FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

The Art of (Large) System Migration in Early 21st Century IT

An Empirical Engineering Analysis of Large Scale Migration Case Studies, Their Solution Concepts and Derived Reasonable Abstractions

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften (Dr.techn.)

eingereicht von

Stefan Strobl
Matrikelnummer 0426201

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuer: Thomas Grechenig

Diese Dissertation haben begutachtet:

(Prof. Dr. Wolfgang Slany) (Prof. Dr. Andreas Rauber)

Wien, 29. Juli 2019

(Stefan Strobl) (Prof. Dr. Thomas Grechenig)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

The Art of (Large) System Migration in Early 21st Century IT

An Empirical Engineering Analysis of Large Scale Migration Case Studies, Their Solution Concepts and Derived Reasonable Abstractions

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

PhD

by

Stefan Strobl
Registration Number 0426201

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Thomas Grechenig

Vienna, July 29, 2019

Technische Universität Wien, Forschungsgruppe INSO
A-1040 Wien ▪ Wiedner Hauptstr. 76/2/2 ▪ Tel. +43-1-587 21 97 ▪ www.inso.tuwien.ac.at

Statement by Author

Stefan Strobl
Allißengasse 73/5, 1210 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

I hereby declare that I am the sole author of this thesis, that I have completely indicated all sources and help used, and that all parts of this work – including tables, maps and figures – if taken from other works or from the internet, whether copied literally or by sense, have been labelled including a citation of the source.

(Place, Date) (Signature of Author)

Acknowledgements

I am beyond grateful to all of my surroundings for making this thesis possible. It has required sacrifices and a lot of understanding from many different sides, too many to list everyone here. However, I would like to mention a few persons that were outstandingly supportive.

My wonderful family, my wife and my two kids, for both giving me the room to accomplish this and the support to keep me going as well as motivating and pushing me to finally deliver this to the finish line.

My mother who has tirelessly read over too many draft versions focusing on a professional language and presentation.

All the reviewers who have provided valuable feedback and especially my advisor who has shown incredible patience with me and belief in me during the whole journey.

Last but not least a big shout out to all my work and research colleagues. It is incredibly powerful to be part of such a great group.

A big thank you to all of you!

Kurzfassung

Regierungseinrichtungen, Organisationen und Industrie sind mit einer stetig wachsenden Herausforderung konfrontiert, die auf die langfristige und intensive Nutzung von Informationstechnologie (IT) zur Bewältigung ihres Geschäftsaufkommens zurückzuführen ist. Altsysteme sind die kodifizierte Inkarnation und das Vermächtnis von früheren Generationen an technologischer Handwerkskunst, die nach wie vor den Großteil aller Transaktionen weltweit abwickeln. Allerdings behindern deren Kerncharakteristika Innovation und Wachstum und konsumieren gleichzeitig überproportional viele Ressourcen. Die Migration dieser Altanwendungen wird nach wie vor als der nachhaltigste Weg betrachtet, um die genannten Defizite zu adressieren ohne eine vollständige Neuentwicklung zu riskieren. Trotz dieser Situation ist das Thema im akademischen Bereich und der Forschung zur Methodik und den Techniken im Software Engineering stark unterrepräsentiert. Dort werden nach wie vor vorrangig zunehmend unrealistische Greenfield-Szenarien betrachtet. Dies bedeutet für den Anwender, dass eine zunehmende Lücke zwischen Theorie und Praxis entsteht, die im Alltag auf den Unterschied zwischen Erfolg und Misserfolg eines Softwareprojekts hinauslaufen kann.

Diese Arbeit präsentiert eine Langzeitfallstudie, die vier Fälle jeweils über einen Zeitraum von mehreren Jahren, in Summe mehr als ein Jahrzehnt, betrachtet. Jedes Untersuchungsobjekt wurde ausgewählt, indem es an einem Kriterienkatalog gemessen wurde, der sicherstellen soll, dass es sich beim fraglichen System um ein Altsystem handelt. In weiterer Folge wird es detailliert analysiert, wobei sowohl technische als auch sozio-ökonomische Aspekte betrachtet werden. Alternative Ergebnisse werden diskutiert, um zentrale Entscheidungen zu unterstreichen, die den Ausgang des Projekts maßgeblich beeinflusst haben. In weiterer Folge werden aus den Fällen sowie aktueller Literatur gemeinsame, vorsichtige Abstraktionen und Generalisierungen herausgearbeitet, um übergreifende Gemeinsamkeiten und Lerneffekte hervorzuheben. Diese Abstraktionen werden in einem zweiten Schritt in Themengruppen zusammengefasst, die jeweils ein eigenständiges Resultat in Form eines Kernpunktes und dazugehörigen Herausforderungen darstellen. Gemeinsam können Themengruppen und zugehörige Generalisierungen als ein Vorschlag einer neuartigen Topologie für das Thema Altsystemmigration (Legacy System Migration) betrachtet werden.

Die resultierende detaillierte Beschreibung jedes Falls ermöglicht einen tiefgreifenden Einblick in die Entstehungsgeschichte jedes Untersuchungsobjekts. Die Herausforderungen, die dabei gemeistert werden mussten, reichen von Organisationen, die weder auf technischer noch organisatorischer Ebene für ein derartiges Projekt vorbereitet waren, bis hin zum ewigen Kampf gegen Königswege und Wunderwaffen beziehungsweise den dadurch verursachten, unangenehmen Nacharbeiten. Die resultierenden Abstraktionen und Generalisierungen verbinden Erfahrungen aus den einzelnen Fallbeispielen und kontrastieren oder erweitern diese soweit vorhanden mit dem aktuellen, akademischen Forschungsstand. Je nach Standpunkt des Lesers können diese auch als Erfahrungsbericht oder vorsichtige Empfehlungen gewertet werden. Gleichzeitig dienen die Abstraktionen auch dazu die thematische Breite der Altsystemmigration zu veranschaulichen. Die Arbeit wird mit der Gruppierung der Ergebnisse in fünf Themengruppen von grundlegenden technischen Mustern bis hin zu fundamentalen, strategischen Überlegungen abgerundet. Jede Gruppe trägt dabei eine zentrale Erkenntnis zum Ergebnis bei, die Erfolgsfaktoren hervorhebt und Herausforderungen im Feld diskutiert, die in zukünftiger Forschung adressiert werden sollten. Diese Topologie dient damit auch als zentrales Kommunikationsmedium zwischen Industrie und Forschung sowie gleichzeitig als Navigationshilfe für den Themenkomplex Altsystemmigration. Die

Zusammenfassung der Arbeit endet mit einem Aufruf gleichermaßen an Forscher und Anwender sowohl in der Forschung als auch in der Ausbildung enger und besser abgestimmt zusammenzuarbeiten. Ein deutlich stärkerer Fokus auf Themen Rund um die Behandlung von Altsystemen ist dringend erforderlich, um eine neuerliche Krise zu vermeiden.

Schlüsselwörter

Migration von Altsystemen, Altsystem, Software Reengineering, Topologie

Abstract

Governments, organisations and enterprises are facing an ever growing set of challenges stemming from their history of using IT systems to solve their business challenges. Legacy systems are the codified incarnations and the remnant of past technological state of the art processing the majority of all transactions worldwide. However, their main characteristics are inhibiting innovation and growth and excessively consuming resources. Migration is still considered the only sustainable approach to handle the inherent deficiencies without starting from scratch. Despite this, the topic is largely absent from research on software engineering methodology and techniques which instead continue to focus on increasingly improbable greenfield scenarios. This leaves practitioners with a mismatch between theory and reality, which can mean the difference between success and failure in a software project.

This thesis represents a long term case study analysing a total of four cases over the course of several years each, spanning almost a decade in total. Each object of investigation was chosen by evaluating it against a set of criteria to ensure that it exhibits common characteristics found in legacy systems. It is then analysed in detail covering both technical and socio-economical aspects. Alternative outcomes are discussed to highlight key decisions which have significantly affected the direction of the respective project. Afterwards a series of cautious abstractions and generalisations are extracted to discuss commonalities and lessons learned. These abstractions are subsequently condensed into topic groups, each representing a result in the form of a central thesis and a set of challenges to be addressed. Together, topics groups and generalisations can be seen as the proposal for a new topology on Legacy System Migration (LSM).

The resulting detailed description of each case gives a deep insight into the evolution of each unit of investigation. The challenges that had to be overcome range from organizations that are not prepared on either a technical or an organizational level or both to fighting the urge to chase “silver bullet” solutions or the need to clean up after past chases. The ensuing abstractions and generalisations link the experiences from each case and contrast or augment them with the academic state of the art if there is sufficient overlap. Depending on the perspective, they can also be read as lessons learned or cautious recommendations. At the same time these abstractions also serve to highlight the thematic diversity of the challenges facing a LSM effort. The thesis is completed by clustering them into a total of five topic groups ranging from low level technical patterns to high level strategic considerations, forming the proposed topology. Each group delivers a resulting thesis that propagates success factors and a discussion on challenges in the field which need to be addressed in future work. The topology therefore serves as a central medium for communication between researchers and practitioners and at the same time as a navigational aid for the topic of LSM. The conclusion culminates in a call to both industry and academia to collaborate more closely both in research and education. A stronger focus on legacy topics is urgently needed to avoid another legacy crisis.

Keywords

Legacy System Migration, Legacy System, Software Reengineering, Topology

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

We are sitting on a "software volcano" [46], a realization already made by Van Deursen et al. in 1999 - the time when the y2k problems as well as the Euro conversion brought to light the sheer amount of skeletons we had been collectively hiding in our closets. Fifteen years later little overall progress is visible, memories of the year 2000 seem to be fading and along with them, the interest in the research topic of software renovation in general. The gap between academics and industry seems to be widening: The academic circles are focusing mainly on technical issues and, as a side note, almost completely neglecting the topic in the curricula [97]. The industry is primarily experiencing organizational and business perspective problems [12, 73]. However, the setting of numerous, real life problems combined with little expertise on the industry side and the desperate need for realistic research scenarios on the academic would suggest a near perfect synergy [86].

A fundamental challenge that enterprise IT is facing at the moment, is the speed of technological evolution. This becomes especially apparent in migration scenarios as new technology is directly compared and also benchmarked against legacy technologies. Existing legacy systems have been in place for decades, undergoing relatively little technological evolution. Of course there were some fundamental changes, like replacing data storage mechanisms by relational databases or text based terminal interfaces by graphical user interfaces, but at the core business functionality was developed and delivered in similar ways over many years.

This is in stark contrast to the technological landscape today. The life expectancy of a software system has shortened significantly, in some areas even radically. Already almost 30 years ago the average of a lifespan was found to be around 6-8 years, slightly increasing with the size of software [121]. No indication could be found that the lifespan has increased since. The size of enterprise systems can likely explain parts of a perceived longer lifespan for these systems when compared to consumer-facing web or mobile applications. However, rapid technological turnaround does affect enterprise systems in similar ways then consumer applications, especially when it comes to frontend technology. This opens another way of reasoning in connecting the lifetime of a system with the evolution of today's presentation technologies based on the web and mobile platforms. The browser and the phone as replacements for classic operating systems and therefore user interface platforms are inherently connected to the pace of evolution of consumer technology. This creates a disparity as these platforms lack the requirements for stability related to significantly longer lifetimes dictated by enterprise RoI calculations. The challenge in the context of enterprise systems is to strike a balance between modern platforms and their end user convenience and technological stability required in building long living applications suitable for the corporate world. Ultimately it comes down to realistically defining the expectation for the lifetime of a system at the planning stages of a project just as any other non-functional requirement while at the same time acknowledging the realities of today's fast paced world.

This thesis will contrast the current state of industry with the academic state of the art based on four cases of successful legacy system migration efforts in representative domains (transportation, insurance and e-government) conducted over a period of ten years. On the one hand this will show newly established but proven industry best practices with a focus on deviations from published

material. On the other hand focusing on lessons learned will highlight mistakes repeatedly made although already published (like [15]) in academic circles.

The overall goal of this thesis is to show the current state of the art of system migration for the 21st century by analysing a series of cases in a multi case study and highlighting common success factors as well as problems that can be translated into research questions to be addressed in the next decades.

While in industry the search for a "silver-bullet" [122] solution (like a completely automated, pain free transformation) is still common place, this work will highlight that large migration and reengineering efforts are subject to the influence of a large number of (soft) factors:

- Existence and suitability of an enterprise IT landscape architecture and long term IT strategy
- Existence of a (proven) architecture for the target IT system
- Awareness of the need to understand business processes
- Understanding of the role the legacy system(s) play in the business processes
- Existence (or ability to create) a suitable team of domain and technical experts

1.2 Approach

To achieve the primary goal of assessing the industry state of performing legacy system migration strategies a case study involving multiple cases following the guidelines of Verner et al. [128] will be conducted and evaluated. This approach is suitable as a "case study is preferred in examining contemporary events, but when the relevant behaviors cannot be manipulated" [136]. After base lining the current state of the art (as drafted in the following section) by doing an extensive analysis of the current literature, the cases will be selected. The selection criteria will include:

- Relevance of the legacy system to the organization: The migration effort has to include core systems vital to the success of the business.
- Age of the legacy system: An evolution of at least 15 years will prove the system fulfils the business requirements.
- Size of the legacy system: All case studies have to be non-trivial, typically having millions of lines of code, executing millions of transactions, managing terabytes of data.
- Need for renovation: Along with "factors such as inflexibility [24, 32], expensive to maintain [12, 24], and even use of obsolete programming languages [12, 31]" [73], the alignment with business requirements is a key requirement for selection.
- No possible (easy) substitution with a COTS product (SAP or similar).

Possible candidates include a campus management system (CaMS) [116] and an airport operational database (AODB) [19] - both successful applications of the Chicken Little [32] incremental approach - as well as systems currently operational in the insurance and government sectors. Following the selection and description of the cases will be the analysis according to the research questions proposed in the previous section.

1.3 Related Work

Chikofsky et al. [37] have provided a level playing ground by creating a taxonomy guiding the researchers through the mine field of expressions and definitions.

Brodie et al. [32] have accumulated incremental, gateway based strategies focusing on the core value holder, the data, of a legacy system. Wu et al. [135] advanced the approaches providing a gateway free approach to legacy system migration.

H.M Sneed [110] conducted a study of 13 migration projects to identify the major risks in reengineering projects being performance, architectural mismatch, testing bottleneck, quality goals and acceptance emphasizing the author's main point on the challenges of legacy system migration being of a non-technical/technological nature.

Khadka et al. [73] have recently (2014) addressed the perception of legacy system modernization, providing interesting insights for academia by highlighting that "the challenges of legacy system modernization in industry largely emerge from the business perspective" but at the same time by providing affirmative data on the technical research focus of the field.

Ulrich et al. [125] propose Architecture Driven Modernization (ADM) as an approach to restore the value of legacy systems including a study of a series of case studies to underpin the method. It builds on earlier, more generic strategies published also by Ulrich in [124]. However, the work is very much focused on the products of a single company which offers automated legacy system documentation and transformation and who the author works for. This greatly reduces the general applicability of the portrayed results.

Lauder et al. [79] have identified a series of architectural anti patterns which they term "petrified" legacy systems and formulated a corresponding set of patterns to address these issues.

Murer et al. [90] presented their experience with an evolutionary approach to migrate legacy software of Credit Suisse Bank. Their findings similarly highlight the need to focus not only on technical details of the modernization, but also on the business and organizational aspects.

1.4 Structure

This thesis is structured as follows:

In Chapter 2 a brief overview of Legacy System Migration Fundamentals is provided including definitions of important terms, reasons for and a history of legacy system migration. The chapter is rounded off with a discussion of some of the most influencing methodologies.

In Chapter 3 the selection of case studies including the underlying method is presented.

In Chapter 4 the four individual case studies are discussed in detail. These are:

- "Case: A Modern Process Management for 6000+ Employees" in Section 4.1
- "Case: A Very Specifically Grown System in an Important Niche Domain" in Section 4.2
- "Case: A Very Old Core" in Section 4.3
- "Case: A Highly Loose Quagmire" in Section 4.4

In Chapter 5 a set of Cautious Abstractions & Generalisations is provided. We distinguish between Technical Experiences & Observations, Architectural & IT Strategy Experiences & Observations and Organizational Experiences & Observations.

As the main result, in Chapter 6 a clustering of the previously presented Cautious Abstractions & Generalisations is suggested to create an initial topological mapping of the field of legacy system migration.

The thesis is rounded off with Chapter 7. It summarizes the abstractions into a concise conclusion and discusses planned future work and potential research directions that can be derived from the findings presented here.

2 Legacy System Migration Fundamentals

When discussing Legacy System Migration (LSM) it is necessary to first take a step back to visualize at what point in a software life cycle it takes place. This is based on the concept of a software life cycle as defined by IEEE [67]:

> The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software life cycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and, sometimes, retirement phase. Note: These phases may overlap or be performed iteratively.

As reference the staged model for a software life cycle as defined by Rajlich and Bennett [98]. They distinguish between the following phases:

- Initial development. Engineers develop the system's first functioning version.
- Evolution. Engineers extend the capabilities and functionality of the system to meet user needs, possibly in major ways.
- Servicing. Engineers make minor defect repairs and simple functional changes.
- Phaseout. The company decides not to undertake any more servicing, seeking to generate revenue from the system as long as possible.
- Closedown. The company withdraws the system from the market and directs users to a replacement system, if one exists.

What the IEEE Glossary summarizes as an optional retirement phase, Rajlich and Bennett split up into Phaseout and Closedown, distinguishing between the period of time when the system is still actively used, but no more effort is put into prolonging its lifespan and the period of actual decommissioning of the system. It is important to note that frequently the decision for phasing out a system is made implicitly. A prime driver is cost reduction, either by directly cutting the budgets to maintain a system, or indirectly by (gradually) reassigning the maintainers of the system to other tasks.

Legacy System Migration is necessary during the Phaseout or Closedown of a software, if this system can not simply be switched off without impacting the business. As this would imply that the previously running system did not provide any significant value to the business it is not a common scenario in this context, unless the business itself is closing down. LSM needs to take place when data and/or functionality of a system facing retirement has to be transferred to a replacement system.

The remainder of this chapter is structured as follows: As a first step important terms are defined. Based on this ground work we discuss the reasons for migrating legacy systems and follow up by discussing the historic evolution of the field. Finally, to complete the picture, established methodologies are reviewed.

2.1 Definitions

To start off and provide sufficient context it is necessary to agree on a concise definition of some core information technology terms and the interpretation that is to be used in the remaining parts of this thesis. Wherever possible, definitions are directly derived from established literature. The following fundamental building blocks will, if executed properly, ensure a clearly defined environment for any legacy system renovation effort.

2.1.1 Legacy System

Bennett [12] defines legacy systems informally as

> large software systems that we don‘t know how to cope with but that are vital to our organisation

Brodie and Stonebraker [32] focus on adaptability:

> Any information system that significantly resists modifications and evolution

Both definitions are very broad and designed to encompass as many scenarios as possible. This is reasonable and necessary at this level of abstraction, as legacy systems can take on many different forms and can come from any technological background or business domain. However, it does also complicate the strategic and technical discussion on how to address challenges found in the context of legacy systems. By definition, any solution that targets these abstract definitions of a legacy system will have to be as generic and abstract as well. The only alternative is to assume certain implicit characteristics or constraints on the definition. Both the Chicken Little (see Section 2.4.1) and the Butterfly Methodology (see Section 2.4.2) exemplify this issue well.

One approach to further refine the picture are the following factors for legacy system classification suggested by [112]:

1. **End of Life**: Parts of the system like technologies, frameworks or components as well as hardware are no longer supported.
2. **Knowledge**: Little is known at the organization about the inner workings of the system or the reasoning behind its design.
3. **Skills**: People with the necessary skills and the will to employ them for maintaining a legacy system are difficult to come by.
4. **Extensibility**: Extending the system is impossible, too risky or too expensive.
5. **Time to Repair**: It takes too long to fix any incidents in the systems.
6. **Scalability**: The system cannot be scaled to handle additional load for new functionality or additional business volume.
7. **Limitations**: The capabilities of the system are limited in a way that effects the business.
8. **Costs**: The cost of keeping the system running is consuming an unjustifiably large amount of the overall IT budget.

