promise` happens, the Confirmation Service can execute `PromiseAct` defining a custom action (e.g., sending an email).

Another example is a tacit execution. Let us consider a situation when the `accept` is tacit, and the `reject` is not allowed. If a confirmator performs the `declare`, the Confirmation Service can react by directly moving the confirmation into the state `accepted` and execute the custom method `AcceptAct`.

[3]A tacit execution of an act means that the act is performed automatically, without being explicitly requested.

```
interface IConfirmationKind {
 Act RequestAct { get; set; }
 Act QuitAct { get; set; }
 Act AcceptAct { get; set; }
 Act RejectAct { get; set; }
 Act PromiseAct { get; set; }
 Act DeclineAct { get; set; }
 Act DeclareAct { get; set; }
 Act StopAct { get; set; }
}

class Act {
 bool IsAllowed { get; }
 bool IsTacit { get; set; }
 Action<...> Execute { get; set; }
}
```

Listing 7.5: Snippet of the IConfirmationKind interface.

An affirmation kind is implemented as a class having a set of custom properties characterising the affirmation. The requester and the confirmator operate with those properties while performing acts (e.g., if the requester is deciding whether the declared affirmation is acceptable or not).

To sum up, confirmation kinds define the process flow and the custom behaviour of the Confirmation Service. Affirmation kinds hold custom properties, whose values constitute the affirmation.

7.3.4 Revocations in the Confirmation Service

In Corima, a revocation is a situation when a requester or a confirmator change their mind just after performing an act. If we want to support such situation, we have to enhance the confirmation senrvice.

In DEMO, four revocation patterns exist in the complete transaction pattern (Section 3.8). We believe that by extending the implementation with revocation patterns, we cover all exceptional cases within the confirmation principle. The changes will affect all the identified components. We do not pursue this topic any further here, let us just show the relevant extension of `RequesterClient`:

```
class RequesterClient<...>
 ...
 void RevokeRequest(object obj);
 void RevokeAccept(object obj);
 ...
}
```

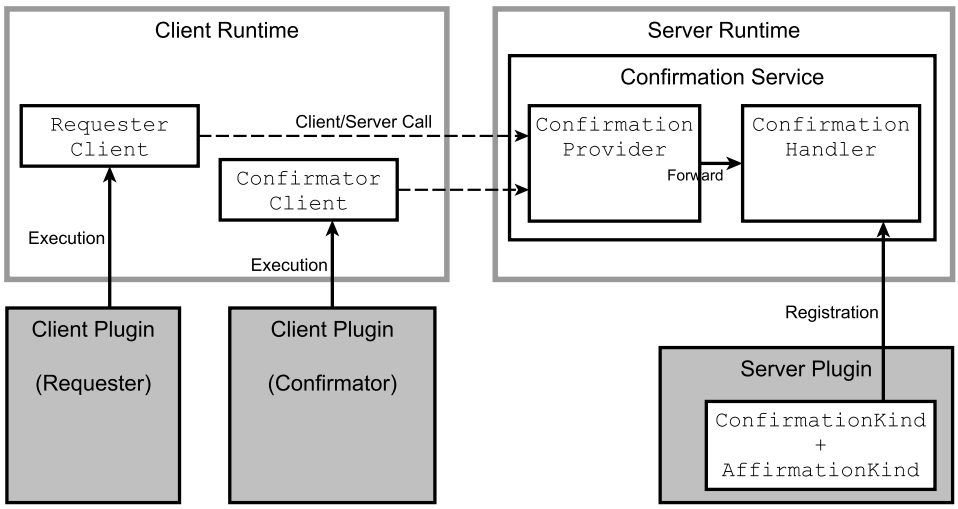

Figure 7.3: Confirmation engine in the Corima infrastructure

Listing 7.6: Snippet of the extension of RequesterClent class.

7.3.5 Confirmation Engine Summary

We conclude our description of the confirmation engine with Fig. 7.3 illustrating how the engine fits into the Corima infrastructure.

The confirmation engine consists of the following key components:

- `RequesterClient`
- `ConfirmatorClient`
- `ConfirmationProvider`
- `ConfirmationHandler`

The client plugin dedicated for requesters uses `RequesterClient` to mediate its requests to the server. The client plugin for confirmators handles its demands via `ConfirmatorClient`.

The server publishes end points using `ConfirmationProvider` that forwards all requests into `ConfirmationHandler`. This is a central unit handling the entire confirmation process. Without an extra information, this unit is useless. It must be supplemented by `ConfirmationKind` paired with `AffirmationKind`. Both are registered with `ConfirmationHandler` during the Corima server startup.

7.4 An Illustrative Example

Let us demonstrate how the engine can be used to address a concrete problem.

Let us consider deals and their confirmations. Each creation of a deal must be confirmed before the deal ends up in the risk management system. It means that the created deal is not automatically persisted in the risk management system. Instead, it stays in a temporary state until a privileged confirmator approves it. In case the confirmator is not satisfied with the deal, he can deny the creation and drop a comment explaining why.

First, we clarify the whole problem deeper and write down rules for deal confirmations. Second, we show how the rules influence the implementation. We conclude with a description of how the entire integration works.

7.4.1 Rules of the Deal Confirmation Case Study

To successfully integrate the confirmation of deals into Corima, we have to scrutinize the problem and specify the rules first.

- Rule (1): The confirmator is not allowed to refuse a request to perform a confirmation. Once he is asked to confirm a deal, he has to produce a result.
- Rule (2): No revocations are allowed. Neither the requester, nor the confirmator can change their minds after performing an act.
- Rule (3): As soon as the confirmator approves or denies the deal, the confirmation is over. The requester cannot express a disagreement with the confirmator's decision.
- Rule (4): An email notification is sent to a confirmator once a request is performed.
- Rule (5): The confirmator does not have other option but to approve or deny a deal. He can just optionally leave a comment.

We described that the confirmation engine (respectively its confirmation handler) does not support any confirmation process on its own. This knowledge is represented by the injected confirmation kind and an affirmation kind. Both the kinds have to be implemented and registered in the confirmation handler. Thus, we have to implement the following:

- `DealConfirmationKind`, a class representing a confirmation kind. The interface `IConfirmationKind` must be implemented by this class.
- `DealAffirmationKind`, a class representing an affirmation kind.

The way the classes are implemented directly depends on the rules stated before. Let us examine each rule and describe how it is implemented:

- Rule (1): The fact that the confirmator cannot refuse a confirmation request means that the `promise` act is performed tacitly. The implementation of `DealConfirmationKind` must consider it and set a value of its `PromiseAct` property to `IsTacit=true`.
- Rule (2): No revocation is allowed, thus all the properties regarding the revocations must be set accordingly
  (e.g., `RevokeRequestAct.IsAllowed=false`).
- Rule (3): Because the requester cannot actually react on a decision done by a confirmator, his `accept` is tacit[4]. That means the property `AcceptAct` must reflect this.
- Rule (4): Sending of an email is a custom action. It must follow the `request`. Thus the method `Execute` of `RequestAct` must be implemented to send an email to the confirmator.
- Rule (5): The look of `DealAffirmationKind` is determined by the options available to a confirmator. It must be a class having two properties. One of a type `enum`, having two possible values: `Approved` and `Denied`. The second property representing a comment, so most probably of a type `string`.

In this part, we described how each of the rules affects the implementation of two types of kinds – `DealConfirmationKind` and `DealAffirmationKind`. In the next part, we register them in a confirmation handler. We show how the requester and the confirmator undertake the expected deal confirmation process.

7.4.2 Deal Confirmation Process

Let us demonstrate the entire deal confirmation process now. It is depicted in Fig. 7.4.

During a Corima server startup, the server-side Deal Confirmation plugin is loaded and `DealConfirmationKind` together with `DealAffirmationKind` are registered in the confirmation handler under a specific identifier `Id`, let us assume the number `248`. This identifier must be known to both the client and the server-side.

A pair of client-side Corima plugins must be present to provide user interfaces for the deal confirmations. One plugin aimed for the creation of the deal, the other is dedicated for its approval. Once a requester creates a deal, the deal must be confirmed. It is done by pressing a Request button. The command related to this button is implemented as follows:

```
OnRequestExecuted(Deal deal) {
  var rct = new RequesterClient(248);
  rct.Request(deal);
 ...
```

[4]This means that the result phase will practically lack the accept act, however, formally, it is present, so the transaction axiom is still valid.

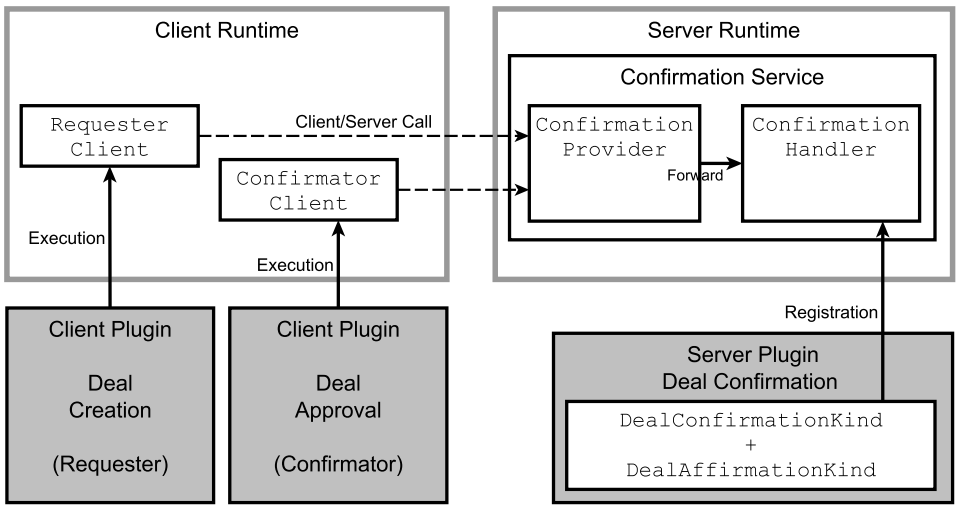

Figure 7.4: Deal Confirmation Process

```
}
```

Listing 7.7: The implementation of a request command

This method results in a client/server call to the confirmation provider. The provider forwards the call to the confirmation handler asking for a confirmation identified by `248`. The confirmation handler searches for the confirmation kind registered under the number `248`. Subsequently, the handler changes the status of the confirmation to `requested`, and it executes the method `Execute` of the corresponding `RequestAct`. Since `RequestAct` of the `DealAffirmationKind` has to send an email to the confirmator, it is done consequently. Because the `PromiseAct` is tacit, the handler instantly adjusts the status of the confirmation to `promised`.

Now, the whole process waits for the confirmator's act. The client-side plugin Deal Approval responsible for making approvals is designed for this purpose. Its implementation is more or less analogical to the previous one, thus we do not elaborate it any further.

7.5 Related Work

Formetis is a Dutch commercial company that has developed and successfully applied a DEMO Engine (also called a DEMO Processor) for its customers, as documented in [97] and [105]. The DEMO Engine enables the construction and execution of DEMO models. It is also implemented in the .NET platform using the C# language. The engine itself is independent on the technological environment. It has been used to implement desktop workflow applications and an educational application `http://demoworld.nl`.

The first DEMO Engine application in production is a case management system for a company that provides energy and utility services. The customers – citizens – are active

co-producers of the service by providing information, coordination of some tasks, approval of decisions, etc. The contract covers issues such as type of services provided, costs, costs calculation procedures, conditions for payments, letters, mails, instructions for the sub-contractors, etc. There is an enforced compliance to legal procedures, policies, conditions, approvals etc.

Formetis's solution is currently the leading industrial solution of a software system based on DEMO and its underlying theories. However, its scope is much broader than our confirmation engine. It is a complete general workflow engine, while our confirmation engine is a rather lightweight, compact module for Corima fully focused on its specific task. Overall, technically, it would be perfectly possible to use Formetis' DEMO Engine as a confirmation engine, however, from the software engineering point, often lightweight focused solutions may be preferred, as is the case of Corima.

Agreement Technologies [167] may seem similar to our approach, however there is a fundamental difference. Our approach addresses confirmations of ontological transactions, which cannot be automated [59], while agreement technologies are focused on automated agents, i.e., the infological level.

7.6 Chapter Summary

In this chapter, we mapped the PSI theory to the confirmation principle. We designed a research artefact prototyping a confirmation engine. We outlined its implementation and described it as an independent unit in the Corima framework. At the time of prototyping the confirmation engine, we were not in a position to evaluate the solution broadly as it was not used in practice. However, later it was deployed to an environment of few banking and corporate customers of Corima. Since 2015, it has been widely used in a production. This gives us an empirical proof that the concepts were designed adequately, and the mapping of PSI theory to SW fulfilled its goal.

The goal of this chapter was to show that utilising the PSI theory for a design of the confirmation engine in Corima brought considerable benefits. Mainly, designing the confirmation pattern by mapping the corresponding concepts from the PSI theory results in a guarantee that all possible confirmation situations are covered.

More importantly, it revealed that each object can be confirmed in an independent GUI defined in a Corima plugin. This means that when the GUI technology of Corima changes, only the GUI plugins need to be recreated, the confirmation process remains unchanged.

Chapter 8

Flexibility-Usability Trade-off

In our research, we cope with designing SW better suited for technology transition. Indeed, an important concept of such a SW might be a split of its function and the construction that we investigated in Section 3.6. The function of SW is typically more stable compared to its construction that often continues to adapt to new technologies indefinitely. Therefore, one direction might be to orchestrate the SW from components having the required function yet differing in a construction. The function of confirming objects described in the previous chapter can be an example. Although its construction (GUI where the user confirms the objects) may change, the function remains the same.

However, as addressed by RO 2.2, we have to understand how flexible the components need to be to stay usable in such an orchestration. In 2017, we inspected the phenomenon of the trade-off between flexibility and usability in CBS development. We published a paper in the proceedings of World Conference on Information Systems and Technologies in Porto Santo [A.5]. In this chapter, we adjust it for the needs of this dissertation thesis.

8.1 Introduction

'Increase flexibility, decrease usability' is a well-known trade-off influencing the effectiveness of reusing artefacts in many engineering disciplines. We claim that software development is influenced, too. We propose a model of building components that can help to decrease the costs of software development, while providing a demanded level of flexibility.

In Section 4.1.1, we explained that NATO Symposium in 1968 was a tipping point in SE. McIlroy [68] brought his vision of software assembling instead of programming it. The overall motivation behind the components organised into standard libraries was to produce a software with a higher quality, while reducing costs on its development. The fundamental question is how to design the reusable component in the best way. Antovski [5] explains that 'to cover different aspects of using components, they have to be sufficiently general, but at the same time, they have to be concrete and simple enough to serve to particular requirements in an efficient way'.

This chapter contains a modified version of: *Ondřej Dvořák, Robert Pergl, and Petr Kroha. Tackling the flexibility-usability trade-off in component-based software development.* Published in: *WorldCIST, Porto Santo, Madeira, 2017. [A.5]*

However, notwithstanding the mass of component libraries created for decades, software projects still exhibit a high rate of failure, as documented by the reports of the renowned Standish Group[1]. They usually address topics like 'unclear project objectives', 'scope creep', 'communication gap', etc. [2]. We argue the next topic is the decision about the kind of reusable components. Since the component libraries exhibit volatile quality, it is important to select the right one.

In general, the trade-off indicates that an increase of a flexibility decreases a usability. Our opinion is that a widespread trade-off between flexibility and usability may bring an explanation of the varying qualities of components.

In the next sections, we elaborate on the definitions of flexibility and usability. We will explain the circumstances under which the mentioned trade-off is valid in the domain of CBSs. Our hypothesis is that the trade-off is legit, however, equally flexible components can significantly vary in their usability. The architecture of components seems to influence the flexibility/usability rate. Our intention is to measure the trade-off based on a constructional decomposition of components. The relations between the semantics of the component and the semantics of the system build another dimension of the problem. As we show below, the existing definitions of flexibility and usability do not consider this aspect in reality, thus they do not answer the problem of costs.

8.2 Overview of Flexibility and Usability

Here we use the *flexibility* definition put forward in Section 4.1: 'the ability of a resource to be used for more than one end product' [69], as it may be easily mapped into the domain of CBS:

Definition 11. Flexibility is the ability of a component to be used for more than one CBS.

Usability is a broad term. We already touched in in Section 4.1. It is often used for user experience topics, i.e., handiness, elegance, clarity, etc. [76], commonly exhibited by a software to its end-users. However, for our purpose we need to understand usability in a narrower sense as defined in [111]: '*Usability* is the degree to which a software can be used by specified consumers to achieve quantified objectives with effectiveness, efficiency, and satisfaction in a quantified context of use'. In terms of CBS, the specified consumers are software engineers using the given component. Therefore, we can translate the definition as:

Definition 12. Usability is the degree to which a component can be used by software engineers to achieve quantified objectives with effectiveness, efficiency, and satisfaction in a quantified context of use.

[1]Standish Group is an international advisory company famous for its research on IT projects failures and improvements.

8.2.1 Flexibility-Usability Trade-off

A cross-disciplinary book [132] reminds that 'as the flexibility of the system increases, its usability decreases' is a design principle trade-off. It compares the trade-off to a well-known maxim 'jack of all, master of none' and explains it as follows: 'flexible designs are, by definition, more complex than inflexible designs, and as a result are more difficult to use'. The principle trade-off between flexibility and usability is widespread, e.g., in science – philosophy has a very high flexibility but a very low usability.

However, we argue that the above definition may not be true depending on the definition of flexibility and usability. The following example illustrates that by a certain slightly modified understanding of the flexibility and usability, we can easily find components that exhibit high flexibility and high usability at the same time.

There are companies (such as SAP) delivering cross-industry, standardised software for business lines like finance, sales, marketing, human resources, and many more. The software is designed to be modular: For example, customers can buy an accounting module and integrate it into their enterprise environment. Following Definition 11, the accounting module is highly flexible, as it can be applied in an accounting department of nearly any company. Simultaneously, according to Definition 12, it is highly usable, as it claims to satisfy all needs of account management efficiently. In this case, the flexibility does not seem to decrease usability. Thus, the trade-off is not valid in this case.

The gist lies in the fact that the given definition of flexibility completely ignores the costs of a component integration. The integration of the accounting module may be enormously expensive in practice. All surrounding systems may need changes. In addition, there may be high costs of integration at the side of the accounting module. Therefore, in this particular case, the costs are mostly related to adopting the surrounding infrastructure to the standardised interfaces of the given accounting module, which is not addressed in Definition 11.

To come to more realistic considerations about modular component systems, we revisit this scenario in Section 8.3, where we reformulate it in a more formal way, including the consideration of integration efforts.

8.3 The Core of the Problem

The definition of flexibility does not say anything about the costs related to the amount of labour of fitting the component into a system. The costs are given by the difference in the functionality offered by the integrated component and the functionality demanded by the system in which the component should be integrated. We can speak about a semantic difference. If the component absolutely fits, no changes are needed, and the costs are marginal. In other cases, we may distinguish two situations[2]:

[2]In practice, we encounter typically mix of the situations, however, this does not affect the reasoning made.

1. The integrated component is changed minimally, but the remaining system is adapted intensively to cooperate with the component.

2. The system stays untouched, but the integrated component is modified thoroughly to fit into the system.

These cases support the idea that changes in a component have another (smaller) weight compared to changes in the system when calculating their costs. Fig. 8.1 depicts the described situations. Suppose that CBS is represented by the nodes in the figure. The node with a question mark in the middle represents the integrated component. Suppose, we have two different components, A and B. Let us assume that when using the component A, all the surrounding components need to be changed. On the other hand, when using the component B, we only need to slightly adopt B to fit in the whole system, but we change the system itself intensively. In the view of Definition 11, A and B are equally flexible because they both can be used in the given system. However, the situations are very different and may significantly differ in costs.

The relation between costs, flexibility, and usability has to be extended to respect the influence of Semantic DiFference (SDF) between the demand of the system and the offer of the component. We can write:

$$costs = f(SDF, flexibility, usability). \tag{8.1}$$

In some cases, the SDF has to be removed by changes of the component. In other cases, the SDF has to be removed by changes in the system, or both.

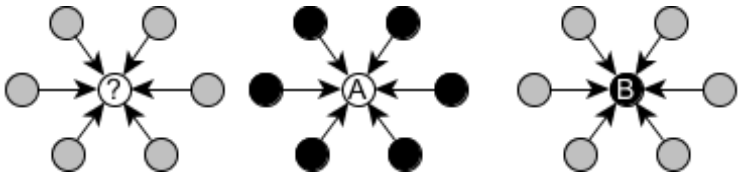

Figure 8.1: Flexibility of components

The first situation refers to the notion of *integrability*. Merriam-Webster defines this term as 'having different parts working together as a unit' – the different parts are related to the surrounding system plus the integrated component, which should work together as a unit.

The second situation refers to the notion of *moldability*. Merriam-Webster explains the roots of this term as the ability 'to knead (dough) into a desired consistency or shape' – the dough is the integrated component, which is kneaded into the shape fitting system.

We believe that the original McIlroy's idea, as well as the proper component-based nature goes the way of *moldability*. Thus, we base our following considerations in this perspective:

Assumption 1. The surrounding system is not changed considerably when integrating a new component.

By including the word 'considerably', we acknowledge that there must always be performed some integration changes to the surrounding system. However, these should be marginal compared to changes of the component itself.

Now, we may offer an alternative definition of flexibility:

Definition 13. Flexibility is the ability of a component to be molded for use in more than one CBS.

By formulating Definition 13, we specified more clearly what does it mean to reuse the component – the component is molded into a shape to be integrated into a new system. Now if we have a high moldability, how do we measure the cost of it?

8.3.1 Two Architectures

To realistically compute the cost of moldability, we argue that we need to distinguish two situations. We call these situations a 'Carpenter model' and a 'Mosaic model': Carpenter builds a (wooden) system by taking various generic components available (logs, beams, planks, etc.) and molds them into a desired shape to be integrated into the resulting system.

Definition 14. Carpenter model in building a component-based system means taking independent components and molding them to be integrated into the resulting system.

These independent components are usually the programming language libraries or some higher-level components, e.g., web development components, or persistence solutions. The building process is typically hierarchical, i.e., we compose more complex components from the simple ones. This forms the *architecture* of the system. This model also applies to building the system from the scratch. The left arrow in Fig. 8.2 illustrates the Carpenter model when building a system.

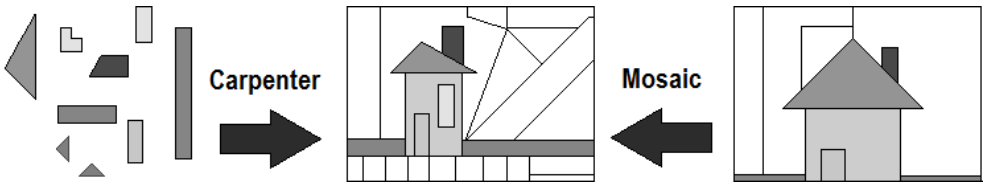

Figure 8.2: Carpenter/Mosaic model of building a component system

On the other hand, imagine a situation where we have a mosaic. The mosaic consists of simple pieces put together to form parts of a picture, like a house. Once we have the 'house component' in the mosaic, we may like to reuse it in another mosaic[3]. To fit the 'house component'[4] into another mosaic, we typically need to mold the component to fit it. It is

[3]In practice, to use the part of the mosaic would mean to recreate it, however, in software, we have the luxury of cloning a piece of code with no effort.

[4]i.e., the house component clone

possible, as we may assume that there exists an original former mosaic that was created using the Carpenter model, i.e., built by combining simple pieces into more complex ones. These complex pieces may be disassembled and molded into the proper new shape. The right arrow in Fig. 8.2 illustrates the Mosaic model when building a system.

To summarise, we may distinguish two models of molding components to be fitted into the resulting system: the Carpenter model, where we mold generic simple components into bigger ones by composing them, and the Mosaic model where we take complex components and mold them by disassembling and changing some parts. These two models obviously differ in how costs should be calculated.

8.4 Proposed Measures

In this section, we discuss and propose measures to calculate the effort/costs of reusing components by means of moldability. Moldability represents the reducing of the semantic difference between the system and the component.

The Carpenter model. This model is essentially a traditional way how software is built. If the components degrade into atomic expressions of a programming language, the composition of the system means putting together these expressions. As we typically assume a text-oriented computer program, we come to well-known counting of Lines of Code (LoC). There are well-established, documented, and discussed methods of assessing the effort/cost of such model of development: Constructive Cost Model (CoCoMo) [88], Function Points Analysis (FPA) [48], and others. CoCoMo computes a development effort as a function of the size of a program. FPA tries to estimate a cost on top of the demanded SW functions. A cost estimation performed by CoCoMo is based on LoC, and it is affected by certain variables. There are several types of CoCoMo models that differ in the selection of variables (basic, intermediate, detailed). As the CBS development increases in its use – mostly by using GUI toolkits – the original CoCoMo model [88] becomes imprecise [24].

FPA incorporates this observation more precisely in the form of the *language gearing factor* [175], which is specified for the most used languages and programming systems. More CBS exhibit lower values – e.g., Powerbuilder scores 26, while assembler scores 119. This illustrates the need to tune the cost estimation methods for various levels of components.

To summarise, the cost of building the Carpenter model component leads to determining a traditional software building effort by taking into account the level of component architecture.

The Mosaic model. This model of molding a component is based on the required functional and non-functional changes to the component. There are several established methods in SE for functional analysis like UML use-case model [93], Extreme Programming User Stories [18], or CoCoMo II Object Points [24]. All these models represent a *functional*

decomposition tree. As explained in Section 3.4, the functional decomposition represents so-called black-box perspective on a system. It is mostly about how the system can be used.

However, the tree nodes represent just abstraction classes of the functionality, which is in the leaves. Thus, we measure the functional changes as the number of changes of the leaves. We may observe that leaves represent a list structure in the end, which is analogous to use-cases or User Stories mentioned above. For the illustration in our example, we use the UML use-case diagram (Fig. 8.3).

The traditional estimation methods cannot be used for the Mosaic model, as molding means here changing some of the (existing) parts of the components.

The effort of molding a Mosaic component can be measured as the number of changes in its *construction decomposition tree* representing its decomposition into sub-components. Construction tree represents so-called white-box perspective of a system, what is 'a model capturing construction and the operation of a system, while abstracting from implementation details; they are assumed to be irrelevant' [58]. A corresponding constructional perspective on a system thus captures of which parts the components are composed, and how the parts are interconnected. We depict construction decomposition using UML component diagram [177] (Fig. 8.4).

The effort of molding corresponds to operations of *adding*, *deleting*, and *moving* a node in the construction decomposition tree [212].

Adding a node means developing a sub-component, which leads to the Carpenter model. Moving of a node is related to restructuring (sub-)components in the system, which is usually done during refactoring of a code. Changing of a node is not listed, as it can be expressed by additions and deletions.

If we denote $E(C_A)$ the total effort of addition operation, $E(C_D)$ the total effort of deletion operation and $E(C_M)$ the effort of moving operation, the resulting effort E of molding a Mosaic component M is:

$$E(M) = \sum_i E(C_{Ai}) + \sum_i E(C_{Di}) + \sum_i E(C_{Mi}), \tag{8.2}$$

$E(C_{Ai})$ may be computed using the Carpenter model estimation method. $E(C_{Di})$ are the efforts of deletions, which are typically much lower, however, in specific cases, the deletion may require additional changes to the component, so we leave it in this general formula. $E(C_{Mi})$ is the effort of modifying a constructional node. It may not be easy to assess it, as it typically is a complex set of additions and deletions. One of the feasible practical approaches is to assess the effort by an expert estimation using a work break-down decomposition [101].

8.5 Assessing Effectiveness and Efficiency in Usability

We may assume that the required efficiency and effectiveness in the usability definition in Definition 12 are concerned by costs. The costs correlate with effort. All the three

components of Equation (8.2) can be transformed to costs. The first one, as usual in the Carpenter model. The second one can be measured in Man-Days (MDs). The third may refer to a fixed cost of contracting or it can also be expressed in MD.

The first example. It presents our approach on an illustrative problem from the GUI of Corima – a tabular data viewer. Let us imagine a situation where we have a system using a tabular data viewer component. Its UML use-case diagram is in Fig. 8.3 (just white use-cases *UC1 - UC5*). Corresponding UML component diagram is depicted in Fig. 8.4 (just white nodes).

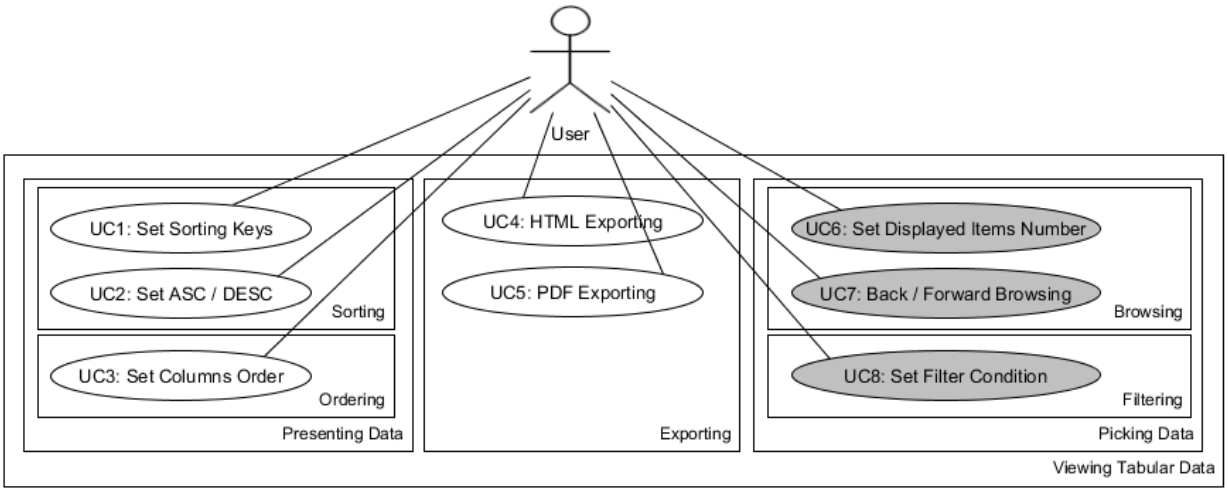

Figure 8.3: Use-case diagram of a tabular data viewer

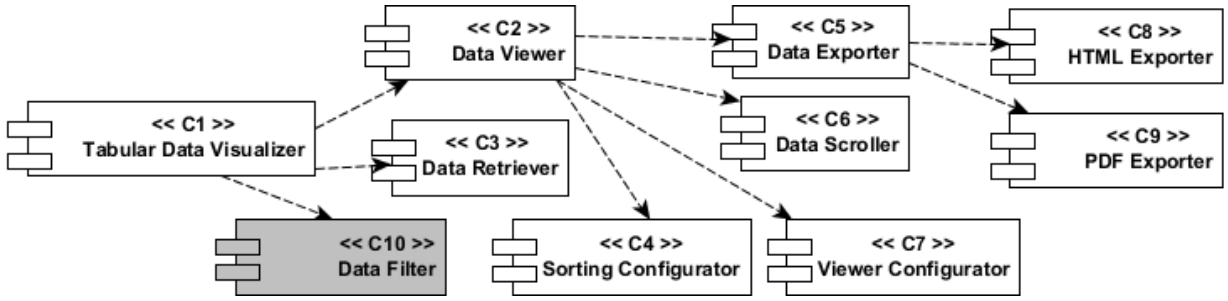

Figure 8.4: UML component diagram

We would like to reuse this component in a different system. However, we need to mold it, as the original system contained just a small number of viewed data, so they always fitted to one screen. On the other hand, the new one contains a big number of data, which do not fit on one screen. By performing a functional analysis, we come to three new use-cases *UC7–UC9* in Fig. 8.3, we dyed them grey. By performing an impact analysis, we come to adding new grey component `Data Filter` in Fig. 8.4. In addition, we need

to change two components `Data Retriever` and `Viewer Configurator`). The resulting effort of molding the tabular data visualiser can be expressed as:

$$E(M) = E(C_{A(C10)}) + E(C_{M(C3)}) + E(C_{M(C7)}). \tag{8.3}$$

We neglected deletion operations in the expression. Let us assume that $E(C_{A(C10)})$ has been computed with the result 10MD, $E(C_{M(C3)}) = 3MD$ and $E(C_{M(C7)}) = 1MD$, thus $E(M) = 14MD$. We can now assess the usability as high, because the effort of molding the component is feasible.