Not all factors have to apply for a system to be classified as a legacy system.

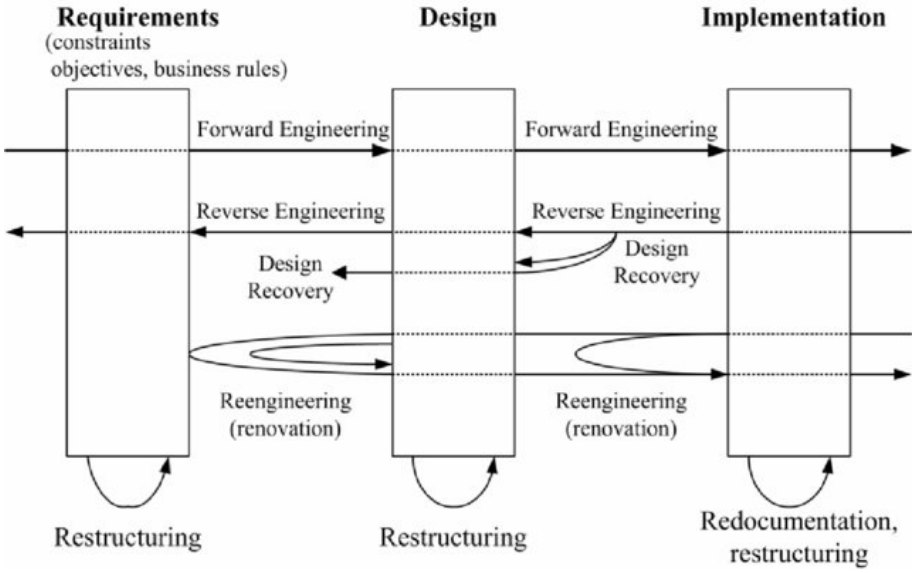

Figure 2.1: The relationship between the terms reverse engineering, forward engineering, reengineering and design recovery [37] (graphic from [102]).

2.1.2 Legacy System Migration

Bisbal et al. [24] define migration as a way to move

> the LIS [Legacy Information System] to a more flexible environment, while retaining the original system's data and functionality.

Brodie and Stonebraker [32] have a looser definition, where:

> Legacy IS migration begins with a legacy IS and ends with a comparable target IS.

They continue to note that "true" legacy IS migration involves functionality and data, anticipating that little of the actual legacy code can be directly reused, but most of the legacy data can and must.

2.1.3 Design Recovery

Biggerstaff et al. [23] define design recovery as follows:

> Design recovery recreates design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains. ... [it] must reproduce all of the information required for a person to fully understand what a program does, how it does it, why it does it, and so forth. Thus, design recovery deals with a far wider range of information than found in conventional software engineering representations or code.

Biggerstaff goes on to note that design recovery is "inherently unstructured and unpredictable" and as a consequence little tool support is available as any tools would need to have access to domain specific knowledge and expertise.

2.1.4 Reverse Engineering

"Reverse Engineering" is a term originally used in the context of hardware [100]:

> The process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system

In the context of software engineering the most commonly used definition was coined in the taxonomy of Chikofsky and Cross [37]:

> Reverse engineering is the process of analyzing a subject system to
>
> - identify the system's components and their interrelationships and
> - create representations of the system in another form or at a higher level of abstraction.

They go on noting that reverse engineering is a process of inspection that does not change the subject system. Furthermore they define two sub areas of reverse engineering. "Redocumentation" and "Design Recovery" (based on the definition presented in section 2.1.3). Redocumentation is defined as the process of recovering information at the same level of abstraction with the intention of creating documentation that should have been kept with the system in the first place, but was either never written, was not maintained or lost over the years. Design recovery on the other hand is defined as the process of recovering information at a higher abstraction level. It thus must include more sources of information than code and similar artifacts to be able to recover domain knowledge and the reasoning behind design decisions. The relationship between those terms is illustrated in Figure 2.1.

Confora et al. [34] additionally highlight that reverse engineering tools and methods need to adapt with technological evolution (in their case Service Oriented Architectures). Most importantly they point out the need for dynamic analysis in addition to existing, static methods to deal with the distributed and heterogeneous nature of today's legacy information systems.

2.1.5 Reengineering

Again the taxonomy by Chikofsky and Cross [37] serves as a reference:

> Reengineering, also known as both renovation and reclamation, is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.

Reengineering utilizes both "Reverse Engineering" (section 2.1.4) and "Forward Engineering" techniques to achieve the goal of creating a new target system. In conjunction they also discuss the relationship with the term "Restructuring" (which today would be more likely be termed "refactoring" as defined by Fowler[51]). The distinction being made is that "Reengineering" includes functional modification between the source and the target system, whereas "Restructuring" is the pure modification of the internal representation of a system without visible change for the end user.

Bergey et al. [14] define the Reengineering Horseshoe Model (as depicted in Figure 2.2) as part of their Option Analysis for Reengineering (OAR). This model is frequently used to visualize

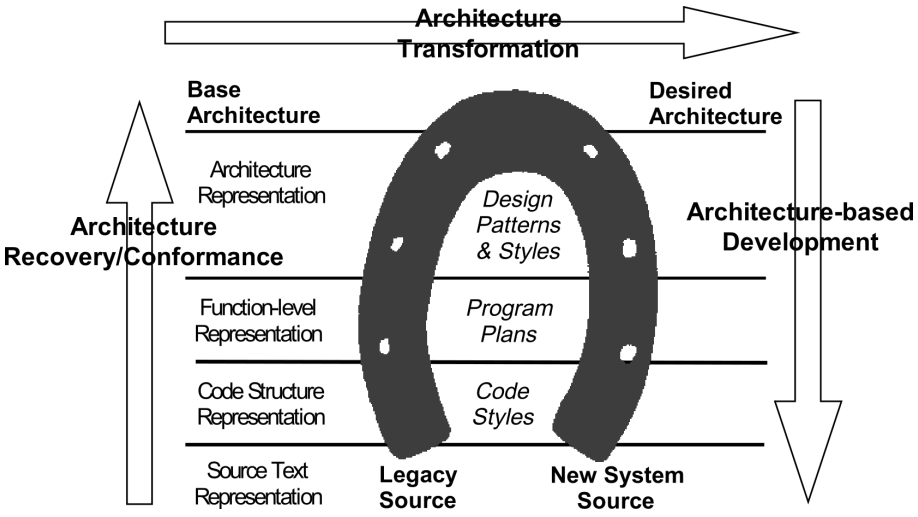

Figure 2.2: The Reengineering Horseshoe Model [14].

reengineering processes. The model depicts a reengnineering method in three stages. The left, vertical side is used to depict reverse engineering effort. The top, horizontal side is used to show the transformation between the source and target systems. The right, vertical side then shows the subsequent forward engineering steps. Based on this three levels of transformations are attainable. The purest form is the architectural transformation where the legacy architecture is recovered and transformed into the target architecture which is subsequently implemented. Two shortcuts are possible on a lower level of abstraction. At the middle, functional level transformation can be performed in the form of repackaging (wrapping for example). At the lowest level, the source level transformation can be realized either on a textual basis or the abstract syntax tree (AST) representation of it. It is important to note that both the functional and architectural level cannot be fully extracted automatically from the source code representation of a legacy system.

It is equally important to differentiate system reengineering from business process reengineering (BPR), as defined by Hammer and Champy [62]:

> Reengineering is the fundamental rethinking and radical redesign of business processes to achieve dramatic improvements in critical contemporary modern measures of performance, such as cost, quality, service, and speed.

The focus of BPR is solely on the organization and does not significantly touch information systems. However, IT will in many cases be directly affected by the results of a BPR effort as legacy systems are no longer able to cope with the redesigned business.

2.1.6 Rehosting & Replatforming

The terms Rehosting and Replatforming are unfortunately not clearly defined in literature. They can be seen as two subsets or possible approaches of (legacy system) migration, both fitting well with the overall definition "... moves an existing, operational system to a new platform, retaining the legacy system's functionality and causing as little disruption to the existing operational and business environment as possible" [24].

We therefore define the two terms as follows:

ReHosting: Migration maintaining the same technical platform (CPU architecture, operating system), but changing to a different (physical) host. For example the existing workload of a dedicated host system is moved to a shared host.

RePlatforming: Migration changing the technical platform. For example migrating from z/OS and zCPU to AIX and PowerPC or from Linux to Windows. Replatforming therefore includes rehosting activities.

2.1.7 Technical Debt

Technical Debt is a financial metaphor originally coined by Ward Cunningham in 1992 [43]. It describes the effects of necessary trade-offs between quality, available resources and time to delivery. Technical debt incurs cost in the form of interest due to higher maintenance and development efforts and principal for eliminating it. Technical debt also entails risk in the form of potentially breaking (parts of) the information system.

2.1.8 Artifact

The term artifact (or artefact) has been widely used and misused due to a long lasting lack of a proper and concise definition. Although not a core term in terms of legacy system migration this thesis will adhere to the definition provided in [87] with the notable exception of using according to the author's opinion the more appropriate spelling "artifact" with an 'i':

> An artefact is a self-contained work result, having a context-specific purpose and constituting a physical representation, a syntactic structure and a semantic content, forming three levels of perception.

2.1.9 Software Entropy

Jacobson draws an analogy between thermodynamics and software systems to illustrate the deteriorating effect of changes on these systems [69]:

> The second law of thermodynamics, in principle, states that a closed system's disorder cannot be reduced, it can only remain unchanged or increased.
>
> A measure of this disorder is entropy.
>
> This law also seems plausible for software systems; as a system is modified, its disorder, or entropy, tends to increase.
>
> This is known as software entropy.

2.1.10 IT Strategy

IT strategy is defined by Gartner in their IT Glossary as follows (via Johanning [71]):

> IT strategy is about how IT will help the enterprise win. This breaks down into IT guiding the business strategy, and IT delivering on the business strategy. Although some or all tasks involved in creating the IT strategy may be separate, and there are normally separate documents, IT strategy it is an integral part of the business strategy.

IT strategy from a legacy system renovation or migration perspective is the principal and initial driver behind any effort. A legacy system must be of strategic interest to be a possible candidate for renovation and the direction of the effort must be towards the long term goals set by the IT strategy to better serve the corresponding business strategy. In the author's point of view the IT strategy is defined and maintained by the corresponding C-level functions inside an organization (CIO, CTO).

2.1.11 Enterprise Architecture

Enterprise Architecture (EA) is defined by Gartner in their IT Glossary as follows (via Bojinca [28]):

> Enterprise architecture (EA) is a discipline for proactively and holistically leading enterprise responses to disruptive forces by identifying and analyzing the execution of change toward desired business vision and outcomes. EA delivers value by presenting business and IT leaders with signature-ready recommendations for adjusting policies and projects to achieve target business outcomes that capitalize on relevant business disruptions.

MIT CISR defines enterprise architecture as (also via Bojinca [28]):

> the organizing logic for business process and IT capabilities reflecting the integration and standardization requirements of the firm's operating model.

The term Enterprise Architect(ure) is attributed to John Zachmann, author and creator of one of the two leading (enterprise) architecture frameworks - the Zachmann Framework [138]. Although originally termed "A framework for information systems architecture", Zachmann only later realized it "should have been referred to as a 'Framework for Enterprise Architecture"' [137]. The other major framework for completeness would be the TOGAF Framework [63].

However, while documentation with the help of frameworks is important, reducing the role of an Enterprise Architect to documenting the current state of the organizational IT landscape is one of the most common mistakes. In the author's point of view an Enterprise Architect is the leading figure behind executing the IT strategy and hence realizing the envisioned benefits for the business. It is for that reason, on the one hand an interface position between business and IT requiring exceptional communication skills and on the other hand a technical position to be able to determine what steps have to be taken to reach the previously defined goals.

From a legacy system renovation or migration perspective, Enterprise Architecture provides the necessary rules and boundaries to enable qualified decision making on a system and application design level and a first level of escalation for resolving conflicts between different systems of the enterprise IT landscape.

2.2 Reasons for Legacy System Migration

25 Years ago David Lorge Parnas talked about "Software Aging", the reasons behind it, the associated costs and possible remedies [92]. He points out that both the lack of change to adopt to an evolving environment and the (badly done) execution of changes are causes of software aging, but distinguishes from leakage problems (memory, file system, database) as these are not directly

related to the age of an application and can potentially be cured completely and permanently. The main costs in connection with aging software systems are an increase in resource usage associated with maintenance and development, decreased performance and lower reliability of the product. Parnas continues to argue that the cost of aging can only be reduced by planning (far) past the initial release of a software system. The primary goal is to design for success (as he rephrases the popular "design for change"). However, he argues that, while this is well agreed on in theory, the specifics of how to apply this in practice are much less clear. He goes on to point to proper documentation and ongoing reviews as additional remedies, but concludes that aging, to a certain degree, is ultimately inevitable. The following "geriatrics", that can be applied to undo the harm done by software aging, are illustrated without sticking to the established terminology (see Section 2.1): addressing technical debt, redocumentation, refactoring, rewriting or reengineering. The concluding remarks revolve around the need to realize that the problem of software aging needs a long term therapy, the need for the software engineering profession to become more of an engineering discipline (as opposed to a science) and finally the need for closer ties and integration between academia and practitioners.

Bennett already pointed out in 1995 there is what he calls a "Legacy Dilemma" [12]: On the one hand you have a legacy system that is a large system reliably delivering value to its users which is too complex and hence costly to be simply replaced. On the other hand you have ever increasing maintenance costs due to a lack of understanding of the system, documentation, human resources with expertise in the technology and a lack of past continuous investment into the maintainability of the system. Bennett concludes that ultimately it comes to a "trade off [between] the cost of continuing to cope with the legacy system against the investment needed to improve it and the benefit of easier subsequent maintenance."

Bisbal et al. summarize the problems that can be caused by legacy information systems (LIS) [24]:

> They [LIS] can cause host organizations several problems:
>
> - LISs usually run on obsolete hardware that is slow and expensive to maintain.
> - Software maintenance can also be expensive: because documentation and understanding of system details is often lacking, tracing faults is costly and time-consuming.
> - A lack of clean interfaces makes integrating LISs with other systems difficult.
> - LISs are also difficult, if not impossible, to extend.

They continue to argue that the migration and therefore ultimate replacement of the legacy information system is the only final solution to the previously named problems. Partial solutions like wrapping, while widely adopted because of their relatively easy applicability, represent only short term remedies that might in fact complicate a subsequently necessary migration effort [132].

Ulrich describes the problems attributed to legacy systems as the main inhibitor between today's reality in computing and the vision drawn by mid twentieth century science fiction. He lists financial institutions, telecommunications firms, utility companies as well as the public sectors as some of the industries "facing a major challenge in being mired down by legacy computing architectures" [124]. On a management level legacy systems are perceived as an obstacle slowing down innovation and transformation.

The problems affecting legacy systems can be argued with Lehman's Laws of software evolution [80, 81]:

> The law of continuing change. – Any software system used in the real-world must change or become less and less useful in that environment.

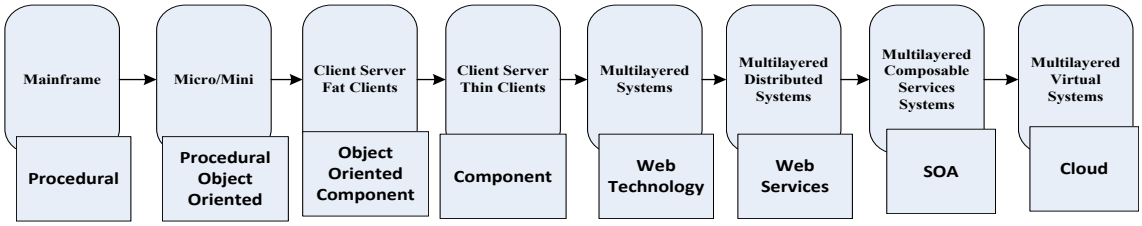

Figure 2.3: The Legacy System Migration Evolution Roadmap [55].

> The law of increasing complexity. – As time flows forwards, entropy increases. That is, as a program evolves, its structure will become more complex. Just as in physics, this effect can, through great cost, be negated in the short term.

To summarize the effect of both laws, software must be continuously adapted to maintain its usefulness to the consumer. However, these changes negatively affect the quality and future adaptability of the software and as a consequence require constant investment to counter this effect. If a system lacks this kind of investment the maintainability of the software decreases until it reaches the state of a legacy system. The term "Software Entropy" (as defined in Section 2.1.9) paints a similar picture.

These observations can be complemented by Wirth's Law [133]:

> With a touch of humor, the following two laws reflect the state of the art admirably well:
>
> - Software expands to fill the available memory. (Parkinson)
> - Software is getting slower more rapidly than hardware becomes faster. (Reiser)

The laws can be used to explain why the ails of legacy information systems will not be mitigated by improvements in (hardware) technology. In other words the legacy of LIS cannot simply be addressed by waiting for future technical developments.

2.3 Legacy System Migration - A History of Histories

Legacy System Migration (LSM) has significantly evolved over the years. It started out with the goal of recovering knowledge from existing system artifacts in the field of program understanding, progressed towards more abstract design recovery and finally a general reengineering approach based on the combined application of reverse and forward engineering techniques. Depending on the individual goal a legacy system migration can mean rehosting, replatforming, refactoring, renovating or completely rebuilding a system. In recent years a major shift in the field has been from the focus on the legacy artifacts towards the characteristics of a specific target architecture or paradigm. The following section will structure the evolution of the field in four distinct phases over approximately the past forty years, similar to the structure used in [55] to elaborate the evolution roadmap depicted in Figure 2.3.

2.3.1 Early Days

The theoretical foundation for the field of reverse engineering and reengineering can be found in Lehman's Laws of software evolution [80, 81]. It is rooted around the deteriorative nature of software throughout its lifespan, as discussed in the previous section.

The genesis of the field of LSM can be traced back to the reverse engineering efforts initially focused on hardware [100]. These techniques were then adopted to the software engineering and primarily the database community [44]. The initial focus on data can quite easily be explained by both the intrinsic and durable value of data, which can be increased with its quality and the relative simplicity of the programs initially manipulating it. Aiken defined data reverse engineering (DRE) as "the use of structured techniques to reconstitute the data assets of an existing system" [2].

In 1990 Chikofsky et al. [37] created a taxonomy defining the terms "Reverse Engineering" (see also section 2.1.4) including the sub-terms "Redocumentation" and "Design Recovery", "Forward Engineering", "Restructuring" and "Reengineering" (see also section 2.1.5) and their relationship in the context of a basic software engineering life cycle. They predict a significant increase in productivity for software engineering techniques that include reverse engineering capabilities.

Hainaut et al. [60], in the context of DRE, elaborate on the many reasons and motivations to perform (data) reverse engineering:

- Knowledge acquisition in system development
- System maintenance
- System reengineering
- System extension
- System migration
- System integration
- Quality assessment
- Data extraction / conversion
- Data administration
- Component reuse

One of the earliest methods found describing LSM is database first and database last in a working paper "Migration of Legacy Systems" by Bateman and Murphey of Dublin City University. However, the working paper seems no longer publicly available and only short summaries can be found [134]. Two of the main issues with both approaches are the requirement for a forward or reverse gateway and the possibility of significant downtimes during data migration.

Brodier and Stonebraker [32] publish their "Chicken Little" approach to legacy information system migration. They acknowledge that replacing an increasing number of legacy systems needs more sophisticated approaches and tools than the common, brute force strategy of big bang replacements, which they term "cold turkey". In addition, they highlight the difference the state of a legacy system makes, differentiating between decomposable, semi-decomposable and non-decomposable legacy systems.

However, the concept of a gateway is frequently not feasible in practice, mainly due to the requirement of maintaining update consistency across heterogeneous information systems and the complexity of developing and integrating the necessary gateways. As a result Wu et al. [135] introduced the "butterfly methodology", a gateway free approach to migrating legacy IS.