The second example. It discusses the situation when we want to reuse the resulting tabular data visualiser from the first example yet for another system. In this situation, we determine that the functionality is equal, however, the system contains an even bigger number of data, and their retrieving is too slow. Thus, we need to mold the component for this situation. The use-cases do not change, but the non-functional attribute of 'Picking Data' use-cases changes. Based on this change, we need to change construction node `Data Retriever` to become more efficient for big amount of data. The resulting effort is thus:

$$E(M) = E(C_{M(C3)}). \tag{8.4}$$

Let us assume that an expert estimation results in 50MDs. Now the usability is assessed as considerably lower due to low efficiency. We may like to explore alternatives, like using a different component or switching to the Carpenter model.

8.5.1 Discussion

Examples 1 and 2 show two different scenarios of Mosaic model component reusability. Example 1 exhibits a situation with high usability of the tabular data visualiser component: the effort of molding the component is not high, thus the usability is high. On the other hand, in Example 2 the effort of moldability is too high, thus the usability is low.

8.6 Revisiting the Flexibility-Usability Trade-off

Let us now elaborate on the mentioned trade-off between flexibility and usability. We argue that the trade-off holds inevitably just for the Carpenter model. In this situation, it is obvious that greater flexibility leads to a more elaborate need of molding (configuring) the component to be used in the building process. Higher effort means lower efficiency and thus lower usability.

On the other hand, in the Mosaic model, the high flexibility is beneficial, because it increases the chances of effective reuse. At the same time, it may be the case that molding the component for a different system may not be elaborate, it may be even trivial, depending on the F/C impact analysis.

8.7 Related Work

Flexibility and usability in component design is heavily discussed in manufacturing systems domain as the *reconfigurable manufacturing* approach. It is known as a *manufacturing systems paradigm* that aims at achieving cost-effective and rapid changes by designing the manufacturing system and its machines ... to facilitate reconfiguration (e.g., as described by ElMaraghy [72]). The challenges faced are very similar to ours [72].

Naab et al. [152] deal with architectural flexibility in SW lifecycle. They also observe that 'realising flexibility scenarios means to be prepared for conducting changes at a later point in time with limited effort'. Evolvability of SW components (see Chapter 2) seems to be the same class of problem as moldability for reuse, as it poses requirements on flexibility.

8.8 Chapter Summary

In this chapter, we clarified the definitions of flexibility, usability, and their trade-offs in the context of CBS. We distinguished the integrability and moldability view on components integration. Next, we followed the way of moldability as the required model of CBS development. We explained that there are two models of CBS building: the Carpenter and the Mosaic. We formulated a method to assess the effort of moldability of each of the models. We illustrated our approach using two examples.

There are several limitations and open questions. First, the applicability of the method relies heavily on the ability to properly assess efforts. We showed that there is a lot of existing work done, but not entirely tailored for component-based development. Here lies a lot of future work. The second issue lies in the ability to perform a constructional decomposition and F/C impact analysis, which is trivial in small systems like we presented, but complicated in a big, heterogeneous system. This is also a topic for future work.

Next, the 'quantified objectives' that are claimed to be achieved in Definition 12 need attention. Sometimes, some functional requirements are consciously ignored to increase efficiency (i.e., decrease the cost). An extreme situation is the shelf software, where the user is forced into the functionality of the system and not vice versa. In this situation, we may theoretically get to high usability, but this is not the desired state.

The chapter culminates in Section 8.6. Based on the presented method, we argued that the famous flexibility-usability trade-off may not hold for the Mosaic model. This may be a good motivation to employ the Mosaic model in programming practice. It represents a prototype-based programming, where existing solutions are molded into new ones. This idea may be seen in the old Self programming language [188] or today's JavaScript's prototype-based inheritance. However, these are mainly language concepts, while true Mosaic model component development should work with higher-level components. A better illustration of it may be REST service web server implementation, where a new implementation may be achieved by taking the original one from a system and molding it for another system. Today, most of the molding is performed by copy-paste, lacking an engineering rigour.

Chapter 9

RPA Bridge to New Technologies

In Chapter 7, we concluded that systems based on PSI theory might be a great basis for technology transitions. More specifically, we presented so-called confirmation principle. We translated it to concepts introduced by General PSI theory and DEMO. This revealed that if a process where people interact is well-described and independent from GUI, the concrete tasks requiring user interaction may be generated. In that particular case, tasks in a confirmation process. This is another example of F/C separation discussed by BETA theory in Section 3.6. In other words, the resulting tasks could be seen as artefacts from TAO theory introduced in Section 3.2. They are typically designed and created with some affordance in mind to provide corresponding functions. In case of the confirmation principle, the tasks are designed with the function of confirming various financial instruments before they land in the risk management system. However, having these tasks described independently from the GUI technology, their GUI may transit from technology to another. Specifically, while the task's function remains the same, its construction in certain GUI technology may evolve. We only need to properly describe the mapping between them.

Nevertheless, this idea of describing tasks as technology-independent activities done by users is not new. It is heavily practised with the help of BPM introduced in Section 4.5.5. It is focused on generic end-to-end processes. In BPM, the user's tasks are described along other activities in an enterprise. It represents a foundation platform for companies to orchestrate users, data, and systems. Therefore, it often comes with big cross-company transformation project. While it aims for integration of many systems in enterprises, so-called RPA introduced in Section 4.5 rather focuses on task-level automation. Although it is not able to provide a powerful end-to-end business process automation as BPM does, it may be very powerful in its combination.

Therefore, in this part of our research, we want to evaluate how RPA technology may contribute to smoother technology transitions. We focus on routine activities requiring users to interact via GUI of a legacy SW. We want to explore if RPA is able to shield users from using legacy SW by creating tasks independent from a specific GUI technology. RPA may translate these tasks into BPM activities, thereby become a bridge between GUI workflows in legacy SW and technically independent processes in BPM.

This chapter contains a modified excerpt of: *Ivana Nacevska. The evolvability of technologies with the help of Robotic Process Automation.* Published in: *CTU in Prague, Faculty of Information Technology. Prague, Czech Republic, 2020. [A.14]*

9.1 Combining RPA with BPM

In Section 4.5, we provided a general information regarding RPA. Even though, both RPA and BPM seek process automation, these two concepts have different areas of influence. Thus, they are not in conflict. Typically, BPM systems support an integration with RPA tools. In this section, we will investigate how this integration is done by analysing examples. Although in the work of Nacevska [A.14], we elaborated on different BPM vendors, here we only provide the most relevant results.

Triggering RPA workflow in BPM process. We will exemplify our findings on a combination of UiPath that we described in Section 4.5.4, and Camunda that we introduced in Section 4.5.6. The Camunda company itself provides official articles regarding the end-to-end workflow automation. They explain how to get the best out of both tools [230].

The process for automation includes legacy systems without APIs, such as Customer Relationship Management (CRM) and Enterprise Resource Planning (ERP). Since updating the BPMS would take a longer time, they use RPA as a short-term solution.

The Fig. 9.1 depicts BPMN diagram with two so-called *Service tasks*, each triggering RPA process. The diagram shows all steps that RPA robot will execute. The author uses BPMN notation to demonstrate the process visually, even though RPA tools do not use BPMN for modelling workflows. The diagram may give an impression that RPA can complete both CRM and ERP task without the need of having BPMS. However, Winters [230] explains that 'Camunda manages the business process as a whole while RPA handles rote manual tasks within a process that can not be completed programmatically'.

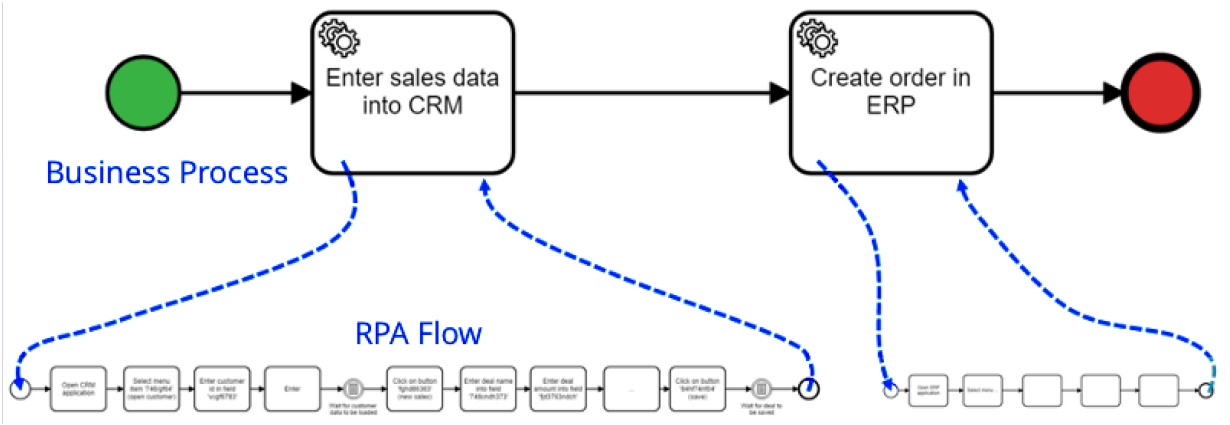

Figure 9.1: RPA as a part of a BPM process [230]

Under this scenario, RPA process is triggered from BPMS. Camunda offers integration with RPA tools via *External task* pattern, which uses REST API for communication.

Triggering BPM process in RPA workflow. From the previous example, it is obvious that BPMSs manage to use RPA functionalities in the process modelling. Integrating RPA

capabilities means that they are able to trigger an automated workflow that has been previously deployed. However, the integration can be also done the other way around. RPA can start the execution of BPM process. If the BPMS provides API for interaction with third-party applications, RPA can define an activity that will trigger the business process. In this way, RPA is able to deal with all exceptions that might arise, as well as gaining the ability to handle activities that need user confirmation.

Combining RPA workflow with BPM process. The Previous examples show that both RPA and BPM systems can trigger a process/workflow in the other one. BPMS can replace rule-based user tasks with robots, while RPA can handle exceptions by exporting tasks to BPMS. Since both ways of interaction are possible, we would like to further explore the idea of combining RPA and BPM processes. For this, we will use a collaboration diagram to model a process that will use both systems. While RPA will be seen as the main automation system, BPMS will be used only for activities that require human decision-making capabilities. Integrating both systems provides a nice workaround for companies to overcome the limitations of both approaches of automation.

The RPA workflow and BPM process notation typically differ. Consequently, the representation of the RPA process shown in Fig. 9.2 is not correct. Despite that, we are using BPMN so that we can present the RPA process visually, rather than mapping it as 'black box'. The process shown in the diagram is a typical invoice processing done in accounting. Each step of the work is mapped to an appropriate activity representing a separate task. Most of the activities are rule-based and executed in the same way, therefore, can be executed by RPA robot. The one that involves working with more sensitive data or making decisions that cannot be done by robot, will be managed as user tasks in BPMS.

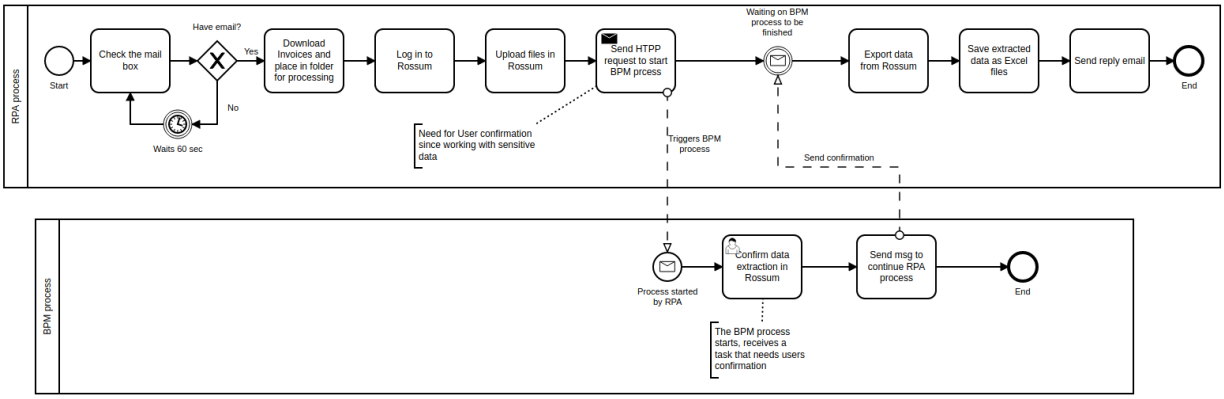

Figure 9.2: Combining RPA and BPM process

The idea of using RPA as main automation system is a bit naive, having in mind that throughout the thesis of Nacevska [A.14], we stressed out that RPA should not be used for end-to-end business processing. Rather, it should focus on single task automation. However, this is a perspective that we investigated further. Moreover, in such a combined workflow, this would move the entire user interaction to BPMS and may build a basis

for mining end-to-end business processes. As a result, the organisation could step-by-step determine the technically independent business process, execute it, and either replace the entire RPA, or still trigger it from BPMS. The implementation of this idea will be presented in the next section.

9.2 The Case Study

In this chapter, we will investigate the scenario of combining RPA with BPM as outlined in Section 9.1. First, we will demonstrate typical administrative process in the area of finance. We will explain each step of the process that is executed manually by the worker. Next, we will implement this process using pure RPA, and later in a combination with BPM. Together with Nacevska [A.14], we assessed this implementation using RPA tools UiPath and Microsoft Power Automate. However, for the scope of this thesis, we only outline the approach with UiPath and we explain its combination with Corima BPM subsystem. Further details can be found in the work of Nacevska [A.14].

Many market studies suggest that RPA is suitable for processing invoices and filling in data tables in Excel. Having this in mind, we map one end-to-end process involving a range of applications. This helps us to understand RPA automation capabilities. The process involves receiving invoices, extracting data from files, generating Excel files from it, and sending back a confirmation. Let us call the undermentioned process 'invoice processing'. Broken down into smaller and simpler steps, it looks as follows:

1. Gets an email with attached invoices (in .pdf format) that must be processed
2. Downloads all invoices
3. Place them in a folder 'Invoices for processing' contains all unprocessed invoices
4. Signs in to software used for keeping track of all invoices and customers
5. Uploads all .pdf files for processing
6. Starts reviewing and extracts necessary data
7. Confirm extraction of important data fields
8. Writes data in Excel sheets from the the accounting book
9. Moves files that are processed in 'Processed invoices' folder
10. Writes reply email (that the invoices were processed)

The RPA processes must be always executed in exactly the same way. Therefore, to automate the invoice processing using RPA tool such as UiPath, some changes need to be applied. To be able to simulate the process from the beginning to end, we slightly adapted it to the scope of this dissertation thesis. Therefore, web-based application was used to

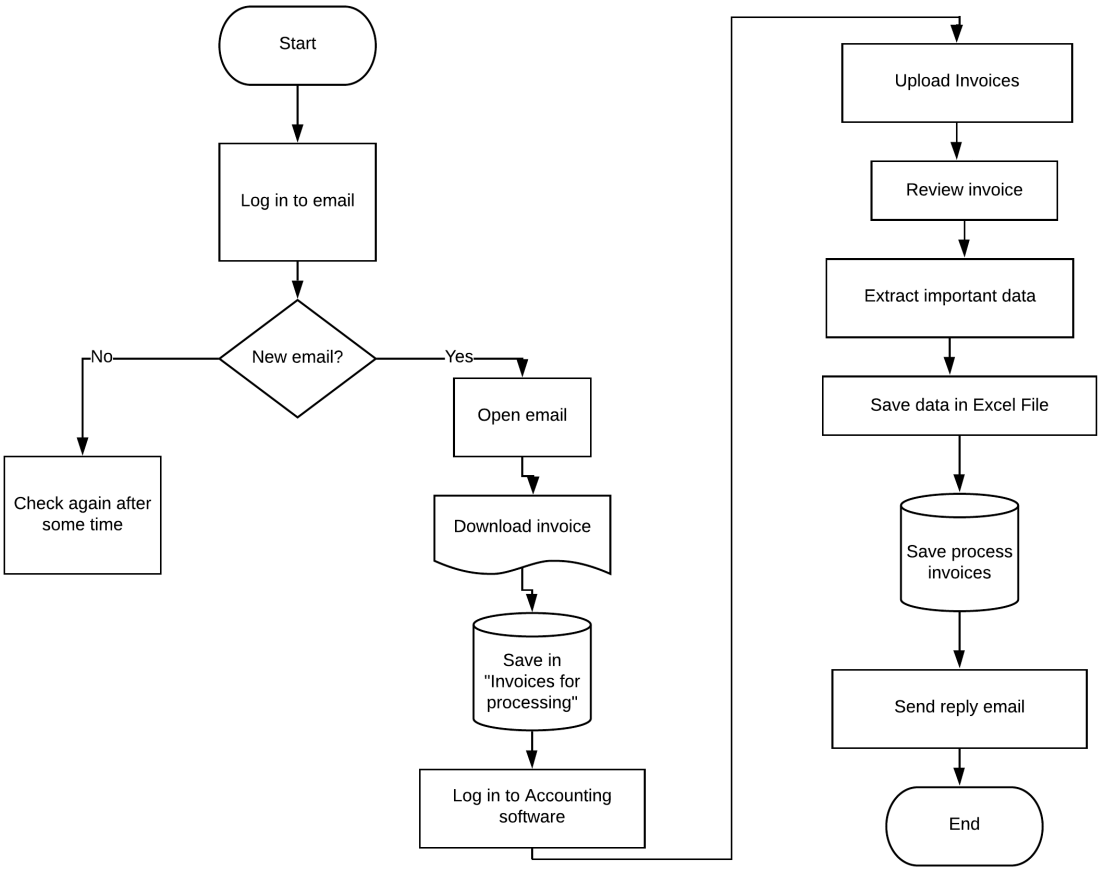

Figure 9.3: Flowchart diagram representation of the case study

simulate professional accounting software. Free online web application *Invoice Simple* was used to generate invoices. This allows us to register some clients, set parameters, generate invoices, and email them to customers [194]. Here, we do not present all details regarding this invoice generator since it is unimportant in the context of process automation.

Rossum is another third-party software we incorporate. It is a cloud-based application using AI to extract data from the invoices [181]. Rossum provides many alternatives to import invoices. It can be done either directly from email, or with a simple upload of scans and pictures. We are using Rossum API in order to upload the invoices from UiPath.

9.2.1 Automating the Invoice Processing with UiPath

As mentioned at the beginning of this section, the process automation will be achieved by using UiPath, and by combining it with BPM. Now, we present UiPath automation. Since this implementation is described in detail by Nacevska [A.14], we only simplify it here.

After defining the steps of the invoice processing clearly, the automation in UiPath is rather straightforward. The whole process executed by UiPath robot is mapped to flowchart layout diagram depicted in Fig. 9.4.

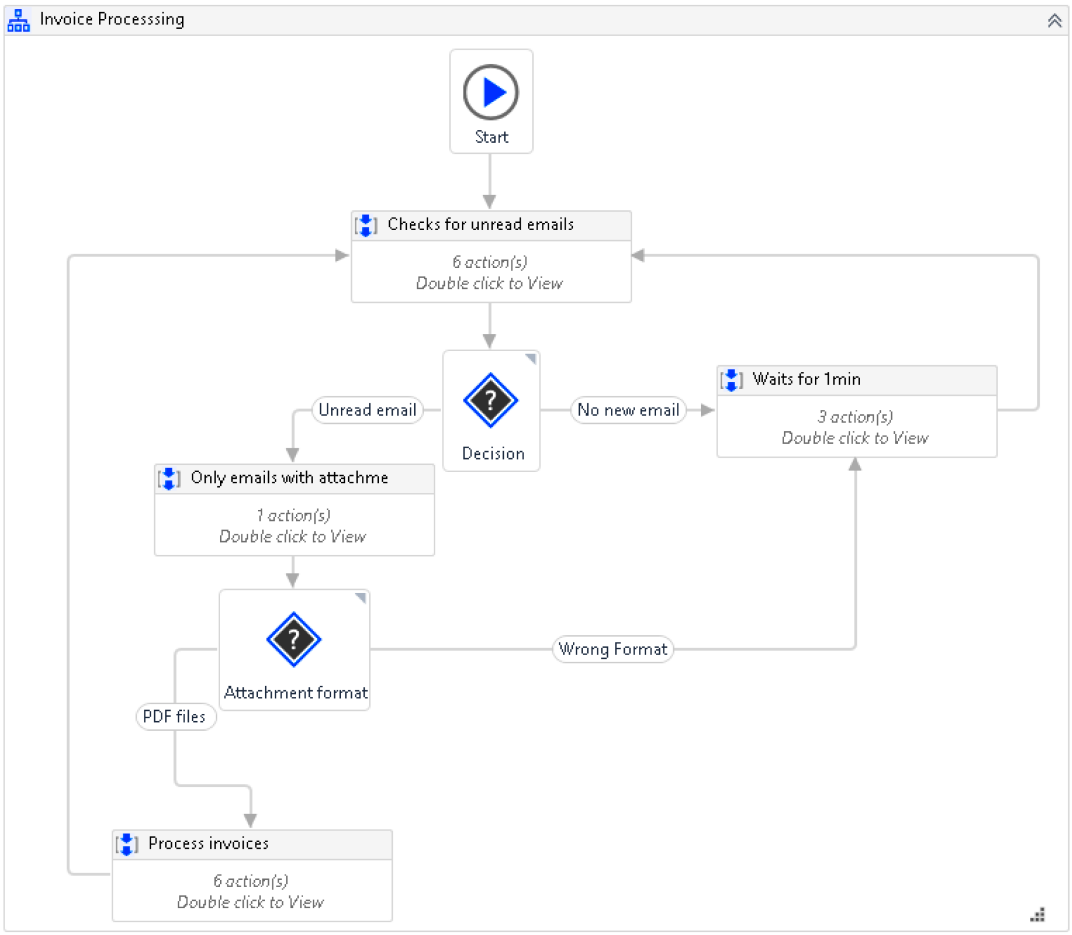

Figure 9.4: The case study implementation in UiPath studio

Every 60 seconds, the robot checks new incoming emails. If there is no new email, the process waits 60 seconds and then returns to the beginning. Otherwise, the email attachment is inspected. If it is in a required format, it is processed further. Otherwise, the process again listens to a new email.

In the thesis of Nacevska [A.14], we explained the entire automation of invoice processing in depth. We tackled the implementation details of how UiPath fetches emails, generates invoices, interacts with external services such as Rossum, etc. However, in the research scope nailed in Section 1.5.1, and in the corresponding research objective RO 2.3, we only want to understand how UiPath supports user interactions. Especially, how these interactions may be done in different technologies. Therefore, we skip all other details and we solely focus on the aspect of user interaction.

In the invoice processing example, the user interaction is required to confirm an invoice processed by Rossum that extracts the structured data from the pdf. This confirmation may be solved in UiPath using either the *Recording option* or using GUI activities such as

clicking, typing, and capturing elements from the GUI design. The Recording option is an important part of UiPath. This functionality enables to easily capture the user's actions on the screen and translates them into sequences. With its help, the user interactions are captured and UiPath automatically maps each step to a suitable activity. In our case, we used only *Message Box* activity that will display a message to remind the user that the files are already uploaded and the extraction of the data, which is done automatically by Rossum, needs to be confirmed. This demonstrates that UiPath supports tasks requiring a certain user's approval. However, they may still be tight to the specific GUI captured when recording that user's interaction in UiPath.

9.2.2 Automating the Invoice Processing with UiPath and Corima BPM

In this section, we will explore the idea of combining RPA with BPM in practice. We will present a prototype that integrates UiPath and a BPM subsystem of Corima.

UiPath has API based on OData protocol[1] and allows GET, POST, PUT, and DELETE request. Similarly, Corima provides APIs enabling us to start a BPM process. The initial idea is to automate the RPA process presented in Fig. 9.4 and to modify it so that it can be combined with BPMS. Instead of having a Message Log activity that pops up to notify the user to confirm the sensitive data in Rossum, we want to activate BPM process where a task to confirm the data will be created.

After the user completes the task, the BPM process notifies RPA to continue. However, this cannot be done in such a way. The tendency of RPA is to automate the process from the beginning to end without any interruption. All user interactions are handled by Message log activities. Because of these reasons, we divide the initial process mapped in UiPath to two different fully automated workflows – the workflow to check emails, and the workflow of exporting data to Rossum.

The first workflow is depicted in Fig. 9.5. It is executed by the unattended robot described in Section 4.5.3. It is constantly running and checking for new emails to be processed. After going over the emails, sorting them in files and uploading them for processing, UiPath starts BPM process in Corima. It executes a sequence 'Trigger BPM process' and continues to check for new emails. By handing over the user activity to Corima, UiPath can execute the process uninterruptedly, while still having handled data-sensitive activities with more attention. This increases the flexibility of the solution.

The Fig. 9.6 shows a part of the process that was previously mapped with *collaboration diagram* in Fig. 9.2. It is designed and executed in the BPM subsystem of Corima. When the BPM process is triggered by UiPath, the corresponding task is created in Corima. In this particular case, the task is represented in a module presenting the extracted information from the invoice such as account number, amount to pay, due date, etc. After

[1]OData (Open Data Protocol) is an ISO/IEC approved, OASIS standard that defines a set of best practices for building and consuming RESTful APIs, which allow resources, identified using Uniform Resource Locators (URLs) and defined in a data model, to be published and edited by web clients using simple Hypertext Transfer Protocol (HTTP) messages. [156]

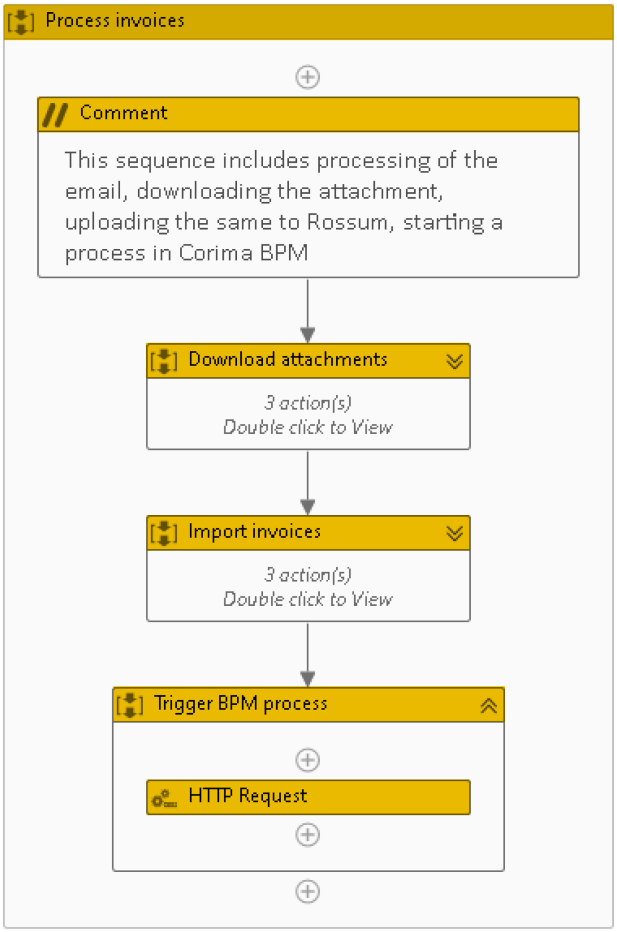

Figure 9.5: Uploading invoices to Rossum

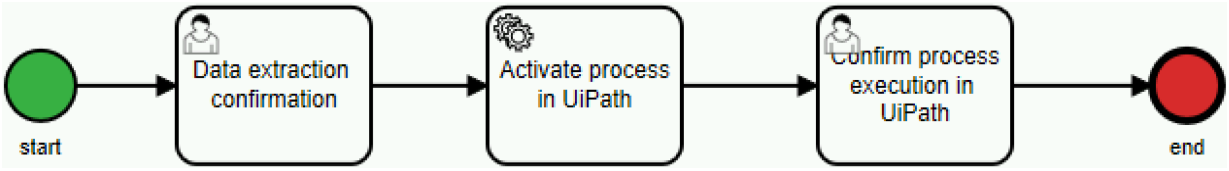

Figure 9.6: BPM process in Corima

the invoice is confirmed in Corima, the BPM subsystem in Corima will move to a next activity, a Service Task. Specifically, it calls an external service to perform another action. As in this case, it triggers a job in UiPath Orchestrator that will write the data into an accounting book, or so.

9.3 Chapter Summary

In this chapter, we presented a typical use-case in the area of accounting. We tried to automate the invoice processing process by using UiPath and later by its combination with BPM. UiPath offers many possibilities for automating tasks, even when working with third-party software for which they do not have connectors, as in the case with Rossum. The mapping of the process with UiPath workflows can be sometimes overwhelming. Even a rather simple task requires many activities.

Next, we integrated UiPath and Corima BPM subsystem to support the same invoice processing process. The goal was to export the tasks that require user attention to BPMS, rather than to use Message windows in UiPath. Since, both UiPath and Corima provide APIs for communication with third-party software, their integration was rather smooth.

UiPath, or better said RPA seems to be a nice complement to BPMS. However, it should not be seen as the main system for managing business processes. By combining RPA with BPMS, the business processes can be improved. BPM can handle exceptions that might come along in RPA process, as well as activities that require human approval, especially when working with sensitive data. In this way, RPA processes will not be interrupted, while all exceptions will be nicely dealt with. On the other hand, RPA can be used to cover all the rule-based, boring User tasks and can serve as API for legacy systems without APIs or web services. So instead of changing the existing system to implement those functionalities, BPM can use RPA as a solution.

Finally, this chapter helped us to address the research objective RO 2.3. It turned out that RPA might be extremely helpful in digitalisation. Recently, Kirchmer et al. [123] published a paper on topic 'Value-Driven RPA'. They discussed opportunities and challenges of applying RPA as process improvement approach. They mentioned that: 'Value-driven RPA is a part of a discipline of process-led digital transformation management, leveraging the capabilities of BPM to realise the full value of digital initiatives, fast and at minimal risk' [123]. We additionally revealed that not just digitalisation, RPA may represent a bridge to BPM activities that are described in a technology-independent BPMN model. The user activities that are present in these BPMN models are typically mapped by BPMS into a specific GUI. Therefore, if the BPMS can map them into different GUI technology, the move to a new one may be easier. Thus, RPA itself does not seem to directly solve the problem of technology transitions, yet it may contribute it.

Chapter 10

Evolvability of Financial Models

As explained in Chapter 2, evolvability is a characteristic dealing with changes in ISs. In time, the complexity of the system may increase with changing requirements. In turn, the ability to change it decreases. Consequently, the cost of a change can become unbearable. A domain model is an important abstraction covering key aspects of ISs. Similarly to IS it represents, it can suffer from the same evolvability issues.