2.3.2 The Post Year 2000 era

Having just survived a widely predicted disaster, both IT industry and academia found time to reflect on the findings of this period. Seacord et al. [104] go so far as to identify a "Legacy Crisis". They argue that enterprises needed the Y2K-era to realize how much (legacy) code they are maintaining and how much of the IT budget is being eaten by associated maintenance tasks. They furthermore highlight that the pile of code is growing quite fast. While certainly true, it is quite easy to argue that given the amount of research in areas like program comprehension and reverse engineering these facts were no news at the time for the academic community. However, there are few indicators that show any kind of improvement on the key characteristics of the perceived industry crisis. The experiences led to a series of published roadmaps to highlight open research challenges:

Müller et al. [89] provide a roadmap which focuses on reverse engineering process, data reverse engineering, reverse engineering tools and their evaluation. They also highlight the problem of low industry tool adoption due to usability concerns. They conclude with a call for improving the process of reverse engineering, integrating it better with software evolution techniques and balancing the curriculum of universities between software construction and evolution.

Benett et al. [13] focus in their roadmap on maintenance and evolution challenges. They highlight the changing business landscape which requires a much more frequent and especially much more rapid possibility for change. They also anticipate a new generation of legacy problems aggravated by the increasingly distributed and integrated nature of information systems. Furthermore they envision software evolution based on a service model bringing the topic of software evolution to the centre of software engineering.

To summarize the already ongoing shift to the web anticipates a more fundamental paradigm shift. With the PC and somewhat later the smartphone ubiquitous computing becomes foreseeable. With it comes a transformation of the whole IT landscape, both for academia and industry. The reverse engineering and legacy system migration fields need to adapt accordingly and as a consequence lessen their focus on migrating enterprise information systems. As the following two sections show migration research pivots towards target architectures.

2.3.3 Legacy to Service Oriented Architecture

Service Oriented Architecture (SOA) was designed with the goal of achieving characteristics as loose coupling, abstraction of underlying logic, flexibility, reusability and discoverability [91]. These characteristics can be used to facilitate the reuse of legacy assets through wrapping. A wide variety of tools and methods were proposed during a period of intensive research. Razavian et al. have categorized studies into eight different families ranging from code transformation over business model transformation to forward engineering [99]. Khadka et al. provide a historic overview of legacy to SOA evolution as part of their systematic literature review [72].

One of the key contributions of this era is the fact that SOA provides a standardized way for encapsulating legacy functionality and as a result making it accessible for everyone. On a low, technical language level a common way of exchanging meta information was found and widely established across industries and technical ecosystems, even as more advanced concepts like service discovery were less widely adopted. In addition, the primary mode of communication with this integration technique was synchronous, providing a major shift from batch/file based integration to remote-procedure call approaches with a focus on online transactions. All of this enabled an unprecedented degree of enterprise and industry wide integration of information systems.

The main criticism of this approach can be characterised with the tendency of hiding legacy assets. By wrapping problematic systems in accessible services two acute pain points are addressed. First, this functionality becomes available to engineers not familiar with the used legacy technologies, enabling them to build on existing features based on a universally understandable contract. Second, this functionality becomes available to the end user without the need to use legacy user interfaces like the VT100 terminal. Instead it can be made accessible through modern, state of the art user interfaces. Both these features are important mechanisms to alleviate immediate pressures. However, they do not address the underlying characteristics of a legacy information system: the respective implementations of these services still significantly resist modification and evolution.

2.3.4 Legacy to Cloud and/or MicroServices

Cloud computing and MicroServices Architectures can be seen as the evolutionary next step up from SOA. The concepts behind cloud computing address the need for further virtualization on an infrastructure (Infrastructure as a Service, IaaS), platform (Platform as a Service, PaaS) and service (Software as a Service, SaaS) level. Micro Services provide the architectural concepts of developing systems that fit with the new cloud based operating model. However, both concepts impose significant new challenges due to their highly distributed nature, large scale redundancy and frequent heterogeneity. In addition, a widely proprietary solution landscape adds the risk of vendor lock in.

Frequently building upon research and findings of legacy to SOA migration approaches as presented in the previous section a strong focus today is on legacy to Cloud and legacy to Micro Services research. Jamshidi et al. have published a systematic literature review on cloud migration research [70]. They distinguish between “cloud-native” software, specifically designed for running (exclusively) on the cloud and “cloud-enabled” systems, which provide the possibility of being (also) run in cloud based environments. One of their key findings is a lack of research addressing cross-cutting concerns like governance and security. In addition, little was found on the topic of automatic runtime resource scaling, which can be considered a key requirement if promises of cost-effectiveness of cloud environments are to be realized. Balalaie et al. [7] have created a pattern repository from their migration experiences for the reuse in other migration efforts. They furthermore highlight the fact that even in a component based micro services environment final integration requires a high level of coordinative effort and might prolong the final delivery of a complete working system.

Tobar et al. [30] have published a systematic literature study focusing on research regarding the path from SOA to the cloud. In other words, SOA systems are the new legacy which needs to be migrated. They found a majority of the studies to be about conventional (i.e. manual) migration strategies with a goal of rehosting to cloud infrastructure with a private or hybrid PaaS or IaaS as a target. Furthermore a very large proportion (>80%) of all studies were conducted in an academic context.

One key observation similar to SOA migration is that mainstream adoption of basic cloud and microservice concepts is visibly increasing whereas adoption of more advanced concepts both on a technical as well as an organizational level are still lacking. One reason for this could be that the former is being conducted to realize quick gains in terms of efficiency and cost-effectiveness, whereas the latter is only feasible in certain settings, like the development of internet-scale mega platforms.

2.4 Methodologies

Many methods and approaches have been proposed over the years to structure the migration of legacy systems. However, only few have seen more widespread adoption or even attention. In this section we give a brief overview of some of the most prominent methodologies. Two classes of methods are highlighted: Concrete, but generally applicable approaches to the actual migration of a legacy system are represented by the Chicken Little (Section 2.4.1) and Butterfly (Section 2.4.2) methods. Architecture Driven (Section 2.4.3) and Risk-Managed Modernization (Section 2.4.4) on the other hand represent more high level, strategic approaches to select, decide on and properly plan modernization efforts. The general overview is rounded off with an example of a specialised methodology with SOA as a target architecture developed in collaboration between academia and industry (section 2.4.5).

2.4.1 Chicken Little Methodology

Brodie and Stonebraker [32] have introduced the "Chicken Little" methodology as one of the first structured and iterative approaches to legacy system migration. It addresses the high risks of what they term the "Cold Turkey" approach of replacing legacy systems with rebuilds in one big bang style cut over. They distinguish between decomposable, semi-decomposable and non-decomposable legacy systems. Decomposable legacy systems have independent modules that can easily be separated and consequently migrated incrementally. The only coupling in between modules is at a database level. Semi-decomposable systems have some kind of coupling at the application level that makes separating them infeasible. User and system interfaces can still be migrated incrementally. Non-decomposable systems on the other hand have inseparable components and interfaces.

In addition, three methods are distinguished: Forward migration, reverse migration and general migration as a combination of the previous two. While forward migration starts with an initial, possibly partial database migration, reverse migration transfers the database last. A general migration combines both to pick the suitable approach for each module or component that can be migrated incrementally.

Depending on the possibility for decomposing the legacy information system gateways are used on different levels. Gateways have three main responsibilities. They guard components from changes made in their surrounding, they translate requests and data between components and they coordinate between components to mediate and ensure update consistency. Database level gateways are easiest to apply for decomposable systems. Depending on the migration method either a forward or reverse gateway is needed. The former provides access to a migrated database in the new system whereas the latter provides access for the new system to the legacy database. Hence a general gateway combines the functionalities of both forward and reverse gateways. Application level gateways enable communication between the legacy and the new system at a component level. Depending on the method it is therefore possible to have legacy components call the new system or vice versa. In the context of non-decomposable legacy systems an information system level is necessary. This type of gateway captures all user interactions and returns results, similar to a proxy. The gateway therefore needs to be aware of user interface requirements and consequently needs to address them.

The actual migration method consists of the following eleven steps. However, it is important to note, that independent steps should be executed in parallel to speed up the process. The individual steps have to be tailored depending on the characteristics of the legacy system and the migration strategy.

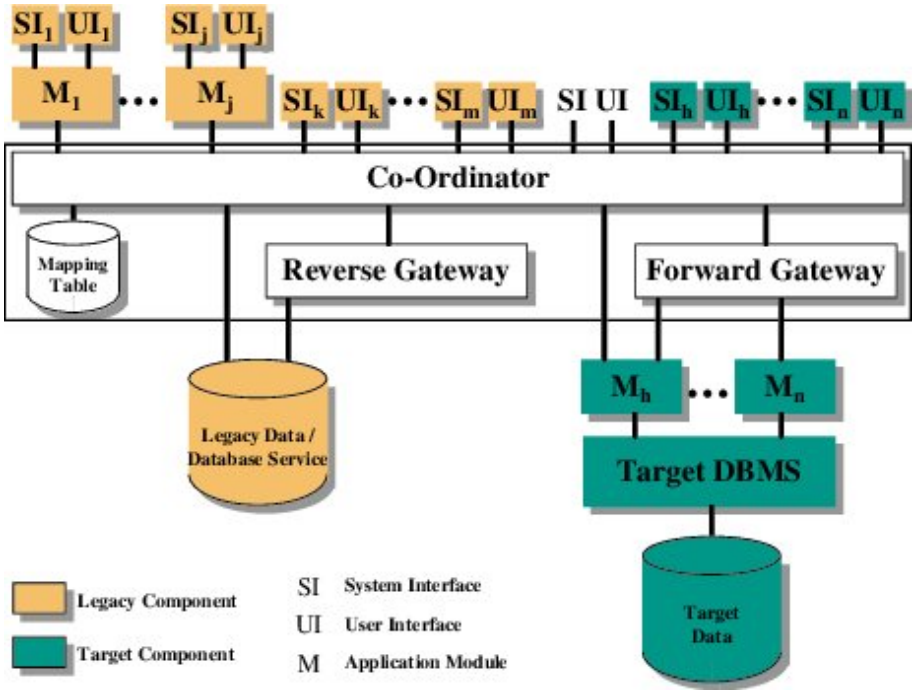

Figure 2.4: The Chicken Little approach applied in a migration architecture including a coordinator [135].

1. Incrementally analyze the legacy IS.
2. Incrementally decompose the legacy IS structure
3. Incrementally design the target interfaces.
4. Incrementally design the target applications.
5. Incrementally design the target database.
6. Incrementally install the target environment.
7. Incrementally create and install the necessary gateways.
8. Incrementally migrate the legacy database.
9. Incrementally migrate the legacy applications.
10. Incrementally migrate the legacy interfaces.
11. Incrementally cut over to the target IS.

One of the key issues with gateways is ensuring update consistency. Brodie and Stonebraker themselves note that no general solution exists to that problem and as a consequence specific solutions need to be manually developed, which constitutes a tedious and error prone task. The second problem is the placement and integration of the gateways. Ideally they need to be placed between application and database to keep them as simple as possible. However, this is only possible in decomposable legacy systems, which are in their pure form, incredibly rare in real life applications. In reality, as pointed out by Wu et al. [135] a coordinator as depicted in Figure 2.4 is frequently needed. Integrating any gateway in the legacy system furthermore requires changes to it, a system that by definition significantly resists modification.

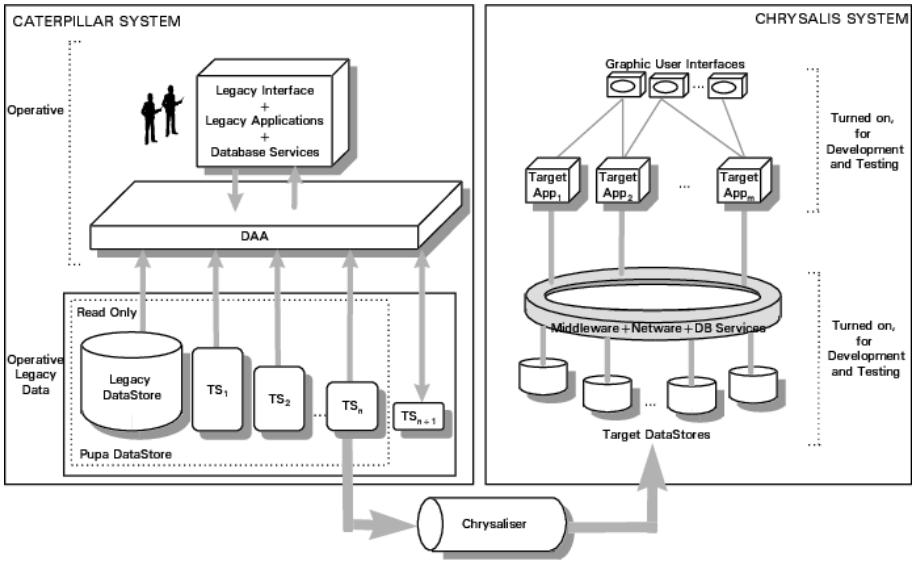

Figure 2.5: The placement of the Crystalizer and DAA components. [135]

2.4.2 Butterfly Methodology

To address these shortcomings of existing methods, specifically the Chicken Little approach Wu et al. have devised the Butterly Methodology [134, 135]. The method eliminates the need for gateways and for the users to simultaneously access both legacy and target system. This is achieved by not having both systems in production at the same time. The focus of this approach lies on the migration of the most valuable legacy assets: the data. The basic strategy is to decide on an initial state of the legacy data that is subsequently frozen to be read-only and then migrated to the target system. Meanwhile, the legacy system is still running in production and changes to the (frozen) data are stored in a separate, temporary data store. Once the main migration is finished (which can take a very long time), the first temp store is frozen, the migration is started again and for new writes a new temp store is created. This process is conducted iteratively until an exit condition is met. The exit condition is typically a function of the size of the temporary data store which is used to predict the amount of time necessary to migrate the available data. The goal is to exit at a point when the remaining migration is fast enough to warrant the shutdown of the legacy system and therefore to obtain a final state for the cut over to the target system.

The Butterfly methodology consists of a total of six phases:

- Phase 0: Prepare for migration
- Phase 1: Understand the semantics of the legacy system and develop the target schema(s)
- Phase 2: Build up a Sample Datastore, based upon the Target SampleData in the target system
- Phase 3: Incrementally migrate all the components (except for data) of the legacy system to the target architecture.
- Phase 4: Gradually migrate the legacy data into the target system and train users in target system.
- Phase 5: Cut-over to the completed target system.

One of the key criticisms of this approach is its abstract level of description and the hidden complexity of its "magic" components (see also Figure 2.5). The "Data-Access-Allocator" (DAA) needs to detect changes to a frozen data store, redirect them to a temporary data store and answer queries on that data correctly. The "Crystalizer" needs to migrate both the initial frozen store and subsequent incremental, temporary stores consistently into the target database. Both of these components are dependent on database technologies as well as the legacy and target database structures and hence specific to the individual migration scenario. Furthermore, they are highly complex and require a large amount of specialised effort to build [85]. No guidance is given to the prospective user of the method on how to design and develop these components. Finally while the method is designed to construct and test the target system incrementally, the cut over between legacy and target system happens in one step, which inherits many of the inherent risks of a big bang ("Cold Turkey") migration approach.

2.4.3 Architecture-Driven Modernization (ADM)

Architecture-Driven Modernization (ADM) [125] is a very high level modernization framework placed in the setting of the Object Management Group (OMG), a standardization body. It is geared towards an IT strategy und Enterprise Architecture perspective found primarily in upper management. While this of course means a higher level of abstraction, it also has the ability to give a full overview and deliver talking points when it comes to convincing decision makers about the benefits of a modernization effort.

Ulrich defines ADM as "a collective set of tool-enabled disciplines that facilitate the analysis, refactoring and transformation of existing software assets to support a wide variety of business and IT-driven scenarios." [125] The main driver behind this approach is the realization that previous industry solutions like Greenfield replacement, COTS deployment and wrapping including integration with a middleware component rarely lead to the desired or promised results.

ADM defines three major categories of modernization disciplines: assessment, refactoring and transformation. The goal of the assessment tasks is primarily the exposure of existing assets to the required level. This can be compared to a reverse engineering of the enterprise. It can lead to the decision for and setup of a subsequent modernization project. The goal of refactoring tasks is to improve the situation of existing systems by addressing findings of the assessment stage. It can be used as a direct and standalone measure to increase the maintainability and adaptability of a system or as a preparatory step for a subsequent transformation. The goal of transformation is to move the existing data and functionality to a new target architecture. In all three stages Ulrich argues strongly for a focus on the business perspective of modernization with a clear need for return on investment (ROI) calculations and an active involvement of business users.

While it is certainly true that some modernization efforts are strongly driven by the IT technical side of an enterprise, the proposed solutions concentrate very heavily on the business side. For the engineer this has the downside of receiving strategic directions and decisions that have not been baselined against technical realities. In addition, a further point of criticism can be the bias towards OMG's own standardized architectures, SOA and especially model driven architecture (MDA). This does not seem to match what is being applied in the field as Botto-Tobar et al. point out in the context of SOA to cloud migration: "MDD approach had been rarely used in the process to migrate SOA applications to Cloud environments" [29].

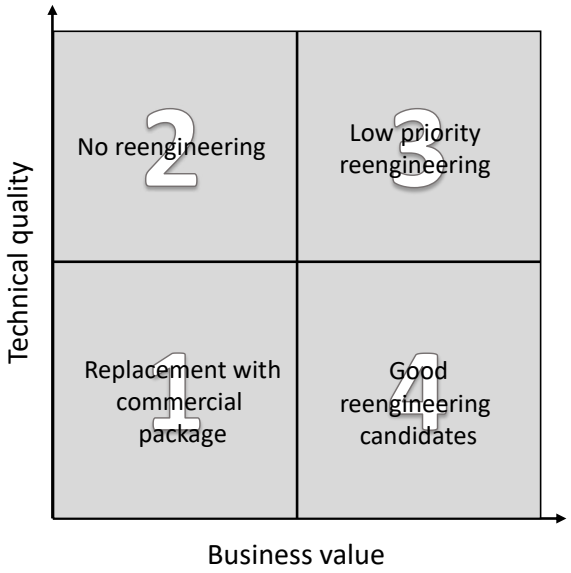

Figure 2.6: The protfolio analysis graph used to identify possible targets for modernization, after [104].

2.4.4 Risk-Managed Modernization (RRM)

Risk-Managed Modernization (RMM) as presented by Seacord et al. [104] is a modernization framework on a similar level as ADM in the previous section. The process starts with stakeholder and requirements analysis and ends with the definition of a modernization plan. However, the presented approach for planning a migration goes into a high level of detail and tries to bridge the gap between business and technical concerns. The portfolio analysis graph as originally defined by Warren et al. [131] is used to identify possible target systems for modernization. It highlights that good reengineering candidates are systems of a high business value combined with low technical quality.

The presented approach can be structured into three different phases. In the first phase, an initial analysis is done until a valid business case can be made. It contains an exit condition that allows to terminate in case this is not possible. The second phase is about establishing the current and target architectures. This means reverse engineering on the one side and designing a suitable target architecture on the other. The third phase iteratively refines the migration plan both from a technical and business perspective until a feasible plan is found and the actual modernization project can be started. Apart from establishing the detailed target architecture and planning and estimating the cost of the overall strategy there are dedicated steps for considering improvements to the legacy system prior to migration and for carefully planning both code and data migration. It concludes with a series of general recommendations on topics like incremental development and deployment, continuous integration or software engineering skill management. Overall the proposed method delivers a detailed way of planning a modernization effort.

One of the key issues that can be found with the given approach is that a detailed planning is not followed through by guidelines for actual execution. While a lot of modernization project failures can be attributed to poor planning and/or a lack of understanding of either the legacy system or the organization using it, the saying “no battle plan survives first contact with the enemy” (by Helmuth von Moltke the Elder) exists for a reason. Especially with risk management as part of the

name, it should be made very obvious to the user that identifying and managing risk throughout the modernization effort is an integral part to ensure success.

2.4.5 SOAMIG

SOAMIG is a research project that provides an adaptive, iterative general migration model [68]. It is structured in four organizational phases. The preparation phase deals with pre-renovation, project setup and tool initialization. The conceptualization phase is focused on establishing technical feasibility including a prototype migration and also includes tool adaptation as necessary. The main migration phase consists of a series of disciplines that are conducted iteratively until the goal of a running target system is achieved. The process concludes with the transition phase which focuses on post-renovation cleanup and refactoring to achieve the goal of a maintainable and adaptable system.

The following disciplines have been identified by the SOAMIG process for iterative execution:

- Business Modelling
- Legacy Analysis
- Target Architecture
- Strategy Selection
- Realization
- Testing
- Cut Over

Criticism of the SOAMIG process model includes that its applicability is narrowed down by focusing on model driven engineering techniques and automated code transformation. In additions neither the case study nor the presented tool set for the execution of the migration seem to be finished [139]. Furthermore, although an incremental approach is presented, no guidance is given to the user with respect to the incremental cut over and deployment of the target system and the technical challenges associated with it.

3 Objects of Investigation

3.1 Approach

In Section 1.2 the general approach and high level selection criteria were outlined. In this section the goal is to highlight the main decision points for including a unit of investigation as a case in this thesis. A total of 11 possible units of investigation were evaluated. In the end four cases were selected as the best matches. Only selected units of investigation will be represented in the remainder of this chapter.

The four selected projects are:

- Insurance: A renovation and consolidation project in a heterogeneous IT landscape with the goal of providing a central platform for handling master data and automate business processes.
- Ministry: A replatforming (see Section 2.1.6) effort with the goal of freeing up resources to subsequently incrementally replace a very large government IT system.
- Airport: An incremental replacement of a highly critical core information system that digitally mirrors the state of an airport.
- University: A replacement project that consolidates a highly heterogeneous landscape into a coherent enterprise management system.

The following selection criteria (see Section 1.2) were set out during the design phase of the concept behind this thesis:

- **Relevance** of the legacy system to the organization.
- **Age** of the legacy system.
- **Size** of the legacy system.
- **Need** for renovation.
- No possible (easy) **substitution** with a **COTS** product (SAP or similar).

The remainder of this chapter will be dedicated to the analysis of the selected units of investigation with respect to the given selection criteria. Please note that only short summary arguments are provided. This has been done to avoid redundancies with the subsequent detailed presentation of the units of investigation in Chapter 4.

3.2 Legacy Characteristics

Table 3.1 gives an overview of the main legacy system characteristics of the selected units of investigation.

		Insurance	Ministry	Airport	University
Need	**LS decomposable**	semi	semi	non	yes
	LSM type	partial replacement, wrapping	replatforming	replacement	replacement
	LS composition	heterogeneous	homogeneous	homogeneous	heterogeneous
	LS main technology	PL/I, COBOL, proprietary, scripting	PL/I, assembly	COBOL	COBOL, PL/I, Smalltalk, scripting
	LS database	relational	relational	file-based	relational
Age	**Age**	> 20 years	> 40 years	> 30 years	> 40 years
Size	**Size**	2.5 MLoC*	5 MLoC	250 KLoC	2 MLoC*
Relevance	**# of Users**	6.000	12.000	1.000	6.000 employees, 40.000 students
	Key purpose	customer acquisition and management	taxation and benefits	central operational database	campus management system
	Significance	complete master data, all claim and file handling	complete inner workings of ministry	manages all information distribution	complete IS
	Downtime until stand still	12 hours	6 hours	4 hours	12 hours
COTS	**COTS available**	no	no	yes, deemed infeasible	yes, deemed infeasible

Table 3.1: Overview of the main (legacy) characteristics of the four selected units of investigation

- **LS decomposable**: The decomposability of a legacy system [32] is the most relevant characteristic to assess the possibility of applying incremental approaches. As illustrated by the analysis of the units of investigation, the distinction between decomposable and semi-decomposable is vague, especially in the context of heterogeneous system landscapes.
- **LSM type**: The type of migration [104] indicates the planned approach and degree of change for the effort in question. Only replacement efforts were selected with the exception of the Ministry case where only the initial replatforming step was included due to the long running nature of the overall replacement effort.
- **LS composition**: The composition of a legacy system affects many of its characteristics. Homogeneous systems might exhibit benefits in reverse engineering but can be harder to decompose. Heterogeneous systems on the other hand are harder to comprehend due to technology and paradigm breaks, but are frequently already decomposed into several subsystems. Both technologically heterogeneous and homogeneous cases were selected to be able to highlight the differences as well as commonalities.
- **LS main technology** is used to highlight the main technologies in use, to bring a perspective on the technological state of the legacy system. All cases make extensive use of old technology that is still widely found in an industrial setting but scarcely used to build new systems.
- **LS database**: The technology used for data storage is key to understand the challenges in the context of data migration, a prominent success factor in any LSM effort. It is also a good indicator if the legacy system has already undergone previous migration towards relational database systems. Only the airport case was still using a file-based database system.
- **Age**: The age in itself is a bad indicator for finding a legacy system. However, it is a helpful measure in judging the amount of technical evolution and functional modification a system has undergone. All of the selected cases represent decades of IT history.
- **Size**: As with age, size alone is not a reason to classify a system as legacy. Nevertheless, it serves as an indicator to distinguish between trivial tasks and monstrous challenges. This characteristic was mainly used to eliminate potential cases that were too small. Values marked with an asterisk ('*') are estimates only, due to the heterogeneous nature and high share of scripting components.
- **# of Users**: The number of users actively engaging with the system on a regular basis is primarily an indicator of relevance. Even a large system displaying strong legacy system characteristics will not be a good candidate for migration if it is hardly used. It can also be treated as a secondary indicator to gauge the size of a system. In this thesis it was utilized to make sure that a significant portion of an organization's workforce is in frequent contact with the system.
- **Key purpose**: The key purpose is used to assess the position of the system relative to business. All of the selected cases are central to the organization and therefore absolutely vital to handle their day to day business operations.
- **Significance**: The significance is used to describe the importance of the data managed by the system. All of the selected cases are handling the key entities of the organization in question and providing them to surrounding systems and business partners alike.
- **Downtime until stand still**: This measure is used to again gauge how essential the LS is to the survival of the business. All cases are mission critical systems. Service disruptions

	Insurance	Ministry	Airport	University
Relevance	✓	✓	✓	✓
Age	✓	✓	✓	✓
Size	✓	✓	✓	✓
Need	see next section			
COTS substitution	✓	✓	✓	✓

Table 3.2: Initial matching of criteria with LS characteristics.

cause significant limitations within short periods of time and lead to a complete stand still of each organization within hours.

- **COTS available**: The evaluation of suitable COTS is a primary strategic task in a LSM effort. The availability of a viable solution on the market is frequently unbeatable in terms of cost and time to market, although it is frequently associated with the need for significant organizational changes. Half of the cases in this thesis did not identify any suitable COTS. The other half did identify such possibilities, but prior evaluations showed that the transition to a COTS solution would be deemed infeasible or a strong interference with a perceived competitive advantage gained through an individualized solution.

To match these characteristics with the selection criteria, a mapping has been created. This mapping is also illustrated by the first two columns of Table 3.1.

- The **Relevance** of the legacy system to the organization can be judged by the characteristics **# of Users**, **Key purpose**, **Significance** and **Downtime until stand still**.
- **Age** of the legacy system is matched by the corresponding characteristic.
- **Size** of the legacy system is matched by the corresponding characteristic.
- **Need** for renovation is affected by the characteristics **LS decomposable**, **LSM type**, **LS composition**, **LS main technology** and **LS database**.
- No possible (easy) **substitution** with a **COTS** product is addressed by the characteristic **COTS available**.

Combining this mapping with the information from Table 3.1 brings us to the results shown in Table 3.2 providing a clear answer for four out of five previously defined criteria.

3.3 Factors for Legacy System Classification

The criterion **Need** for renovation can be inspected more deeply, although the already analysed characteristics might be considered sufficient to argue a need. For a closer look the following factors (as defined in Section 2.1.1) that play a role in classifying a system as a legacy system can be utilized. It must be noted that there is a strong relationship between the previously analysed

	Insurance	Ministry	Airport	University
End of Life	partly	no	yes	yes
Knowledge	yes	yes	no	yes
Skills	yes	yes	yes	partly
Extensibility	yes	yes	yes	yes
Time to Repair	partly	partly	no	yes
Scalability	yes	yes	yes	yes
Limitations	yes	yes	yes	yes
Costs	yes	yes	yes	yes
Overall Need	✓	✓	✓	✓

Table 3.3: Factors classifying the selected projects as legacy systems.

characteristics related to need and the factors discussed in this section. For instance **End of Life** is directly related with the characteristics **LS main technology** and **LS database**. It can therefore be argued that the factors are used to evaluate the given characteristics of a system in order to reach the classification as a legacy system.

Judging most of the given factors can only be done in a subjective manner. In the instances where it is theoretically possible to perform measurements, the lack of data in a legacy environment makes it practically impossible. However, despite these constraints, a definitive "yes" can be argued in most cases (see Table 3.3).

The insurance case does use technology which has reached end of life, both software in terms of programming languages, environments and frameworks and hardware. A continuous loss of knowledge can be observed as most of it is only in the heads of the product teams. The loss is speeding up as team members are reaching retirement age. Replacing them is increasingly difficult due to a lack of availability and the fact that the organization cannot compete in terms of salary packages with stronger competitors (mainly banks). Extensibility and Scalability are affected by lack of resources and ageing design. A strong factor here is that systems were not designed to directly interact with the customer and therefore incompatible with upcoming e-Government strategies. This can also be seen as a major limitation. Both time to repair and costs are secondary factors in this case, but are far away from figures expected for a modern information system.

The ministry case revolves primarily around the cost and limitations for running the information system. While the platform is still supported and therefore not technically end of life the costs are prohibitive. Knowledge and skills are a second major factor as it becomes increasingly difficult to obtain and retain professionals with Cobol and Host-Assembly skills, which also indirectly affects the time to repair and extensibility. A key limitation is the difficulty of integrating the information system with modern parts of the IT landscape. Scalability is indirectly affected by a related direct increase in cost due to the licensing model of the platform.

The airport case has a strong focus on end of life. Only a handful host systems of this vendor were in production at the time. Licensing costs were accordingly high. Intimate knowledge of the system was retained by no more than two engineers, both of them close to retirement. No

	Insurance	Ministry	Airport	University
Relevance	✓	✓	✓	✓
Age	✓	✓	✓	✓
Size	✓	✓	✓	✓
Need	✓	✓	✓	✓
COTS substitution	✓	✓	✓	✓

Table 3.4: Complete matching of criteria with LS characteristics and classification factors.

new engineers were hired due to a lack of experts on the market and long standing plans to decommission the platform. Extensibility and scalability were also drivers, as it deemed infeasible to adapt the legacy system to upcoming changes introduced by a major expansion of the airport. Some limitations were secondary drivers like the lack of interfaces to retrieve flight information and some decade old design decisions that lead to costly workaround for certain corner cases.

The university case mainly revolved around the heterogeneous nature of the information system. Some parts of it were end of life. Other parts incurred licensing costs. Knowledge was distributed among some veterans, but increasingly scarce. The heterogeneous ecosystem also required a broad and unusual combination of skills that were near impossible to find on the market. Due to the extremely long history of the system, extensibility and scalability were severely constrained. The system was exceeding its capacities at the start of every semester. Limitations were numerous, especially when it came to the integration with modern e-Learning tools or other student services. Most non-critical errors were no longer repaired, the ones that were, were perceived as open-heart surgery and workarounds were strongly preferred by most engineers on the team.

Having completed the analysis of the factors for classifying the units of investigation as legacy systems, the results can now be combined with the findings of Section 3.2 to complete the picture. The final outcome is illustrated in Table 3.4. It shows that the chosen units of investigation in fact exhibit the expected characteristics and therefore fulfil the initially outlined criteria.

4 Case Study

4.1 Case: A Modern Process Management for 6000+ Employees

This case has been the main industry focus of the author for more than four years. Unlike the cases in sections 4.3 and 4.4 it has a stronger focus on the general renovation and modernization of the IT landscape of a large organization than on the reengineering of a single system. However, this overall picture involves several medium to large size system modernization efforts in various stages of completeness. The systems of particular interest will be highlighted in the course of this chapter.

The organization behind this case is one of Austria's main social security agencies. This organization constitutes one of several agencies, responsible for handling a core aspect of social security. In this ecosystem it is by far the largest representative and as a consequence taking a lead role. It annually handles and distributes a double digit billion Euro amount to millions of customers. We will refer to it as "the fund" in short in this case.

4.1.1 Status / Analysis

The fund is organized federally, similar to the state of Austria itself. The head office is situated in Vienna, collocated with one of its nine regional branches responsible for the city state of Vienna. The other branches are running offices located in the capital of the federal state. All branches together employed a total of around 6200 employees in 2013.

In 2010 the fund initiated the project in cooperation with the partner agencies with the main goal of, for the first time, providing a single process oriented front end application and integration platform for all business processes related to benefits of age. A major secondary factor in this effort is the inception of a "paperless" process - a radical step from an established, file based process involving the logistics of moving tons of paper every day including a sophisticated file moving conveyor system. This can be seen as a single project with the final aim to fully digitalize one of the core business processes.

Not included in the initial scope of the project was a major legislative change, the result of the largest administrative reform in Austria to date. It however had a major effect on the project forcing it to prematurely deliver several artifacts. The main goal of this change was to ensure transparency for the end user in a timely manner. To achieve this a large scale initial data gathering and validation effort was necessary as well as deep changes to the underlying business processes. It can be regarded as the major disruptive event of this case.

Organization

The complete development and operation of the core IT systems of the fund is currently done in-house, although especially in the area of software development, but also operations, the organization significantly relies on external resources. Traditionally there are three departments all located in the main branch charged with this task, a fourth was added towards the end of the project:

- ORG (Organisation): this department is responsible for business analysis and IT business alignment
- OPS (Operations): this department is responsible for operations and customer care
- SDE (Software Development): this department is responsible for software development and maintenance
- QSE (Quality Assurance): this department was created in the late stages of the project with the goal of consolidating all quality assurance aspects in a competence centre

To accommodate the additional resources for the project (about 90 internal and 60 external team members) administratively a separate department was established for the duration of the project. This department can be abbreviated with PRJ. Responsible for all of the mentioned departments and reporting directly to the board of directors is the director of IT.

The main focus of the organization and thus the way it is overall organized is not geared towards the execution of IT projects. The remaining organization largely represents a strictly hierarchical order to ensure processing of large amounts of claims and requests. These core business processes have been very well established. As a result IT together with other supporting disciplines are marginalized and politically underrepresented as well as frequently underfunded. Developer pay levels, for example, are aligned with case workers, which results in a non-competitive salary compared to similar roles in industry. The role of IT is seen as one of many helper functions (like mail carriers) that are completely subordinated to the interests and goals of the core business processes. This is important to understand as it makes the position of IT (and hence the project) much harder since the core digitalization effort is not primarily driven by those people running the core business.

Enterprise Architecture

The management of a customer base as large as the fund's (and its predecessor institutions') entails highly specific requirements for its information system landscape. This section will give a short and high level overview of the systems in place at the fund around 2013 as depicted in illustration 4.1. This illustrates clearly the interaction pattern between a case worker and the systems. The user is responsible for correctly orchestrating the individual steps of the business process. During this process users need to switch between different applications and navigate to the proper menu items/screens. Screens are typically geared towards technical modules, for example manipulating a specific data type and so might be used (differently) in various situations. Most of the screens are fat client systems running on the PC or thin client of the end user, which have replaced prior VT100 or 3270 terminals.

Being a "file-oriented" institution, at the core of the enterprise architecture one can find the systems for authoring (1) and storing/archiving (2) documents. Both of them are commonly used COTS products, but have received very different amounts of tailoring.

For the templating and document creation system (1) the tailoring (apart from definition and maintenance of document templates) is limited to providing web service APIs for producing and sending documents and data-warehousing related tasks. Basically it consists of four modules: the engine for producing documents and exposing the API, a rich client designer for producing and maintaining document templates, a repository for storing and versioning document definitions and a standalone, web based client for generating documents out of templates.

For the document archive (2) the main focus of tailoring should be the maintenance of a consistent and well defined meta-data structure for documents and the retention of documents in formats

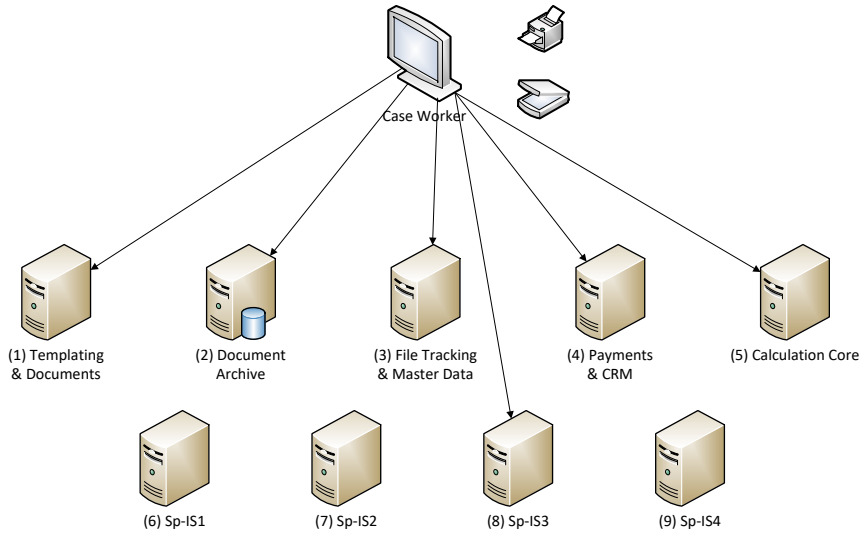

Figure 4.1: The enterprise architecture prior to the project.

that are highly resistant to change (legal requirements force retention times of up to 30 years for paper and digital documents) and the handling of very high volumes of data. However, mainly due to the lack of adaptability in the core information systems, a whole series of data management applications (3) for maintaining customer, claim and paper file data has been created (or rather grown). These applications are named after the company selling the original product for the lack of a more common denominator, have no strong cohesion with the core document management system (DMS) but are responsible for most of the maintenance effort.

The real core information system of the fund however is the payments and CRM system (4). The official name is the abbreviation of a German phrase which freely translates to “consistent application of state of the art technologies”, a statement that might have been true approximately 25 years ago. It is based on the Siemens OpenUTM transaction processing facilities running on OS360 with its programs mainly implemented in COBOL and PL/I. It is backed by an Oracle database with queries being translated from Sesam to SQL by a proprietary and closed source component. The front end is realized as a fat client using SyBase Powerbuilder, an integrated development environment focused on easy development of 2-tier data manipulation applications now owned by SAP. The core task of this information system is the management and issuance of payouts and it is therefore by far the most mission critical system in the institution. It currently handles the payment of more than 1.9 million benefits per month. While it has been considered a prime candidate for a large reengineering or renovation effort for years the high risks combined with the criticality to day to day operations have so far limited the decision makers’ ambitions in this direction.

The closely related application (6) Sp-IS1 (Specialized Information System 1) runs on the same technology and provides additional, niche functionality for specialized services of the fund.

A special status has been attributed to core calculation system (5). This information system is used to calculate the aggregated data from an individual's history (typically between 30 and 45 years in annual statements) into the form necessary for processing a claim. Originally developed as a COBOL application and run by the umbrella organisation it underwent a major migration effort in 2010. Its source code has been semi-automatically translated from COBOL to Java in an ongoing effort to migrate all mainframe applications by 2020. While the front end has retained its Sybase PowerBuilder fat client, the back end uses the institution's standard Oracle database. The Java part however is strongly influenced by its history of automatic source code translation and strongly resembles its procedural COBOL structure. Also large amounts of glue and framework code are necessary to enable the COBOL-like data access and manipulation in Java forcing the execution of the application on a legacy application server using the Java EE 5 standard.