The scope of our research is a financial industry. At the same time, our research goal is to design and develop a new methodical framework that aids in the construction of software solutions enabling controlled technology transition. Technology transitions often come with CEs defined in Definition 1. Therefore in 2017, we assessed CEs in finance. In conjunction with the University of Antwerp, we created a domain model of a financial risk management domain. It was settled thanks to the experience of the authors from Belfius bank, and the knowledge of COPS in banking area. On Enterprise Engineering Working Conference in Antwerp, with Deryck et al. [A.3], we published a joined paper. Its adjusted version is provided in this chapter. It reveals difficulties related to identifying CEs in domain models in general and presents some insights on the nature of CEs on this level. By this, it also helps us to address RO 2.4.

10.1 Introduction

As introduced in Section 2.2, NST proposes a systematic methodology for a modular design with the objective of creating evolvable systems [137]. Its applicability as a theory for evolvable modular software systems has been proven by the development of critical software systems for multiple organisations [109]. The use of the theory in the broader scope of EE has been demonstrated by the research performed by De Bruyn, Huysmans and Van Nuffel [214]; [107]; [51]. However, the successful identification of CEs in some EE instruments does not guarantee the general applicability. CEs typically emerge at a very low and fine-grained level. Aggregating this in higher-level abstractions may hide the underlying impacts.

This chapter contains a modified version of: *Marjolein Deryck, Ondřej Dvořák, Peter De Bruyn, and Jan Verelst. Investigating the evolvability of financial domain models.* Published in: *EEWC. Springer, Antwerp, Belgium, 2017. [A.3]*

Moreover, the application of NST to specific industries only took off recently, e.g., as researched by Vanhoof [216]. Its mission is to demonstrate the factors that may hamper evolvable modular design. Such an analysis has not been done yet on the level of domain models, neither in the financial industry.

Thus, the purpose of this chapter is to investigate the evolvability of domain models in the finance industry. We will focus on the sub-domain of market risk management that is expected to be subject to regulatory changes in the coming years. Due to the importance of domain models in software development in general, the main focus is on analysing the corresponding reference models using NST. We present typical change requests, and we show their possible implementation. By doing so, CE inherent in the models are uncovered and described. The presence (or absence) of these effects indicates how hard (or not) it is for companies to implement changes within a reasonable time frame. Ergo, CE in this case may point to an increased risk for regulatory penalties if short-term changes are imposed.

In Section 10.2, we elaborate in general on what is meant by *a change*, and what might be its consequences. Next, in Section 10.3, we deeply introduce a finance domain model and we outline its possible changes. In Section 10.4 we revisit evolvability in domain models. We present related work in Section 10.5, and we conclude and summarise the chapter in Section 10.6.

10.2 Evolvability

In Chapter 2, we mentioned that the complexity of ISs increases due to new functional requirements. However, in itself the requirement does not incline the increase of complexity. Rather, the corresponding changes effect it. Thus, in this section, we will clarify what it is meant by a change.

Clearly, ISs can be changed at various levels. On the level of its source code, we usually refactor, optimise, add, or delete certain code constructs, e.g., functions in SP [47], or classes in OOP [195]. On the level of a database, we alter, drop, or create new database objects, e.g., tables, triggers, and views in relational databases [36]. The modification of a software configuration is a change as well. Moreover, several cloud computing services offer a scalability option. For example, Microsoft Azure platform can adapt the system to an unexpected amount of workload by increasing or decreasing resources for an application [225]. Therefore, by introducing a new functional requirement 'adopt to a workload automatically', we do not change the system itself at all. Yet the changes in the surrounding environment can affect it significantly.

Thus, in any kind of system, the formalisation of what a change means is crucial. NST formalises it by a term *task* as a subject to an independent change [137]. In SP, such a task is represented by a function. In OOP, a method plays that role. A number of code lines usually implements the given function, respective method. These can be logically grouped into a sub-function, respective a sub-method, to signify they belong to a different change driver. Therefore, similarly to SP, or OOP, we have to formalise a change in the area of domain models, e.g., in the area of finance domain model.

10.3 Finance Domain Model

This chapter covers the domain of risk management in financial institutions. Risk management has always been an essential activity of the banking sector, and since the financial crisis in 2008, it is under even closer scrutiny of local and international regulators [185]. Furthermore, banks themselves seek to optimise the internal models they use to calculate the regulatory capital, to avoid losses and capital punishment (the so-called plus-factor in case the actual loss exceeds the loss predicted by the internal model more than five times in one year) [17].

The Basel regulations discern three types of risk in the financial sector: credit risk (i.e., the risk that a counterparty will not honor his obligations), market risk (i.e., the negative financial impact of changing market conditions), and operational risk (e.g., fraud, settlement risk, etc.) [15]. Each of these risks cover multiple risk factors. For example, some factors that contribute to the market risk are changes in interest rates, share prices, commodity prices, inflation, FX rates, volatility and credit spread.

The scope of this chapter is the measurement of market risk using Value-at-Risk (VaR). The choice for this scope emanates first from the fact that this instrument shows clear disadvantages (i.e., lack of sub-additivity) and has been said to have played a significant role in the 2008 financial crisis. Second, eight out of the ten largest Belgian banks report the use of VaR in their annual (risk) report and the measure is accepted to calculate the regulatory capital for market risk. Therefore, VaR might continue to play an important role in market risk management, but it will probably be subject to changes in the years to come.

The VaR is a single currency amount that reflects the maximal loss that is expected in the given time period. Regulators require at least 99% confidence on a ten day period, so a 10day VaR(99%) of 500k means that the bank is 99% certain that the loss on the considered portfolio over the next 10 days will not exceed 500k. Note that VaR does not give any indication of the amplitude of the loss in case it is exceeded. The expected shortfall, the calculation of which will be mandatory as from 2018, is adequate to that end [16].

10.3.1 Establishment of the Domain Model

The focus of this chapter is a domain model of the market risk domain extended with a focus on market data import and trade repository. The model does not represent the situation in a single case company, but rather constitutes a realistic representation of common parts, based on the experience of the authors in multiple cases. The advantages of this approach are twofold. On the one hand, this generalisation allows the abstraction of company-specific implementations that are not only the result of business requirements, but also of the company history, its specific systems and the quality of its implementation decisions. Even though the importance of these factors is recognised, they are not relevant in the light of this chapter that aims to demonstrate the identification of CEs in reference models. On the other hand, a real and detailed data model of a single case would require

extensive access to the company's IS architecture, which might even not be readily available in the company.

10.3.2 Overview of the Domain Model

The domain model depicted in Fig. 10.1 abstracts the overall finance model. It displays three large parts. Situated in the upper left part (in gray) is a part related to the import of market data from an external market data supplier. In the lower right part (with black cubes) a trade repository for foreign exchange and interest rate trades is depicted. The part in between relates to the calculation of VaR following the historical method. The paragraph below explains how these blocks fit together in the process to calculate the VaR.

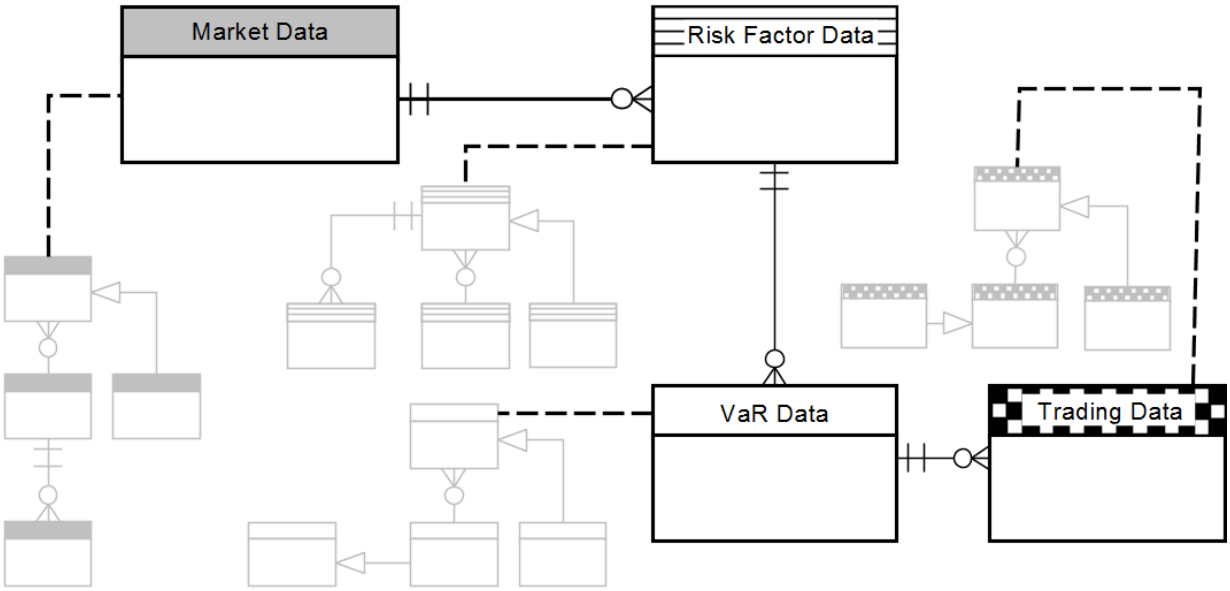

Figure 10.1: Abstraction of a finance domain model

10.3.3 Business Process Introduction

Fig. 10.2 schematically represents the VaR-calculation process with the use of the historical method. In this method, the historical changes of the risk factors that have been observed during the last x days (often 300 to 500 days) are applied to the current trade portfolio. The process starts with the upload of relevant market information from external market data suppliers, such as Bloomberg, Reuters or others. The bank needs to specify which data should be downloaded at which moment. This is done by so-called schedulers. Certainly, also corresponding data entities to store the information are needed. In the next step, the shift from one day to the other is calculated. This is nothing more than calculating the difference between yesterday's and today's value for, let's say, the 300 last days. Afterwards,

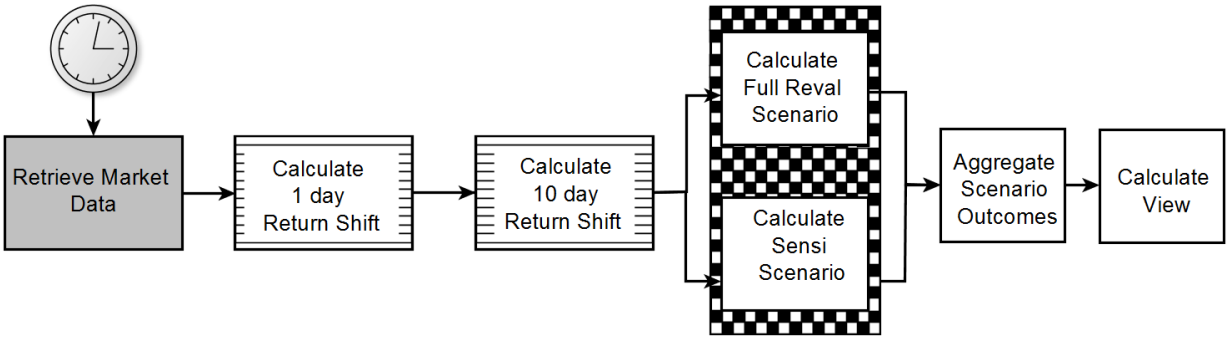

Figure 10.2: High-level overview of the HVaR-proces

the one-day shifts are scaled up with the factor $\sqrt{10}$ to obtain the ten-day shifts necessary for the calculation of the ten-day VaR. In the full reval scenario, the outstanding positions are valued against the ten-day shifts. It means that for each outstanding position, 300 possible profit and loss scenarios are calculated. The method is very simple, but it is heavy on calculating resources and available market data. For some deals with heavy pricing models, it can be beneficial to calculate a proxy. This can be done with the use of sensitivities, e.g., delta, which reflects the change in value of a derivative when the value of the underlying changes. The sensitivities are used in the calculation of the profit and loss scenarios. Their VaR-calculations have their own parameters, including the alpha that indicates the certainty level. In Fig. 10.2 the method is represented by the rectangle below the full reval scenario- rectangle. They are situated in front of a background with black cubes that represent the trades in the trade repository. The resulting outcomes of both the so-called *full reval* and *sensi method* need to be aggregated with the purpose of obtaining a single VaR-number in the end. As VaR is not sub-additive, this aggregation needs to follow strict business rules determining the appropriate calculations for the appropriate positions. Afterwards, the possible aggregated profit and loss outcomes are sorted from the most negative (i.e., loss) to the most positive. Based on the desired certainty level (usually 99%) the appropriate cut-off value corresponding to the alpha is selected as the VaR.

This section offered a high-level overview of the general VaR-process. The individual process phases are covered by the corresponding paragraphs in Section 10.4.

10.4 Revisiting Evolvability of Domain Models

The models described in Section 10.3 breaks down the VaR-calculation in blocks and classes needed to execute it. The different parts are linked with each other through interfaces. In short, the domain model exhibits a modular structure. As explained above, the specific scope of the model was chosen because of the expected regulatory changes in this domain. The characteristics of change and modularity are exactly two fundamental concepts in NST. Therefore, in the section below, we investigate the applicability of the theory on the

VaR-domain model by applying some changes as defined below:

Definition 15. [Changes to ISs are] (1) the addition of new requirements; (2) the modification of existing requirements; and 3) the obsolescence of existing requirements ([137], p. 258).

This results in changes in the domain model, e.g., adding or renaming an attribute, adding a relation, changing cardinality, etc. More specifically, in this case, the appended changes are the addition of equity market data, a new version of 10day VaR calculation, amendment of the alpha, and the addition of a new product.

10.4.1 Revealing Combinatorial Effects

The proposed domain model is limited to the follow-up of the foreign exchange risk factor and the interest rate risk factor. The choice of the risk factors included in the model depends on the nature of the business conducted by the bank. However, these two are most commonly measured by the VaR. In this section, we will introduce a few changes to the model to investigate how the model reacts.

Addition of the equity risk factor. The first change is the addition of a new risk factor, e.g., the equity risk factor. This means that the share prices need to be uploaded in the system. To this end, a new data-entity *Share* needs to be introduced. Usually, this type of market data, along with information on bonds, futures, and funds, will inherit some general attributes from an *instrument* class. To induce the upload of share prices from an external market data supplier, the schedulers that start the retrieval of the data need to be amended. This means the impact of the five schedulers that are currently identified. Moreover, the number of schedulers is not cast in the stone itself. It is thinkable that new schedulers, e.g., quarterly or biyearly schedules, need to be introduced. That means that introducing a new data entity is not only dependent on the size of the system, but also grows along with the growth of the system. This demonstrates that the addition of a new data entity of equity risk factor leads to CEs. Fig. 10.3 schematically shows the discussed changes.

Going further down the process, the market data are translated into Risk Factor Data (RFD). In the drafted domain model, this is represented by a single RFD – entity. In fact, this hides two possible solutions – *generic entity*, or *multiversion entity*.

The generic entity is configured in a way that it can include all necessary details on FX, interest rate, and equity risk. It means that in this case a data structure with superfluous data fields are sent as an input for the return shifts. This is an example of stamp coupling. Even though at first sight it may be tempting to tolerate this kind of coupling at the start, the risks associated with this structure are:

1. The overly large data structure using an extravagant amount of resources.

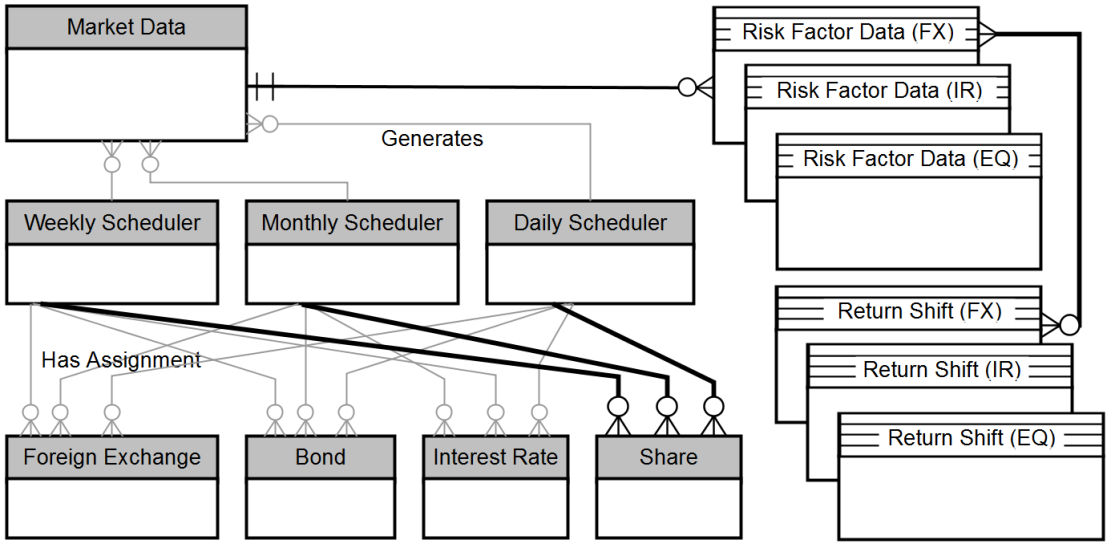

Figure 10.3: Abstraction of a market data sub-model

2. If one of the attributes changes or an additional attribute needs to be included, additional versions of the data-structure will need to be created. This leads to CEs if updates need to be done on the different versions.

The multiversion entity solution helps to avoid this kind of coupling. We can create new versions of this RFD for each product. This would mean that at first there exists RFD for FX, and RFD for Interest Rate (IR). We denote them RFD(FX) and RFD(IR) respectively. Upon the addition of the EQuity (EQ) risk factor, a new version, RFD(EQ), needs to be created. Going forward, when using separate versions of risk factor data, this logically leads to different versions of return shifts. An additional return shift would thus need to be created for equity risk. Fig. 10.3 depicts the multiversion entity solution of RFD.

This shows that the addition of a new risk factor leads to multiple amendments in the system. To ascertain that these amendments are truly CEs, the number of amendments should even increase with the growth of the system. This is the case, which is demonstrated by the example below.

Amendment of 10day VaR calculation method. In the current way of working, we recognise two return shifts for each risk factor, i.e., the one-day shift that is scaled to the ten-day return shift. However, this kind of scaling will probably not be allowed anymore in new risk models. Regulators ask for a full calculation of the ten day VaR, and the adaptation of the existing domain models is unavoidable. This means that the return shifts in the new system will represent only one-day changes. Therefore, next to the one-day return shifts, a new data entity to capture ten-day changes must be introduced. Such an adaptation must cover FX risk, IR risk, and EQ risk. Conversely, if a new risk factor is added at this point in time, as before, the risk factor data and the one-day return

shift need to be created. Furthermore, a 10-day return shift and possibly even more return shifts will be affected. This may happen if regulators estimate that the liquidity on the market has structurally changed and, e.g., 1-month VaRss are necessary. If a new risk factor is added at this point in time, it requires more changes than the ones described for the addition of the EQ risk factor.

The amendment of calculation methodology for +1-day VaRs shows another CE when a full calculation (in contrast to the scaling calculation) is used. These changes are represented in Fig. 10.4.

At a basic level, the difference between two consecutive days can either be absolute or relative. In the scaling method, this difference can be multiplied with the square root of the number of days, both for the 1-day calculation (as $\sqrt{1} = 1$), as for the 10-day calculation (with $\delta_{10day} = \delta_{1day} * \sqrt{10}$). However, if a full calculation of a multiple day VaR is mandatory, this would mean that separate formulas for one-day and ten-day variations need to be created.

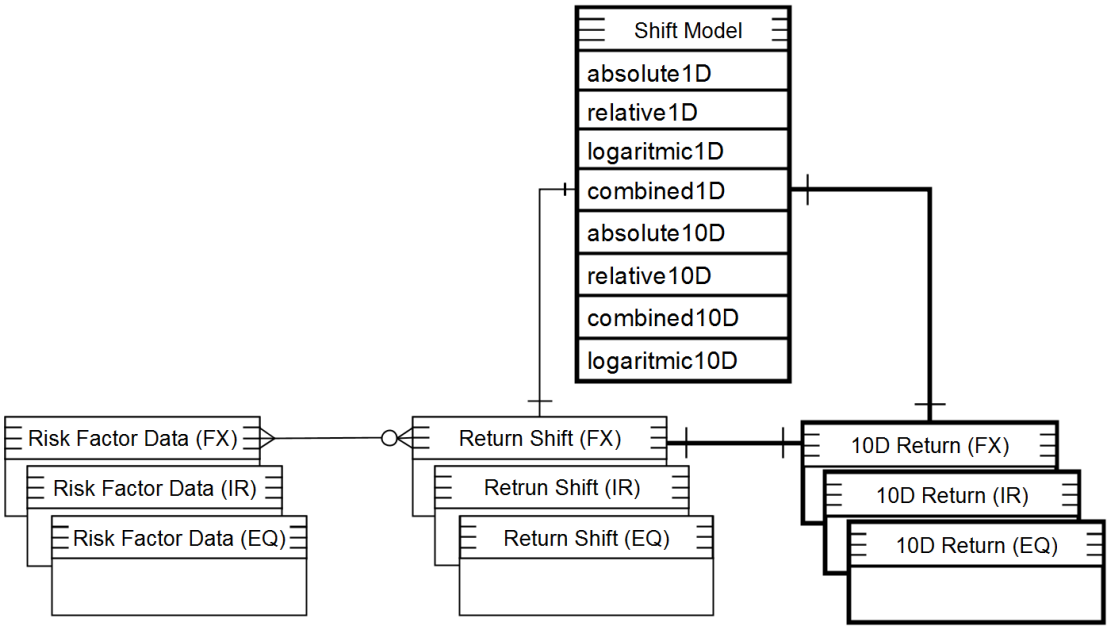

Figure 10.4: Abstraction of return shift calculation sub-model

Unfortunately, this is not the end of the story. Again, the impact of another VaR horizon does not always limit itself to the amendment of the two calculation methods. Although there are not an unlimited amount of possibilities, the number of calculation methods itself might increase as well. This consciousness emerged recently, with prolonged low interest rates as an example. If the interest rate is at 0.01% at day 1, and rises to 0.02% at day 2, this is a relative change of +100%, and an absolute change of 0.01. Whereas the relative change exaggerates the impact, the absolute change would not show any differentiating power. A combination of relative and absolute elements in (one or more) 'mixed' calculation methods could be implemented to remediate this. The introduction of logar-

ithmic calculations could be envisaged as well. If these different methods are implemented, it means that a change in VaR horizon would be needed for each of them, hence demonstrating the definition of a CE.

Addition of a new product. Another example is a change in the certainly level of VaR, which is the second important characteristic next to the time period under consideration. Currently, the required alpha for the 10day VaR is 1% maximum. However, if longer time-horizons are envisaged or in combination with other risk measures, regulators might be satisfied with a 2.5% alpha. Or conversely, they might require a higher level of certainty by lowering alpha to 0.5% maximum. In the current domain model, this would mean that the alpha needs to be amended at two places: one time in the VaR-parameters (necessary for the full reval method), and one time for the calculation based on sensitivities (in the analysis parameters). The choice between the two methods is implemented as a business rule at the full reval ccenario versus sensi scenario level.

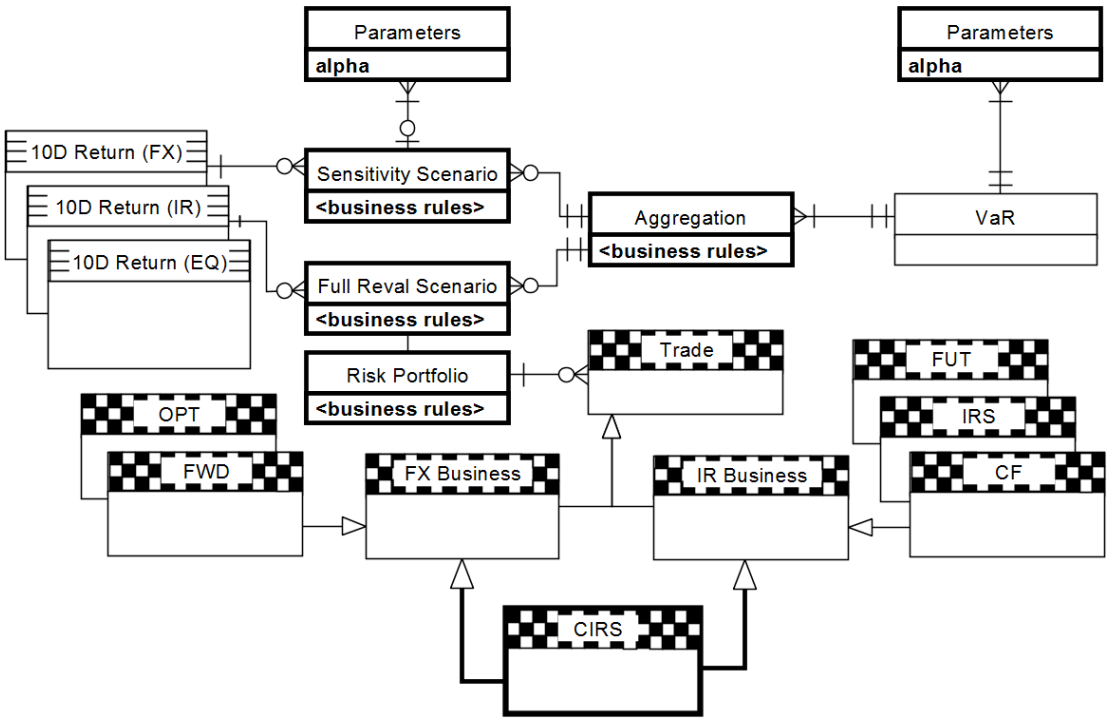

Figure 10.5: Abstraction of model with new product type and changed alpha.

Now imagine yet another kind of change, e.g., the creation of a new type of product. In the current model only the clear-cut instruments from a risk factor point of view were described: two FX instruments (option and forward) and three interest rate instruments (swap, capfloor and future).

Let us see what happens if we add Currency Interest Rate Swap (CIRS) to the portfolio. As the name suggests, this instrument contains both characteristics of the interest rate

business (e.g., the respective IRs of the currencies) and of the foreign exchange business (FX rate between the two legs). Therefore, a new data entity will inherit some attributes from both businesses. Furthermore, traders will need to be able to price this new instrument, so a new valuation model will be developed. This model will contain parts of IR-models and FX-models. As the CIRS contains IR-characteristics, the most probable scenario is the full reval Scenario, which means that the business rule that determines the calculation scenario (full reval or sensi) needs to be updated. Moreover, in this scenario a new Risk Portfolio needs to be created. This results in the adaptation of *aggregation scenario*, as it defines aggregation rules per portfolio. Again, as with the previous examples, it is the reaction of these changes to changes on another dimension.This clarifies whether these are true CEs or effects due to the original change. Imagine for example that the expression of change in foreign exchange rates does not happen in percentages anymore, but in pips (0.0001). It means that the original FX valuation models will need to be adapted. However, the CIRS model, that contains parts of a FX model, will be impacted as well. This applies to every new valuation model annexed to a new product. On the level of the business rules in the full reval / sensi scenario, the straightforward structure (with only one business rule) gets increasingly complicated each time a new product is introduced.

10.4.2 Insights in Exploring the Domain Model

NST aims to promote evolvability in large and complex systems. One of its key concepts is CE. In analyzing the domain model, some insights on key characteristics of CE stemming from their purpose were provoked. To start with, CE are found in complex systems. Complex systems deal with entities that are prone to changes in different dimensions. Because of this, it seems difficult at first sight to uncover the CEs in a domain model. Both knowledge on the domain to recognise the different change dimensions as insight on the reasoning to reveal CEs are necessary. In turn, this implies that the use of CEs to analyse a small system that mainly consist of uni-dimensional constructs is less fit. Next, CEs appear at a very basic level of analysis. Models, which are essentially abstractions of reality, may thus hide this complexity. This was shown in the examples above in which we identified CEs on the level of business rules and attributes.

10.5 Related Work

Originally, NST defines CEs on the software level. In subsequent research, the concepts and theorems of the theory proved useful to evaluate evolvability on the enterprise level and in process models [51, 107, 214]. Eesaar [71, 70] focuses on the database level of ISs, and tackles the common ground between database normalisation and NST. His research concludes that even though both theories overlap at some extent in their goals (avoid update anomalies), manner (multistep nonloss-decomposition), and results (an increased number of smaller tables/modules), NST covers a broader field with its focus on CEs [70].

Domain models are frequently used as the input for the creation of data models, and they often display the desired normalisation form of the final data model.

In line with the conclusion of Eesaar [71] the current chapter illustrates that normalisation is a first, but by no means sufficient, step to avoid CEs.

The related work in the area of risk management does not seem to provide relevant research on evolvability and implementation. Implementation in this domain focuses mainly on the choice of unbiased parameters and financial calculations, which is not the scope of our research.

10.6 Chapter Summary

This chapter started by establishing the importance of evolvability for current ISs. ISs are built to fulfil certain requirements, but these requirements tend to change over time. NST proposes four design theorems to create evolvable software, defined as software free from CEs. CE on this level is the impact from a change that does not only depend on the change itself, but also depends on the size of the system. CE can also be identified on the level of process models, and guidelines exist to avoid them at this level [214].

However, when applying the concept of CE to business models, we need to take into account that a high level of abstraction may hide CE present at lower levels. Therefore, the research for CE typically needs to happen on a low level of aggregation. The current chapter investigates whether the concept of CEs can be applied on the level of domain models, which represent an abstraction of the ISs. And if this is the case, what can be concluded with regard to the evolvability of domain models. By investigating a partial domain model of financial risk management, it is demonstrated that CEs do exist in this domain. Earlier research revealed the existence of CEs in, e.g., accounting systems, education programs, and ERP-systems [216, 162, 31, 52]. It supports the belief that CEs exist in more, if not all, economic sectors and types of ISs.

It also demonstrates that the application of the concept of CE may not always be as straightforward as one might be led to believe by the simplicity of its definition. In practice, it is not always clear which effects are 'proprietary' effects of the change itself, and which ones are due to the size of the system. The reasoning build-up in the examples of this chapter demonstrates a possible way to proceed. It leans on the definition of a CE as being dependent on the size of the system, hence, its amplitude needs to grow when the system grows. The system grows by adding changes of another dimension. Here, it is demonstrated by considering different scenarios for one change. This way of working emphasises the multidimensional nature of CE. Hence, it also contributes to the understanding of the difference with database normalisation, which removes some data redundancy but in a one-dimensional way. In fact, only the CE emanating from the definition of the alpha at two different places would be solved by simply applying database normalisation rules and isolating it in a new entity.

The changes that were applied to the original domain model show that CEs manifest themselves on multiple levels. In the most visible form, a change leads to the amendment

of the domain model by the necessity to add additional classes (see change of 10 day VaR calculation) or multiple relations (2 inheritance relations for CIRS). In other cases, the effect is situated on the level of the attributes of (a) class(es). The increase in shift methods and the adjustment of the alphas are examples of this. In yet other cases, the business rules linked to certain classes need to be amended. This is shown by the addition of the new product type CIRS, which alters business rules in multiple classes.

Because of this, future research should formalise the definition of 'change' at domain model level. Moreover, future research should offer recommendations to avoid the described CEs. Moreover, the current chapter is strictly limited to a well-defined scope that reflects common practices in the sector, but nevertheless disregards others. Therefore, future research should broaden the scope of the current chapter. The model should be expanded to encompass the entire scope of market risk, including multiple risk factors, other VaR-calculation methodologies, backtesing and stress testing practices, and future requirements such as expected shortfall calculation. Moreover, as we know now that CEs do exist in one sub-part of the financial industry, their existence in other sub-parts may be investigated. In addition, these sub-parts can be considered as modules in themselves. The information passing from one part to the other acts as the interface. Therefore, future research could focus on possible CEs at a higher level. Of course, difficulties regarding the hiding of CEs at higher levels of abstraction need to be taken into account.

In this chapter, we focused mainly on the data requirements in the domain. Future work could consider the same domain from a dynamic point of view and focus on, e.g., process models like the HVaR process in Fig. 10.2.

Chapter 11

Architectural Concepts Limiting GUI Transitions

Our research goal stated in Section 1.5 aims at improving technology transitions of GUI. Regardless the specific technology, GUI frameworks typically use certain design or architectural patterns. However, as we showed in Section 4.2, there is a widespread confusion between them. There is no clear consensus on their implementation nor their definition. Therefore, we provided by no means a complete and exhaustive analysis of patterns that have been commonly used in recent decades. Our review continued in Section 4.3, where we introduced some of the concrete frameworks and libraries trending on the market. Finally, in the previous chapter Chapter 10, we explained that certain software artefacts, e.g., domain models, may be analysed with respect to CEs defined in Definition 1.

Similarly to that, in this part of dissertation thesis, we want to reveal GUI concepts limiting the shift to another GUI technology. One of the objectives is to understand CEs caused by necessary changes throughout such a transition. However, we do not intend to calculate the exact number of classes that need to be refactored, nor we do not intend to provide a formula for an estimation of man hours required to modernise GUI. Instead, we want to explore evolvability of GUIs. If possible, we want to provide an advice on what to look for and what to avoid when designing them.

We researched this together with Mareš [A.13] who later published the results in his master thesis. Its adjusted version is provided in this chapter. It also helps us tackling RO 2.5 that we put forward in Section 1.5.2.

11.1 Analysing GUI Frameworks

The scope of this research part is large. In Section 4.2, we put forward a number of GUI design and architecture patterns commonly used. We observed that the complexity of these patterns graduates. Thus, it makes their understanding increasingly difficult and their adaption imperfect. Since the analysis of a particular technology is a costly endeavour, we

This chapter contains a modified excerpt of: *Václav Mareš. Evolvability of UI technologies, MSc Thesis.* Published in: *CTU in Prague, Faculty of Information Technology. Prague, Czech Republic, 2019. [A.13]*

decided to demonstrate our ideas on two sample technologies. Anyone anxious to analyse and evaluate another technology can continue analogically.

We decided to use the following frameworks – WinForms introduced in Section 4.3.2 and WPF presented in Section 4.3.3. The decision comes from the expertise of authors, and its possible contribution to Corima research that is largely using WPF. Although some might consider these technologies ageing, Google Trends [96] shows both topics still actively searched worldwide, just about half as much as was their peak search volume. The Fig. 11.1 depicts it.

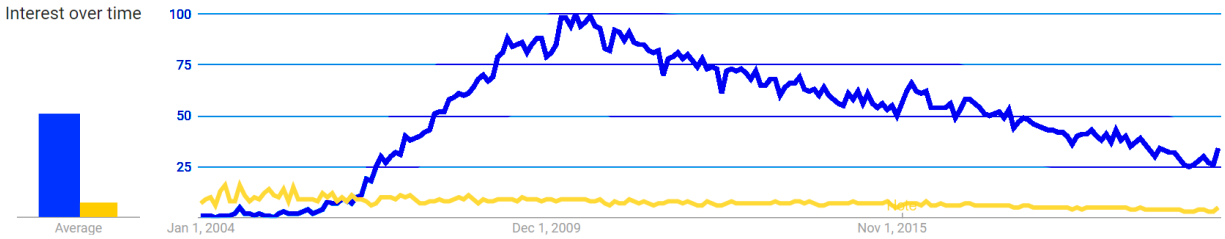

Figure 11.1: Search trend of WPF and WinForms captured by Google Trends [96]

Additionally, a quick search through GitHub[1] shows several very active and highly rated repositories [229, 231] extending or consuming these technologies. Not to mention, products like DevExpress [56], Telerik [205], or Syncfusion [160] discussed in Section 4.4.2 that set their business models on extending these Microsoft's frameworks and produce new versions for both. Thereby, we conclude that WPF and WinForms are alive and actively used. Furthermore, we are not reviewing the myriad of different libraries, extensions, and frameworks built on top of what Microsoft provides as their GUI technologies. We limit ourselves to the documentation, principles, and code samples provided directly by Microsoft, mostly what can be found on their documentation website [144, 145].

11.2 Modifying WinForms GUI

In Section 4.3.2, we introduced WinForms based on FaC architectural pattern explained in Section 4.2.3. Let us now present its simple example provided by the official documentation from Microsoft. We show a code snippet in Listing 11.1. For brevity, we only present part of the official example.

The example shows a source code of an application that displays a message box upon clicking a button. The events raised by users are handled by event handler functions. Even this simplistic code discloses clues of how the framework is meant to be used. The constructor of the `Form1` sets the control, i.e., `button1`, and fills up its properties. It also hooks the `button1_Click` event handler function to the `Click` event of `button1`. In this particular case, the message box dialog is directly shown when the button is clicked.

[1]GitHub is a provider of Internet hosting for software development and version control using Git. [1]

```
using System.Windows.Forms;
namespace FormWithButton {
  public class Form1 : Form {
    public Button button1;
    public Form1() {
      button1 = new Button();
      button1.Size = new Size(40, 40);
      button1.Location = new Point(30, 30);
      button1.Text = "Click me";
      this.Controls.Add(button1);
      button1.Click += new EventHandler(button1_Click);
    }
    private void button1_Click(object sender, EventArgs e) {
      MessageBox.Show("Hello World");
    }
    [STAThread]
    static void Main() {
      Application.EnableVisualStyles();
      Application.Run(new Form1());
    }
  }
}
```

Listing 11.1: Simple form example in WinForms

Unfortunately, this is not a toy example tailored for illustrative purposes. Microsoft presents many examples where the code in the event handler function modifies other elements directly or manipulates session state data. For instance, the code snippet in Listing 11.2 is also provided by Microsoft. It demonstrates a code manipulating a visual of a form upon clicking the button.

It shows the principle FaC is adhered to. The `Form1` is the all knowing entity that manipulates its controls. Its logic dictates what the controls do and how they behave.

11.2.1 WinForms in NST and EA Lens

Let us have a critical look at WinForms in the lens of NST and EA methodologies. Starting with Separation of Concerns (see Theorem 2.2.1), the `Form` class might be even worthy of the anti-pattern label 'God class'[2]. These are its most obvious responsibilities:

- **Visual representation** – This responsibility might not be obvious at first sight. WinForms developers often use Visual Studio Integrated Development Environment (IDE) [3] from Microsoft. It provides so-called form designer where they drag and drop

[2] In OOP, a 'God object' is an object that knows too much or does too much. [1]

[3] IDE is software for building applications that combines common developer tools into a single GUI

```
private void button1_Click(object sender, System.EventArgs e) {
   // Get the control the Button control is located in.
   // In this case a GroupBox.
   Control control = button1.Parent;
   // Set the text and backcolor of the parent control.
   control.Text = "My Groupbox";
   control.BackColor = Color.Blue;
   // Get the form that the Button control is contained
   // within.
   Form myForm = button1.FindForm();
   // Set the text and color of the form containing
   // the Button.
   myForm.Text = "The Form of My Control";
   myForm.BackColor = Color.Red;
}
```

Listing 11.2: Event handler function in WinForms

controls, set their properties and even generate event handlers. Finally, the `Form` class remains to be the one object responsible for handling the complex initialisation of the controls. It takes care of the layout of controls and the population of their properties.

- **Data Binding** – In case of simple binding, the `Binding` object is created during the initialisation of the `Form`. It knows what data binds to what control's properties. Considering a complex binding, it behaves similarly, only the instantiation of data sources might be delayed. It may proceed during the form load or other startup event.

- **User input** – Controls raise events that can be received and handled. Again, it is in the realm of the form. Form designer from Visual Studio generates these event handlers directly to the `Form` class leading developers to write the event handling logic. This can expand to calling into business components and down the rabbit hole. Potentially, this design can lead to freezing the application waiting for some back-end response.

This is a staggering number and scope of responsibilities gathered in one class. In terms of NST, we may recognise many change drivers. Considering the tight coupling between components and methods fulfilling the different requirements, the CEs defined in Definition 1 are present all over the place. Any change in a control triggers changes to the initialisation of the form, its data binding, and event handlers. This propagates to every form that use the control. This CE is also present in the other direction. If the data changes, all different bindings and assignments to controls have to change too. There is also no notion of view state. Only the controls hold the data they display. This again mixes responsibilities and hinders any possibility to share the state across multiple views.

Lastly, there is a lack of modularity. Any time we would like to reuse an existing form, add something to it, or adjust few controls, the framework forces us to create a new form entirely. Composition exists only on the level of whole controls and even tweaking an existing control leads to creating a completely new custom one.

11.2.2 WinForms Resume

WinForms was meant to support concepts of FaC architectural pattern. It is rather simple starting a WinForms project and to develop a small application in a relatively short time. However, the system is fundamentally flawed. There are change drivers that will lead to CEs. This is mostly apparent in the core `Form` class that comes with too many responsibilities, thereby too many possible change drivers.

However, we have to be critical to ourselves here. If anyone considers writing an application with WinForms today, the community will suggest much better practices. The most valuable professionals[4] often voice their suggestions on how to isolate and limit certain areas that we marked as sources of CEs. We must admit they are right. However, this may count for the applications developed in WinForms now. Since our goal is to investigate transitioning and evolving ageing applications, we dare to expect they may suffer from the old troublesome aforementioned approaches.

11.3 Modifying WPF

In Section 4.3.3, we briefly introduced WPF. Let us now present its basic concepts on the code snippet in Listing 11.3. We are using the example from Microsoft's documentation of the basic application [232].

Logical and Visual Tree. From the technical perspective, XAML defines so-called *logical* and *visual Tree*. The logical tree is the description of higher level elements that create the GUI. It is also used for Dependency Properties, Static and Dynamic Resources, and Data Binding. All of that results in dynamic layouts as indicated by the row's `Height` parameter being set to `Auto`. This helps to render a responsive layout.

The visual tree is a super set of the logical tree with all different elements that are being rendered. Therefore, a single node `Button` from the logical tree is expanded to a subtree with `Border`, `ContainerPresenter`, and a `TextBlock` that we see on the screen. The visual tree is utilised when rendering objects, layouts, and for routed events that travel across it. Let us briefly explain certain WPF concepts.

- *Dependency Property* – an object property that is managed by WPF framework. It is registered with its metadata and allows for new functionalities that were not possible

[4]Most Valuable Professionals are people awarded by Microsoft for 'actively sharing their ... technical expertise with the different technology communities related directly, or indirectly to Microsoft'. [1]

```
<Page x:Class="ExpenseIt.ExpenseReportPage" xmlns="http://..."
    ...
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="300"
    Title="ExpenseIt - View Expense">

  <Grid Margin="10,0,10,10">
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition />
      <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>

    <!-- People list -->
    <Border Grid.Column="0" Grid.Row="0" Height="35" Padding="5"
       Background="#4E87D4">
      <Label VerticalAlignment="Center"
         Foreground="White">Names</Label>
    </Border>
    <ListBox Name="peopleListBox" Grid.Column="0" Grid.Row="1">
      <ListBoxItem>Mike</ListBoxItem>
      <ListBoxItem>Lisa</ListBoxItem>
      <ListBoxItem>John</ListBoxItem>
    </ListBox>

    <!-- View report button -->
  <Button Grid.Column="1" Grid.Row="3" Margin="0,10,0,0"
     Width="125" Height="25" HorizontalAlignment="Right"
     Click="Button_Click">View</Button>
  </Grid>
</Page>
```

Listing 11.3: XAML example

with the ordinary object properties such as: styling, data binding, using dynamic resources, and animation.

- *Resources* – objects can be defined as resources making them available for reuse in other XAML files. Static resources cause a single lookup where the dynamic resource creates a link to the value and updates on change. Availability scope depends on the declaration point and spans only the subtree. Resource dictionaries, separate files, are a common practice that allows easy reuse of defined objects.
- *Data Binding* – the concept is identical to the one presented in this thesis. It is a way to link data of GUI controls to data in objects behind GUI. XAML supports multiple modes, the most used are one-way and two-way.
- *Routed Events* – this feature allows for CoR succession of event handlers to react to an event. The chain progresses along the visual tree and can be bubbling (up) or tunneling (down) depending on its direction. This allows for reacting only to events that interest certain control and pass the others.
- *Commands* – a semantic level approach to actions that application should execute. It allows for removing logic code from event handlers and sharing these objects encapsulating the actions across GUI. There is a little more complexity for `RoutedCommands`, but are the most useful in terms of reuse and isolation.

Code behind. There is one more concept worth explaining. It is called *code behind*. We present its snippet in Listing 11.4. Essentially, all classes generated when adding new XAML files have the extension `.xaml.cs`. All of them are *code behind* classes. These are partial classes[5] that are also described by the XAML files and one can write logic and business code into them and their event handlers, but the best practice is to leave them bare and empty, and use commands instead of other objects that encapsulate just logic.

11.3.1 WPF in NST and EA Lens

Let us have a critical look at WPF in the lens of NST and EA methodologies. The principle of Separation of Concerns is fulfilled with XAML describing visualisation and layout, and the code behind doing the rest. The user input is intercepted by controls and routed events that traverse the tree structure. This isolation of elements allows components to focus only on events relevant to them.

The action following the input could still be implemented into the event handlers, but if we allow the influence of MVVM there should be a ViewModel objects with the relevant commands. This makes a big difference as we can now bind to the commands directly from the Controls even in XAML. This allows for complete separation of business logic, but costs some extra effort.

[5]In languages supporting the feature, a partial class is a class whose definition may be split into multiple pieces, within a single source-code file or across multiple files. [147]

```
using System;
...
namespace ExpenseIt {
  /// <summary>
  /// Interaction logic for ExpenseItHome.xaml
  /// </summary>
  public partial class ExpenseItHome : Page {
    public ExpenseItHome() {
      InitializeComponent();
    }

    private void Button_Click(object sender, RoutedEventArgs e) {
      // View Expense Report
      ExpenseReportPage expenseReportPage = new
         ExpenseReportPage();
      this.NavigationService.Navigate(expenseReportPage);
    }
  }
}
```

Listing 11.4: Code behind file ExpenseItHome.xaml.cs

The weak point is in the relation of Model and ViewModel that was not much discussed here. The idea of MVVM is that the ViewModel is a wholesome description for the GUI to render. Some of this might be very fragile depending on the exact implementation of passing data between Model and ViewModel, but WPF does not describe this part. Nonetheless this relationship can be a source of CEs if multiple ViewModels feed into a shared Model, change to the Model can have unbounded impact.

The binding to data is also extended, allowing not only multiple modes, but also a wide variety of targets. The most notable is the dependency property that allows for all dynamic links. This results in evaluating these properties just-in-time even based on multiple inputs that are not tightly coupled. This contributes to lowering the CEs since instead of propagating and copying data there is a link to a central dictionary. Limiting the possible number of object relationships from n^2 to just n.

The last topic of our interest was modularity of the system. Looking at the View side of the framework XAML is working very well in this direction. It allows for resource dictionaries, separate styles, and each control to be defined independently. On the side of business logic, it can be direct and simple written in the code behind, loosing the modularity and reusability, but embracing the ViewModel concept that is supported with commands and wide available data binding all the logic of the application can be made reusable and wrapped in appropriate objects.

11.3.2 WPF Resume

From the evaluation above, it might look that WPF is clearly much better in terms of functionality and also in terms of evolvability. We agree that WPF with the MVVM architecture seems like a really good approach to GUI in general. However, we have to add that the devil is in detail. We tried to describe the most notable concepts and make some arguments for what they mean in terms of our scope. The point to get from this analysis is the approach we took. We advise to be much more thorough in the case of real project analysis that should be done on a specific code base.

11.4 Transition Approaches

Let us now elaborate on the typical approaches to use when migrating SW from one technology to another. We can imagine a SW application having its presentation layer written in a specific GUI framework, e.g., the aforementioned WinForms, or WPF. Let us denote this framework by F_1. This framework needs to be replaced by framework F_2. Although the vision of the application employing the new framework F_2 is clear, the approach to accomplish it is not entirely obvious.

Before transitioning $F_1 \rightarrow F_2$, we typically formulate the *initial state* A and *intended state* B of the application. This may encompass the required functionality, appearance, and behavior. Also, it should cover the technical specification of the application using F_1, including its architecture, best practices, etc. Thus, a deep analysis of the frameworks F_1 and F_2 themselves is desired. Next to other activities, it may comprise discovering of CEs exhibited by the particular technology as we outlined in Section 11.2 and Section 11.3.

However, the states A and B alone do not determine the approach to move between them. Therefore, together with Mareš [A.13], we inspected two well-known transition approaches:

- *Rewrite from scratch* – the presentation layer in state B would be developed from scratch using F_2. The application would not be shipped to the production until it reaches the state B fully.
- *Change incrementally* – the application would be developed in steps. Each step represents its intermediate state between A and B. Thus, the application schematically evolves as follows: $A \rightarrow A' \rightarrow A'' \rightarrow ... \rightarrow B$. It remains operational at each intermediate state, and it is shipped incrementally to a production.

In the next sections, we will discuss and reason about the benefits and drawbacks of each particular approach.

11.4.1 Rewrite from Scratch

This approach leads to a new development of the corresponding application. This requires at least a rewrite of the presentation layer implemented using the framework F_1. In practice,

we have to reverse engineer the application in its state A. We need to understand its conceptual model and technology. We have to build a new presentation layer thereby reaching the state B while using F_2.

This idea of rewriting applications in need of an update or severe maintenance is not new. Netscape 6.0 is a real-world example of such a rewrite. Joel Spolsky, co-founder and CEO of StackOverflow shared the story of Netscape in *Things You Should Never Do* [198]. He explained that in the 90's, Netscape was the most dominant web browser on the market. When the company was developing version 5.0 of its browser, the code proved very difficult to deal with new requirements. They decided to scrap it and create a new version 6.0 from scratch. However, this full rewrite of the code base took almost three years, enough for the company to lose its independence on the market. When Netscape 6.0 was finally released, the product was rushed and not fully developed. Back then, Netscape was competing with Internet Explorer from Microsoft. In the meantime, it already flooded the market, thereby marking the end of Netscape era. These are also not unique consequences of such decisions to rewrite an application. Similar stories can be found with Microsoft's Word in 1991, that effort was abandoned, and many others.

Nevertheless, our conclusion is that one needs to be cautious judging if this approach is the right path forward. Rewriting implicitly could result in a great time delay, unexpected expenses, reverse engineering costs, and usually a broad scope even with only partial rewrites (e.g., GUI), and lastly a great risk of failing at any point in time. This conclusion is confirmed by the aforementioned Standish Group *Endless Modernisation: How Infinite Flow Keeps Software Fresh* [116]. They conducted a research about failures of IT companies modernising software. The report was based on 50.000 projects in the CHAOS 2020 database including six individual attributes of success: on budget, on time, on target, value, on goal, and on satisfaction. The study concluded that organisations starting from scratch had a 26% success rate versus a 20% failure rate. Comparing to other approaches to modernise SW, this one was the least successful and the most failing.

If we would want to relate to the agile approach outlined in Section 2.1, we cannot be much further apart. Therefore, we mostly share the opinion of Spolsky who concluded that 'A functioning application should never ever be rewritten from the ground up' [198]. His opinion is based on two reasons – first, 'crufty-looking parts of the application's code base often embed hard-earned knowledge about corner cases and weird bugs', and the second 'a rewrite is a lengthy undertaking that keeps you from improving on your existing product, during which time the competition is gaining on you'.

11.4.2 Change Incrementally

Incremental change is an alternative to rewriting the application from scratch. It is an approach employing gradual changes to achieve the intended state. As earlier discussed, we have an initial state A and the intended state B of the application. The path is a series of intermediate states altering the current state until B is reached. The key principle here is that the application remains operational at each state, thus it is potentially production ready. Schematically, the application evolves as follows:

$$A \rightarrow A' \rightarrow A'' \rightarrow A''' \rightarrow ... \rightarrow B$$

Martin Fowler [86] calls this approach the *Strangler Application*. There are real world examples that show success using this approach and promote the reduced risks that it provides. For instance, Stevenson and Pols [202] gives an insight into the transition of the legacy financial application InkBlot by employing this approach.

In their InkBlot example, they emphasise the importance of delivering the core functionality first. It is built on the 80/20 rule saying that the most used 20% of features satisfy 80% of users. Each step must help us with one of the two objectives below:

- Bring us closer to the intended state B. This can mean that some part of GUI is migrated to the new framework F_2 or some part of the architecture is transformed to the new paradigm.
- Enable us to move closer to the intended state in the next step. This can mean restructuring the code and opening new options.

There are several case studies [169], where we can see the the strangler pattern in practice. The core idea is illustrated in Fig. 11.2. It shows that the abstraction layer is developed between a consumer and a component providing a certain functionality. This abstraction layer is often referred to as *Anti-Corruption Layer pattern* [154].

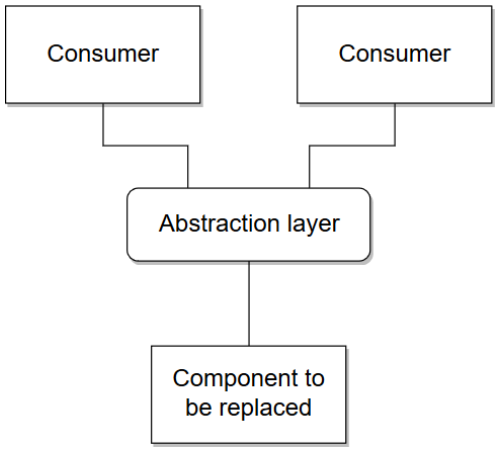

Figure 11.2: Abstraction layer placement

The step of creating the abstraction layer is beneficial on its own. It can point to what is used and needed from the consumer perspective. Thanks to the existence of the abstraction layer we can gradually replace the old component with the new one as depicted in Fig. 11.3. When all the functionality is used from the new component, we can remove the old one and potentially also the abstraction layer. This depends on the decision of a development team that must consider possible penalties on the application performance.

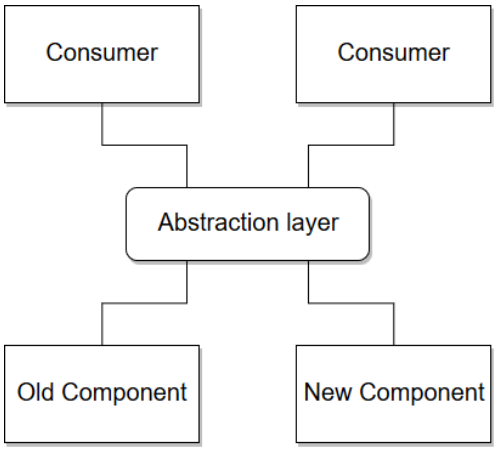

Figure 11.3: Abstraction layer usage

We intentionally do not mention how broad the abstraction layer is and what granularity the consumer and components are. This can be chosen during the transition and depends on technical details and other contexts.

As we concluded with Mareš [A.13], if we step back a bit from the problem, we may observe that this approach looks like combating technical debt. We argue that the root problem is that the old framework F_1 became a technical dept and we are now paying interest to it during the migration. Furthermore, referring again to the report of Standish Group [116], this approach can be seen as an incremental flow-like modernisation. According to that report, 71% of the companies enjoying this approach succeeded while only 1% of them failed.

Section Takeaway

At least two approaches of transforming an application from one technology to another are commonly used – rewrite from scratch, and change incrementally. Both are used in real world projects. It seems the incremental approach is more popular with the rise of agile in software engineering. However, it is difficult to make convincing conclusions of which approach is better. However, according to the Standish Group, when employing the iterative change, the numbers are in our favour.

11.5 Transition in Practice

In Section 11.1, we reinforced the importance of understanding the architecture of the specific GUI framework and its weaknesses in terms of CEs. Next, in Section 11.4, we introduced possible transition approaches that deemed necessary to realise a transition between these frameworks. Now, we will outline a case study done by Mareš [A.13] demonstrating a transition in practice. Although Mareš in his master thesis describes the case study of Car Dealership in detail, we only indicate the outcomes and conceptual steps.

11.5.1 The Case Study

The case study demonstrates a naive Car Dealership information system. It covers the needs of multiple branches in different locations. It offers functionality to manipulate different car models and it supports roles such as manager and a sales representative.

The information system is written in C# running on Windows. The application was migrated from WinForms to WPF.

We decided for an incremental transition approach described in Section 11.4.2. Therefore, we conceptually defined what needs to happen along the way:

- A – initial state of the application covering the use-cases from its functional specification.
- F_1 – WinForms framework used by the application in its state A.
- B – intended state of the application covering the same use-cases as in A.
- F_2 – WPF framework used by the application in its state B.
- $A'..A''''$ – intermediate states of the application during its transition. Each state is either enabling, or it brings us closed to B.

Our takeaways from the case study resulted in a couple of observations. We realised that even a simple Car Dealership application is challenging to be migrated if all presented aspects are reflected. This means that in a real-life complex application, such a systematic approach must tackle several topics. The code base should be analysed on multiple levels – chosen architectural patterns, the framework's support for the said pattern, the framework's functionality, and adherence to the implicit implementation rules that come from these aspects. All these levels should be looked at on both ends of the transition, meaning the starting state and the intended state. Adding to that, in real-world projects, we need to pose the question whether the code base is fragmented and full of edge cases and ad hoc solutions. This is one more overarching quality that needs to be reviewed before we even consider transition at all. All of these angles are necessary in order to understand what does the transition from one GUI technology to another actually mean for a given project.

11.6 Conclusion

In this chapter, we addressed the RO 1. We wanted to understand concepts that limit GUI transitions. We gave a basic overview of two technologies that were used in the exploration of the transition process. The descriptions were rather brief, primarily focused on capturing concepts rather than the implementation details. However, one of the important realisations that came from the analysis is that frameworks can be leading developers towards a certain GUI architectural pattern, but they can be adapted to others as well.

Chapter 12

Building methodical framework: ADA

At this stage of our research, we are finally in a position to propose a methodical framework suited better for technology transitions – ADA. All the prerequisites depicted in Fig. 6.5 has already been tackled. Therefore, now we can address the research objective RO 3. Let us first recapitulate what we learned from the research artefacts developed in the previous chapters.

Understand the purpose. In Chapter 7, we addressed RO 2.1 concerning how user interactions may be depicted in SW. For the first time, we applied EE PSI theory into SE. We explained that one can focus on the business process where actors interact. Actors are the ones that will carry out the tasks involved in a process. This brought two important findings. First, at least one EE theory can be directly applied into SE. Second, the tasks emerging from business processes may come with own independent GUI.

The first finding made us curious of how to describe the tasks, and possibly other means where actors interact. In this case, we may focus on the purpose of their interaction, describe it, and generate GUI in possibly any current and future technology. Therefore, in Chapter 8, we inspected this from the perspective of flexibility-usability trade-off what constitutes our next research objective RO 2.2. We generally discussed how complex the descriptions of those purposes should be to remain usable for the developer.

The second finding made us to realise that even though we look forward the purpose-driven approach in SE, the industry will still suffer with legacy SW solutions. Thus, we need to bridge the gap between ageing SW used for people to interact, and new SW solutions built with purpose in mind. Therefore, we started another track of our research – RPA. We defined another research objective RO 2.3 to understand whether we can create business processes from interactions with legacy systems. If we could, again, we can create a GUI for the task in practically any technology. These ideas were researched in Chapter 9.

Experience the evolvability. Another prerequisite to build a methodical framework is the evolvability. To justify that the system is aiming at a better technology transition, we need to prove it. In Chapter 2, we already got a rough picture of evolvability. We decided

This chapter contains an extended version of: *Ondřej Dvořák, Robert Pergl, and Petr Kroha. Affordance-driven software assembling.* Published in: *EEWC, Luxembourg, 2018. [A.6]*

to use NST to evaluate evolvability. Yet, we never tried that in practice. Therefore, since we exemplify our research on financial system Corima, in Chapter 10 we elaborated on the evolvability of financial domain models. This helped us to address RO 2.4.

Based on that understanding, we revealed a lack of knowledge in nowadays practices in GUI development itself. In Chapter 4, we did by no means a complete and exhaustive analysis of each design and architecture patterns, and of certain GUI technology. However, we did not explicitly derive concepts that limit the resulting SW to evolve its GUI. We addressed that by the research objective RO 2.5 that was tackled Chapter 11.

Having the research objectives RO 2.1, RO 2.2, RO 2.3, RO 2.4, and RO 2.5 tackled in the previous chapters, we can now design the heart of technology transitions – methodical framework ADA. It should be grounded in a notion of purpose as it turns to be essential when actors interact throughout the task's GUI, or in general throughout any GUI.

The TAO theory and BETA theory discussed in Section 3.2 and Section 3.6 might be crucial to this. They offer concepts of function, construction, and their relationships. Moreover, they work with affordances that concern purposes. Therefore, let us now investigate how to build SW based on these theories and concepts.

12.1 Running Example (part 1)

In Section 14.1, we discuss a substantial application of the research in a commercial TMS Corima developed by the COPS company. However, a lasting Non-Disclosure Agreement (NDA) does not allow us to disclose all the implementation details. Therefore, to illustrate the results sufficiently, we present a hypothetical application for a cryptocurrency trading. In Section 12.4, we discuss how it would be implemented using the actual Corima architecture meant for this type of financial applications.

Running Example (part 1). *Imagine a cryptocurrency trading application. Traders trade cryptocurrencies for other assets, such as conventional fiat money or other digital currencies. A cryptocurrency exchange can be a market maker that typically takes the bid–ask spreads as a transaction commission for its service or, as a matching platform, simply charges fees. In such a trading platform, traders work with so-called trades – bids or asks for a cryptocurrency. They exchange a specific amount of a cryptocurrency for another one. These trades are typically used for a purpose of new buy or sell order, or historical outline of the latest bids close to where they trade. One of the objectives is an efficient user interaction. They want to do the common actions without using a mouse. Also, it is important to clearly visualise the fee charged for the exchange, so they may do the decision quickly. The task at hand is to migrate an original desktop application into a web environment.*

A possible GUI of such a cryptocurrency trading application is depicted in Fig. 12.1. We refer to it throughout the next sections.

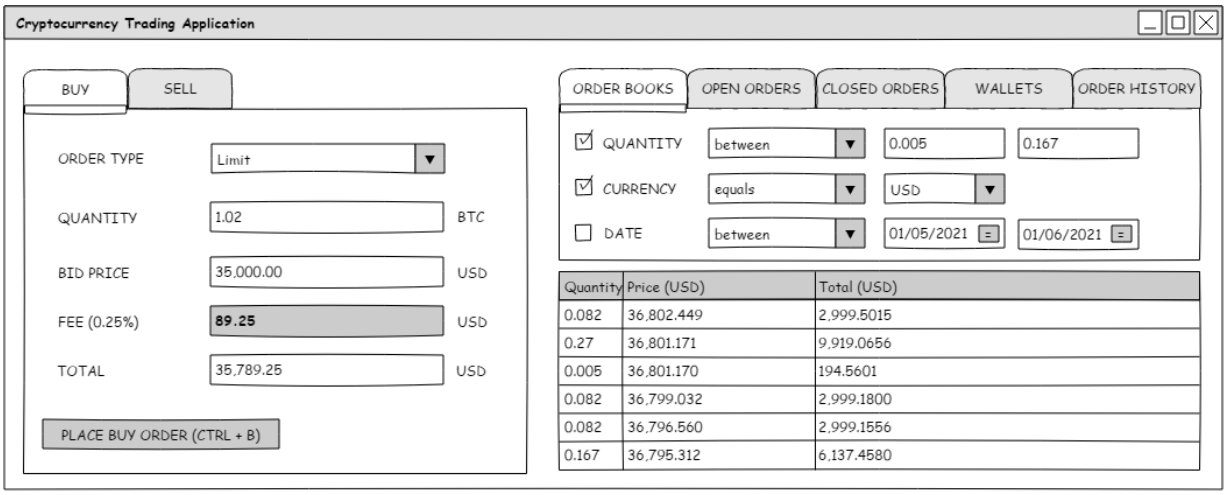

Figure 12.1: Wireframe of an application for cryptocurrency trading

12.2 SW Based on the TAO Theory and BETA Theory

We assume that the concepts of TAO theory and BETA theory play a crucial role for SW aiming at technology transitions. Let us investigate the characteristics of systems built on top of them. A system founded in TAO theory and BETA theory must be composed of components that suit the needs of a given user with a specific purpose. As we remember from Section 3.6, such a relationship is captured by the term *affordance* defined in Definition 6 as subject-object relationship represented by a formula (subject * purpose) * (object * properties) where the symbol '$*$' denotes the concept 'is in relation'.