Sp-IS2 (7) and SP-IS3 (8) are again used to support specialized business processes including re-funding costs to public and private institutions and as a result handling significant sums of money. Their functional scope coincides with legislative responsibilities passed to the fund at various stages. IS2 is a mainframe-based system running on an IBM OS390 host, using an Object Star integration layer and storing data in an IMS database system. IS3 is mostly used to process claims and to handle the ensuing correspondence with government offices and doctors. Again the system consists of an Oracle database and a fat client developed with Sybase PowerBuilder. This fat client is tightly integrated with IS2 using the Object Service Bus of Object Star to communicate.

The remaining applications IS4 (9), IS5 and IS6 (not depicted) are basic two-tiered data processing systems. All of them are composed of a PowerBuilder fat client front end and an Oracle RDBMS back end. However, these systems are of lesser relevance and would as a consequence be placed in the first quadrant of Warren's portfolio analysis graph (see section 2.4.4).

Apart from the communication between the user and the individual information systems through the fat client, the systems themselves are integrated mostly at the database level. This is all done directly doing point to point communication, resulting in a star like communication pattern. Cross-database access is not documented. Frequent reuse of credentials without access restrictions makes it difficult to reverse engineer detailed communication patterns. A further common communication pattern is the orchestration through automated batch tasks. This frequently entails extraction of information from several information systems, aggregating and filtering the data and subsequently creating reports or documents to be sent out to customers or other institutions. Batch processing can be scheduled or manually triggered depending on the use case. This is also a common pattern to digest data originating from or going to third party organizations or the umbrella holding. Only a few online interfaces (e.g. web services or message queues) are integrated.

4.1.2 Solution

The question at this point may be, what exactly this project has to do with legacy system migration? It is true that from reading project definitions or goals it is not initially discernible from a general business process automatization project. The main reason is that the reengineering and migration "part" of the project has been strongly neglected during the planning stages of the project. It is therefore a migration project "in disguise". The remainder of this section will explain in detail, how the project was retroactively outfitted to accommodate the needs of a legacy system migration project. This is best done by looking at the evolution of the projects over time, as both goals and approaches have been refined significantly along the way.

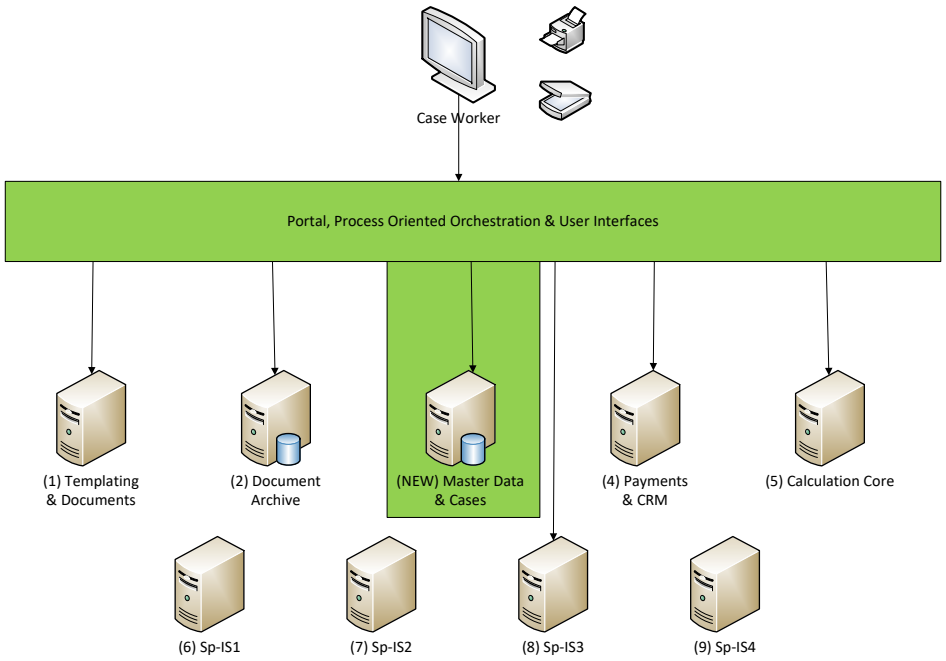

Figure 4.2: The target enterprise architecture.

Target Architecture

From an architecture perspective the target was the abstraction of the underlying specialized information system from the end user. The core business processes were to be automated to move the task of orchestrating from the end user to the information system. A high-level overview of the concept is shown in Figure 4.2. In addition, a centralised storage of master and file data replaces some systems completely and carves out significant pieces from others. This target architecture is consequently a core enabler for later modernization efforts of specialized information systems, with one of the main goals being to abstract away immediate effects from the end user.

The project

As already indicated in the introductory part of this case, the inception of the project dates back to around 2007 and its official kick off was in 2010 with the two main goals of creating an integrated, process driven, multi-tenant platform for all business processes supporting the long term goal of a paper-less office. A high level overview of the timeline is given in Figure 4.3.

The complete project is divided into a total of seven sub-projects. However, detailed planning and financing has, at the time of writing, been only secured for sub-project 1 (SP1), which was originally scheduled for completion within four years of the project kick off and has undergone several replanning, restructuring and rescoping efforts. The following section will thus focus exclusively on SP1 and will only address past project-plans where necessary to understand the complete picture.

Scope

The scope of SP1 comprises two main goals which have been used to break down the organization of the project into teams.

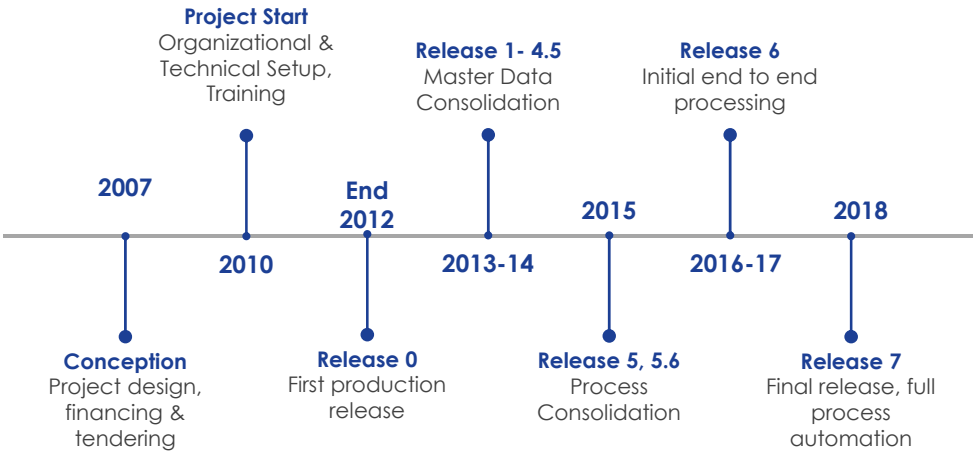

Figure 4.3: A decade of software project. The timeline of the insurance case.

"Central Services" has the goal of providing the necessary (software) infrastructure for the remaining project and constitutes the initial phase of the project. This includes the integration of the process engine, consolidation of master data, integration with external systems and the design and implementation of a series of cross cutting concerns.

"Core area or competency" has the goal of fully supporting all accession cases for the core lines of business of the fund. The support is defined by the process starting with the receipt of the claim and typically ending with a positive or negative decree.

Inception Phase

The focus of the inception phase was the acquisition of all necessary software and hardware components, the development of a first proof of concept implementation, the creation of a first detailed project plan and the project team organization.

The acquisition was done with a single tender process comprising both hardware and software. The hardware part required a two-location, virtualized, Unix-based solution. The software part required a full scale Java SOA-Stack comprised of identity management, application server, rule engine, process engine, enterprise service bus, service registry, portal server as well as all necessary design and development tools with a strong focus on model driven development. This design limited the possible candidates to the few big vendors that were able to provide the full portfolio and the required level of support. The requirement for providing all tools as well as support in German language further limited the range of possible candidates. However, the tender expressly only included the inception phase of the project including initial training, forcing any vendor to go through separate tendering processes for providing consulting services throughout the remainder of the project.

After completion of the tender, a proof of concept (PoC) was done using the newly acquired tools. Apart from the setup of the necessary development environments the goal of the PoC was to completely automate an (albeit simplified) business process as well as to create drafts for the main architecture and design documents. On the organizational side a waterfall like process model was adapted and the project members structured and staffed into a total of nine teams. (requirements & business analysis central services, requirements & business analysis accession, process modelling,

service modelling, domain modelling, development, test and project administration & controlling, operations[1]).

All internal staff was sourced in-house with little to no requirements on previous experience in the respective field (for example some of the people staffed in a developer's position did not have any previous programming experience). As a result a significant amount of time and resources was necessary to provide the training to bring the internal team members to (or at least near) the required skill levels.

During this training period, external support was acquired with a series of additional tenders. Approximately one third of the staff was planned to be provided by external partners, with the bulk of the resources going into the modelling, development and test teams. However, none of the key project positions (project lead, team lead, business or enterprise architect) were staffed with external resources.

The initial project plan for SP1 was comprised of a total of 9 incremental releases, evenly distributed over the originally planned duration of four years, resulting in the order of one release every second quarter. The first five releases were dedicated to central services, the remaining one to the features summarized under "core area". Only the releases relevant to this case study will be presented in some detail. It is important to note, that these plans only contained features of the new system, but not the dependencies and effects on the surrounding systems. For example the decommissioning of parts of a legacy system no longer needed after a certain release would not be visible from this plan.

Release 0

As already indicated in the introduction to this case study, a significant administrative reform was passed by parliament not long after the project started. This legislation fundamentally changed how the benefits of the fund would be calculated and were aimed at providing greater transparency to the citizens, the fund's customers. This decision forced the fund to perform a significant data acquisition and validation project. This effectively meant contacting all eligible persons, tracking the status of their responses, evaluating these responses, asking for additional documents where necessary and sending out a final, official statement once the process was finished. Several million people were contacted in this way. All of this was, in style and method, very similar to well established business processes. The scale, however, was completely different and the amount of work to be done orders of magnitude higher than anything previously handled by the organization. This process and the associated requirements were added after the project had officially started.

The subsequently added Release 0 was accordingly the main focus of the whole project (excluding the second business analysis and requirements team) for the first two years resulting in a (little big bang) release at the end of 2012. None of the software was properly tested under production-like conditions. For example no performance or load tests were conducted. In addition, bringing live the complete architecture at once represented a completely new situation for all involved stakeholders. This combined with the necessity to produce and handle millions of cases without any kind of ramp up period were the foundations for a very rough start bringing the whole project to the edge of cancellation multiple times. The ensuing months were completely defined by highly critical fixes, functional adaptations, massive performance problems and, to top it all off, a load of bad press as the system was sending out erroneous mail to thousands of recipients countrywide including some who had already passed away.

[1] this team was dissolved early on in the project and its responsibilities and staff reassigned to the OPS line organizational unit

Release 1 (Master Data consolidation)

In the wake of the disastrous Release 0, Release 1 was the beginning of the effort to bring the project back on track (and to redefine the track). Building on the experiences of the previous months, Release 1 was done using an alternative approach. A team staffed completely with external resources was tasked with building a prototype on a more lightweight architecture. This step was necessary as apart from having to address the architectural and technical shortcomings of the previous release, all existing resources were still fully bound on maintaining Release 0.

Unlike the previous, strongly process oriented release, the key goal of Release 1 was the consolidation of master data and subsequently providing a new user interface to manage all the data. This addresses one of the key weaknesses of the original system architecture which forced each expert system to hold and manage its own master data.

The resulting prototype was subsequently enhanced to be a completely separate application. The main effort however consisted of a major data migration consolidating the master data in the new database as well as a complex, three way synchronization application attempting to keep the data consistent over three systems (two of which allowed simultaneous editing).

Release 2, 3, 3.1, 3.5

These two releases as well as two subsequent minor releases focused on feature development in the area of customer service, document creation and editing as well as document scanning and distribution. The main goal was to provide additional business value based on what we had learned from the initial two releases as well as base functionality needed for subsequent releases.

Only 3.1 featured a major reengineering task. This was the complete rewrite of the user interface produced in Release 0 on the basis of the technologies introduced in Release 1. This can be seen as a major step towards a full technological consolidation. It also highlights the two levels of legacy the project was dealing with. On the one hand, the legacy provided by the old systems with legacy technologies, on the other hand the legacy introduced during the early stages of the project.

Release 4

This represents the final step in supporting the consolidation of the master data establishing the new application as the single leading system for customer data for the whole organization. This means that the editing support for customer data was deactivated in all other systems, except a single outlier which was to be replaced by a re-engineered system by the end of that year as part of a project running parallel to it. Instead of the (bidirectional) synchronization mechanism of Release 1, a new “push style” notification was designed to inform connected systems of changes. Therefore the highest complexity in this release was the coordination with all receiving systems and the respective regression testing. In addition, the data model was extended to support full versioning and historization and synchronization capabilities with the master data at the umbrella organization.

Since not all types of changes can automatically be applied to the receiving systems in all cases, technical processes were devised to alert users to manually perform the relevant changes and initiate subsequent actions. For example one system requires certain changes to master data to trigger a manual re-evaluation of ongoing payments. All in all the complexity of the release caused a series of delays cumulating in a delivery more than six months after the original planning.

Release 4.5

This is another strategic release, planned and introduced last minute and functionally carved out of the subsequent Release 5. This was mainly necessary because of the aftereffects of the massive Release 4 delay causing Release 5 to be pushed back by a quarter. The main functional driver of this release was a single feature, an annual correspondence with recipients of payments living

abroad to make sure they are still alive, originally not in scope of SP1. However, due to the planned shutdown of the respective legacy system with Release 5, this feature had to be re-engineered and brought into production in advance to the yearly batch of more than 100 000 requests being sent by postal mail.

On a technical side, this feature was the first to go live on the replacement process engine. It therefore concluded a half year parallel development effort to bring in a lightweight, integrated and open source BPMN 2.0 process engine instead of the bulky and costly BPEL process servers in use to date. The new process engine and its process definitions are now bundled with the remaining application in a single deployable artifact. As a proof of concept for the possibility of a seamless switch from one engine to the other, the external customer service features developed in Release 2 were reengineered on the new engine.

Release 5

From a functional perspective, this release is the first large chunk of functionality after Release 1. From a reengineering perspective, as already indicated in the description of Release 4.5, the main focus of this release is the replacement of the legacy file tracking and master data subsystem. It mainly consists of two blocks: The first is the complete file tracking application, enabling users to order up, trace and manage paper files. The second is the basic claim management supporting the user from claim entry (claims are at this point still paper based) to settlement. Both of these functionalities are provided with little to no support in the process engine, with the exception of a new file editing feature.

Additional complexity in this release comes from the required data migration efforts affecting both functional blocks. Both claims and files have been tracked in the legacy system for years and accordingly billions of records exist that have to be consolidated and brought into the new database schema. Especially the claim data is also used by many other systems. Significant effort and coordination were thus necessary to ensure the proper integration with these systems.

Finally as the user base of the application is increased more than five-fold to all previous releases both user training and a specific focus on performance, load and scalability testing are major focus points.

Release 5.6

This release marks the end of the legacy mode for Release 0 features. While database (Release 1), services (Release 2) and user interfaces (Release 3.1) all have been replaced to conform with the target architecture in previous releases, the processes are still running in the original BPEL process engine. With this release all definitions of all automated processes will be ported to BPMN 2.0. Subsequently all previously existing process instances will be brought over to the new engine at appropriate cut off points. As soon as all process instances are migrated the legacy process engine can be decommissioned just in time for its end of life. The approach was published in detail in [115].

Release 6

Following a major replanning this release constitutes the first of two in an effort to reach the goal of processing all accessions by the end of SP1. The main goal of this release is the digital processing of all incoming and outgoing postal correspondence, including those that will not be handled (functionally) throughout SP1. All incoming mail items are scanned, classified, tagged and archived and then routed through preprocessing to the respective departments for final processing.

The preprocessing consists of two sub-processes, one for the validation (or creation) of customer master data and the other for initial claim processing. Both of these steps are optional. A signi-

ficant part of the release is furthermore the fully automated processing and correct correlation of digitally coded (QR code) reply mail for inquiries sent from the new system.

As the main business processes are to be realized with later releases, a lightweight substitute was realized. This process consists of a single, re-entrant human task that can be seen as a kind of cockpit for the expert user to perform the tasks necessary for a specific item of mail. This includes limited functionality already in the final, process oriented state, which can be triggered as a subprocess. This functionality, which will be extended or at least revised during the release, includes handling of inquiries and their replies, review of all outgoing correspondence, external translation request, internal translation and request for medical assessment or statement.

From a reengineering perspective, a related system has been automatically transformed from COBOL/Object Star to a web-based Java application during the previous year. This enables much tighter integration between the two systems (e.g. single sign on and data exchange via web-services). The release of these two systems was aligned due to political reasons and to save on testing efforts, as integration testing between the two systems was considered to be a large part of the overall quality assurance effort.

Release 7

This (final) release of SP1 covers the core business cases of accession processing. Building on the foundation of the previous releases (especially Release 6) all main accession business cases will be fully supported and handled by the process engine. All necessary data can be collected in the main system and most interaction with surrounding systems is handled through web services resulting in a consistent, single workspace experience for all users. The success of this release has only been made possible by the continuous and intensive reengineering during the previous releases, paving the way for the final strong focus on functionality.

Ramp Down

After this release the organization of the project was itself dissolved. The project members were reassigned to their previous line organizations or posted to new ones. The further ongoing development is now again bundled with the respective organizational units and therefore in a tighter integration with the roadmaps for the other systems.

4.1.3 Experience

Architecture Transformation

As already indicated in the previous section, the system architecture envisioned at the beginning of the project was a complete SOA stack, a large array of semi-integrateable tools provided by a vendor to check off all the necessary items on a list. Without addressing the individual advantages and disadvantages of the specific tools, the critical aspect in this case is the massive mismatch between the power and complexity of the tool stack and the collective capabilities and experience of the organization. Basically the team was equipped with textbook knowledge of how to do "SOA" with the vendor's stack, a large volume of documents (several thousand pages) was created based on the official vendor technical white papers and then the project team was more or less consigned to its fate. In other words, while the tools themselves were possibly capable of solving the technical part of the challenge, the combination of tools and organization certainly was not.