By searching for a mapping from the general TAO theory and BETA theory in EE, we made the following observations:

- A subject corresponds to a software user.
- The purpose can remain general.
- An object is a software component as defined in Definition 8.
- Properties can remain general.

Now, we can reformulate Definition 6 in terms of CBSs as follows:

Definition 16. An affordance is a user-component relationship that can be represented by the following formula: *affordance: (user * purpose) * (component * properties).*

This definition is the key to constructing systems considering F/C division. We can now reformulate this relationship as a function that takes users, their purposes, and components with properties as inputs, and outputs a final construction. Fig. 12.2 visually represents this function in a CBS. One can see that the components and properties are inherently

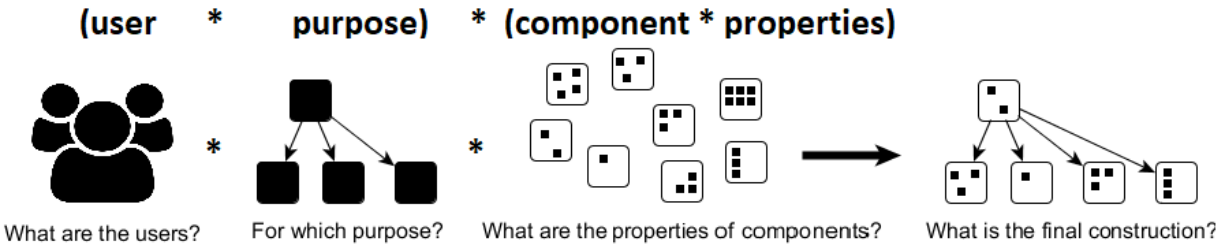

Figure 12.2: Affordances in CBSs [A.6]

bound together. This function expresses a software design process that results in the construction of a CBS from the appropriate components.

A mapping algorithm that selects the proper components for a user-purpose relationship is the key element of the software design process. Such an algorithm can be manual (assisted), semi-automated, or even fully automated (as is the case for the NSX mentioned in Section 5.1). Fig. 12.3 visualises the core of such an algorithm in a three-dimensional space. One can see that each component can satisfy the specific purpose of a given user to some extent (affordance). The degree of satisfaction is fuzzy. This means that it can range from completely true (best fit) to completely false (worst fit), as indicated by shades of grey in the figure. Here, we present the mapping algorithm on a theoretical level. This allows us to generally consider possible automation. In practice, such an algorithm may map appropriate components depending on the limitations of a user and the purpose he or she has. Therefore, all the three – components, users, purposes, are bound to the domain of SE, in particular to the typical use-cases we face in the GUI. For instance, while different users may have the same purpose, i.e., to exchange cryptocurrency, everyone of them may benefit from slightly different component satisfying this purpose. A good example might be a health barrier such as color blindness, or mental challenge discussed in Section 14.4, etc. These limitations may influence the choice of an appropriate GUI component.

Finally, when designing systems based on the TAO theory and BETA theory, we must define the semantic meanings of objects on the axes of Fig. 12.3. It requires a description of specific users and their purposes on one hand, and the components and their properties on the other hand. We call a software design approach based on these descriptions *affordance-driven* design and we investigate it deeper in the following section. This design approach is consistent with numerous model-driven techniques in which software artefacts are generated from abstract models to reduce development costs and shrink the error space [49].

12.3 ADA: The Way of Thinking

Before we jump into ADA, we again need to stress what Dietz concluded in Section 3.6 – an individual can identify an unlimited number of purposes for which a system can be used. There is no way to simply map functional decomposition to a constructional one. Thereby, the constructional designers must bridge the gap between them.

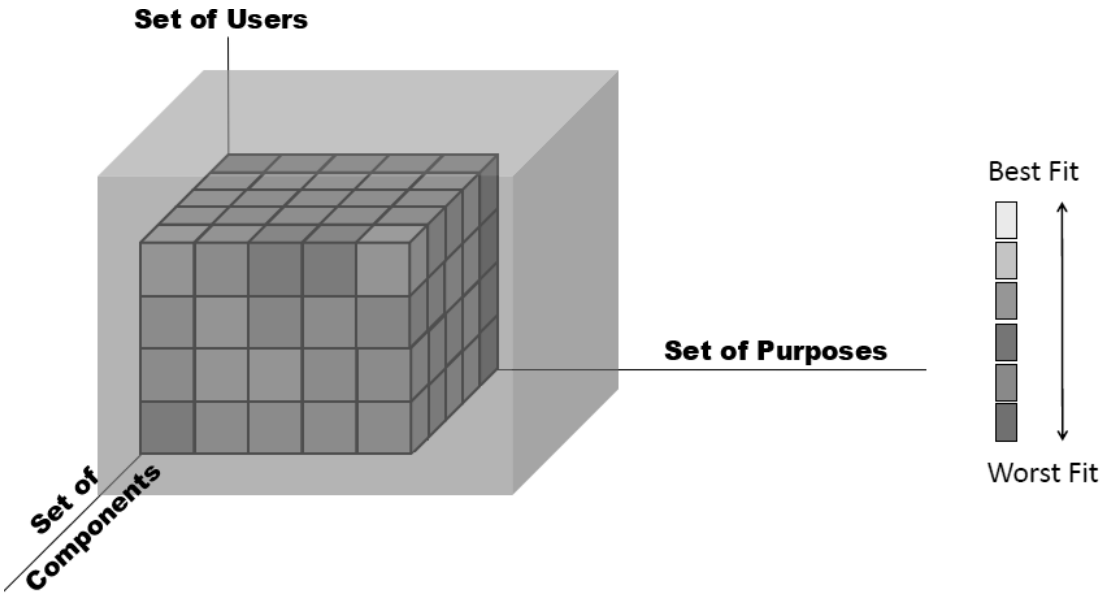

Figure 12.3: CBS affordances in a three-dimensional space

Having said that, we argue that the hard-to-grasp mental bridge between function and construction is one of the main reasons for difficult switches from old technology to the new one. Although we fully agree with Dietz at the general level, we suggest that if one is able to limit the scope of conditions to a certain degree, this gap can be bridged in a manageable and systematic manner in a software design approach, where subjective aspects can be minimised and the set of construction elements is limited and clearly defined. This suggestion is based on the following simple reasoning. Let us assume that the naturally infinite set of functions is (arbitrarily) limited. Then, by calculating a Cartesian product with a (naturally) limited set of possible constructions and limited design freedom (implied by SW development best practices), we can develop a bounded relationship between functions and construction.

Therefore, to make F/C relationships manageable, we deliberately limit the number of imaginable users, purposes, and components. We now formulate the following definition based on the previous discussion [A.6]:

Definition 17. ADA is a software design approach for the development of CBSs following Definition 7, where all of the following points hold:

1. There is a bounded set of **ADA-users** AU that we are able to describe formally.
2. There is a bounded set of **ADA-purposes** AP that we are able to describe formally.
3. There is a bounded set of **ADA-components** AC. Each component has its own construction and properties and it manifests its possible ADA-purposes for all possible ADA-users.

4. ADA-components are either atomic or they consist of other ADA- components representing their constructional decomposition according to the TAO theory.

5. There is a relation called an **ADA-relation** defined as $(AU * AP) * AC$.

Definition 18. The union of ADA-users, ADA-purposes, and ADA-components forms the set of **ADA-elements** AE, which can be captured by the following formula: $AD = AU \cup AP \cup AC$.

Similar to an industrial assembly line, we use the term 'assembling' to express that a product is merely assembled from ready-to-use components instead of requiring heavily human-dependent design principles, as shown in Fig. 3.7.

To illustrate on the running example introduced in Section 12.1, ADA-users is a set of traders. Each ADA-user may be characterised by their limitations such as health limitations, and personal preferences influencing the environment where they want to trade. ADA-purposes is a set of purposes they typically have during trading such as to buy & sell, to filter latest bids, to show their crypto wallet, etc. ADA-Components is a set of components in a specific technology and environment that may satisfy the trading purposes of the traders. In this particular case, it may contain charts, widgets for viewing various aspects of the crypto wallet, configurable forms to exchange cryptocurrency, etc. The ADA-relation is the pairing up (mapping) relating traders with purposes to components with properties. Thereby, this relation determines the extent to which the specific component may satisfy a given trader having a specific purpose. Altogether, these elements constitute ADA-elements. ADA-elements may be then implemented using various technical approaches, for example a low-code solution later discussed in Section 5.2.

Next, we elaborate more on ADA-elements and ADA-relation.

12.3.1 Realising ADA-relation

The principles according to which a 'process of organising knowledge regarding an application domain into hierarchical rankings or orderings of abstractions' are commonly referred to as *abstraction principles* [204]. Because purposes (i.e., ADA-purposes) can be viewed as a type of knowledge, the same principles can be applied to them. Therefore, purposes can be logically grouped into a higher level of abstraction that is organised hierarchically [204] 'to obtain a better understanding of the phenomena of concern'. The set of ADA-purposes forms a tree structure, which is a functional decomposition according to the TAO theory. This tree describes all possible ADA-purposes related to all possible users for which a system within a specific domain can be used.

To realise ADA-relation, there are two possible approaches.

1. Developing a function $(AU * AP) \rightarrow AC$ that assigns a set of ADA-components to ADA-users and ADA-purposes. This scenario corresponds to Fig. 12.2 and can be viewed as generating software from specifications, such as the press described in [137].

2. Developing a function $AC \rightarrow (AU * AP)$ that returns possible ADA-users and ADA-purposes for a given ADA-component. This function is a means of reusability. In other words, it is a means of discovering existing components that are suitable for a specific combination of ADA-users and ADA-purposes, thereby facilitating the software design process.

12.3.2 Objectified ADA (O-ADA)

In the Unified Foundational Ontology of Social Entities [98], a purpose can be mapped to a *desire*, which is an entity type that is existentially dependent on its bearer. This means that in ADA, we must independently classify various types of users and purposes, and relate them. However, we can simplify this process by considering only 'standard' users and their typical purposes. This means that the type of a common desire can be 'extracted' from its bearer to form an existentially independent object called an objectified desire. According to the TAO theory, this is an intended affordance disconnected from the purpose of a user. The theory refers to this concept using the term 'function'. Therefore, we can formulate the following simplified notion of ADA:

Definition 19. Objectified Affordance Driven Assembling (O-ADA) is a special type of ADA with the following changes:

1. Instead of ADA-users and ADA-purposes, there is a bounded set of **O-ADA-functions** OAF that we are able to describe formally.

2. We replace ADA-relation with the following **O-ADA-relation**: $OAF * AC$.

Again, we must handle an unlimited number of purposes, as discussed by Dietz and Hoogervorst [65]. However, we can assume that we can objectify the decomposition into a form on which everyone can agree. Additionally, we can limit the decomposition to the domain of a specific system.

Therefore, although the TAO theory considers an affordance as something that 'subjects perceive in their pursuit of satisfying needs' [65], which can lead to an unlimited number of purposes for using an object according to its properties, we do not consider such a broad definition in ADA. We assume that the number of purposes is limited[1].

O-ADA is strongly related to existing requirements engineering principles. In our previous work [A.6], we explained that ADA-users, ADA-purposes, and O-ADA-functions can be mapped to use-cases that are traditionally captured by UML use-case diagrams. In the waterfall model of the software development lifecycle, such a description corresponds to the phase of analysing requirements. In contrast, in the agile model of software development, O-ADA-functions can be translated from user stories written in the following format canonised by Mike Cohn [37]: 'As an [actor], I want [action] so that [achievement]'.

[1]We consider the fact that a situation in which certain software artefacts or technologies are used for something they were not designed for is not impossible. For example, HTML was supposed to be a markup language for displaying text in a web browser. However, it is often used for the purpose of constructing complex interactive GUIs

Again, let us refer to the running example from Section 12.1. The O-ADA-function might be the following: 'as a [trader], I want [sell cryptocurrency] so that [I can quickly see the exchange fee]'. Another one could be 'as a [trader], I want to [view crypto wallet] so that [I can filter it by balance and by name of the currency]'. O-ADA-relation determines to which extent the available GUI components fulfil these functions.

Section Takeaway

In this section, we presented the way of thinking about ADA. We explained that systems built on top of TAO theory and BETA theory may support technology transitions. This requires defining sets such as ADA-users, ADA-purposes, ADA-components, and capturing the ADA-relation between them. We altogether call them ADA-elements. They help us to realise the hard-to-grasp mental bridge between F/C that Dietz finds impossible to build. Moreover, if we can afford objectifying ADA-users and their ADA-purposes into a form on which everyone can agree, we can simplify ADA into O-ADA. This simplification limits the degree of freedom to a set of O-ADA-functions that can be mapped into well-known user stories.

12.4 ADA: The Way of Working

Previously, we demonstrated that when constructing a software system based on the notions of EE theories, we must consider concepts captured by the affordances defined in Definition 17. In such a system, we must be able to identify ADA-users, describe their ADA-purposes, and construct a final solution consisting of ADA-components. This corresponds to the function $(AU * AP) \rightarrow AC$.

In this section, we first propose a potential software architecture in which ADA can be realised. To exemplify our descriptions, we will again use the running example put forward in Section 12.1.

12.4.1 Designing a Software Architecture

To develop a sample ADA system architecture, we employed a model-driven development approach. Such an approach is consistent with our approach in terms of constructing a platform-independent model. This is a model that typically consists of conceptual data, models, use-case models, descriptions of functions, and processes. However, these are 'defined in a general form abstracted from concrete technologies and platforms'. [172]. This model can be realised using UML component and activity diagrams [25]. The *component diagram* in Fig. 12.4 presents basic high-level components and their relationships in an ADA-based CBS. Additionally, Fig. 12.5 depicts the flow between components.

The proposed architecture has been evolving over time in the form of a design artefact from the applied DSRM. Its very original version emerged from our previous research on projectional editor for Domain-Specific Language (DSL) [A.9]. In this stage, we aimed at having a DSL to describe GUI. This preliminary version already contained the idea of

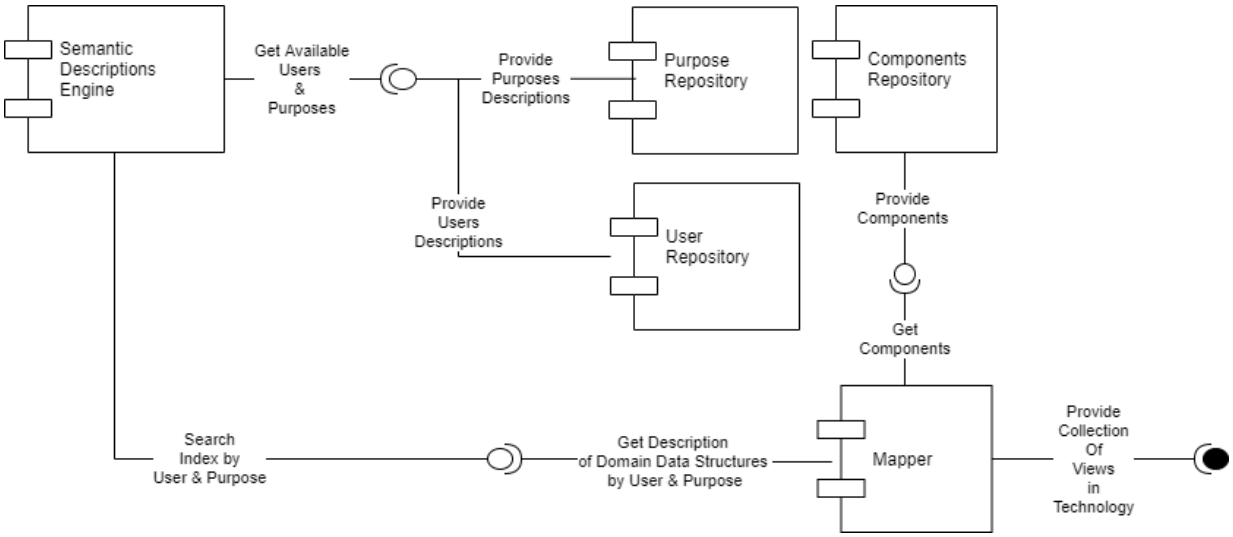

Figure 12.4: A possible high-level architecture for a system applying ADA

generating SW components from descriptions. However, when we analysed the topic of technology transitions conceptually, essential concepts such as users, their purposes, and components defined in Definition 17 were formulated.

Let us walk through the architecture. First, we explain the components that directly reflect ADA-elements from its definition, each of which must be described, e.g., in XML, DSL, annotation, or other descriptive tools. In our running example, traders use trades for the purpose of exchanging cryptocurrencies without using a mouse, or to view and filter the latest bids. The `Purpose Repository` is the storage containing possible purposes typical for cryptocurrency trading. Additionally, each trader may have certain restrictions effecting him during cryptocurrency trading – physical disorders such as color blindness, mental challenges such as a lack of focus. Moreover, they may be of different age. All these characteristics might influence the final choice of GUI components dealing with trading. Similarly, the `User repository` is the storage containing these user characteristics that might be reflected in GUI of trading application. Finally, given a cryptocurrency trade, the `Mapper` component is responsible for finding an appropriate GUI component with respect to the trader's purpose and their user characteristics. These components are provided by the `Component Repository`. In our example, mappers may return JavaScript or WPF controls for filtering data, grids for viewing them, widgets for bidding and selling, etc. The `Semantic Description Engine` plays a different role. It does not correspond to any ADA-element. It was only introduced to provide a technical infrastructure to work with them. It is responsible to process ADA-elements regardless the form used to describe them. For example, in our finance domain, this engine can process information about types of objects such as trade. In this case, the system will likely represent it as a domain data structure, e.g., a class in OOP [43]. This class is accompanied by the so-called semantic description based on those available in the `User Repository` and `Purpose Repository`.

It covers all user characteristics and purposes for which trade objects are used by traders. Respectively, the purpose or user characteristics are expressed as a combination of domain data structures and operations on those structures. Additionally, the engine must be able to store descriptions of domain data structures in a manner that can be efficiently searched using a combination of a user and its purpose(s). Typically, some type of dictionary is implemented.

The *activity diagram* in Fig. 12.5 illustrates the typical flow among components in Fig. 12.4. It represents a situation in which the system needs to generate GUI to manipulate instances of domain data structures in a given context, such as GUI to satisfy certain purposes. In our situation, the system needs to generate GUI for traders to satisfy their cryptocurrency trading purposes. An example of this GUI is in Fig. 12.1. The left side shows a widget satisfying a purpose to buy or sell cryptocurrency (to create a new trade object). The Fig. 12.1 illustrates that the 'PLACE BUY ORDER' button comes with a shortcut. This is again due to the requirement of satisfying that purpose without using a mouse. The right side contains a table satisfying a purpose to show recent bids (viewing and filtering trade objects).

The entire process begins with a request to 'get a collection of views in technology'. For example, this could be a request to obtain a collection of JavaScript views to orchestrate an editor, where a trader having the purpose to deal trades does not need to use a mouse.

The engine handles this request. It does two things. First, it parses the semantic descriptions of the domain data structure on its inputs. Second, for every single attribute of the domain data structure, it generates a list of how the property should behave for certain type of users having a specific purpose dealing with that. Again, let us explain it on building GUI dealing with trades. Two purposes enter the engine – purpose to buy and sell cryptocurrency, purpose to show recent bids. The engine opens the domain data structure representing the trade. For every single attribute (order type, quantity, bid price, fee, total amount, data, ...), it assigns a matrix which tells how such an attribute satisfies the purpose users could use it for. For instance, the attribute 'fee' must be clearly visible when satisfying a purpose to buy or sell a cryptocurrency.

Next, the mapper proceeds. Based on the detailed parsing of the domain data structures, it searches the component repository to find the best GUI satisfying required purposes. In our case, it generates a JavaScript view similar to Fig. 12.1.

Section Takeaway

In this section, we covered a software architecture where ADA can be realised. We showed that ADA concepts such as ADA-users, ADA-purposes, and ADA-components are to be stored in certain SW repositories. Each repository may use own notation to describe the ADA-elements, e.g., DSL, XML, annotations, etc. Therefore, the SW architecture will likely contain a component standardising these specifics. We called this component `Semantic Description Engine`. It outputs data to a `Mapper` generating the final GUI from ADA-components. Thereby, it satisfies the ADA-purpose of given ADA-user(s).

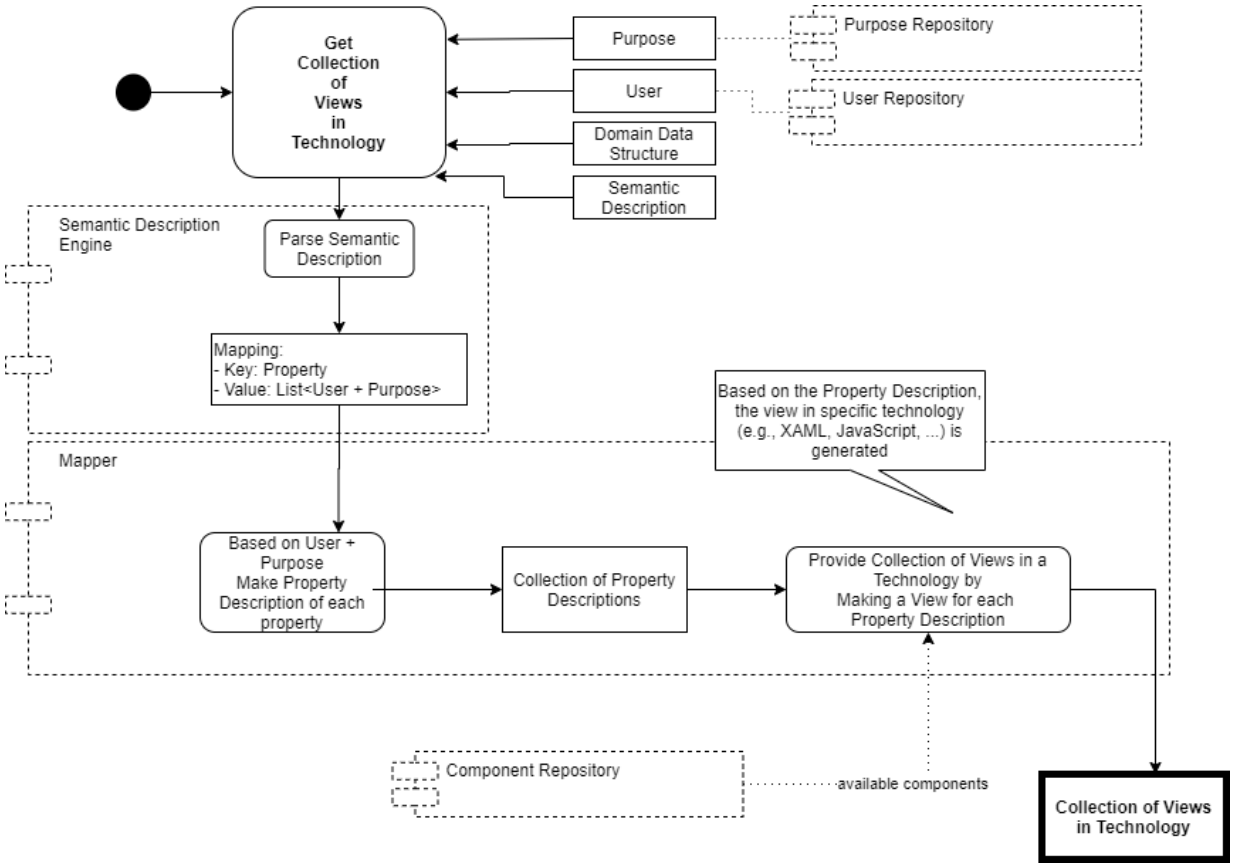

Figure 12.5: ADA process

12.5 Chapter Summary

In this chapter, we synthesised all findings addressing particular research objectives of RO 2. In Section 12.3, we introduced a methodical framework ADA. ADA is created with a notion of purpose of a user in mind. It allows for constructing GUI based on mapping these purposes to a limited set of SW components. Next, in Section 12.4, we proposed a SW architecture where ADA may be realised in practice.

In the next chapter, we demonstrate its application in an industry-scale Corima application.

Chapter 13

Demonstrating ADA in Corima

Throughout our research, we already examined Corima framework multiple times. Now, we will present how it implements and widely uses ADA. Corima is a multi-user client-server application and an application platform at once. Its applications cover the needs of the whole treasury department of banking/corporate customers of COPS. Most of the applications are data-centric, focused on displaying data in standardised components, e.g., pivot tables, grids, charts, edit forms, etc. They typically cover use-cases in which several users communicate their demands to a server that cooperates with a risk management banking system contending with the processing of various operations carrying essential information (e.g., deals, FX, and balances).

Corima traditionally offered WPF. Lately, we started porting it to a new web technology (Angular). This is where we realised that expressing ADA-purposes in a descriptive language (or a DSL) is a great help. The future technological switches will not require rewriting it (possibly just syntactically), just the code of high-cohesive, low-coupled ADA-components must be adjusted.

In the following sections, we relate the traditional requirements analysis performed in SE to the concepts formulated by ADA. For simplicity, let us work with O-ADA[1]. First, in Section 13.1, we extend our running example. We analyse it using the traditional UML Use-Case Diagram, and demonstrate a sample wireframe of the demanded system. Next, in Section 13.2, we map the corresponding user requirements into O-ADA-functions. In Section 13.3, we show how we describe them in Corima. Finally, in Section 13.4, we outline the Corima architecture enabling O-ADA.

13.1 Running Example (part 2)

Let us now extend the running example put forward in Section 12.1. We zoom into the area of a cryptocurrency trading where traders may filter for the latest bids. These bids are

[1]This will be probably the most common situation in practice, as we need to work with ADA just in case we need to distinguish special types of users, such as users with various inabilities.

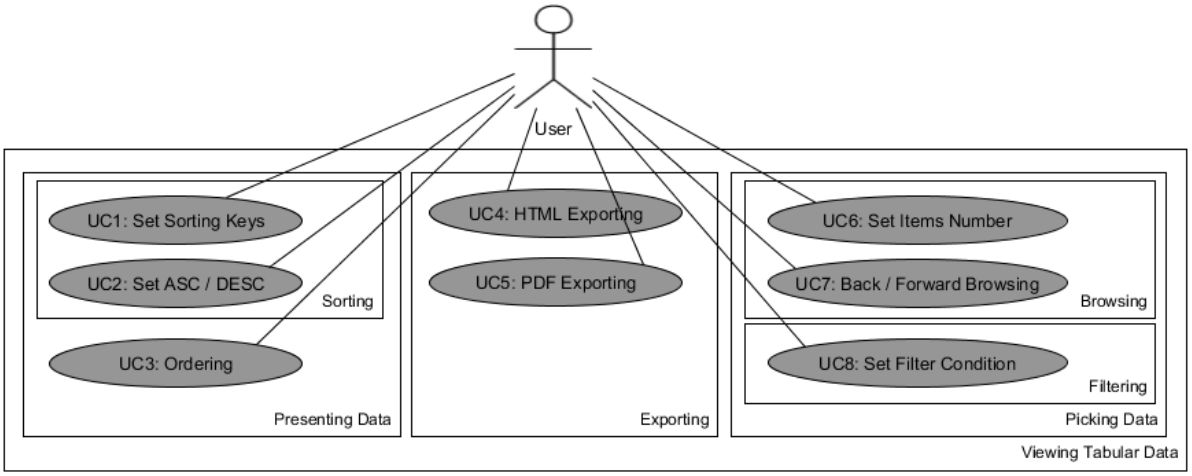

Figure 13.1: Use-case diagram of a tabular data viewer

stored in a relational database and the users manipulate them with the tabular data viewer.

Running Example (part 2): *Tabular data viewer is a component for viewing, filtering, and exporting historical data about cryptocurrency bids. The system must implement use-cases depicted in Fig. 13.1. The wireframe of its possible construction can look like in Fig. 13.2. We can see that* UC_1 *and* UC_2 *can be performed by using the header of each column, one can sort data ascending or descending using a little black arrow.* UC_3 *is covered by using the left / right arrows directly in the header – they switch the order of columns. The exporting functionality demanded by* UC_3 *and* UC_4 *is triggered by HTML / PDF buttons. The number of displayed items demanded by* UC_6 *can be set in the footer using a page size selector. The back and forward browsing in* UC_7 *can be performed using a scroll bar, and the filter condition in* UC_8 *can be set in a filter.*

Additionally, we declare that tabular data viewer should ignore the name of cryptocurrency while filtering, i.e., it must handle the notion of ignoring things. Per default, the cryptocurrencies should be sorted in ascending order by price.

13.2 Mapping User Requirements to O-ADA-Functions

Now, we need to map the aforementioned use-cases onto O-ADA-functions that determine the scope in which the application operates. In a general situation, this would result in a mapping onto several O-ADA-function decomposition trees (called a *forest* in computer science). However, in our simple case, all the use-cases map onto a single O-ADA-function decomposition of the tabular data viewer. The result is depicted in Fig. 13.3. The black nodes mark O-ADA-functions of the tabular data viewer in the running example. The grey nodes mark O-ADA-functions that are not needed, yet generally, they belong to the same domain of viewing tabular data. Clearly, the involved O-ADA-functions result in a subtree. Again, in a common situation, we end up with a forest of O-ADA-function trees.