The subsequently necessary transformation had to happen on two different levels. On a system architecture level the tools and components had to be adapted or replaced to suit the needs of the project. This can be seen as the "internal" architecture transformation. On an enterprise level the system landscape had to be adapted to make room for a new central system. Most systems

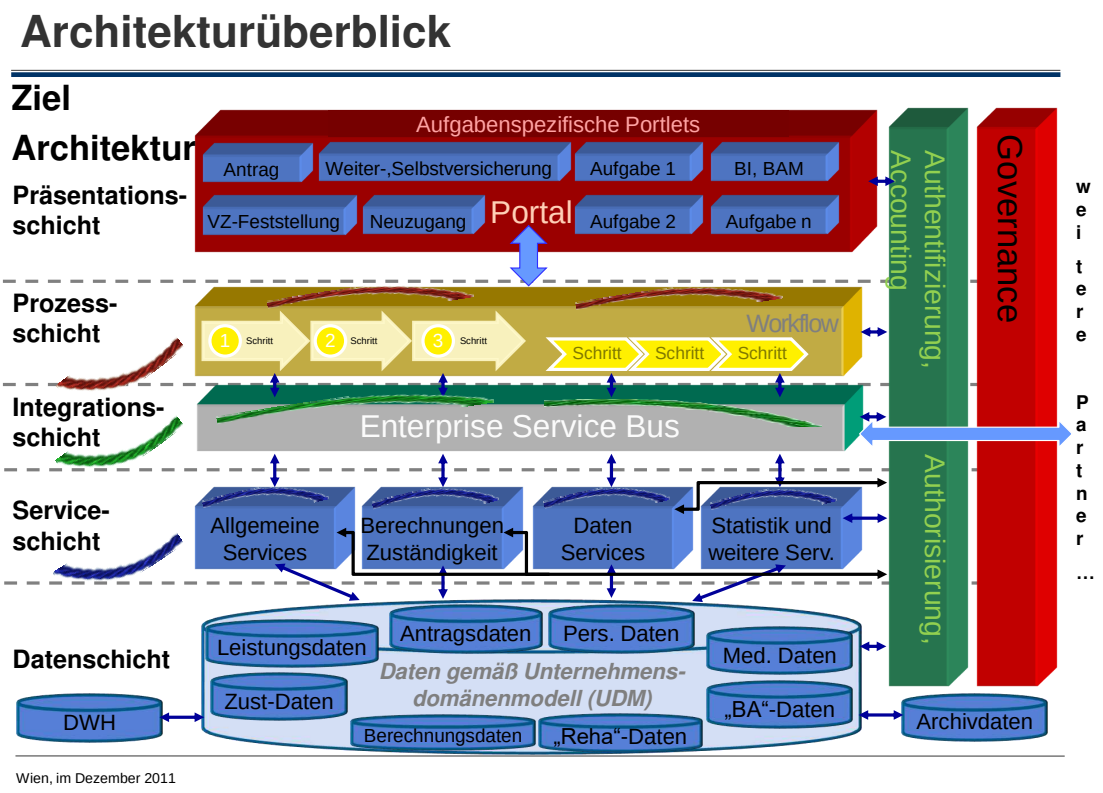

Figure 4.4: The original system architecture at project kickoff.

were affected in several key areas, mainly: data ownership, system integration, data processing paradigm and system load. The essence was the transformation of a batch oriented collection of isolated, monolithic systems into a collaborative, online processing landscape.

Internal architecture transformation

The following figures depict the initial system architecture (Figure 4.4) and development environment (Figure 4.5) for the target system as shown at the initial architecture presentation to the project team. These exports have intentionally been left unchanged from the original presentation and are hence in German language. They were only slightly modified to ensure the anonymity of the author and organisation.

The original target architecture can be summarized as a multi-layered application with each layer being represented with a separate tool chain and runtime environment. In addition, it was strongly influenced by model driven approaches with several attempts at code generation (e.g. database level, domain models, service skeletons). The individual tool chains are depicted in Figure 4.5. For each tool chain and layer a team of engineers was responsible. Creating the deployable artifacts involved numerous manual steps in each environment. Automation of these steps was not possible. Creating a complete release of the target system was consequently an onerous, error prone and time consuming task. Some of the tools were so complex that one or at most two of the team members were capable of producing correct deliverables.

To mitigate these issues a stepwise but radical simplification of the architecture took place between releases 1 and 7. One after the other the individual runtime environments were thrown out. The build process was gradually automated and consolidated. Different runtime components were transformed into modules. By Release 7 a completely automated build and deploy pipeline was

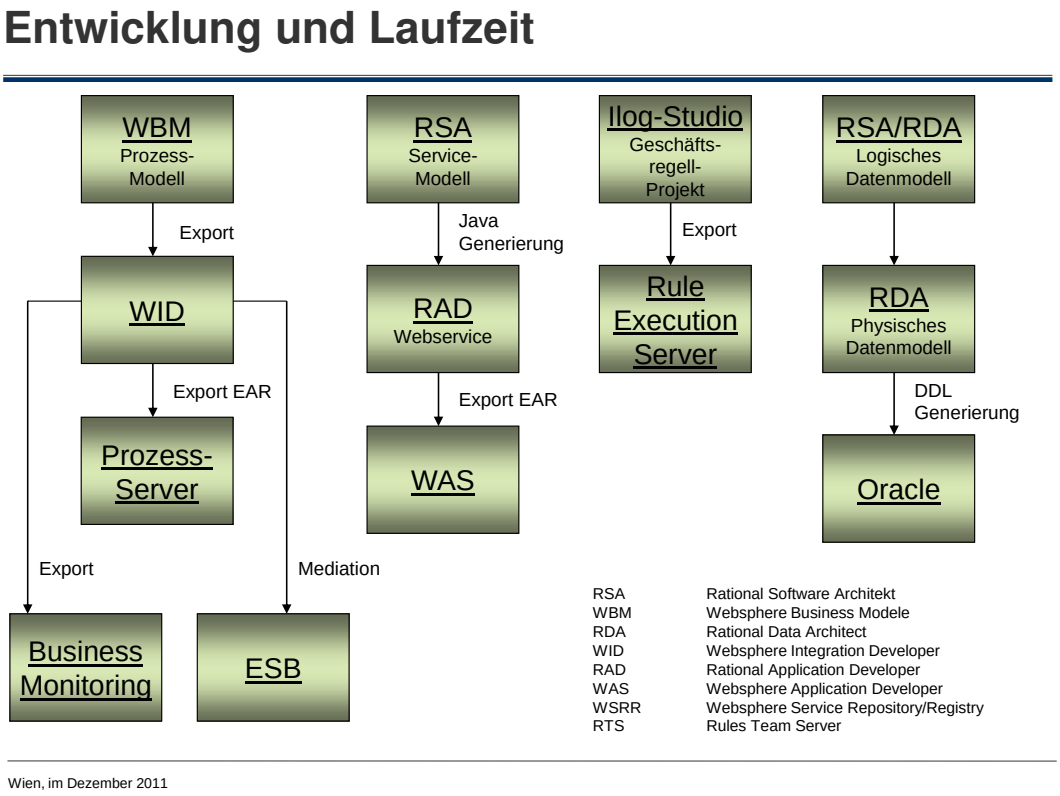

Figure 4.5: The original development environment at project kickoff.

able to deliver a new release to any environment at the click of a button. This was achieved while retaining most of the logical layers of the original architecture (the integration layer was discarded) and as a consequence conserving the original architectural vision. At the same time the tool set relevant for developers was reduced significantly with most of the tasks being handled by a single integrated development environment (IDE). Specialised tools were only necessary for specific tasks like business process modelling.

External architecture transformation

The external architecture transformation is, at a very high level, already depicted in Figure 4.2. Over the course of the project some systems were replaced completely, most of the others had to be significantly modified. The key driver here was the consolidation of master and file data. While legacy systems were allowed to have read-only copies of this data in their stores for performance and compatibility reasons, write operations were no longer allowed. Therefore modifications had to be done through the new system. The legacy applications were modified so they would be able to handle change notifications. This helped reduce the redundant maintenance of master data over several systems reducing overall data maintenance effort and at the same time reducing errors and inconsistencies.

The second main part of the external transformation was the transition from batch processing and database access to online, service based interfaces. This was necessary to enable orchestration through the business process engine and integration into a unified, task oriented user interface. This transition is a gradual process as more and more of the core business processes are automated and brought to the new platform. This is true not only for the information systems of the fund itself but also for its partner- and umbrella organizations.

Three main consolidation efforts will be highlighted in greater detail in the next section.

Migration strategy and architecture

Phase 1 - Master Data consolidation

One of the primary concerns with the legacy architecture was the fact that there was no consolidated view on customer master data. In fact each division and its respective system had its own master data management built in. As each division treated a subset of all customers throughout different points in the customer life cycle, often with years of inactivity in between, significant deviations in the quality and actuality of the data were the norm. Historically this was a minor problem as the possibilities for overlap were limited and the paper based file for each customer formed a unifying factor. However, with increasing efforts to keep people out of early retirement and the goal of creating a paperless, electronic file system, both of these constraints were no longer valid.

The consolidation effort started with the design of a completely new database schema, uniting the requirements of all legacy systems along with the additional goal of keeping compatibility with the centralized national system. One of the first main applications realized in the new system was consequently the management and information interface for customer master data. Featuring full historization and versioning this application manages personal customer information, addresses, relationships including legal entities and customer data per provider. The big selling point however was the full support for (semi-)automated alignment of the data between the fund and the national system using web services as well as providing web services for all in-house systems to access the data and a notification mechanism to distribute changes.

The main challenge on the other hand was the migration and cleanup of the legacy data. The legacy system's customer data schemas were highly different and often agreed on little more than the social security number and a couple of basic fields. Consistent however was the use of a single table for all data with very weak to no constraints to ensure consistency, which resulted in a significant gap between the legacy and the new schema. To accumulate data with as much consistency as possible the decision was made to enrich each person with the more up to date data from the national system for all customer data that did not currently receive any form of payments by the fund. For all other systems the order of precedence was defined as the last point of contact. This means that the system that has a current interaction with the customer is believed to have the most current data and as a consequence used before other systems that might have older records of the same customer.

In the end the complete elimination of decentralized writes to customer data was achieved. All data was initially written and later edited within the target application and all legacy systems only had the option between directly accessing the centralized customer data or keeping a read-only (cache) copy in their local database. A total of more than 8 million customer records was consolidated with a much higher data quality than any of the previous systems could have provided.

Phase 2 - Claim and File data consolidation

Similar to the first phase all claim data and (physical) file data were consolidated in a second step. These two data sets (connected, but with very different life cycles) are vital to the inner workings of an agency like fund.

First, claims are virtually always the trigger for business processes, as by its own self understanding, the organization never actively initiates a customer process. In other words, without a claim and the corresponding data, not one of the core accession processes can be triggered. An incomplete set of all claims was previously handled in file tracking and master data system, which was

retired with Release 5. However, data was only completed as far as necessary to satisfy the requirements of the statistics department. Furthermore only some of the system's processing claims reported back the status of a claim resulting in a significant portion of the claims never being resolved. Besides some of the claims affected more than one system resulting in conflicting writes and race conditions.

Second the (physical) file data is no less crucial to day to day operations. The fund has absolutely perfected the handling of large amounts of paper based files - including a sophisticated system based on conveyor belts for automatically transferring files between departments. With millions of customers and requirements for traceability over decades (resulting in "historization" features for legacy systems where a printout of a screenshot is added to the file) the amount of files and their size is immense. Additional complexity stems from past attempts to reduce the amount of paper by archiving old files on film. So, although previously only meta data was stored for these physical files, the criticality of the data for tracing and retrieving information about a customer and past business processes was significant. Furthermore the files served as a "token of work" moving from department to department and being handed down from manager to case handler to distribute the workload. For scrutiny the file (and with it the case) was handed to a senior member of staff before any final decree was sent out to the customer.

During the project both sets of data were consolidated, cleaned up and again a clean web service API was provided for all dependent systems to access the data. All systems can "pick up" a claim, consequently locking the claim for editing in the core system and thus avoiding duplicate pick ups and report back, either by returning the claim for further processing or with the necessary handling data to mark the claim as completed. The file meta data is used to synchronize the physical, paper based world with the electronic one by tracing the movements of the physical file and ensuring that the physical file (as long as one exists) is with the case worker. Besides it supports a large scale digitalization process enabling case workers to scan and properly archive whole or partial files while handling them. When the overall goal of a paperless process is achieved, paper based files and the accompanying meta data will gradually become obsolete.

Phase 3 - Business Process consolidation

The final phase 3 of the consolidation effort is ensuring a continuous, process based and electronic handling of a claim from the incoming mail (or in future possibly through online claims) to the official (positive or negative) decree.

Currently the process is of a purely organizational nature and only supported by specialized, isolated expert IT systems. As already mentioned the steps of the process are passed by handing the gradually filling file from department to department and case handler to case handler. The involved personnel is using the systems described in the section Enterprise Architecture as part of their work but without any systematic guidance. For example, if an error is made during an earlier step, the current case handler writes a (possibly handwritten) note and directs the physical file back to the previous handler (or clerk) for corrections. This process can easily take days, even weeks if the backlog of files is sufficiently large or the file is stuck at some point due to the absence of an employee. The proper execution of the processes is largely dependent on governance and individual discipline.

During the project the business processes were documented, modelled and therefore formalized. Tasks were mainly distinguished between tasks needing human attention and tasks suitable for automation. In addition, the rule set for assigning tasks to groups or individuals was formalized as well. Through this effort a high level of standardization and transparency was achieved. On the one hand the solution constitutes a disruptive change that significantly altered the daily routine of all case workers. On the other hand it aims at taking away repetitive tasks to leave more time for focusing on the areas that need their highly specialized expertise.

4.1.4 Alternative Outcomes

As mentioned in the introduction to this case, the major transparency change during the early phase of the project was a significant disruption to the whole organisation and especially the project. There are two major views on the topic.

On the one hand it can be seen as a destructive force, ripping through neat project plans and a well oiled machine of a big government project. Time lines had to be readjusted significantly, teams reorganized and priorities shifted significantly. Parts of the system that were not to be built for years now had to be completed in months. This certainly pushed the organisation and the projects to their boundaries. In fact in retrospect the project was on the brink of failure for an extended period of time, probably somewhere close to two and a half years.

On the other hand the event can be seen as wake-up call, a destructive event that, like a wildfire, was necessary to make room for the effort to even have a chance at succeeding. It forced the project to deliver. Not in small increments and cautious baby steps (as initially planned), leaving the hard parts for the late project stages, but in a way that required touching all functional areas and a way that required for it to work at scale. The ensuing phase was intensive for all participants. Errors were made. However, it also brought to light several fault lines that might have otherwise stayed hidden in the shadows of a large scale project, possibly for years to come.

The question that will be discussed in this section is what the scenario would have been without this disruptive event. The metaphor of forcing a quick and sharp turn of a large ocean tanker was frequently used within the context of this project. To think this further, the turn can in retrospect either be regarded as a lifesaving necessity that maybe damaged the overall performance of the vehicle, but kept it afloat or a significant overreaction to a problem that could have been solved in a much less drastic way. Out of the many possibilities, these two will be discussed in some more detail.

First the assumption that the disruption would simply not have happened, would lead to the project going on as planned. The tanker is holding course. The original plans called for a series of incremental releases. However, the hard part of digitizing the end to end flow would have been years away. In this scenario one likely outcome is a late and dramatic failure due to late integration. At this point, years would have been spent writing large amounts of code and developing highly complex solutions for side topics. One actual example of this might be the sophisticated and highly over-engineered subsystem for handling all kinds of events from deadlines to appointments including of course escalations. This system is developed completely generic to handle all possible use cases and future needs, resulting in a system that is very powerful in theory but not usable in practice. Over the course of the years a large project without delivery practice tends to engineer exactly these kinds of subsystems. At the point when the end to end processes are finally ready to be integrated and everything needs to come together, the most likely outcome are severe issues during integration. Subsystems do not perform as others assumed, the individual pieces do not fit together and there is overlap as well as gaps in functionality from an end to end perspective. All of this happened at a smaller scale due to the disruptive change. However, the big difference is, that it forced early integration and accordingly happened at a time, when corrections were still possible without throwing away years of work. So the outcome of the first scenario is that many of the problems that were exposed by the change would have otherwise persisted for a large part of the remaining project. The chances of late integration and late emergence of the fault lines leading to a negative outcome for the project (either cancellation or extreme cost overrun) are considered to be highly likely. In other words without the abrupt turn the ocean tanker would have been headed for disaster.

The second assumption would be that the disruption could have been handled without incurring (the observed amount of) long lasting damage. This scenario might easily go in a direction where the change is actually outsourced to a different organization or team - leading to the first scenario again, or even to a premature cancellation of the project due to the realization that it had become obsolete due to the change. The alternative would have been a longer period of actually planning and thinking about how to realize the change with the existing project resources. This in itself however is a very disruptive event. It might have easily split the project team completely and as a consequence might have resulted in a setup that would have been better equipped to deliver the change, but with no real chance of subsequently holding on to the goals of the original project. This can again be seen as a chance on one side, resulting in a learning experience and a possibility to realign. On the other side it can easily be seen as a disguised form of project failure where the overall goal of digitalization of the business processes is no longer in reach. In addition, the risk of stopping the tanker to think about where to go, in continuation of the metaphor, is that by the time it has stopped, reoriented itself and started again, too much time will have passed for it to still be able to even reach the intermediate goal set by the change.

To summarize, this could easily be made into a case for agile software development, but due to the traditional, top-down nature of the organization this would be beside the point. As the discussed alternative scenarios have shown the chances of project failure are high in almost any case. Accordingly the author is of the opinion that the disruptive change was a necessary catalyst to ensure the survival and eventual success of the project. It can be regarded as a good example of how to deliver early and in the process learn from mistakes. It had a profoundly transformative effect on project organization, stakeholder expectations as well as the (enterprise) architecture.

4.2 Case: A Very Specifically Grown System in an Important Niche Domain

This case study presents the chance to take a deep look into governmental IT and the very specific challenges associated with it. Without going too much into politics, this case can be described as an archaeological treasure hidden from the eyes of the public for decades, while still affecting almost every citizen of the country. Behind the frequently changing facade of the cabinet, its ministers and secretaries a cadre of solid performance has ensured consistent quality and services for a long time. However, the requirements of e-Government and web-enabling for millions of citizens on the one hand and continuously increasing budgetary restrictions have brought this well oiled machine almost to a halt. This case study is about enabling the necessary changes to bring a 20th century IT into the 21st century.

4.2.1 Status / Analysis

Transforming the IT of a whole ministry from one paradigm to another is a process that encompasses many years, if not decades. Two major problems arise that can affect the suitability as a case study. First the tedious political discussions and decision finding processes tend to prolong the project significantly, rendering time frames unsuitable for comparison. Second these long periods render certain past decisions obsolete at the time of execution. To mitigate these problems, the case study will focus on a limited period of time (about one year) that was necessary to devise and implement a strategy for the transformation of the legacy systems. Complementing this will be retrospective analysis of past decisions that proved to be unsustainable (without looking for someone to blame).

Organization

While the working title of this case study designates it as “ministry”, it actually encompasses multiple, loosely coupled but highly dependent organizations. The cabinet itself is responsible for the strategic orientation as well as providing the budget. A separate organization is responsible for business analysis and IT management. This includes all contact with end users as well as all coordination with the many other governmental organizations that have to interact with the business processes. The actual development or acquisition of software and hardware as well as its operation and maintenance is outsourced to a company owned by the government. In theory the main reason is to be able to perform IT operations more efficiently across government agencies and ministries. However, in practice the groups associated with individual agencies are still closely coupled and only little by little synergies are being realized. In addition, the ministry is by far the largest customer accounting for more than 70% of the total revenue. Over this construct a multitude of IT projects (about 700 concurrently active) are handled, from little maintenance and change projects to very large introductions or modernizations of complete IT landscapes.

Breaking it further down, the main organizational concept is application. This typically signifies a system (or system of systems) handling one business domain or legislative responsibility. This could be levying of a specific tax or fee or the grant and payment of a benefit or subsidy. Each of these applications has a responsible group of managers and business analysts in the government agency, reporting to the responsible secretary in the cabinet and delegating the actual realization of a project to the company where typically a designated group is handling all projects of an application. Peaks are typically bridged by hiring external support on both sides.

Enterprise Architecture Transformation

On an enterprise level the architectural strategy over many decades has been a monolithic host application used by virtually all groups. Especially on a hardware level and concerning centralized services (as backups and storage) this allowed for a high level of synergies and a homogeneous setup and operation. On a software architecture level this strategy resulted in a highly monolithic, interconnected and interdependent code base of more than 5 million LOC with a significant amount of duplicate code and identical or at least very similar features. A large majority of this code was written in PL/I, but a significant portion (almost 20%), especially some of the oldest parts were written in native assembler code. In addition, some applications were developed using a 4GL programming system which was used to generate COBOL code. Almost all user interfaces were accessed via standard 3270 terminals, later replaced by a terminal emulation product. [130]

The requirements of emerging e-Government features and best practices and a focus on the web started the slow erosion of the old architectural model. A high rate of batch processing and therefore limited amount of transactions available online in combination with a lack of modern integration technologies on the mainframe were key factors to bring the organization to the point where it is standing now. Additional economic factors were the continuing increase in transaction and data volumes which were causing disproportional rises in the cost to run the legacy systems. Most web-facing applications were therefore already developed in the Java ecosystem, slowly establishing a second alternative line of technology to the legacy host, however without a clear strategy for a full scale transition. This state is depicted in Figure 4.6.

As a result a large infrastructure, architecture and application development program was launched to establish a new Java based platform suitable to one day run all the applications now running on the mainframe. This included everything from acquisition of the necessary hardware and software infrastructure to the development of commonly needed base components and features. The latter was a direct result of (correctly) perceived feature and code duplicate in the legacy system. How-

ever, while the transition of host based applications was one of the main goals, the priorities set by the program lacked a clear focus on achieving this in a timely manner. Instead the individual projects were cut along base component and master data lines as most applications creating business value were depending on some of the features provided by them, leaving little resources to develop actual end user facing applications. From an outside point of view the relation of end user visible benefits to actual cost was prohibitive.

Again, as a direct result of cost estimates for redeveloping all remaining legacy applications with the previously mentioned approach in the new target architecture combined with increasing pressure to quickly reduce the cost of running the mainframe a new strategy was devised. In a volteface strategic reorientation the decision was made to seek a silver bullet solution by finding a vendor that will translate the existing host-code to Java in a fully automated fashion. Instead of more than a decade of projected development time for the previous approach, this was expected to reduce the time to the final shutdown for the mainframe to less than three years. However, during the bidding process serious doubts were raised concerning the feasibility and associated risks of the chosen approach. In combination with disappointing bidder turnout this led to the decision to seek yet another opinion which signifies the start of the focus period of this case study.

4.2.2 Solution

To start with, the focus was laid on the results to be anticipated from the automated transformation tender. As mentioned in the previous section, the main intention of the tender was a quick and painless solution to the prospect of having to run a cost intensive mainframe over many years to come. The problem was intensified by a lack of development performance in the target architecture. Some of the key issues with this approach were identified in the early stages of the case study, the strategic analysis.

Strategic Analysis

The output quality of the transformed code base could not be verified. While most vendors do promise readable and maintainable code, there is no formal standard that could be taken as a reference. In addition, the result of the transformation greatly depends on the quality of the input, the legacy code. Previous analysis by an independent third party revealed that the legacy systems code quality was significantly below industry standards, even for systems of this age. Especially a high volume of redundancies and code duplication would intensify this issue. Comparison with the code quality in similar efforts, amongst these the automatically transformed code described in the insurance case study (see section 4.1), further underpinned this impression.

The resulting system would have to mimic the behaviour of the legacy system in every single detail. While this is certainly doable, solutions have been documented and published [108] and a multitude of companies have it on offer, the implications in detail are massive. For correct operation an intermediate library has to be used to simulate behaviour of basic programming language constructs like loops or data types going as far as subtle unintended bugs with side effects that a programmer in the legacy system might have intentionally or unintentionally used. This essentially means that the data types of the target language along with their well known behaviour to programmers familiar with that language may not be used. The resulting code is only syntactically valid code in the target language whereas the simulated semantics strongly resemble the character of the legacy language. Furthermore even a minor glitch in that layer will result in subtle and hence hard to pinpoint side effects like calculations going slightly off in border cases. This additional library has to be maintained either by the customer or the solution provider, piling on the already existing high level of complexity.

Generally, language construct translation is only a small subset of the full problem. The overall picture is complicated by the use of sophisticated framework for transaction processing, printing or batch control and execution. While the common host transaction monitors have been ported to other operating systems, print and batch do not have any drop in replacements. For printing this means replacing all calls to print routines with a modern document creation system, which comes down to re-engineering all existing templates and formats. In this case this would represent a highly infeasible scenario just considering the amount of document types. For batch processing this means rewriting all the batch control code in a modern batch environment. However, this still does not address the inherently problematic nature of a batch focused architecture. While the transformed code and batch control might be run on the target platform it will not be any closer to the kind of synchronous online processing that is expected of modern day architectures. Exposing these features for example as web services to other systems is therefore still not easily possible.

As mentioned in the previous section, significant effort has already gone into developing the target architecture. This includes a series of core and base components intended to maximize reuse and consistency across applications. With a fully automated transformation of the legacy system, none of these components, including a sophisticated master data management system would be immediately reused. Instead follow up projects would have to be planned to adapt the transformed source code in effect requiring a partial rewrite of these applications. This would significantly delay the adoption of the target architecture or even make it unlikely, especially for the more complex legacy applications.

Besides the obvious benefits of being able to move away from a mainframe based system, a popular argument is frequently the lack of experts available for the legacy system. While this is certainly true, an automated transformation might not solve this problem as expected. While the availability of experts for modern day programming languages is certainly higher and as a result might be positively impacting the cost of hiring, it is unlikely that an automatically translated system can be maintained by entry level college graduates. As mentioned previously, the resulting code most likely has little to do with the typical modern day system. On the contrary, the simulated behaviour of the legacy platform will require intricate knowledge of exactly this platform in addition to solid knowhow in the target platform. In the end a transformation might consequently have the exact opposite of the intended effect on expert availability.

The last one of the major factors had been left out of the tender: quality assurance. Even with automated transformation intricate tests are required to ensure the transformed solution works in exactly the same way as the legacy system. Unlike in regular software development where quality assurance nowadays is mostly an integrated parallel activity this basically means that a full retest of all features has to be done after the delivery of the transformed system. Needless to say these tests had to be fully automated to be repeatable after every increment of the delivered software artifacts. The only feasible way of handling this workload is by running automated capture - replay tests. This means that an initial state (in practice a full copy of the production database), a series of transactions (typically at least a day) and the final state (again a full copy of the production database) are captured in the live legacy production system. The transactions are then transformed to inputs to the target system and executed against the same baseline data. The resulting database is then compared to the copy of the final state of the production system. Any deviation has to be analysed and explained manually to verify if it is a bug in the new system or maybe a side effect of the testing process itself, for example the fact that the transaction is executed at a different time or day, or a valid deviation, caused for example by non-deterministic order of execution.

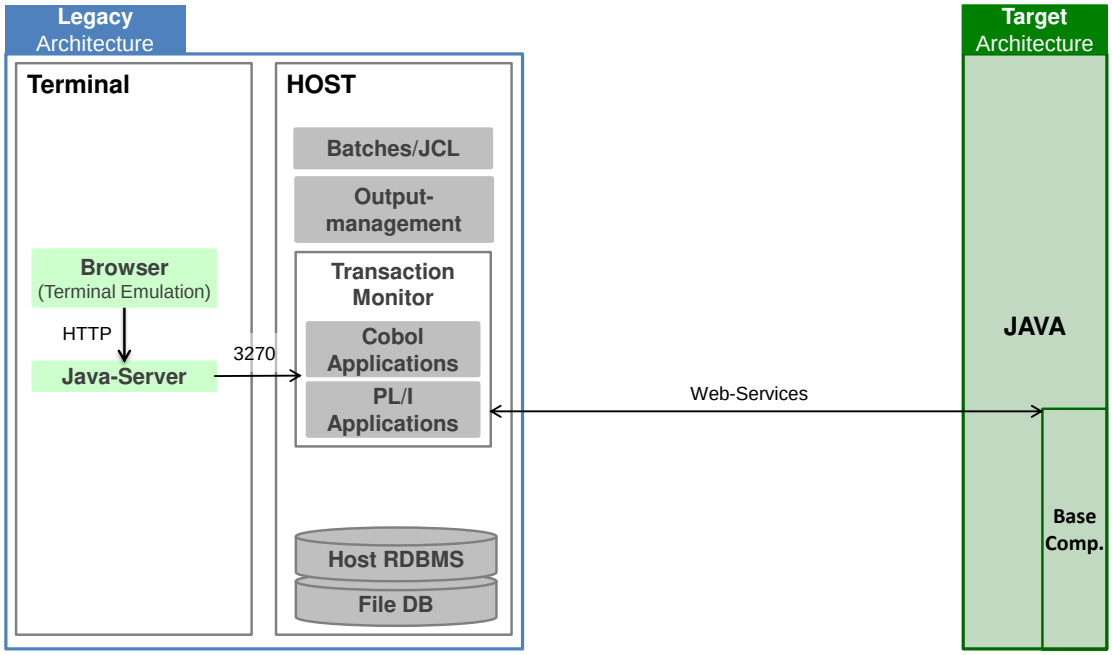

Figure 4.6: The legacy architecture as encountered at the beginning of the case study

Migration Architecture

Considering all the issues and uncertainties mentioned in the previous section the following solution was devised to achieve three major goals:

1. In the short term a replacement platform for the mainframe applications had to be found to reduce the operational cost.
2. In the medium term the existing investment in the target architecture should yield returns in the form of maximum possible reuse.
3. In the long term both the maintainability and adaptability of the existing applications should be improved.

The idea behind the solution architecture (as depicted in Figure 4.7), which in itself is merely a transitional step towards the final target architecture, is based on the fact that automatically translated legacy code is by nature at best very close to the legacy. Therefore translating the legacy code does not bring sufficient benefits to justify the cost and risks involved with the automated transition. Instead the majority of the existing legacy code is merely recompiled on a different, Unix based platform. This has the major advantage of not having to deal with syntactic and semantic differences between programming environments. In addition, the transaction monitor does not have to be replaced or reengineered. Instead a platform is selected where the transaction monitor is also provided, thus mitigating one of the major transitional risks. Furthermore the user interface does not have to undergo any changes as the terminal protocol (3270) is supported

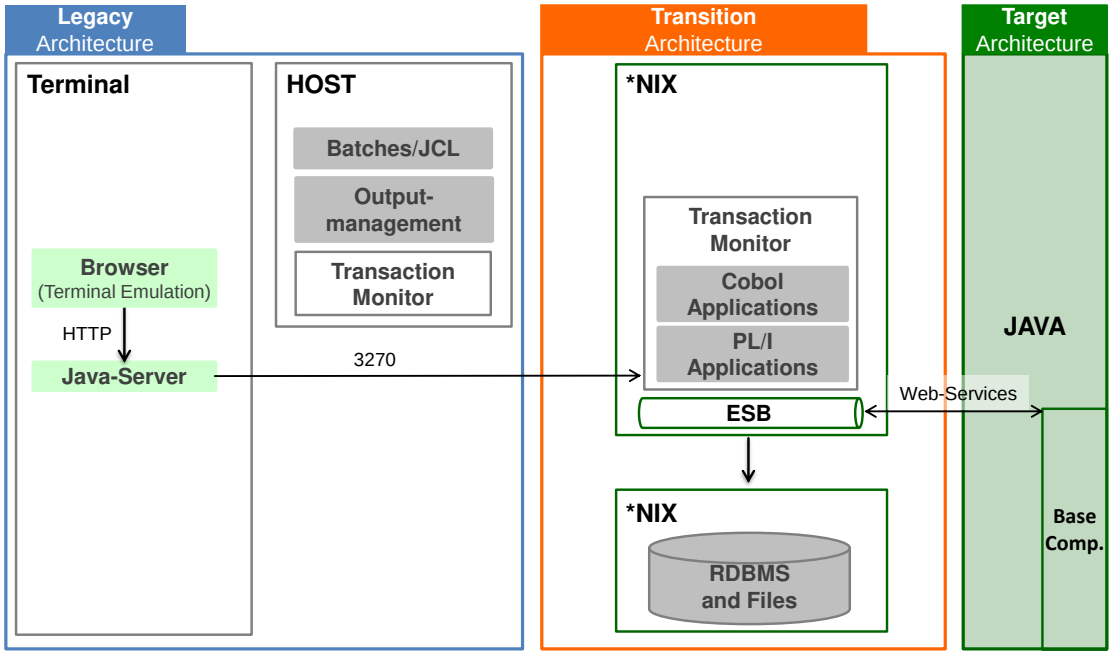

Figure 4.7: The solution architecture as proposed during the case study

by the transaction monitor eliminating the need for user training. All of these aspects greatly reduce the amount of quality assurance necessary to provide sufficient confidence in the moved application. From a hardware point of view, the necessary servers are capable of running the transitional as well as the target environment and can be thus reused even after the transition architecture becomes obsolete. Still there are several hidden obstacles that need to be overcome. Most prominently, the code page used for character encoding on the legacy platform (EBCDIC, Extended Binary Coded Decimal Interchange Code) and as part of the terminal protocol is not available on the target platform. This means that a suitable encoding has to be selected, the data has to be transformed during migration and code used for character comparison with hexadecimal values and sorting needs to be adapted. Another obstacle is the difference in caller stack design between legacy and target architecture which can lead to previously hidden occurrences of illegal memory access due to array index out of bounds errors.

However, as mentioned in the previous sections, there are three problematic areas that cannot be solved with this approach: Assembler code, batch control and output management (printing). All of them can be solved by the second strategic idea behind the proposed approach. All of the code dependent on these three problem areas is moved to a new mainframe which is shared by multiple government agencies. The main benefit of this move is the fact that little extra cost is generated, as long as the ministry's share of the host resources is sufficiently small. Early estimates put the cap at a maximum of no more than 20% of the total processing power consumption. The mainframe has to be acquired in any case, so this will not cause additional costs. The operational costs can furthermore be shared among all agencies, consequently benefiting all participants.

The third pillar is an evolutionary improvement of the target architecture to address the deficits mentioned earlier. This will affect both the development process which will get a stronger focus

towards reengineering (instead of regular green field development) and the solution architecture itself. In addition, the earlier integration of end users both during reverse engineering of existing applications and the development of the new solutions will provide earlier feedback to developers and create better visibility to the prospective users.

The design of the respective applications will reflect and leverage the transition architecture. One of the unique opportunities in this setup is the possibility to incorporate legacy transactions in the new system provided out of the box by the transaction management framework of the transition platform. This approach, termed "mosaic coding" is especially useful in situations where old calculations have to be retained for a number of years (in practice this can mean up to 10 years or even more), but modifications are highly unlikely. By not migrating these calculations the risk of regressions is greatly reduced as is the amount of code that has to be developed on the target platform. As these parts of the code will age out gracefully over the retention period the amount of legacy code will decrease gradually.

Transition Strategy

The migration architecture can be achieved through a series of parallel projects that will be consolidated in a program.

The first major stream is (incorrectly, as by our definition in Section 2.1.6 it would more fittingly be replatforming) called *rehosting* and involves the move to a different hardware platform. This is the stream with the highest risk and uncertainties. To mitigate these a complete vertical breakthrough is done as a proof of concept. In a first stage this is done with a few isolated transactions. In a second stage a whole application is moved from the host to the new platform. This includes migrating the data to the new database and configuring the transaction container to behave in a similar fashion to the legacy transaction manager. Of course the major task is recompiling and then evaluating the legacy code on the new platform. At the same time all parts that cannot be moved to the new platform are identified.

The second stream is called *shared host* and focuses on preparing the move from the existing mainframe to the shared host and would technically represent the actual rehosting (as defined in Section 2.1.6) part of the overall migration effort. Mainly organizational in nature (selecting the right partner and setting up a suitable contract) the technical side includes planning and testing the move of components and ensuring the correct network and infrastructure setup. One of the key concerns with this part of the solution is a performance deterioration due to additional network latency. In addition, many operational procedures have to be adapted to the new setting. Furthermore the new points of integration, especially the now externally hosted database system, will require some adaptation.

The third stream is the development of a viable *reengineering program* to achieve the long term goal of having a modern, maintainable and adaptable application landscape. In a first stage the necessary architectural and development process changes are identified. The existing applications and their dependencies are analysed to identify a viable sequence to migrate the existing applications. Apart from technological dependencies, major factors for prioritization include legal and contractual obligations, the early removable of exotic niche products (e.g. 4GL environments), external financing opportunities through cooperation with other government agencies or international institutions as well as a high amount of pending changes. This stream then selects a suitable candidate project to be executed as a flagship reengineering effort intended to server as a role model for all subsequent migration projects. As with any major process improvement a need for ongoing tailoring is to be expected. The flagship project will have both the time and expertise ready to adopt the initially devised process to the organizational challenges that arise during the

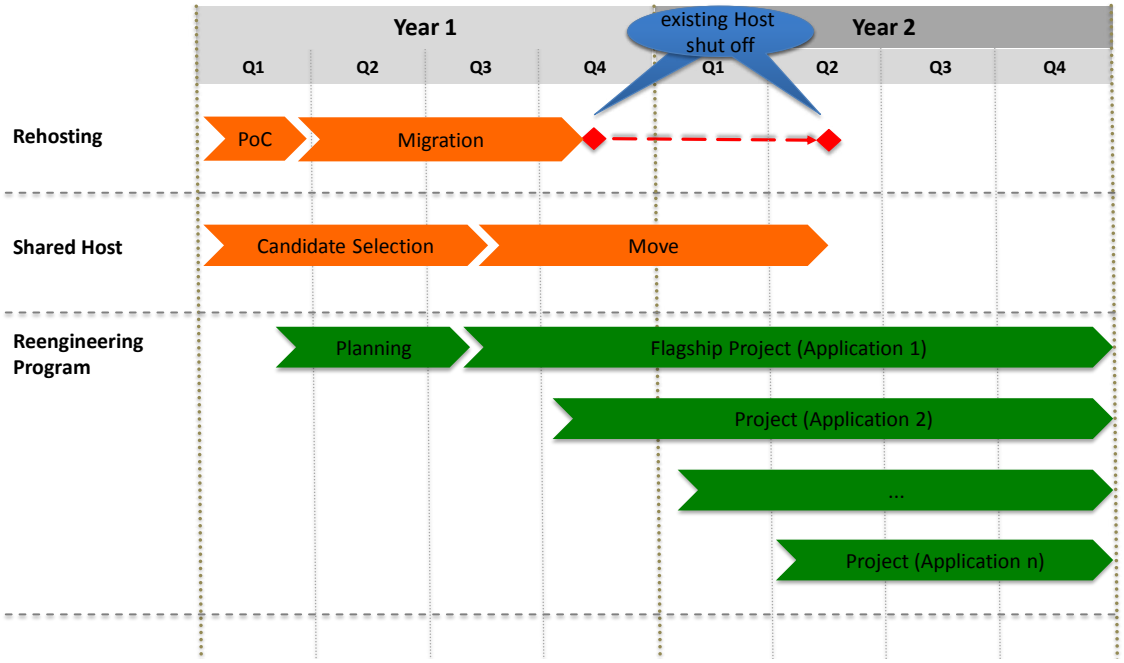

Figure 4.8: The timeline of the transition strategy for the first two years

execution of the project. All subsequent projects can benefit from the experience and lessons learned in this flagship endeavour accordingly. Even more importantly this project will serve as proof that significant portions of the mainframe can be redeveloped within reasonable budgetary and temporal constraints. The individual reengineering projects would then use a common incremental and iterative approach with specific adoptions as necessary. Cross functional teams would consist of business analysts, legacy system engineers and developers for the target platform. It is jointly managed by a team lead, handling organizational issues and a product lead who is responsible for the selection, prioritization and documentation of all features. Furthermore a strong end user involvement throughout all stages will ease the transition and ensure the suitability of the end product.

Figure 4.8 depicts the planned timeline of the initial two years of the transition strategy. While the first two streams are rather short lived with an expected duration of about 12 months, the third stream will evolve into a program to gradually redevelop all applications until neither a shard host nor the legacy code running on the new hardware platform are needed. The exact duration of this effort is highly dependent on the availability of sufficient resources as well as the political climate. Budgetary constraints will limit the room for movement while new legislature will create the need (and for that reason provide the funds) to migrate the applications one after another. From a technological standpoint, with a high level of parallelization the reengineering effort could be completed in a matter of a few years.

4.2.3 Experience

4.2.4 Alternative Outcomes

In this section two hypothetical scenarios and their outcome will be depicted. This will enable a clear view as to why it was necessary and the right decision to change course with the ongoing tender at a very late stage in the process.