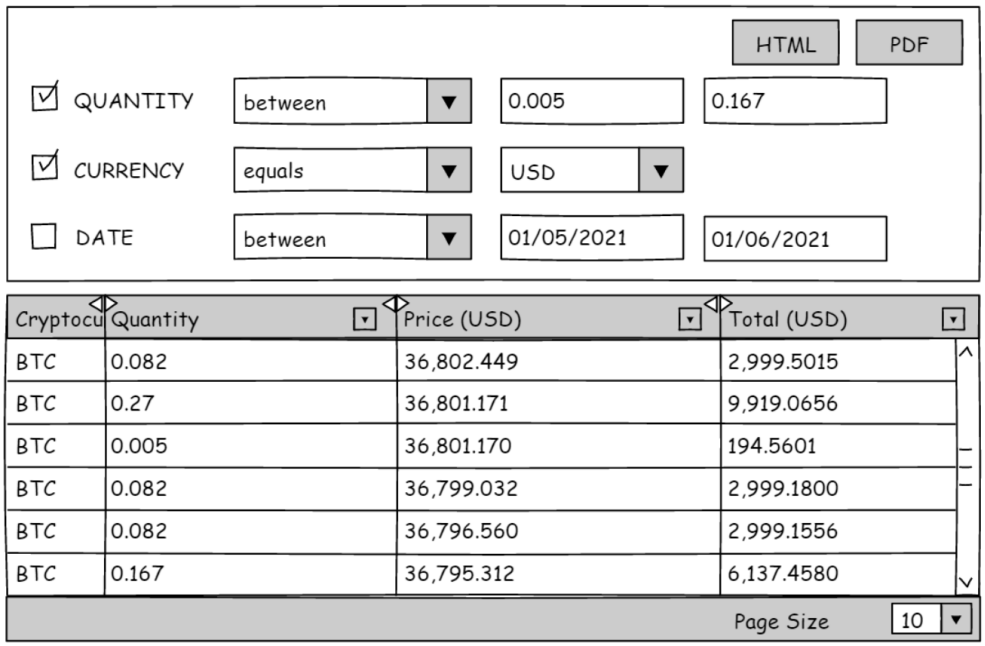

Figure 13.2: Wireframe of a tabular data viewer

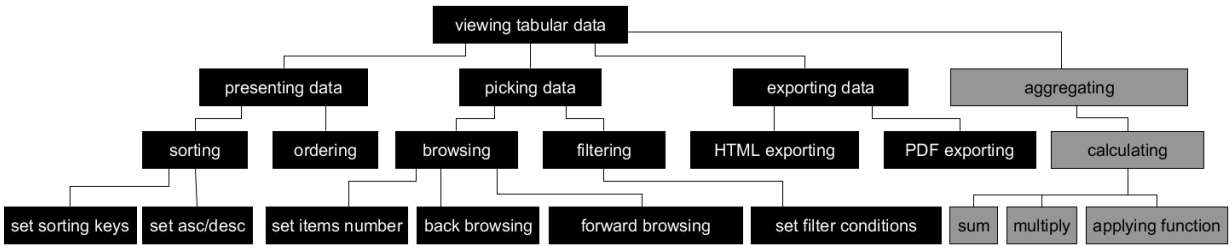

Figure 13.3: O-ADA-Functions decomposition of the tabular data viewer used in the cryptocurrency trading application

In Fig. 13.4, the constructional decomposition is displayed using a traditional UML Component Diagram. Generally, the relation between a node from the functional decomposition and the nodes in the constructional decomposition is M:N, as one function may be (and it typically is) realised by several constructional nodes and vice versa, one constructional node may realise several functions (reusability). In this particular case, the constructional decomposition is very similar to the functional one. However, the nodes represent constructional elements, and as such they may be reused in different systems.

Similar mapping to O-ADA-functions could be done for use-cases emerging from the Section 12.1 of the first part of our running example. However, in this chapter, we describe our reasoning on these extended use-cases to mainpulate tabular data of cryptocurrency historical bids.

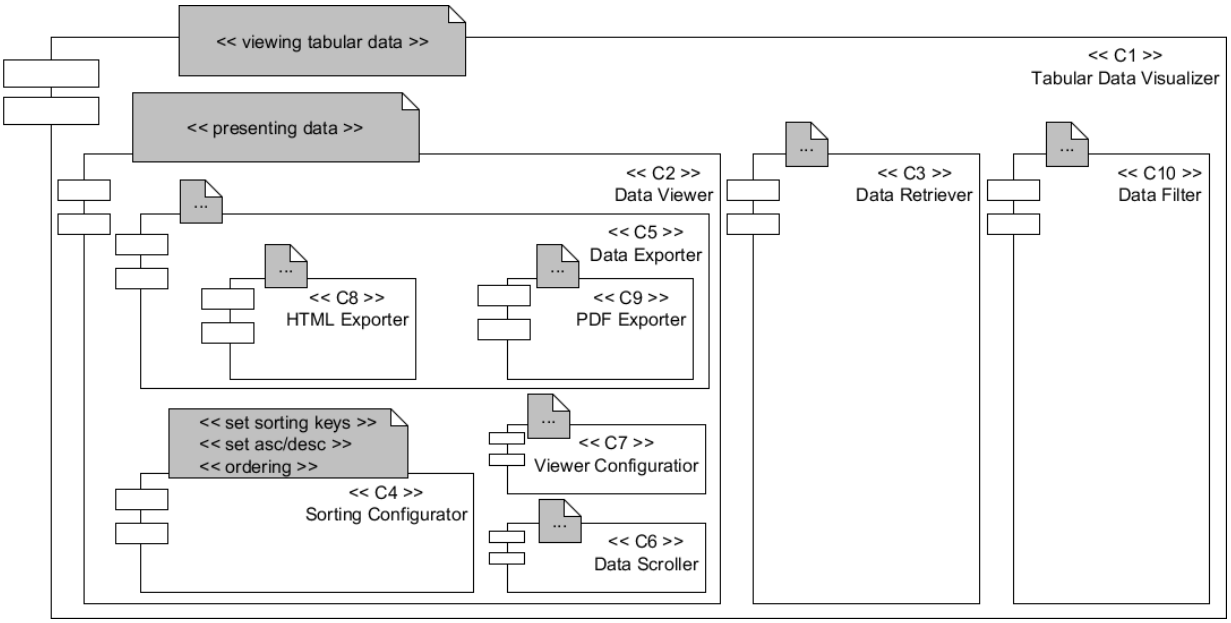

Figure 13.4: UML Component diagram representing a constructional decomposition

Section Takeaway

In this section, we showed that O-ADA-functions correspond to the functional decomposition of a system. The functional decomposition is based on the use-cases the system should cover. The construction decomposition in O-ADA-system must support a mapping to O-ADA-functions for the O-ADA-system to be realised.

13.3 Semantic Descriptions

Now, having an O-ADA-functions defined, the next question is how to describe them. In Section 12.4.1, we said that domain data structure such as cryptocurrency 'trade' must be described semantically, e.g., using XML, DSL, annotation, etc. This description enriches the domain data structure with information necessary for O-ADA-based system. When ADA-components deal with instances of these structures, the description helps them to understand how to use them for a certain O-ADA-function.

Let us move back to general ADA. For instance, ADA-component satisfying ADA-purpose of filtering trades would use the trade description to understand the sorting preferences of a certain ADA-user. An ADA-component supporting ADA-user to buy and sell cryptocurrency would benefit from a description of a required number of decimal places, highlights, limit options, etc. In Corima, we call these descriptions Semantic Descriptions (SDs). We define them as follows:

Definition 20. A Semantic Description (SD) in ADA is a formal description of an ADA-element that specifies its essence in such a way that the ADA-relation can be realised.

Then, similarly:

Definition 21. A Semantic Description (SD) in O-ADA is a formal description of an O-ADA-element that specifies its essence in such a way that the O-ADA-relation can be realised.

For machine processing, we need to encode the SDs into a formal language: General-Purpose Language (GPL), a descriptive language (like XML), or DSL.

Expressing SD in DSL (or in a descriptive language) instead of a GPL, brings additional effort of implementing a parser/processor. However, for a certain number of SDs, this pays off, because once a switch to a new programming language occurs, the SD does not have to be re-coded as it will be apparent from the Chapter 14. For typical large enterprise systems like the one described below, the savings of costs and decrease of risks of bugs can be considerable. Moreover, as we described in our previous work [A.9], due to the simplicity of DSL, several visual and textual forms can be used for its presentation. These are known as DSL projections [220] that could be maintained in so-called *projectional editor*. Hence some of the projections do not require any deeper technical programming knowledge and can be easily understood by a non-developer, the projections can improve a cooperation between tech people and domain experts. However, in a scope of ADA implementation, we decided not build an independent DSL for SD and maintain it with a projectional editor. Rather, we sticked on SD in C# that is more common in Corima.

13.4 ADA Architecture

Corima implements ADA architecture similarly to the one described in Section 12.4.1. Therefore, we will now shift away from the simplified O-ADA we referred to above. In Fig. 12.4, we introduced the components required to implement ADA. The UML class diagram in Fig. 13.5 illustrates our design decisions in Corima. Below, we detail these decisions and present specific examples of ADA-purposes. A more detailed summary of the various ADA-purposes available in Corima is beyond the scope of this study. We solely focus on a general architecture for ADA implementation, rather than on various ADA-purposes in different contexts.

Semantic Description Engine. In Corima, there are two types of SDs enabling us to describe the domain data structure semantically: using *class descriptions* or using *class annotations*. Without elaboration, these descriptions can be simply represented by objects. We refer to such an object as a `DomainClassDescription`.

`DescriptionController` is a crucial class in the `Semantic Description Engine`. Corima uses it to register all available types of ADA-users from the user repository combined with all possible ADA-purposes they may have. In our running example, the user repository includes the types of trading users with all their aforementioned characteristics, the purpose repository contains types of purposes to satisfy traders. For instance, specific buy

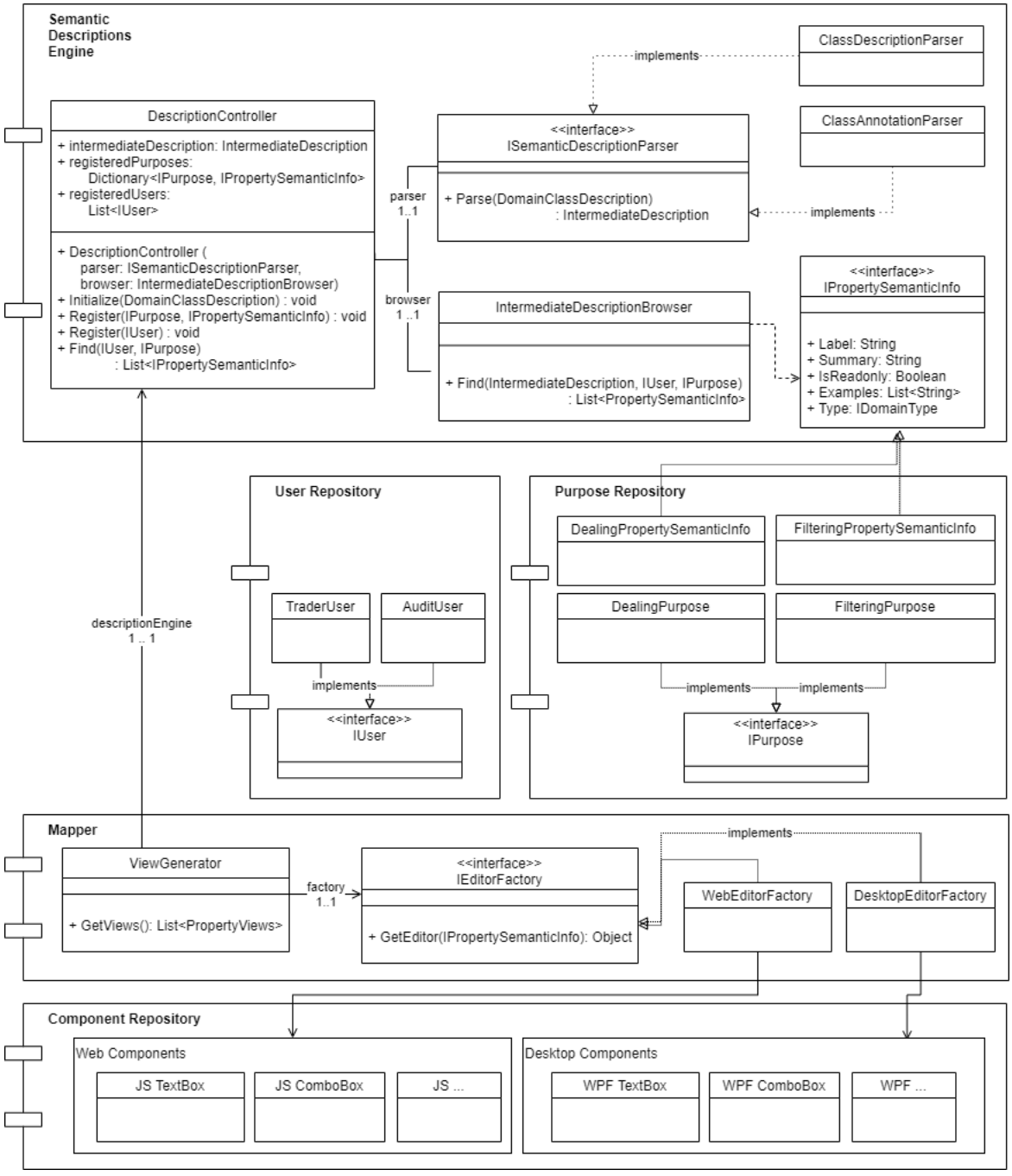

Figure 13.5: ADA process in Corima

and sell cryptocurrency purpose, and generic purpose to filter and view data regardless the type of data it manipulates.

Upon request, `DescriptionController` will be initialised with a specific instance of a `DomainClassDescription`. The controller will parse this description using a specialised parser such as `ClassDescriptionParser` or `ClassAnnotationParser` and then store it in the form of an `IntermediateDescription`. Similarly, if COPS architects would decide to store these descriptions in XML or DSL, obviously `XmlParser` or `DSLParser` would need to be introduced. However, the description would always need to be transformed into an independent `IntermediateDescription`. It is optimised for further searching by the `IntermediateDescriptionBrowser`. Finally, the `DescriptionController` is responsible for finding semantic information that corresponds to a given user with a specific purpose. This information is stored in an instance of the interface `IPropertySemanticInfo` for each property of the described domain data structure. For example, let us reffer to our running example Section 13.1. If we consider a purpose of filtering data, then the semantic information contains an indication of whether one can filter by a specific field (property). If we consider the purpose of exchanging trades, the semantic information contains a constraint regarding the ability to exchange cryptocurrency without using a mouse, disable filtering by 'fee' when viewing trades in a filter table, etc. Similarly, the preference on visible decimal numbers shown to the trader would also be covered by the semantic information.

User Repository. As explained previously, CBS constructed according to the notion of *affordances* must select components that are well suited a specific user with a given purpose. Therefore, CBS utilising ADA must contain a repository defining ADA-users, such as a trader, controller, front, back, or middle office employee, risk officer, accountant, etc. In Corima, a specific class for each user can be created and stored in a repository. Again, each class representing ADA-user may come with characteristics that must be considered in Corima such as age, health restrictions, or preferences influencing the appearance (shape, color, contrast, GUI patterns, etc.). It may include clues that indicate not obvious functions that are not displayed until the action is being taken. These may influence a drop-down menu or other clickable feature that only appears when the user is hovering over it.

Purpose Repository. ADA-Purposes are expressed as a combination of domain data structures and operations. Each purpose is represented by a class. These classes constitute the purpose repository. Additionally, each purpose is linked to a special (technical) class representing the semantic information, such as `ExchangeCryptocurrencySemanticInfo`, `FilteringSemanticInfo`, `ViewingSemanticInfo`, etc. The instances of these classes are by no means the objects that semantically enrich the instance of domain data objects. When performing the `Find` operation, the `DescriptionController` returns their instances with help from its `browser`. For example, the implementation of the aforementioned exchange cryptocurrency purpose is shown below.

```
class ExchangeCryptocurrencyPurpose<T, SI> : IPurpose
    where T : object // Data type representing the data structure
```

```
    where SI : IExchangeCryptocurrencySemanticInfo {

    // Dictionary mapping properties of their SemanticInfo classes
    Dictionary<Prop, SI> SIDictionary { get; }

    void Highlighted(T obj, Prop prop) {
      SIDictionary[prop].Highlighted = true;
    }
    ...
}

class ExchangeCryptocurrencySemanticInfo :
   IExchangeCryptocurrencySemanticInfo {
   bool CanUseMouse { get; }
   string Label { get; }
   bool Highlighted { get; }
   ...
}
```

Listing 13.1: Sample implementation of purpose and semantic info class.

The `ExchangeCryptocurrencyPurpose` is instantiated in a `DescriptionController` and parsed by a `parser` having a type `ClassDescriptionParser`. This instance is evaluated within a `browser` generating concrete semantic information for the given user. The semantic information is created per each attribute and type of the purpose satisfied by trades. The specific trade description may look as on the draft realised using `ClassDescriptions` and `ClassAnnotations`. They heavily utilise the language features such as lambda expressions or annotations in C#. Referring to the Section 13.1, it shows how the requirement to disable filtering by cryptocurrency name, and the default sorting by price is fulfilled.

```
class TradeDescription : IClassDescription<Trade> {
   public void Describe(IClassDescription<Trade> d) {
        ...
        d.Purpose<IExchangeCryptocurrencyPurpose>()
            .DoNotUserMouse();
        d.Purpose<IExchangeCryptocurrencyPurpose>()
            .Field(x => x.Fee).Highlighted();
        d.Purpose<IFilteringPurpose>()
            .Field(x => x.Cryptocurrency).DisableSorting();
            .Field(x => x.BidPrice).SortPerDefault();
        ...
    }
}
[DoNotUseMouse] // Annotation of ExchangingCryptocurrencyPurpose
class Trade {
    ...
```

```
    [Highlighted] // Annotation of ExchangingCryptocurrencyPurpose
    [SortPerDefault] // Annotation of FilteringPurpose
    decimal Fee { get; set; }
    decimal Quantity { get; set; }
    decimal BidPrice { get; set; }
    ...
}
```

Listing 13.2: Sample implementation of trade descriptions using ClassDescriptions and ClassAnnotations

In the same way, the majority of domain data structures in Corima are described with respect to the purposes they satisfy throughout components in JavaScript, WPF, Angular, AngularJS, etc. A more detailed summary of the various ADA purposes available in Corima is beyond the scope of this study.

Component Repository. Components are specific to the technology used and are collected in a repository. For example, in Corima, there are repositories of web and desktop components for JavaScript (web) and WPF (desktop), respectively. These components cover typical atomic GUI controls for data inputs, such as drop-down menus, text boxes, and list views, buttons, radio buttons, etc. Additionally, it contains generic components for filtering and editing data, viewing data in charts, pivots, etc., and it provides specialised components to view cashflows, liquidity, and risk figures, or to support typical operations of front office, back office, middle office, risk, accounting, and other departments using Corima. These specialised components are ready to work with classes representing the corresponding semantic information. For instance, components for filtering data require `FilteringSemanticInfo` for its operation, those dealing with cashflows would require `CashflowSemanticInfo`, and components for liquidity planning would rely on `LiquiditySemanticInfo`. In our running example illustrated in Fig. 12.1, the application for cryptocurrency trading is orchestrated from one generic component used to filter trades, and one specialised component for exchanging cryptocurrencies. Both of them selects the atomic components to work with trades, e.g., atomic components for displaying quantity, bid price, and fee while exchanging cryptocurrency, or components to display date, currency, etc. when filering trades.

Mapper. Corima uses a mapper to generate final views for a specific technology. Its main responsibility is to pick suitable implementations of an interface called `IEditorFactory` (e.g., `WebEditorFactory` or `DesktopEditorFactory`) and trigger them to generate final views using components from the corresponding component repository. Any new future technology would need to come with the similar factory for Corima to switch to it. As we outlined in Section 12.2, this mapping could be in principle manual (assisted), semi-automated, or even fully automated. In Corima, it happens semi-automatically. Specialised complex components like filter tables, pivot tables are explicitly assigned, however the

atomic components are selected automatically based on the global mapping in Corima.

Section Takeaway

In this section, we outlined ADA architecture in Corima. We explained that in Corima, SD can be created using two different approaches. However, both of them results in a standardised SD format recognised by Corima. Finally, all the components described here can be mapped to the general architecture presented in Section 12.4.1. We explained them on the running example put forward in Section 12.1, and continued in Section 13.1.

13.5 Chapter Summary

In this chapter, we tackled the first part of RO 4. We demonstrated the methodical framework ADA in practice. Within an industrial-scale system Corima, we implemented a subsystem based on ADA. First, in Section 13.1, we extended a running example. In Section 13.2, we demonstrated, how such an example would be approached in terms of ADA. We explained the nature of user requirements in the area of finance, in particular the requirements for viewing and manipulating relational data. These requirements were mapped onto O-ADA-functions. Next, in Section 13.3, we exemplified semantic descriptions as a mean to express ADA-elements. In the scope of this thesis, we simplified them and we concentrated on the essential principle, rather than diving into the detailed implementation in Corima. Finally, in Section 13.4, we presented a convenient architecture of a system where ADA can be realised. We also described how we tackled certain architecture components in Corima.

Chapter 14

Evaluating ADA

In Chapter 12, we described the foundations of systems based on EE theories and ADA. In the same chapter, we elaborated on a potential high-level architecture for their implementation. A sample technical realisation was demonstrated on a Corima case study in Chapter 13. We now evaluate ADA-based systems in terms of their industrial scale, and in terms of *evolvability*. Specifically, we elaborate on the impact of technological changes on entire systems. To evaluate this impact, we performed measurements on the code base of Corima. We used NST to evaluate our approach from the perspective of improved evolvability, which is the main goal of this work. Therefore, we did not ground ADA in NST. We only used its principles to minimise CEs when transiting systems from one technology to another.

14.1 Embedding in Practice

Corima has been actively developed for more than a decade. Its Corima.cfs package contains around 20 web and 40 desktop applications [41]. These applications are used by numerous customers across DACH[1] region and Czech Republic. The DerTreasurer and HSBC [55] conducted a survey among companies with a turnover of up to five billion euros. They listed COPS among 10 market leaders in Germany with a market cap equal to 3.1%. The Corima applications are focused on the following aspects.

- *Cash management* – to administer, transfer, and optimise payment transactions, including the definition of derivation rules, pooling structures, configuration of the arrangement process, and parametrisation of payment transaction functionalities.
- *Front-office and Back-office* – to obtain an overview of the current status of treasury positions, process financial transactions, and control the treasury business workflow. This includes definitions of FX derivatives, money market trading, linking to external trading systems, and matching business information to corresponding platforms.

[1] DACH is an acronym used to describe Germany (D), Austria (A), and Switzerland (CH)

- *Liquidity planning* – to facilitate the efficient display and planning of liquidity positions. It helps the treasury department to ensure that the company remains solvent at all times for the foreseeable future.
- *Risk management* – to analyse and control financial risks. This includes market risk, credit risk, VaR, Interest Rate Risk in the Banking Book (IRRBB), and other types of risks.
- *Accounting* – to implement booking concepts on the basis of national and international requirements in parallel and integrate them into the overall treasury process. This includes the entry of derivatives such as FX, IR, and commodities, and the setup of loan accounting concepts.

Many applications are accommodated by modules designed to assemble components based on the ADA approach, such as the data editor, data filter, and data viewer. Currently, across approximately 40 desktop applications, the following numbers of modules are based on ADA: data editor (630 usages), data filter (513 usages), and data viewer (564 usages). In contrast to these modules, only approximately 262 modules are fully custom. Therefore, 86% of all modules are driven by ADA and their possible migration to other technologies will be accompanied with the benefits and limitations evaluated in the sections below.

14.2 Evaluating ADA in Terms of NST

Now, we want to evaluate ADA with respect to its goal – to construct software systems enabling controlled technology transitions. NST does not directly speak about technology transitions. However, it investigates so-called change drivers. In our case, the target change driver is a transition from one technology to another, such as a move from one framework or library to another, or a change from one programming language to another. Therefore, we decided to use NST as a benchmark to evaluate ADA. We want to show that if the system is designed according to ADA, the CEs driven by the technology change is bounded.

We now evaluate the impact of changes in an ADA-based system. We present it on the component diagram depicted in Fig. 12.4. First, we define some symbols that will help us discuss changes.

- S_n is a system using technology T_n.
- SD is a semantic description of a domain data structure.
- L is a language used for expressing SD.
- R_n is a component repository in technology T_n.
- M_n is a mapper fetching SD and creating a component using R_n.

- P_n is a purpose repository.
- U_n is a user repository.

Let us explain the symbols on our running example from Section 12.1. We also schematically included the symbols to Fig. 13.5. S_1 is a cryptocurrency trading application developed in a former desktop technology T_1 such as WPF. S_2 should be moved to a new web technology T_2 such as JavaScript. The application is dealing with a trade object to satisfy two purposes – P_1 to create a trade while exchanging cryptocurrency, and P_2 to filter historical trades or wallet history. The trading application behaves the same regardless the user. Thus, it expects to work with a trader user U_1 having specific characteristics, visual preferences, etc. In Corima, COPS developed a language L of `ClassDescriptions` to enrich the trade object with certain semantic meaning. The source code in this language may describe the trade with respect to certain purposes, e.g., P_1 and P_2 of the user U_1. The compiler of this code will produce a semantic description SD of the trade domain structure. These semantic descriptions will be used by components in a repository R_1 that contains all WPF components available in Corima, thus those using a technology T_1. On the other hand, R_2 is a repository of components in JavaScript that are available in Corima, thus they use technology T_2. M_1, respectively M_2 maps the SD to WPF, respectively to JavaScript components.

When realising a transition $S_1 \rightarrow S_2$, we must also realise the transition $M_1 \rightarrow M_2$ to keep the system consistent and running smoothly. Instead of T_1, the S_2 mapper must find components in R_2 for the new technology T_2 and organise them properly (e.g., by creating a view when working with GUI components). This could also potentially require a transition $R_1 \rightarrow R_2$ in the case where the new technology T_2 requires updates of components in the repository.

Based on the application of the *Separation of Concerns* theorem, the required changes for replacing a technology T_1 with a technology T_2 only touch the mapper and (potentially) the repository modules. Therefore, the impact of a change is bounded by the effort of transitioning M_1 to M_2, which maps SD into components in technology T_2 instead of T_1. However, if the change driver is in the same domain, only the purpose repository (P_n) and user repository (U_n) are affected, and the effects on the system are bounded by the effort of transitioning M_1 to map the new user and purpose repositories to R_1 components.

14.3 Evaluating ADA in Terms of Impact measurements

To demonstrate the impact of the technology transitions previously evaluated using NST, we now investigate the statistics of the real code base of a representative subset of applications in Corima. A_{D_1} denotes the source code of all desktop applications in Corima. A_{W_1} denotes the source code of all the applications in Corima.

The Corima system was originally implemented using WPF. Additionally, it was ported to the web using JavaScript and AngularJS[2]. Such a transformation is typically very

[2]AngularJS is a JavaScript-based open-source frontend web framework.

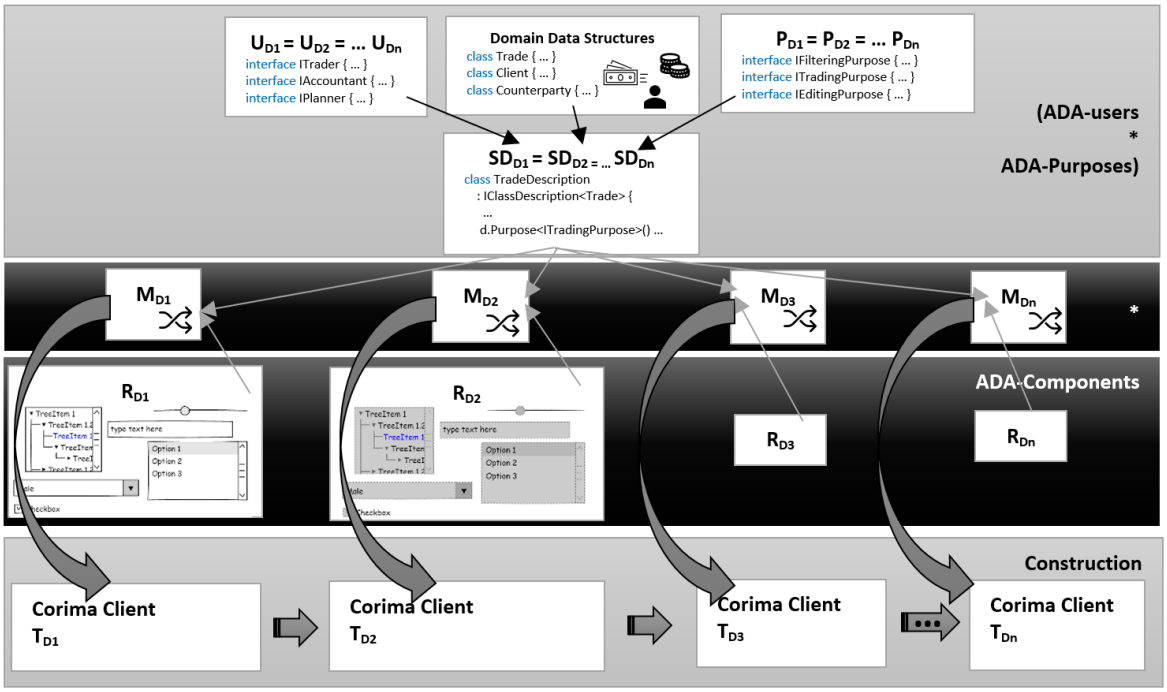

Figure 14.1: Corima transitions

challenging both technologically and economically. Let us demonstrate this transition in an ADA-based Corima. We list code base statistics for both web and desktop applications. Corima is a client-server platform. Because the focus of this research is on GUI frameworks, only the client side is relevant for our calculations. Therefore, we omit the size of the server-side code base and only list code base statistics for the client side.

We detail the calculations for the code base statistics of the desktop client. The same reasoning can be applied equally to the web client. We present that code base schematically in Fig. 14.1 with respect to the ADA definition in Definition 17 as well as the symbolic put forward above. One can see that the GUI code base of all desktop applications (A_{D_1}) is ∼618K LoC, which account for approximately 92% of the entire desktop code base. The repository of desktop-specific components (R_{D_1}) is covered by ∼11K LoC, accounting for less than 2% of the code base. The mapper (M_{D_1}) for fetching semantic descriptions (SD_{D_1}) for desktop applications and creating components using R_{D_1} only requires ∼1K LoC, accounting for a marginal 0.2% of the entire code base. Finally, the sizes of the purpose repository (P_{D_1}) and user repository (U_{D_1}) are ∼500 and ∼600 LoC, respectively.

Now, let us assume that we must re-implement Corima using components in a new GUI technology (T_{D_2}). The impact of this change is bounded. We only have to update the mapper $M_{D_1} \rightarrow M_{D_2}$ and create a new repository of components R_{D_2} using the new technology (T_{D_2}). Therefore, only ∼2.2% of the desktop code base is touched while 98% remains the same. Despite these relatively small changes, the most important NST aspect is that the number of changes is bounded with respect to the application size. As long as new applications in A_{D_1} use the components from R_{D_1}, their transition to T_{D_2} does not

	Desktop (T_{D_1})			Web (T_{W_1})		
	Symbol	Code Lines	Rate	Symbol	Code Lines	Rate
GUI of all Applications	A_{D_1}	618,230	92.34 %	A_{W_1}	38,250	77.41 %
All Semantic Descriptions	SD_{D_1}	37,529	5.61 %	SD_{W_1}	3,354	6.79 %
Mapper	M_{D_1}	1235	0.18 %	M_{W_1}	330	0.67 %
Repository	R_{D_1}	11,398	1.70 %	R_{W_1}	6,320	12.79 %
Purpose Repository	P_{D_1}	544	0.08 %	P_{W_1}	544	1.10 %
User Repository	U_{D_1}	613	0.09 %	U_{W_1}	613	1.24 %
TOTAL		**669,549**	**100 %**		**49,411**	**100 %**

Table 14.1: Code base for a subset of Corima treasury applications

influence the number of changes needed to transition $M_{D_1} \rightarrow M_{D_2}$ and $R_{D_1} \rightarrow R_{D_2}$.

However, when the change driver is within the same domain, the adjustment of the purpose and user repositories is bounded to ∼1.7% of the entire code base and does not change with an increasing number of new Corima applications.

Similar calculations can be performed for web applications in Corima that use JavaScript and AngularJS. Here, the percentages are slightly different (∼13.4% when re-implementing Corima using another web technology and ∼2.3% when the domain changes). This discrepancy in calculations between the desktop and web clients is irrelevant to our point because it simply represents differences between the technologies used (WPF and C# as T_{D_1} versus JavaScript and AngularJS as T_{W_1}). Overall, the change impact is similar.

14.4 Limitations of ADA

ADA limitations result from its ability to describe the dimensions presented in Fig. 12.3.

The first dimension is the dimension of *users*. In this dimension, ADA is limited by the ability to describe ADA-users so that the resulting GUI reflects their constraints. This is not required in Corima, the GUI is the same regardless of their constraints. However, together with Rašovský, we developed a prototype of an affordance-based system reflecting the needs of people with mental challenges. By explicitly capturing the disorders of potential users, suitable GUI components to serve their purposes were identified. We grounded the work in the research on *computer therapy*, which was introduced by Fiala [77]. We learned that the GUI for mentally challenged people respects the specific construction of elements, for example, colours, shapes, and distances between elements are required to have specific relations. This approach may be extremely beneficial for information systems provided by governments. Typically, by law, these systems must be accessible to all citizens, including those with certain limitations. Examples include ISO 17049, which focuses on the use of braille in accessible design, ISO 23599, which focuses on assistive products for blind and vision-impaired persons, and the ISO 21902 focused on accessible tourism.

The second dimension is the dimension of *purposes*. Here, ADA is limited by its ability to describe ADA-purposes so that the resulting GUI can satisfy them. We are often

unable to capture all the purposes for which GUI may be hypothetically used. However, in many cases, it is sufficient to describe only the purposes that are common in a given context. For instance, in Corima, we covered all required purposes dealing with Create, Read, Update, and Delete (CRUD) operations of tabular data in the context of filtering, displaying charts, presenting a master-detail view, etc. Considering the scale in which we used this approach in practice (see Section 14.1), we can cautiously argue that for an enterprise-wide application similar to Corima (i.e., dealing with large sets of tabular-data), we can describe all the necessary purposes. Therefore, if the purposes are tangible and well-described, ADA may help to 'bridge the mental gap between function and construction'.

The third dimension is the dimension of *components*. It relates to the implementation of the GUI itself. Therefore, we can better address its limitations using well-known SW development practices. With Mareš, in Chapter 11, we researched the state-of-the-art of GUI development. We realised that most GUI technologies and frameworks are based on a common architectural or design pattern. However, similar to Verelst and Mannaert [137], we concluded that these patterns do not themselves provide rigorous guidelines to ensure evolvability, and they are typically not consistent with respect to principles and ideas. In the same line of thought, there are multiple possible variations when implementing specific architectural patterns, each with different trade-offs. Therefore, in ADA research, we introduced a general architectural pattern designed for technology transition. Obviously, its implementation is limited by the technology, framework, language expressiveness, and experiences of the COPS. For example, Corima implemented the ADA with the .NET technology stack. It combines a declarative and imperative style of programming using C# and Language INtegrated Query (LINQ)[3]. Thus, the corresponding trade-offs must be considered when migrating Corima from one technology to another.

Finally, we can observe the overall limitations of ADA. It is a general design approach. Although it aids in the construction of SW designed for a technology transition, its implementation may vary significantly. Therefore, we cannot argue that an ADA-based system is completely free of CEs, and its evolvability depends on how it implements the NS theorems, which are not explicitly address. However, because technological evolvability is the main goal of this research, we demonstrated that an ADA-based system exhibits the characteristics of a bounded impact of the transition $T_1 \rightarrow T_2$ discussed in Section 14.2.

14.5 Chapter Summary

In this chapter, we evaluated the final ADA artefact emerging from our research. First, in Section 14.1, we tackled our research objectives RO 4.1 by presenting how ADA-based system Corima is embedded in practice. Next, we addressed the research objective RO 4.2 by a theoretical evaluation of ADA in terms of NST presented in Section 14.2, and in terms of Corima code-base measurement in Section 14.3. We closed this chapter by discussing the limitations of ADA in Section 14.4.

[3]LINQ is a Microsoft .NET Framework component that adds native data querying capabilities to .NET languages. [1]

Part V

Conclusion

Chapter 15

Discussion

In this chapter, we summarise the achievements of our research and we reflect on the conclusions. In addition, we demonstrate that the main goal presented in Section 1.5 and objectives of the research have been achieved and the research problem put forward in Section 1.4 got addressed. We also highlight the main contributions to the SE and EE body of knowledge. Furthermore, we propose and discuss the areas of improvements as well as possible future research directions.

15.1 Addressing the Research Goal and Objectives

The main goal of this research specified in Section 1.5 was set out to:

> *'Design and develop a new methodical framework that aids in the construction of software solutions enabling controlled technology transition.'*

As presented in Chapter 12, this aim was achieved and a framework ADA for the construction of software solutions enabling controlled technology transition was designed. This framework represents a main artefact developed by employing DSRM from Hevner [103, 102] and DSRP from Peffers [170].

The proposed framework builds on findings of EE, and it defines the concepts that need to be present in the SW architecture that supports controlled technology changes. In this way, not only the SW concepts can be determined but also the mapping to new technologies can be derived.

The framework was evaluated and its applicability to a real-life problem was further tested on an industry-scale application Corima. The framework can be effectively used by SW developers and architects for a development of GUI solutions that allow for controlled technology transition.

While we designed the framework by following the DSRP, we took a number of steps in few research phases put forward in Section 1.5.3. We discussed these steps in Chapter 6 in detail. They included (a) conceptualising our research, formulating the problem in our

research scope, populating the knowledge base of the research, and the identification of possible solutions, (b) design and development of the methodical framework, (c) demonstration and evaluating its real-life applicability, and finally (d) communication using suitable channels. The last step (d) was addressed by various publications and presentations. The other steps were translated into the major research objectives (and sub-objectives) of this research. In the next section, we will revisit how we achieved them.

15.1.1 Research Objective RO 1

The first objective of this research comprised four parts. It intended to investigate and understand the state-of-the-art of:

- *RO 1.1: research on SW evolvability;*
- *RO 1.2: EE theories from which SE may potentially benefit;*
- *RO 1.3: technological practices aiming at better technology transition;*
- *RO 1.4: related research aiming at better technology transition.*

In the text below, we will answer these research objectives and thereby answer the research question RQ 1.

The first research objective RO 1.1 was tackled in Chapter 2 by inspecting the evolvability from two perspectives – NST and EAs. NST helped us to understand the solid theoretical foundation, the notion of CEs, and practices focused on building evolvable SW. EAs revealed the emphasis on building DevOps and CI/CD infrastructure instead of dwelling on static architecture diagrams.

The second research objective RO 1.2 was addressed by Chapter 3. We reviewed different EE theories and we realised the importance of affordance, function, construction, and their separation.

The research objective RO 1.3 was met in Chapter 4. In Section 4.1, we started with a thorough and exhaustive narrative review of literature focused on CBSs development what unveiled the historical development in our research domain. This gave us a better inside into a topic of modularity and reusability. Next, in Section 4.2, we undertook a comprehensive elaboration of various architecture and design patterns commonly used to build GUI. Some of the frameworks we refer to in this dissertation thesis were also briefly analysed in Section 4.3. These two section helped us to understand the challenges concerning patterns and related technologies. We realised the complexity grows with each new pattern what might be the reason for their imprecise implementation and dissensus in their definition. Finally, we wanted to meet the research objective RO 1.3 from slightly different angle. In Section 4.5, we wondered how the current RPA and BPM technology may support technology transitions. Again, we undertook a comprehensive review of corresponding tools, practices, and challenges they tackle. It was clarified that BPM can

help organisations to increase their maturity to the level when their business processes are known and correctly managed. This by itself does not solve the problem of technology transitions, however, it builds a great basis for further technological innovations. RPA saves the labour time by automating routine activities that can be later bridged to BPM.

Finally, the RO 1.4 was met in Chapter 5. Here we briefly presented the linkage of our research with the approach of other researchers.

15.1.2 Research Objective RO 2

Our second objective was formulated as follows:

'To design a methodical framework aiming at controlled technology transition.'

In Chapter 12, we completed the framework ADA based on the careful integration of various concepts from EE. However, to come up with it, as the core of this research, multiple sub-objectives were defined and addressed. These include:

- *RO 2.1: To investigate how user-interactions can be captured in SW development.*
- *RO 2.2: To describe trade-off between flexibility and usability of SW components when having their function and construction devised.*
- *RO 2.3: To show how RPA technologies can bridge the gap between legacy SW and SW built with technology transition.*
- *RO 2.4: To demonstrate how to measure the evolvability of systems.*
- *RO 2.5: To describe architecture concepts that limit GUI transitions.*

By responding the research objectives above, we also answer the research question RQ 2. For achieving the sub-objective RO 2.1, in Chapter 7, we presented our article focused on adapting PSI theory to support a confirmation principle in Corima. This was our first attempt to map EE theory into SW. It further highlighted the value of a business process in an environment where people interact, in this particular case a confirmation process where people interact to confirm certain financial instruments in Corima. Whenever the user input was required in this process, the GUI was generated for a specific employee to fulfil this input. It was realised that this may help to move the GUI from one technology to another. Regardless the changing technology, the process remains the same. These were the first observations that made us curious of how to apply the same principle in SW development systematically. According to the findings, a number of knowledge gaps and problems in this research were unveiled. It was unclear how to describe GUI independently from the technology, so that it may get orchestrated from various components of frameworks that we do not know yet.

Therefore, we formulated a second research sub-objective RO 2.2. It was obvious that the flexibility of the GUI components used in the generator plays a significant role. However, we noticed that their complexity increases with their flexibility. Furthermore, the more flexible the components are, the more difficult it is to use them. This trade-off should be kept in mind while designing reusable GUI components.

In parallel to tackling the first two sub-objectives, we started a separate stream of our research. While we were moving forward designing a methodical framework, we identified a gap in terms of how the industry may move the legacy SW to a SW designed for better technology transitions. We defined our next research sub-objective RO 2.3. We investigated RPA and BPM technology that are trending on the market. A thorough systematic review of the existing body of knowledge was already undertaken in Section 4.5.3 to understand their state-of-the-art. We implemented a solution bridging the work with legacy SW into BPM having the GUI generated based on the similar principle as we used for the aforementioned confirmation process.

Since our focus is on technology transitions, we had to understand how to measure them. Therefore, we defined our next research sub-objective RO 2.4. We measured CEs in a complex financial risk management field. In conjunction with the University of Antwerp, we established a domain model of a certain area of risk management. We published a joined paper discussing its changeability and CEs. This gave us the first practical experience with NST and measurements of CEs.

Next, we put forward our last research sub-objective RO 2.5. In Section 4.2, a systematic literature review was employed to identify GUI patterns. The output of this investigation underlined many design and architecture patterns used across variety of GUI frameworks. It revealed their gradual complexity, dissensus in their definition, and imprecision in their implementation. To meet this research sub-objective, we outlined a transition of a simple application from one technology to another. On that example, we pointed out the difficult parts of the process of transitioning between two technologies. We evaluated that with the help of NST and EA, and we provided an overview of all aspects that play a role in a GUI transition.

15.1.3 Research Objective RO 3

The third objective of this research included the development of a prototype system using ADA. It was formulated as follows:

> *'To design & develop a prototype of the designed framework in an industrial-scale system.'*

To achieve this objective, we took an industrial-scale financial application Corima. We implemented its infrastructure according to the ADA concepts. It included the implementation of components serving as ADA-users, ADA-purposes, ADA-components, and ADA-relation that are present in the ADA Definition 17. While this ADA infrastructure enabled Corima to transit between desktop and web technologies, the particular ADA-elements

supported filtering and editing data, viewing charts, and performing treasury operations such as managing risk, cash, cashflow, etc. The prototype facilitated the achievement of the research objective RO 2 of this research and was presented in detail in Chapter 13.

15.1.4 Research Objective RO 4

The last objective in this research was to evaluate the framework in a two-step evaluation process including

- *the demonstration of the designed framework in a real-life industrial-scale environment; and*
- *the validation of the framework from a theoretical point of view.*

These steps were further used to respond to research question RQ 2. For the first step of the evaluation, the application of the framework ADA was tested in a case study of Corima. The results were presented in Chapter 13 and showcased the strength of ADA and the developed prototype in Corima. The initial feedback from the industry in Section 14.1 illustrated the benefits of this framework for improving the technology transitions.

For achieving the verification aspect of the second component of the framework evaluation, its processes as well as its implementation were verified by two-step analysis (refer to Section 14.2 and Section 14.3). We calculated CEs in a proposed architecture of ADA-based system, and we provided measurements in a code-base of Corima.

The outcomes of the validation highlighted a number of limitations of ADA. Section 14.4 details them, and presents future directions where ADA can be further improved.

The presented accomplishments in each objective of this research led to achieving the goal of this research, and subsequently addressing the formulated research problem.

15.2 Responding to Research Problem

The research problem defined in Section 1.4 recognised that despite the benefits of existing GUI frameworks, organisations of different sizes are often lacking the capacity to adapt to, create, and leverage changes to use these latest technologies for their customer's benefit. Specifically, we formulated the research problem as follows:

> *'Despite the potential in modern technologies, SE do not offer guidance and architectural pattern on adapting software artefacts into the latest technologies in a more efficient and manageable manner.'*

To address the highlighted problem and underlined need for a more rigorous and detailed analysis of technology transitions, various technologies, methods, and methodologies were identified and a methodical framework ADA was designed. This framework could address the fundamental building blocks of the defined problem, namely:

- concepts that need to be present in the construction of a software solutions enabling controlled technology transition;
- the notion of affordance in such a software solution;
- the understanding of how to build a software architecture optimised for technology transitions.

The development and the evaluation of ADA advocated its feasibility of and effectiveness for the discussed GUI applications. It is a complement to the current SW development approaches (e.g., common best practices while using some of the GUI frameworks that are not directly optimised for technology transitions) where their capabilities fall short in providing desired outcomes. With consideration of their benefits, ADA could guide SW developers to see these frameworks as an intermediate technology that may be replaced in time. Therefore, ADA may facilitate more controlled transitions between them.

15.3 Main Outcomes and Contributions to Knowledge

This research led to a number of outcomes which are also considered as contributions to the knowledge. These contributions were made to different disciplines and include:

- Development of a new methodical framework that aids in the construction of software solutions enabling controlled technology transition.
- Development of a prototype system confirming the feasibility of the implementation of the framework.
- Clarifying the value of RPA and BPM when tackling challenges concerning technology transitions.
- Identification of a trade-off between flexibility and usability when designing SW solutions.
- Evaluating the evolvability of financial models.
- Understanding architectural concepts limiting transitions between GUI frameworks.
- Determining the role of technology transitions in a context of business agility.

Additionally, although this research primarily focused on software evolvability, the results of our analysis indicate that ADA can contribute to overcoming various challenges in SE. We discuss some of these challenges below.

15.3.1 Supporting an Agile Way of Working

As indicated in Chapter 2, to thrive in the presence of uncertainty, organisations of any size are encouraged to 'align their business strategy with their IT strategy' [140]. On one hand, they are 'advised to be agile across all levels' [161]. On the other hand, as organisations grow, they begin to limit the customisation opportunities in their software solutions significantly because they cannot accommodate the whims of every user individually. This leads to a situation where 'the justification for existence is the responsiveness to the individual needs of customers who cannot have their needs satisfied by mass-produced products'[203]. However, modern organisations can satisfy many customers through 'mass customisation', which is a term coined by Zipkin [234] addressing the capability to offer individually tailored products or services on a large scale. We argue that ADA can contribute to this area because it maps the purposes of users to user stories that change continuously, implying that it is well aligned with the agile way of working. Simultaneously, it maps user purposes to components that can be individualised according to user needs, meaning it also supports mass customisation.

15.3.2 Supporting Technological Transitions

As stated in the research overview in Chapter 1, there are many frameworks, technologies, and systems for creating GUI, and new systems are introduced almost daily. As indicated in the work by Mareš [A.13], systems are often developed based on specific architectural patterns, for such as MVC or the MVVM pattern, which are broadly identified as patterns contributing to the construction of evolvable software [150, 83, 137]. However, there is no clear consensus regarding their implementation or definition.

Therefore, together with Verelst et al. [137], we argue that design patterns themselves do not guarantee evolvable software. A more rigorous split between the technologies and functions of software must be achieved. In the case study presented in Chapter 13, we demonstrated how ADA can tackle this challenge by reflecting on the concepts of affordances in EE theories. However, as evaluated in Section 14.2, an ADA-based system is not completely CE free unless the three remaining NS theorems are addressed in its implementation.

15.3.3 Enhancing Model-Driven Engineering

Modern model-driven engineering efforts are mostly framed by the concept of Model-Driven Architectures (MDAs). Kleppe et al. [125] clarified that within MDAs, the SW development process is driven by the activities of modelling SW. SW development methods based on MDAs are collectively referred to as Model-Driven Development (MDD) [6].

ADA is aligned with this concept. However, ADA goes a step further because it proposes the assembly of software from components fitting given user purposes. To realise this goal, software components must be aware of the extent to which they suit specific combinations of ADA-users and ADA-purposes.

Although this direction has not been elaborated thoroughly, the potential of ADA is clear. When a developer wishes to identify relevant components in large open-source repositories of components, such as NPM, they must search for tags such as 'database persistence' and 'mongo'. Suppose that the components in a repository were semantically annotated with ADA-users and ADA-purposes. In this scenario, components could be automatically or semi-automatically matched to development needs in a more precise manner. This would be a step toward the ideal of assembling software from components that match the required purposes of a given user. From this perspective, we do not consider ADA to be a clear MDD because its main concept is that only models are ever manipulated and the code is always fully generated [122]. ADA is essentially a method supporting software evolvability by providing technologically independent descriptions of user purposes that can be mapped to components in different technologies.

15.4 Future Work

While conducting this research, a number of future research opportunities were identified. However, they could not be addressed within the scope of this research. Although some of them would contribute to addressing the limitations of ADA, others underline the areas to which this work can be extended. Below, we summarise some of those.

- In Section 14.4, we said that a prototype of an affordance-based system reflecting the needs of people with mental challenges was developed. Future extension of ADA would benefit from more detailed elaboration and concrete examples of these challenges in terms of ADA-users. This could be further extended with other, but mental challenges, thereby bringing cross-disciplinary contributions to build ADA-based systems directly reflecting various needs of users.

- This research limited the scope and accordingly only a justified subset of technologies, methodologies, etc., were researched. However, other technologies should be reviewed, and prototypes in different technology stacks should be implemented to prove the applicability of ADA.

- In Section 15.3.1, we explained a tight connection between ADA and agile way of working. This direction should also be inspected and a more tight connection between user stories and ADA should be proposed and exemplified.

- The future work may also focus on elaborating on the change drivers in ADA from additional perspectives. We could consider different drivers such as languages and technologies in different GUI stacks (web, desktop), which would enable the aforementioned potential for reasoning and automation.

- Finally, the unification of NSX expander principles with ADA could be an interesting endeavour in future studies on software evolvability.

Chapter 16

Thesis Summary

In this research, we began by observing that the pace of introducing new technologies is exponentially accelerating, which poses a serious challenge for the software industry, where systems are becoming obsolete increasingly rapidly. We addressed this challenge by developing and demonstrating how ADA can make technological transitions more efficient and manageable.

We formulated a possible high-level architecture for software systems following ADA and discussed it in the context of a case study on a real system. We introduced NST with a focus on improving software evolvability and used it to evaluate ADA theoretically. According to NST, we concluded that ADA improves the evolvability of software during technological transitions by defining clear boundaries for the impact of changes. Additionally, this architecture provides new ways of reasoning about and automating such transitions.

Similar to NST and its applications, the approach used in ADA is not entirely novel. It follows well-established SE best practices, but in a managed, guaranteed, and semi-automated, or even automated manner that eliminates opportunities for poor discipline and/or incompetence on the part of programmers.

Part VI

Publications

Bibliography

[1] Wikipedia, 2020. URL `https://wikipedia.com`.

[2] N. Abbasi, I. Wajid, Z. Iqbal, and F. Zafar. Project failure case studies and suggestion. *International Journal of Computer Applications*, 86(6), 2014.

[3] N. Alija. Justification of software maintenance costs. *International Journal*, 7:15–23, march 2017. doi: 10.23956/ijarcsse/V7I2/01207.

[4] S. Amirebrahimi. *A framework for micro level assessment and 3D visualisation of flood damage to a building*. PhD thesis, The University of Melbourne, Victoria, Australia, 2016.

[5] L. Antovski and F. Imeri. Review of software reuse processes. *International Journal of Computer Science Issues (IJCSI)*, 10(6):83, 2013.

[6] C. Atkinson and T. Kuhne. Model-driven development: a metamodeling foundation. *IEEE software*, 20(5):36–41, 2003.

[7] C. Atkinson, D. Stoll, and P. Bostan. Orthographic software modeling: a practical approach to view-based development. In *Evaluation of Novel Approaches to Software Engineering*, pages 206–219. Springer, 2009.

[8] K. I. Awa. Functional structure and operational issues: An examination of core challenges and remedies. *IOSR Journal of Business and Management*, 18(1):1–4, 2016.

[9] F. J. Ayala. *Evolution, explanation, ethics and aesthetics: towards a philosophy of biology*. Academic Press, 2016. ISBN 9780128037317.

[10] BAI Editorial Team, editor. *Journal of Business Agility Emergence*, volume 01. Sntio Press, 2020. ISSN 2694-5320.

[11] BAI Editorial Team, editor. *Journal of Business Agility Emergence*, volume 02. Sntio Press, 2021. ISSN 2694-5320.

[12] C. Baldwin and K. Clark. Managing in an age of modularity. *Harvard Business Review*, 75(5):84–93, September 1997.

[13] M. Barash. Specifying software languages: Grammars, projectional editors, and unconventional approaches. In *Norsk IKT-konferanse for forskning og utdanning*, number 1 in 1, 2020.

[14] F. Barbier, B. Henderson-Sellers, A. Le Parc-Lacayrelle, and J.-M. Bruel. Formalization of the whole-part relationship in the unified modeling language. *IEEE Transactions on software engineering*, 29(5):459–470, 2003.

[15] Basel Committee on Banking Supervision, (BIS). International convergence of capital measurement and capital standards. Standard, Bank for International Settlements, June 2006. URL `http://www.bis.org/publ/bcbs128.pdf`.

[16] Basel Committee on Banking Supervision, (BIS). Fundamental review of the trading book. Standard, Bank for International Settlements, May 2012. URL `http://www.bis.org/publ/bcbs219.pdf`.

[17] Basel Committee on Banking Supervision, (BIS). Minimum capital requirements for market risk. Standard, Bank for International Settlements, Jan. 2016. URL `http://www.bis.org/bcbs/publ/d352.pdf`.

[18] K. Beck. *Extreme Programming Explained: Embrace Change, 2nd Edition (The XP Series)*. Addison-Wesley, Nov. 2004. ISBN 9780321278654.

[19] M. Beer, M. Finnström, and D. Schrader. Why leadership training fails—and what to do about it. *Harvard Business Review*, 94(10):50–57, 2016.

[20] W. N. Behutiye, P. Rodríguez, M. Oivo, and A. Tosun. Analyzing the concept of technical debt in the context of agile software development: A systematic literature review. *Information and Software Technology*, 82:139 – 158, 2017. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2016.10.004.

[21] B. Bemer. The software factory principle. `http://www.bobbemer.com/`, 2001.

[22] K. Bennett. Legacy systems: Coping with success. *IEEE software*, 12(1):19–23, 1995.

[23] J. Bisbal, D. Lawless, B. Wu, and J. Grimson. Legacy information systems: Issues and directions. *IEEE software*, 16(5):103–111, 1999.

[24] B. W. Boehm. *Software Cost Estimation with CoCoMo II*. Prentice Hall, August 2000. ISBN 9780130266927.

[25] G. Booch. *The unified modeling language user guide*. Pearson Education India, 2005. ISBN 0321267974.

[26] M. Bunge. *Treatise on basic philosophy: Ontology II: A world of systems*, volume 4. Springer Science & Business Media, 2012. ISBN 978-94-009-9392-1.

[27] D. Campagnolo and A. Camuffo. The concept of modularity in management studies: A literature review. *International Journal of Management Reviews*, pages 259–283, 2010. ISSN 1468-2370. doi: 10.1111/j.1468-2370.2009.00260.x.

[28] Camunda. Camunda BPM products [online], 2020. URL `https://camunda.com/products/`. [Cited 2020-04-20].

[29] S. A. Carlsson, S. Henningsson, S. Hrastinski, and C. Keller. Socio-technical is design science research: developing design theory for is integration management. *Information Systems and e-Business Management*, 9(1):109–131, 2011.

[30] A. Cater-Steel, M. Toleman, and M. M. Rajaeian. Design science research in doctoral projects: An analysis of australian theses. *Journal of the Association for Information Systems*, 20(12):3, 2019.

[31] O. Chongsombut, J. Verelst, P. De Bruyn, H. Mannaert, and H. Philip. Towards applying normalized systems theory to create evolvable enterprise resource planning software: a case study. In *The Eleventh International Conference on Software Engineering Advances : ICSEA, Rome, Italy*, pages 172–177, August 2016.

[32] S. Ciraci and P. V. D. Broek. Evolvability as a quality attribute of software architectures. *Journal of Physics Conference Series*, pages 29–31, 2006.

[33] C. L. Clair, G. O'Donnell, A. Lipson, and D. Lynch. The Forrester Wave: Robotic Process Automation, the 15 providers that matter most and how they stack up. Technical report, Forrester, 2019.

[34] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming, 2nd edition*. Addison-Wesley, 2002. ISBN 978-0321753021.

[35] CMMI Institute. What is CMMI [online], 2020. URL `https://cmmiinstitute.com/cmmi/intro`. [Cited 2020-05-15].

[36] E. F. Codd. Relational database: a practical foundation for productivity. *Communications of the ACM*, 25(2):109–117, 1982.

[37] M. Cohn. *User stories applied: For agile software development*. Addison-Wesley Professional, 2004. ISBN 0321205685.

[38] M. E. Conway. How do committees invent? *Datamation*, pages 28–31, Apr. 1968. URL `http://www.melconway.com/Home/pdf/committees.pdf`.

[39] S. Cook, H. Ji, and R. Harrison. Software evolution and software evolvability. *University of Reading, UK*, pages 1–12, 2000.

[40] COPS Financial Systems s.r.o., 2020. `https://cops.solutions`.

[41] COPS GmbH. *Corima: Treasury Management System*, 2020. `https://corima.solutions`.

[42] J. Cordeiro. Analysing enterprise ontology and its suitability for model-based software development. In *International Symposium on Business Modeling and Software Design*, pages 257–269. Springer, 2019.

[43] B. J. Cox. *Object-Oriented Programming: an evolutionary approach*. Addison-Wesley, 1986. ISBN 0201548348.

[44] J. Crotty and I. Horrocks. Managing legacy system costs: A case study of a meta-assessment model to identify solutions in a large financial services company. *Applied computing and informatics*, 13(2):175–183, 2017.

[45] R. M. Curtice, R. Donabue, and J. C. Weiss. Enterprise systems: A report from the field. *PRISM-CAMBRIDGE MASSACHUSETTS*, pages 39–54, 1997.

[46] Cycle.JS. Model-view-intent, [online], 2019. URL `https://cycle.js.org/model-view-intent.html`. [cit. 2019-03-19].

[47] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured programming*. Academic Press Ltd., 1972. ISBN 0122005503.

[48] David Garmus. *Function Point Analysis: Measurement Practices for Successful Software Projects., 1th edition*. Addison-Wesley Professional, 2000. ISBN 0201699443.

[49] J. Davies, D. Milward, C.-W. Wang, and J. Welch. Formal model-driven engineering of critical information systems. *Science of Computer Programming*, 103:88–113, 2015.

[50] T. de Bruin and G. Doebeli. Transitioning from functional silos to process centric-learnings from australian organizations. *BPTrends*, 2008.

[51] P. De Bruyn. *Generalizing normalized systems theory: towards a foundational Theory for enterprise engineering*. PhD thesis, University of Antwerp - Faculty of Business and Economics, 2014.

[52] P. De Bruyn, D. Van Nuffel, J. Verelst, and H. Mannaert. Towards applying normalized systems theory implications to enterprise process reference models. *Lecture Notes in Business Information Processing*, pages 31–45, 2012. doi: 10.1007/978-3-642-29903-2_3.

[53] P. De Bruyn, H. Mannaert, J. Verelst, and P. Huysmans. Enabling normalized systems in practice–exploring a modeling approach. *Business & Information Systems Engineering*, 60(1):55–67, 2018.

[54] C. Décosse, W. A. Molnar, and H. A. Proper. What does demo do? a qualitative analysis about demo in practice: founders, modellers and beneficiaries. In *Enterprise Engineering Working Conference*, pages 16–30. Springer, 2014.

[55] Der Tresurer. The 2020 TMS market leaders. *Der Treasurer*, 12, 2020.

[56] DevExpress, 2019. URL `https://www.devexpress.com/#ui`. [cit. 2019-03-29].

[57] S. di Paola, E. Cohen, and I. Farrar. Global treasury benchmarking survey, digital treasury - it takes two to tango. Technical report, PwC ILP, 2019.

[58] J. Dietz. *Enterprise Ontology: Theory and Methodology*. Springer, May 2006. ISBN 3540291695.

[59] J. Dietz. *Red garden gnomes don't exist*. The Netherlands: Sapio Enterprise Engineering, 2012. ISBN 9081544926. URL `www.sapio.nl`.

[60] J. Dietz. The FI theory - understanding information and factual knowledge. Technical report, Delft University of Technology, October 2017.

[61] J. Dietz. The PSI theory - understanding human collaboration. Technical report, Delft University of Technology, October 2017.

[62] J. Dietz and J. Hoogervorst. BETA theory: Theories in Enterprise Engineering Memorandum. Technical report, CIAO! Enterprise Engineering Network (CEEN), 2014.

[63] J. Dietz and J. Hoogervorst. TAO theory: Theories in Enterprise Engineering Memorandum. Technical report, CIAO! Enterprise Engineering Network (CEEN), 2014.

[64] J. Dietz and J. Hoogervorst. Technical report TR-FIT-15-01. Technical report, CIAO! Enterprise Engineering Network (CEEN), 2015.

[65] J. Dietz and H. Mulder. *The DEMO Methodology*, pages 261–299. Springer International Publishing, Cham, 2020. ISBN 978-3-030-38854-6. doi: 10.1007/978-3-030-38854-6_12. URL `https://doi.org/10.1007/978-3-030-38854-6_12`.

[66] J. Dietz and H. B. Mulder. Introduction to Enterprise Ontology. In *Enterprise Ontology*, pages 13–19. Springer, 2020. ISBN 978-3-030-38853-9.

[67] J. Dietz and H. B. Mulder. The PSI theory: Understanding the operation of organisations. In *Enterprise Ontology*, pages 119–157. Springer, 2020.

[68] Doughlas McIlroy. *Mass produced software components*. NATO Conference on Software Engineering, Garmish, Germany, 1968.

[69] N. B. Duncan. Capturing flexibility of information technology infrastructure: A study of resource characteristics and their measure. *Journal of management information systems*, 12(2):37–57, 1995.

[70] E. Eesaar. On applying normalized systems theory to the business architectures of information systems. *Baltic J. Modern Computing*, 2(3):132–149, 2014.

[71] E. Eesaar. The database normalization theory and the theory of normalized systems: finding a common ground. *Baltic J. modern computing*, 4(1):5–33, 2016.

[72] H. A. ElMaraghy. Flexible and reconfigurable manufacturing systems paradigms. *International journal of flexible manufacturing systems*, 17(4):261–276, 2005.

[73] P. Ensor. The functional silo syndrome. *AmE Target*, 16(Spring Issue):16, 1988.

[74] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, USA, 2005. ISBN 0131858580.

[75] E. Evans and E. J. Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004. ISBN 0321125215.

[76] X. Ferré, N. Juristo, H. Windl, and L. Constantine. Usability basics for software developers. *IEEE software*, 18(1):22, 2001. doi: 10.1109/52.903160.

[77] J. Fiala and R. Kočí. Computer as Therapy in role of alternative and augmentative communication. In *Proceedings of 4th International Conference on Advanced in Computing and Emerging E-Learning Technology*, pages 34–42, 2015.

[78] R. T. Fielding. Chapter 5: Representational state transfer (rest). In *Architectural Styles and Design of Network-based Software Architecture*, pages 76–106. University of California, Irvine, 200.

[79] G. Figueiredo, A. Duchardt, M. M. Hedblom, and G. Guizzardi. Breaking into pieces: An ontological approach to conceptual model complexity management. In *2018 12th International Conference on Research Challenges in Information Science (RCIS)*, pages 1–10. IEEE, 2018.

[80] L. Fischer. *Delivering BPM Excellence: Business Process Management in Practice*. Excellence in practice series. Future Strategies Incorporated, 2011. ISBN 9780981987095.

[81] D. M. Fisher. Getting started on the path to process-driven enterprise optimization. *BP Trends*, 2005, 2005.

[82] B. Foote and J. Yoder. Big ball of mud. In *Pattern Languages of Program Design*, pages 653–692. Addison-Wesley, 1999.

[83] N. Ford, R. Parsons, and P. Kua. *Building Evolutionary Architectures: Support Constant Change*. O'Reilly Media, Inc., 2017. ISBN 1491986360.

[84] M. Fowler. GUI architectures, [online], 2019. URL `https://www.martinfowler.com/eaaDev/uiArchs.html`. [cit. 2019-02-28].

[85] M. Fowler. Presentation Model, [online], 2019. URL `https://www.martinfowler.com/eaaDev/PresentationModel.html`. [cit. 2019-03-01].

[86] M. Fowler. Strangler Application, [online], 2019. URL `https://martinfowler.com/bliki/StranglerApplication.html`. [cit. 2019-11-04].

[87] C. Fuchs. *Internet and society. Social theory in the information age*. Routledge, 2008. ISBN 0415961327.

[88] P. Gajender, K. Manish, and B. Kuldeep. A review paper on Cocomo model. *International Journal of Research & Development Organisation*, 1(Issue 4.):83–87, 2014.

[89] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 75 Arlington Street, Suite 300, Boston, 1995. ISBN 0201633612.

[90] E. Gamma, J. Vlissides, R. Johnson, and R. Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc, 1998. ISBN 0201455633.

[91] Gartner. Robotic Process Automation (RPA) [online], 2020. URL `https://www.gartner.com/en/information-technology/glossary/robotic-process-automation-rpa`. [Cited 2020-02-05].

[92] G. L. Geerts. A design science research methodology and its application to accounting information systems research. *International Journal of Accounting Information Systems*, 12(2):142–151, 2011.

[93] A. Gemino and D. Parker. Use case diagrams in support of use case modeling: Deriving understanding from the picture. *Journal of Database Management*, 20(1): 1–24, Jan. 2009. doi: 10.4018/jdm.2009010101. URL `https://doi.org/10.4018/jdm.2009010101`.

[94] J. K. Gershenson, G. J. Prasad, and Y. Zhang. Product modularity: Definitions and benefits. *Journal of Engineering Design*, 14(3):295–313, 2003. doi: 10.1080/0954482031000091068. URL `http://dx.doi.org/10.1080/0954482031000091068`.

[95] R. E. Giachetti. *Design of enterprise systems: Theory, architecture, and methods*. CRC Press, 2011. ISBN 9781032099439.

[96] Google. Google trends - wpf, winforms, [online], 2019. URL `https://trends.google.com/`. [cit. 2019-03-29].

[97] S. Guerreiro, S. J. van Kervel, A. Vasconcelos, and J. Tribolet. Executing enterprise dynamic systems control with the demo processor: the business transactions transition space validation. In *Mediterranean Conference on Information Systems*, pages 97–112. Springer, 2012.

[98] G. Guizzardi, R. de Almeida Falbo, and R. S. Guizzardi. Grounding Software Domain Ontologies in the Unified Foundational Ontology (UFO): The case of the ODE Software Process Ontology. In *CIbSE*, pages 127–140, 2008.

[99] J. Habermas. *The Theory of Communicative Action: Lifeworld and Systems, a Critique of Functionalist Reason, Volume 2*, volume 2. John wiley & sons, 2015.

[100] P. Harmon. The State of Business Process Management. Technical report, Red Hat, 2018.

[101] G. T. Haugan. *Effective work breakdown structures*. Berrett-Koehler Publishers, October 2001. ISBN 1567261353.

[102] A. R. Hevner. A three cycle view of design science research. *Scandinavian journal of information systems*, 19(2):4, 2007.

[103] A. R. Hevner, S. T. March, J. Park, and S. Ram. Design science in information systems research. *MIS quarterly*, pages 75–105, 2004.

[104] M. Hilbert and P. López. The world's technological capacity to store, communicate, and compute information. *Science*, 332(6025):60–65, 2011. ISSN 0036-8075. doi: 10.1126/science.1200970.

[105] J. Hintzen, S. J. Van Kervel, T. Van Meeuwen, J. Vermolen, and B. Zijlstra. A professional case management system in production, modeled and implemented using demo. In *Proceedings of 16th IEEE Conference on Business Informatics*, volume 1182, pages 1613–0073, 2014.

[106] House of Commons Treasury Committee. IT failures in the financial services sector, second report of session 2019–20. Technical report, 2019.

[107] Huysmans. *On the feasibility of Normalized Enterprises: applying Normalized Systems Theory to the high-level design of enterprises*. PhD thesis, University of Antwerp, 2011.

[108] P. Huysmans, K. Ven, and J. Verelst. Modularity in enterprise architecture projects: An exploratory case study. In *Enterprise Engineering Working Conference*, pages 106–120. Springer, 2011.

[109] P. Huysmans, J. Verelst, and H. M. A. Oost. Integrating information systems using normalized systems theory : four case studies. In *17th IEEE Conference on Business Informatics, JUL 13-16, 2015, Lisbon, Portugal*. IEEE, 2015. doi: 10.1109/CBI.2015.43.

[110] J. Iivari. A paradigmatic analysis of information systems as a design science. *Scandinavian Journal of Information Systems*, 19:5, 2007.

[111] ISO. Ergonomic requirements for office work with visual display terminals. Standard, International Organization for Standardization, Geneva, CH, 1998.

[112] ISO. Quality Management Systems — Fundamentals and vocabulary. Standard, International Organization for Standardization, Geneva, CH, 2005.

[113] L. James. A watershed moment for payments. Technical report, The Record, 2020.

[114] James W. Hooper, Rowena O. Chester. *Software Reuse: Guidelines and Methods (Software Science and Engineering)*. Springer, 1991.

[115] G. J.J. *The Theory of Affordances. In Perceiving, Acting and Knowing. Towards an Ecological Psychology*. Hoboken, NJ: John Wiley & Sons Inc., 1977.

[116] J. Johnson and H. Mulder. Endless modernization. Technical report, The Standish Group International, Incorporated, 2020.

[117] T. Joseph. What Makes a Business Process Apt for Automation [online], 2019. URL `https://www.fingent.com/blog/what-makes-a-business-process-apt-for-automation/`. [Cited 2020-05-01].

[118] A. Josey, M. Lankhorst, I. Band, H. Jonkers, and D. Quartel. An introduction to the archimate® 3.0 specification. *White Paper from The Open Group*, 2016.

[119] S. H. Kaisler. *Software Paradigms*. Wiley-Interscience, April 2008. ISBN 0471483478.

[120] C. Kapil, P. Frank, and S. Ishaan. Memo to the CFO: Get in front of digital finance—or get left back. Technical Report 67, McKinsey, July 2018.

[121] D. Karagiannis. Bpms: Business Process Management Systems. *ACM SIGOIS Bulletin*, 16(1):10–13, 1995.

[122] S. Kelly and J.-P. Tolvanen. Visual domain-specific modelling: Benefits and experiences of using metacase tools. In *International Workshop on Model Engineering, at ECOOP*, volume 2000, pages 1–9. Citeseer, 2000.

[123] M. Kirchmer and P. Franz. Value-driven robotic process automation (RPA). In B. Shishkov, editor, *Business Modeling and Software Design*, pages 31–46, Cham, 2019. Springer International Publishing. ISBN 978-3-030-24854-3.

[124] H. K. Klein and M. D. Myers. A set of principles for conducting and evaluating interpretive field studies in information systems. *MIS quarterly*, pages 67–93, 1999.

[125] A. G. Kleppe, J. Warmer, J. B. Warmer, and W. Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[126] KPMG. Robotic Process Automation (RPA): On Entering an Age of Automation of White-collar Work Through Advances in AI and Robotics. Technical report, KPMG Consulting Co., Ltd, 2018.

[127] G. E. Krasner. A cookbook for using model-view-controller user interface paradigmin smalltalk-80. *J. Object Oriented Programming*, 1(3):26–49, 1988.

[128] R. Kurzweil. The law of accelerating returns. In *Alan Turing: Life and legacy of a great thinker*, pages 381–416. Springer, 2004.

[129] R. Kurzweil. *The singularity is near*. Penguin Books, 2006. ISBN 0143037889.

[130] Laiye. How AI is redefining RPA, 2021. URL `https://laiye.com/en/blog/how-ai-is-redefining-rpa.html`. [Cited 2021-08-01].

[131] M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept. 1980. ISSN 0018-9219. doi: 10.1109/PROC.1980.11805.

[132] W. Lidwell, K. Holden, and J. Butler. *Universal principles of design, revised and updated: 125 ways to enhance usability, influence perception, increase appeal, make better design decisions, and teach through design*. Rockport Pub, 2010.

[133] M. A. Linton, J. M. Vlissides, and P. R. Calder. Composing user interfaces with interviews. *Computer*, 22(2):8–22, 1989.

[134] H. Mannaert and J. Verelst. *Normalized Systems: re-creating information technology based on laws for software evolvability*. Koppa, 2009. ISBN 978-90-77160-008.

[135] H. Mannaert, J. Verelst, and K. Ven. The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability. *Sci. Comput. Program.*, 76(12):1210–1222, Dec. 2011. ISSN 0167-6423. doi: 10.1016/j.scico.2010.11.009. URL `http://dx.doi.org/10.1016/j.scico.2010.11.009`.

[136] H. Mannaert, J. Verelst, and K. Ven. Towards evolvable software architectures based on systems theoretic stability. *Software: Practice and Experience*, 42(1):89–116, 2012. doi: 10.1002/spe.1051.

[137] H. Mannaert, J. Verelst, and P. De Bruyn. *Normalized Systems Theory, From Foundations for Evolvable Software Towards a General Theory for Evolvable Design*. Normalized Systems Institute, 2016.

[138] S. T. March and G. F. Smith. Design and natural science research on information technology. *Decision support systems*, 15(4):251–266, 1995.

[139] o. Marketline. Mobile apps in the United States, Feb. 2016. URL www.marketline.com.

[140] L. Mathiassen and J. Pries-Heje. Business agility and diffusion of information technology. *European Journal of Information Systems*, 2006. doi: 10.1057/palgrave.ejis.3000610.

[141] Merriam Webster, [online], 2021. URL https://www.merriam-webster.com/.

[142] Michael A. Cusumano. *An Entry for the Encyclopedia in Software Engineering*. Massachusetts Institute of Technology, Sloan school, 1991.

[143] B. M. Michelson. Event-driven architecture overview. *Patricia Seybold Group*, 2(12): 10–1571, 2006.

[144] Microsoft. Windows Forms, [online], 2019. [cit. 2019-03-29].

[145] Microsoft. Windows presentation foundation, [online], 2019. URL https://docs.microsoft.com/en-us/dotnet/framework/wpf. [cit. 2019-03-29].

[146] Microsoft. Introduction to model/view/viewmodel pattern for building wpf apps, [online], 2019. URL https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/. [cit. 2019-03-02].

[147] Microsoft. Partial classes and methods (c# programming guide), 2021. URL https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/partial-classes-and-methods. [Cited 2021-06-19].

[148] P. Mike. MVP: Model-view-presenter the taligent programming model for c++ and java, [online], 2019. URL http://www.wildcrest.com/Potel/Portfolio/mvp.pdf. [cit. 2019-03-01].

[149] R. Miller. Camunda hauls in $28m investment as workflow automation remains hot [online], 2018. URL https://techcrunch.com/2018/12/05/camunda-hauls-in-28m-investment-as-workflow-automation-remains-hot/. [Cited 2020-04-20].

[150] T. Mitsa. An evolvable software framework for an internet-based telediagnostic system. In *4th International IEEE EMBS Special Topic Conference on Information Technology Applications in Biomedicine*, pages 203–206. IEEE, 2003.

[151] A. Multiple. RPA market size and popular vendors in 2021, 2021. URL https://research.aimultiple.com/rpa-market/. [Cited 2021-07-27].

[152] M. Naab and J. Stammel. Architectural flexibility in a software-system's life-cycle: systematic construction and exploitation of flexibility. In *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures*, pages 13–22. ACM, 2012.

[153] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen. *Microservice architecture: aligning principles, practices, and culture.* O'Reilly Media, Inc., 2016. ISBN 1491956259.

[154] M. Narumoto, D. Lee, E. Price, A. Buck, and N. Peterson. Anti-corruption layer pattern, [online], 2017. URL `https://docs.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer`. [cit. 2019-11-04].

[155] NSX. *NSX: Normalized Systems*, 2020 (accessed November 21, 2020). `https://normalizedsystems.org`.

[156] OASIS. OData version 4.0. part 1: Protocol plus errata 03 [online], 2020. URL `http://docs.oasis-open.org/odata/odata/v4.0/errata03/os/complete/part1-protocol/odata-v4.0-errata03-os-part1-protocol-complete.html`. [cit. 2020-05-18].

[157] H. Olfat. *Automatic spatial metadata updating and enrichment.* PhD thesis, The University of Melbourne, 2013.

[158] OMG. Business model process and notation (BPMN), 2011. URL `https://www.omg.org/spec/BPMN/2.0/PDF`. [Cited 2020-04-28].

[159] OMG. Semantics of business vocabulary and business rules. Technical report, Object Management Group, 2015.

[160] S. [online], 2019. URL `https://www.syncfusion.com/`. [cit. 2019-03-29].

[161] G. Oorts, P. Huysmans, P. De Bruyn, H. Mannaert, J. Verelst, and A. Oost. Building evolvable software using normalized systems theory: A case study. In *2014 47th Hawaii International Conference on System Sciences*, pages 4760–4769. IEEE, 2014.

[162] G. Oorts, H. Mannaert, P. De Bruyn, and I. Franquet. *On the Evolvable and Traceable Design of (Under)graduate Education Programs*, pages 86–100. Springer International Publishing, Cham, 2016. ISBN 978-3-319-39567-8. doi: 10.1007/978-3-319-39567-8_6.

[163] M. Op't Land and J. Dietz. Benefits of enterprise ontology in governing complex enterprise transformations. In *Enterprise Engineering Working Conference*, pages 77–92. Springer, 2012.

[164] M. Op't Land, M. R. Krouwel, E. van Dipten, and J. Verelst. Exploring normalized systems potential for dutch mod's agility. In *Working Conference on Practice-Driven Research on Enterprise Transformation*, pages 110–121. Springer, 2011.

[165] M. Op't Land, M. R. Krouwel, E. van Dipten, and J. Verelst. Exploring normalized systems potential for dutch mod's agility. In *Working Conference on Practice-Driven Research on Enterprise Transformation*, pages 110–121. Springer, 2011.

[166] OrangeScape Technologies. Business Process Management Software (BPMS) – everything you need to know [online], 2019. URL `https://kissflow.com/bpm/what-is-bpms/`. [Cited 2020-05-02].

[167] S. Ossowski. *Agreement technologies*, volume 8. Springer Science & Business Media, 2012.

[168] N. Palmer. What is BPM? [online], March 26, 2014. URL `https://bpm.com/what-is-bpm`. [cit. 2020-04-20].

[169] H. Paul. Strangler applications, [online], 2019. URL `https://paulhammant.com/2013/07/14/legacy-application-strangulation-case-studies/`. [cit. 2019-15-04].

[170] K. Peffers, T. Tuunanen, C. E. Gengler, M. Rossi, W. Hui, V. Virtanen, and J. Bragge. The design science research process: A model for producing and presenting information systems research. In *First International Conference on Design Science Research in Information Systems and Technology*, pages 83–16, 2006.

[171] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

[172] R. Pergl. *Conceptualisation: Chapters from Harmonising Enterprise and Software Engineering*. PhD thesis, Habilitation thesis, Faculty of Information Technology, Czech Technical University, april 2019.

[173] K. Preiss, S. Goldman, and R. Nagel. Agile competitors and virtual organizations. *Strategies for enriching the customer*, 1, 1995.

[174] PwC. Digital masters. Technical report, PwC ILP, November 2019.

[175] o. Quantitative Software Management. Function point languages table, 2021. URL `http://www.qsm.com/resources/function-point-languages-table`. [cit. 2021-07-27].

[176] S. Ray, C. Tornbohm, D. Miers, and M. Kerremans. Magic Quadrant for Robotic Process Automation Software. Technical report, Gartner Research, 2019.

[177] M. Rayner, B. A. Hockey, N. Chatzichrisafis, and K. Farrell. OMG unified modeling language specification. In *Version 1.3,© 1999 Object Management Group, Inc*, page 710. Citeseer, 2005.

[178] S. Reynard. *Flowcharts: Plain and Simple*. Oriel Incorporated, 1995. ISBN 1884731031.

[179] M. Richards. *Software architecture patterns*, volume 4. O'Reilly Media, Inc., 2015. ISBN 9781491924242.

[180] Rober William Bemer. *The economics of program production*. Proc. IFIP Congress 68, Booklet I, 13-14, 1968.

[181] Rossum.ai. Data extraction from business documents [online], 2020. URL `https://rossum.ai/`. [cit. 2020-04-26].

[182] Rubén Prieto Díaz. *Status Report: Software Reusability*. IEEE Software, 10(3):61-66, May, 1993.

[183] M. Sako. *The Business of Systems Integration*, chapter Modularity and Outsourcing, pages 229–253. Oxford University Press, 2003.

[184] R. Sanchez and J. T. Mahoney. Modularity, flexibility, and knowledge management in product and organization design. *Strategic Management Journal*, pages 63–76, 1996. ISSN 1097-0266. doi: 10.1002/smj.4250171107.

[185] A. Saunders and M. Cornett. *Financial institutions management. A risk management approach*. McGraw-Hill, 7th ed. edition, 2011.

[186] I. Schlueter, L. Voss, and R. Boothby. Npm. `https://www.npmjs.com`, 2016.

[187] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing legacy systems: software technologies, engineering processes, and business practices*. Addison-Wesley Professional, 2003.

[188] Self. Self programming language, 2021. URL `http://www.selflanguage.org/`.

[189] M. Shahin, M. A. Babar, and L. Zhu. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.

[190] S. Sharma. Ovum decision matrix: Selecting a Robotic Process Automation (RPA) platform, 2018–19. Technical report, Ovum TMT Intelligence, 2018-2019.

[191] B. Silver. *BPMN method and style*. Cody-Cassidy Press, 2009.

[192] H. Simon. The architecture of complexity. *Proceedings of the American Philosophical Society*, 106, 1962.

[193] H. Simon. *The Sciences of Artificial, Cambridge MA and London*. The MIT Press, 1996.

[194] I. Simple. Invoicing software [online], 2020. URL https://www.invoicesimple.com/. [cit. 2020-04-26].

[195] B. Smith. Object-Oriented Programming. In *Advanced ActionScript 3*, pages 1–23. Springer, 2015.

[196] M. Sobers and Y. Petras. Staying relevant 2020 hot topics for it internal audit in financial services. Technical report, Deloitte LLP, London EC4A 3HQ, United Kingdom, 2019.

[197] Software Robotic. The difference between front office and back office robots – and why this is important to understand [online], 2020. URL https://softwarerobotics.blog/the-difference-between-front-office-and-back-office-robots-and-why-this-is-important-to-understand/. [Cited 2020-05-01].

[198] J. Spolsky. Things you should never do, part one. In *Joel on Software*, pages 183–187. Springer, 2004.

[199] M. Srinivas, G. Ramakrishna, K. R. Rao, and E. S. Babu. Analysis of legacy system in software application development: A comparative survey. *International Journal of Electrical and Computer Engineering (IJECE)*, 6(1):292, Feb. 2016. doi: 10.11591/ijece.v6i1.8367.

[200] I. C. Station. Robotic Process Automation (RPA), buyer's guide and reviews. Technical report, IT Central Station, January 2020.

[201] P. Steve Burbeck. Applications programming in smalltalk-80 (TM): How to use model-view-controller (MVC), [online], 2019. URL https://web.archive.org/web/20120729161926/http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html. [cit. 2019-03-01].

[202] C. Stevenson and A. Pols. An agile approach to a legacy system. In *XP 2004: Extreme Programming and Agile Processes in Software Engineering (EDS)*, pages 123–129. Springer, 2004. URL http://cdn.pols.co.uk/papers/agile-approach-to-legacy-systems.pdf.

[203] C. Svensson and A. Barfod. Limits and opportunities in mass customization for "build to order" SMEs. *Computers in industry*, 49(1):77–89, 2002.

[204] A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys (CSUR)*, 28 (3):438–479, 1996.

[205] Telerik, 2019. URL https://www.telerik.com/. [cit. 2019-03-29].

[206] W. Tracz. Software reuse myths. *ACM SIGSOFT Software Engineering Notes*, 13 (1):17–21, 1988.

[207] T. T. Tun, T. Trew, M. Jackson, R. Laney, and B. Nuseibeh. Specifying features of an evolving software system. *Software: Practice and Experience*, 39(11):973–1002, 2009.

[208] UiPath. What is Robotic Process Automation [online], 2020. URL `https://www.uipath.com/rpa/robotic-process-automation`. [Cited 2020-05-01].

[209] UiPath. Robotic Process Automation [online], 2020. URL `https://www.uipath.com/company/about-us`. [Cited 2020-03-20].

[210] UiPath. UiPath RPA technologies [online], 2020. URL `https://www.uipath.com/solutions/technology`. [Cited 2020-03-20].

[211] United States Government Accountability Office. Information technology: Agencies need to develop modernization plans for critical legacy systems. Technical Report GAO-19-471, 1998. URL `https://www.gao.gov/assets/gao-19-471.pdf`.

[212] G. Valiente. *Algorithms on trees and graphs*. Springer Science & Business Media, 2013.

[213] J. E. Van Aken. Management research as a design science: Articulating the research products of mode 2 knowledge production in management. *British journal of management*, 16(1):19–36, 2005.

[214] D. Van Nuffel. *Towards desdesign modular and evolvable business processes*. PhD thesis, University of Antwerp, 2011.

[215] M. Van Oosterhout, E. Waarts, and J. van Hillegersberg. Change factors requiring agility and implications for it. *European Journal of Information Systems*, 15(2): 132–145, 2006.

[216] E. Vanhoof. *Evolvable accounting information systems: applying design science methodology and Normalized Systems theory to tackle combinatorial effects of multiple GAAP*. Universiteit Antwerpen, 2016.

[217] P. H. Vervest, E. Van Heck, K. Preiss, and L.-F. Pau. *Smart business networks*. Springer Science & Business Media, 2005.

[218] P. Vincent, V. Baker, Y. Natis, K. Iijima, M. Driver, R. Dunie, J. Wong, and A. Gupta. Magic Quadrant for Enterprise high-productivity application platform as a service. Technical report, Technical report, Gartner, 2018.

[219] S. Vinoski. Chain of Responsibility. *IEEE Internet Computing*, 6(6):80–83, 2002. doi: 10.1109/MIC.2002.1067742.

[220] M. Voelter and K. Solomatov. Language modularization and composition with projectional language workbenches illustrated with MPS. *Software Language Engineering, SLE*, 16(3), 2010.

[221] E. von Glaswersfeld. *Teleology and the Concepts of Causation*. Philosophica, 46 (2), 17–43, 1990.

[222] J. G. Walls, G. R. Widmeyer, and O. A. El Sawy. Building an information system design theory for vigilant EIS. *Information systems research*, 3(1):36–59, 1992.

[223] M. Ward, R. Nicholson, and C. Stephenson. 2019 CFO Survey report: All systems go: CFOs lead the way to digital world". Technical report, Grand Thornton LLP, 2019.

[224] R. Waszkowski. Low-code platform for automating business processes in manufacturing. *IFAC-PapersOnLine*, 52(10):376–381, 2019.

[225] G. Webber-Cross. *Learning Microsoft Azure*. Packt Publishing Ltd, 2014.

[226] A. Weinand, E. Gamma, and R. Marty. ET++ an object oriented application framework in C++. *ACM Sigplan Notices*, 23(11):46–57, 1988. doi: 10.1145/62084.62089.

[227] M. Wilkes. The edsac computer. In *Papers and discussions presented at the Dec. 10-12, 1951, joint AIEE-IRE computer conference: Review of electronic digital computers*, pages 79–83. ACM, 1951.

[228] M. Wilkes and D. Wheeler. *The Preparation of Programs for an Electronic Digital Computer*. Addison–Wesley, Edition 1, 1951. ASIN B0007DWTT0.

[229] WinForms. Github - topic: WinForms, [online], 2019. URL `https://github.com/topics/winforms`. [cit. 2019-03-29].

[230] M. Winters. End-to-end workflow automation with RPA and Camunda BPM [online], 2018. URL `https://blog.camunda.com/post/2018/05/combining-bpm-rpa-workflow-automation/`. [Cited 2020-04-26].

[231] WPF. Github - topic: WPF, [online], 2019. URL `https://github.com/topics/wpf`. [cit. 2019-03-29].

[232] WPF. Walkthrough: My first WPF desktop application, [online], 2019. [cit. 2019-04-04].

[233] L. Yu and S. Ramaswamy. Software and biological evolvability: a comparison using key properties. In *2006 Second International IEEE Workshop on Software Evolvability (SE'06)*, pages 82–88. IEEE, 2006.

[234] P. Zipkin. The limits of mass customization. *MIT Sloan management review*, 42(3): 81, 2001.

[235] Zkoss. ZK, [online], 2019. URL `https://www.zkoss.org/`. [cit. 2019-03-01].

Reviewed Publications of the Author Relevant to the Thesis

[A.1] Ondřej Dvořák, Robert Pergl, and Petr Kroha. Confirmation engine design based on PSI theory. In: *17th IEEE Conference on Business Informatics, Workshop on Cross-Organizational and Crosscompany BPM (XOC-BPM)*. Lisbon, Portugal, 2015.

The paper has been cited in:

- Duarte Gouveia and David Aveiro. Towards an Executable Artefact for Organizations based on DEMO Paradigm. In: *EEWC Doctoral Consortium*. Antwerp, Belgium, 2017.
- [42] José Cordeiro. Analysing Enterprise Ontology and Its Suitability for Model-Based Software Development. In: *International Symposium on Business Modeling and Software Design*. Springer, Lisbon, Portugal, 2019.

[A.2] Ondřej Dvořák, Applying EE Theories to Component-Based Software Design and Development. In: *Doctoral Consortium, Enterprise Engineering Working Conference*. Prague, Czech Republic, 2015.

[A.3] Marjolein Deryck, Ondřej Dvořák, Peter De Bruyn, and Jan Verelst Investigating the evolvability of financial domain models. In: *Enterprise Engineering Working Conference*. Springer, Antwerp, Belgium, 2017.

The paper has been cited in:

- [79] Guylerme Figueiredo, Amelie Duchardt, Maria Hedblom, and Giancarlo Guizzardi. Breaking into pieces: An ontological approach to conceptual model complexity management. In: *12th International Conference on Research Challenges in Information Science (RCIS)*, IEEE, Nantes, France, 2018.

[A.4] Ondřej Dvořák, Towards Semantic Descriptions of Component-Based Systems. In: *Doctoral Consortium, Enterprise Engineering Working Conference*. Antwerp, Belgium, 2017.

[A.5] Ondřej Dvořák, Robert Pergl, and Petr Kroha. Tackling the flexibility-usability trade-off in component-based software development. In: *World Conference on Information Systems and Technologies*. Springer, Porto Santo, Madeira, 2017.

The paper has been cited in:

- Anders Klingberg, Lee Alan Wallis, Marie Hasselberg, Po-Yin Yen, and Sara Fritzell. Teleconsultation using mobile phones for diagnosis and acute care of burn injuries among emergency physicians: Mixed-methods study. In: *JMIR mHealth and uHealth Journal*. JMIR Publications Inc., Toronto, Canada, 2018.

[A.6] Ondřej Dvořák, Robert Pergl, and Petr Kroha. Affordance-driven software assembling. In: *Enterprise Engineering Working Conference*. Springer, Luxembourg, 2018.

The paper has been cited in:

- Siamak Farshidi, Slinger Jansen, and Sven Fortuin. Model-driven development platform selection: four industry case studies. In: *Software and Systems Modeling Journal*, Springer, 2021.
- Siamak Farshidi, Slinger Jansen, and Mahdi Deldar. A decision model for programming language ecosystem selection: Seven industry case studies. In: *Information and Software Technology Journal*. Springer, 2021.
- Elena Baninemeh, Slinger Jansen, and Siamak Farshidi. A Decision Model for Decentralized Autonomous Organization Platform Selection: Three Industry Case Studies. In: *arXiv preprint arXiv:2107.14093*. 2021.
- Siamak Farshidi. Multi-Criteria Decision-Making in Software Production. In: *PhD Thesis*. Utrecht University, 2020.

[A.7] Ondřej Dvořák, Robert Pergl, and Petr Kroha. ADA: Embracing technology change acceleration. In: *CIAO! Doctoral Consortium and EEWC Forum and EEWC Posters 2019*. CEUR, Lisbon, Portugal, 2019.

[A.8] Ondřej Dvořák and Robert Pergl Tackling Rapid Technology Changes by Applying Enterprise Engineering Theories. In: *Submitted to Journal of Science of Computer Programming*. Elsevier, Expected Acceptance in 2021.

Remaining Publications of the Author Relevant to the Thesis

[A.9] Ondřej Dvořák Projectional editor for domain-specific languages, Charles University, Faculty of Mathematics and Physics, Prague, 2013.

The master thesis has been cited in:

- Mikhail Barash. Specifying Software Languages: Grammars, Projectional Editors, and Unconventional Approaches. In: *Norsk IKT-konferanse for forskning og utdanning*, 2020.

[A.10] Ondřej Dvořák Component-based framework for software development and design. Ph.D Minimum Thesis, Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2016.

Selected Relevant Supervised Theses

[A.11] Christián Golian Migration of relational databases using CodiScent's Projective Technologies. BSc Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2015.

[A.12] Martin Rašovský Language for high-level description of user interface requirements. MSc Thesis. Brno University of Technology, Faculty of Information Technology, Brno, Czech Republic, 2018.

[A.13] Václav Mareš Evolvability of UI technologies. MSc Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2019.

The master thesis has been awarded:

- Dean's award for the best master thesis of summer semester in the school year 2018/2019 on Czech Technical University in Prague, Faculty of Information Technology

[A.14] Ivana Nacevska The evolvability of technologies with the help of Robotic Process Automation. MSc Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2020.

[A.15] Tomáš Vahalik Robotic Process Automation in practice. MSc Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2021.

[A.16] Manasa Woolla RPA and OCR integration. MSc Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2021.

Selected Relevant Reviewed Theses

[A.17] Jakub Červenka Utilising projective technologies for object-oriented development of WEB UI. BSc Thesis. Czech Technical University in Prague, Faculty of Information Technology, Prague, Czech Republic, 2015.

[A.18] Richard Strnad Editor XML konfigurací modulárních systému. BSc Thesis. Brno University of Technology, Faculty of Information Technology, Brno, Czech Republic, 2017.