Automated Code Migration

Assume the decision to reevaluate the strategy to let an external company provide a translated version of the legacy code would not have been overthrown. As the tender was already in the final stages, a winner would have been selected within a relatively short time frame (months). Even without the not unlikely appeal considering the total volume of the tender, the first phase, a proof of concept (PoC), would have taken at least half a year. This PoC would have then yielded one of three possible results (in the order of likelihood):

- The decision to cancel the project. An option that the ministry could draw in case it is not satisfied with the result and is not convinced that the vendor can actually perform the migration (in time or at all).
- The decision not to migrate towards the target platform, but instead just recompile the legacy code on a different platform.
- The decision to proceed with the migration as planned.

The first two options can be considered worst case scenarios. Cancelling the project would result in significant cost especially in the form of time loss towards the strategic goal of removing the mainframe. Most likely no usable output would have remained except a painful lesson learned. Migrating by recompiling is not that different from the strategy proposed in this case study. However, instead of doing it oneself and using existing resources (both human and hardware) everything is done by a single vendor, bringing up both time to realization and cost. In addition, the strategic element of moving some parts of the legacy applications to a different mainframe would not have been feasible as the contract would have forced the vendor to provide all code in executable form on the target platform. This would implicate a rewrite of the complete assembler code (in PL/I or COBOL) as well as costly intermediate solutions for the batch execution and output management problem areas.

Assuming a decision to proceed, the vendor would have started to generate source code in the target language. This would have resulted in huge amounts of code handed over (in several increments or even as a single delivery) to the ministry, possibly years after the awarding of the contract. The code as required by said contract would have to be runnable so a running version would be installed on the ministry's servers by the vendor. The most prominent problem at this point is verification of both code and functionality.

To start with (functional) completeness: To prove that every needed functionality is contained in the delivered artifacts, a complete analysis of existing functionality has to be conducted while the vendor is performing the migration. This must be in a great level of detail in order to ensure that neither special cases nor seldom used functionality is omitted. Second functional equivalence will have to be verified, translating into thousands of test cases. All of this for a legacy system that has scarce to no documentation and significant areas with no or just rudimentary formally documented test coverage. As a third pillar, non-functional requirements would also have to be examined.

Analysing and assessing the maintainability and change-readiness of the delivered code will not be any easier. Even cautious estimates put the expected volume of code at about 2.5 to 3 times the LOC of the original source code. This would translate to approximately 12 to 13 mLOC. Manual code reviews over any significant part thereof seems unrealistic, albeit spot testing would surely take place. This leaves static code analysis with established tools. However, as these tools easily yield thousands of indications for much smaller projects, filtering of the results into relevant and irrelevant alone will be a mammoth task further complicated by the fact that these tools are tailored towards manually written code and not machine translated code originally written in a completely different programming language.

Any of the previous verifications could well end the project in a long lasting legal battle over fine print in the contract or questions as to how maintainable code will look. All of this with very uncertain outcome except the fact that the delivery of a working solution will most likely be delayed for years.

But assuming furthermore that the ministry is actually able to verify the delivered artifacts and accepts them, two more challenges lie ahead. First, all of the code has to be brought into production. A huge transition for both the technical and business employees of the organization. After all, the last challenge is on how to proceed. Not very clean legacy code has been machine translated (and most likely machine verified) to a target language and platform and is now running, doing exactly the same as the legacy system in the most optimal of all cases. Several years have passed with little opportunity to apply changes to the system translating to a significant backlog. The ministry's developers now have to become acquainted and fundamentally understand more than 12 mLOC plus any additional frameworks necessary to simulate the behaviour of the legacy system to be able to even perform simple changes. A daunting task from an engineer's perspective.

To conclude this outlook: Even in the very optimistic scenario depicted in this section the benefits of having code running on the target platform comes at significant cost concerning both time and money. However, chances are that significant rework to the code base up to a level which comes close to a complete rewrite of some applications will be necessary in order to be able to incorporate the kind of fundamental changes which are in the pipeline to enable 21st century e-Government features.

Successful Redevelopment

This second scenario is a thought experiment about how things could have been done right in the first place. In other words, could the tendering for an automated migration and the current transitional tactical strategy have been avoided all together? As mentioned in section 4.2.1 the initial intent already was to do a complete architectural and redevelopment program to rid the organization of the legacy mainframe applications. In fact a significant effort has already been made in developing the target architecture and the first applications are already out in the field. However, user feedback as well as development performance and the perceived cost to benefit ratio all have been negative, leading to the current course of events.

The following three improvements could have avoided this situation, revolving around the planning and execution of the redevelopment project to be focused on the fact that it is a legacy system migration effort. The existence of a legacy system has to be reflected in every part of the project.

The breakdown into manageable projects and releases has to respect the decomposability of the legacy system [32]. This will most likely not be along the established lines of teams and applications, but will rather be reflected by external political factors and internal technical aspects, especially the dependency tree and data flows. User groups and their work habits and patterns play an equally important role. Avoiding frequent context switches between the legacy and the new

system is an important criterion to achieve adequate user acceptance. The established business architecture, especially the partitioning of functionality into applications needs a critical scrutiny while at the same time acknowledging the value of decade long experience of orchestrated business processes and established procedures.

The business analysis needs to ensure that existing functionality is exactly retained unless an explicit change is required and documented. The focus on the legacy system highlights the importance to view specialists in the legacy technologies and functionality as one of the key assets of any migration project. The potential of their knowledge can only be realized by fully immersing them in the project setup. From a functional perspective this will lead to a cautious, evolutionary approach to the provided functionality. Any non-essential improvements to usability and similar wishes from the end users should be carefully analysed as to the actual impact. Only low-hanging fruits should be implemented during the reengineering phase if their realization is expected to have a significant positive effect on user acceptance, everything else can be saved for a later, maintenance or feature release after the critical phase of migration.

The technical architecture and especially the integration points to the legacy systems have to concede to the fact that a transitional phase will be necessary to reach the target architecture. Temporary constructs are required to allow an incremental migration and should be treated as first level architectural and design artifacts. This means documenting their use, providing rules, patterns and best practices to follow during design and development as well as maintaining their temporal status by planning their timely removal. The transitional or migration architecture will play a central role in the success or demise of the project(s) warranting a dedicated architect resource to be tasked solely with maintaining a consistent strategy and execution. By providing a counter part to the architect(s) of the final target platform this role will ensure that the necessary compromises can be realized with adequate official backing as to not thwart the migration effort over a clean target architecture.

These strategic and organizational adjustments will ensure a continuing focus on the legacy system and the ultimate goal of replacing it with an appropriate solution that respects its heritage without being stuck in the past. Many more factors could further improve the performance of the project. However, due to the fact that they are not immediately related to the legacy system migration nature their treatment would exceed the scope of this case study.

4.3 Case: A Very Old Core

This case study describes a classic core IT system migration. The system in question is the Airport Operational Database (AODB) of a 20-million passenger per year airport in central Europe as part of its expansion to a target capacity of 30 million passengers. It has been developed and maintained in-house alongside several other IT-systems (cargo management or load balancing as two examples) for just over 40 years. Triggered by the upcoming opening of a new terminal building and the related changes the airport decided to modernize its IT landscape including a renovation of its core AODB.

4.3.1 Status / Analysis

An Airport Operational Database (AODB) is the very core of the IT landscape at any airport. Its main purpose is the near real time representation of the current state of all (currently relevant) flights to and from the airport. This includes planned, expected, approaching, landed and arrived flights for the incoming part as well as planned, taxiing, departing and departed flights for the outgoing life cycle. In airport operation terms this is called the tactical traffic image. In addition,

to the current state a multitude of other data is tracked for each flight, from basic information as to which airline and aircraft is operating the flight, delay information, runway and position information, basic load data or deicing.

In addition to the traditional AODB use cases the AODB in this case study also provided support for flight and resource planning. Resources that could be planned and administrated through the system included positions and gates as well as baggage belts. However, all of these features will most likely be gradually replaced by specialized IT systems.

The sources of information for an AODB are highly diverse. The main sources of outside information are other airports, air traffic control, the airlines operating the flights and ground handling. Most of this communication space is highly regulated and standardized by International Air Transport Association (IATA), a trade organization for airlines representing over 80% of the total, worldwide air traffic. These messages are then transported via a specialized and dedicated network operated by Société Internationale de Télécommunication Aéronautique (SITA), a non profit organization based in Belgium. Their network transmits and delivers over 800 million messages a year.

Additional information is generated by other IT systems at the airport. This can include Resource Management Systems (RMS), Cargo Systems, Baggage Handling Systems, Ground Radar, Docking Guidance Systems or Dispatching and Ground Handling Systems to name a few.

All the information that comes together in an AODB is used for a multitude of purposes that can be broken into three parts: First, the information is used to validate the consistency of any incoming information (e.g. a flight can only park at a position if it has previously landed). This part is absolutely crucial to ensure that the following two parts deliver valuable and reliable information. Second, the information stored in the AODB is provided to virtually all people working in and around an airport as well as to passengers and the general public. Third and most importantly the AODB provides the information to all other IT systems necessary to operate the airport. In this case study "there was a total of 26 systems that depend on the core AODB and, the other way around, there were 10 dependencies on other systems by the new AODB." [19]. For a high level overview of an AODB architecture see figure 4.9 clearly highlighting the central integration nature of such a system.

The task for the team of migration specialists was to renovate the AODB system within the given time frame with less than 20 months available to the planned opening of the new terminal. Already significantly delayed any coinciding delay in the AODB migration would have brought the blame on this effort - justifiable or not.

The core legacy COBOL GCOS8 mainframe system "consists of 114 transactional processing routines (TPR) and 819 library routines together with about 250 KLOC COBOL code. Further 109 IDS-II database configurations with 3500 LOC. The average load is 124 transactions per second and there are about 1000 concurrent users connected with a VT100 terminal." [19]. The replacement of this legacy system had already been tried unsuccessfully three times, including one attempt at transforming the code automatically [111].

However, the most critical constraint for the effort was the minimal possible downtime. As the airport as any major airport was operating de facto 24 hours 7 days a week the system had to be constantly available and operational. Any downtime exceeding more than one hour would automatically lead to severe restrictions in the operations of the airport. For a start one of the two runways would be closed to limit the amount of concurrent events and throughput. After four hours the whole airport would be shut down until the system is back up and running. During the busiest times of the day the cost of an airport shutting down can easily reach millions of Euros. The only exception to this pattern is a nighttime gap of about two to three hours with a low volume

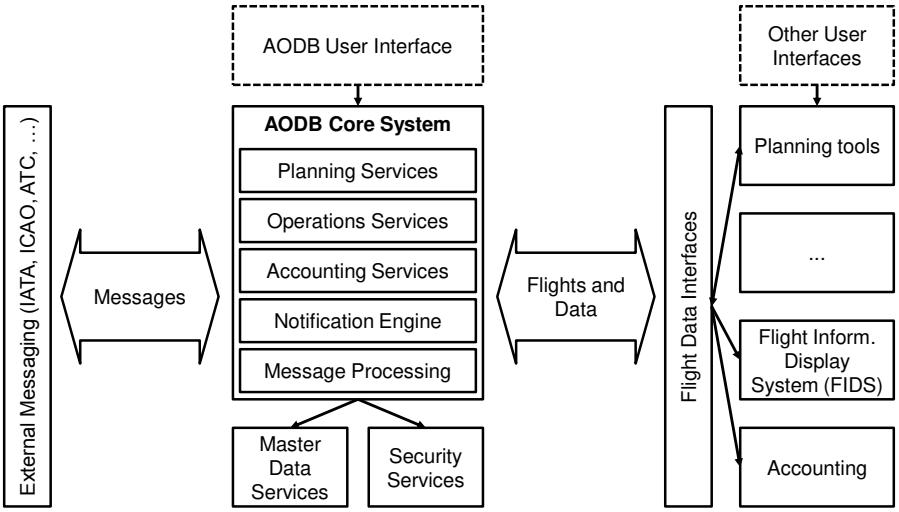

Figure 4.9: A generic and simplified AODB architecture similar to the one in this case study [19].

of mainly cargo traffic that can, if planned accordingly, be handled manually without the use of the AODB.

The required solution accordingly had to take as careful an approach as possible disqualifying any solutions based on a big bang (cold turkey [32]) strategy right from the beginning. This also included any considerations towards COTS products.

4.3.2 Solution

The following section will illustrate the solution from two main views. The first is the method of incremental migration and the second takes a more technical perspective focusing on the parts necessary for parallel operation.

Migration Strategy

The solution outlined during the planning phase of the contract proposed an incremental migration with parallel operations of the legacy and the new system for an extended period of time. The main focus was on eliminating the risk of unplanned downtimes leading to the key strategic element of allowing de facto instantaneous fall back to the legacy system at any time. This possibility acted as an insurance policy providing the team with the necessary space to build and deploy the new system. However, as with any insurance policy the price was according to the mitigated risk. In other words the necessary overhead to provide this level of flexibility was substantial. "To achieve the required degree of functional and technical compatibility a strict one-to-one strategy was implemented. This was an indispensable cornerstone for this project." [19]. The implications of this approach had to be considered from the detailed planning phases of the individual releases to the verification of the developed features. The extra complexity for testing alone was enormous. Every test case required compatible state in both the test stage and the corresponding legacy system and had to be validated in both systems as well requiring intimate logic of both worlds.

From a planning perspective, the "key problem is how to decompose the system into reasonable parts that may be migrated one after the other. An ideal process would be a fine-grained incremental strategy, but in practice the atomic decomposable modules are rather large (little big bangs)." [19]. While the initial plan did propose a series of releases and had the features mapped to each of them, only during the detailed planning stages and the execution of the prototype features the full scale of the interdependencies was revealed. This required a fundamental rethinking of the migration strategy and resulted in the subsequent series of steps.

Selecting a prototype feature was a key challenge in itself. It had to be of limited scope to ensure timely success or failure. At the same time it had to be complex enough to be both functionally and technically representative. "In this case we chose the block-off operation of an outbound aircraft for several reasons: It is a central step in the business process of handling outbound flights with many calculations such as the rotational delay and sending of external SITA messages such as the movement departure message. The block-off also updates the position and gate occupations data, which is a primary data structure for an airport and usually relevant for accounting. The internal notifications system was also strongly involved with sending and withdrawing a set of system messages." [19]. The (expected) failure in implementing this feature as a (potentially) shippable artifact left the team with a lot of valuable experience (analysed in some more detail in section 5.2.3) for the later planning and execution of the migration. Vital for the remaining project however was that the existing, passive flight core was not suitable to perform the tasks necessary to become an active flight core.

The step-wise migration strategy was based on the decoupling of input and output communication. This meant that the input could be (pre)processed by the new system while the legacy system was still responsible for the main calculations as well as the output. The following three steps were necessary to incrementally move from the legacy system as the leading source of information to the new system with consistent parallel operation.

Step 1: Flight Data Interfaces

As described in section 4.3.1 providing data to the surrounding IT systems is the primary concern of an AODB. The first step therefore consisted of creating an intermediate component responsible for preprocessing the data and distributing it to all other systems. The legacy AODB had a single, text based interface providing the full flight record on any change. This was necessary at the time as some of the components like the legacy Flight Information Display System (FIDS) did not hold any state (or business logic) themselves and as a result needed extensive preprocessing of the data down to the line number the specific flight has to occupy on the physical display. Following established enterprise integration patterns (EIP) [66] the interface was deconstructed into several building blocks.

The whole available flight data was provided via query-able services. Any system could consequently retrieve the current state at any time selecting exactly the flights it needed. For example the cargo system could retrieve only the relevant flights of a day marked with a flag to indicate the flight will be carrying cargo. Changes to the data were provided as event messages only carrying the information needed to classify and handle the change. Unlike the old interface it carried both the old and the new value of the changed field or fields. This allowed extensive filtering and routing of change information to the affected consumers and thus provided enough benefits to give sufficient incentive for the consumer systems to quickly perform the switch.

The resulting setup is shown in figure 4.10. Any output on the legacy systems flight interface is processed and transformed into the new database schema by the passive flight data component. It is then provided to any dependent system via the before described services and update mechanisms. Any communication back towards the legacy system is happening directly or mediated as before.

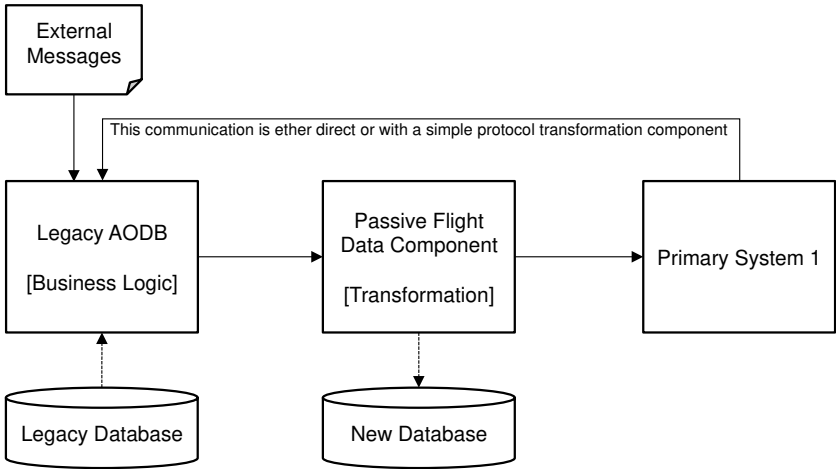

Figure 4.10: Step 1 of the incremental migration strategy. The main element is the Flight Data Component reading flight data updates from the legacy interface and transforming them to a new model before forwarding it to the primary systems [19].

Step 2: Service Interfaces

In Step 2 a series of facades are built to decouple any kind of input from the legacy system. Every facade represents a later business service for the AODB. Any input received is first validated using the same business rules as the legacy system and then transformed to calls on the legacy system. A functional one to one mapping is absolutely crucial for this step as any mismatch between the legacy and the new system is rapidly revealed as a problem where either the available functionality is reduced (too strict validation) or a call to the legacy system fails (not strict enough validation). To perform the validations all flight data is needed in the new database, therefore this step has Step 1 as a strict precondition.

As a consequence the initial service interface design for the new AODB closely resembles the individual transactions of the legacy system. However, both data formats and service cuts can be consolidated and optimized for the target language. For example, instead of having an eight character string to represent a date plus a four character time a date and time object is used to represent the data more accurately. Similar, two transactions that do the same operation, once in bulk and once for a single flight can easily be consolidated in one service.

After having completed Step 1 and Step 2 a primary system can interact with the new AODB as if the legacy system does not even exist. Any interaction with it is wrapped [24], as depicted in figure 4.11. However, to calculate the tactical traffic image and execute the needed business logic the legacy system is still needed.

Step 3: Master AODB

In the last step the two components created in the previous steps are consolidated and enriched with the missing business logic to form a fully functional set of business services. This means that the previously validated input is processed to actively write the flight data in the new database. Only afterwards is the operation synchronized to the legacy system. This happens asynchronously

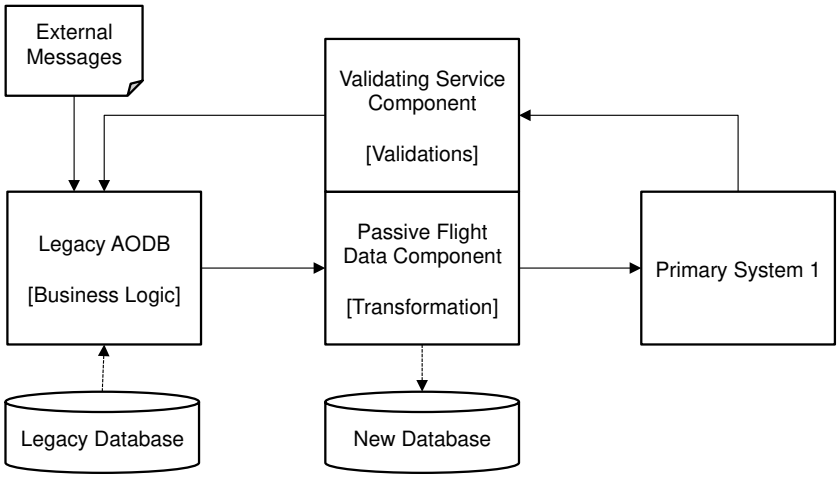

Figure 4.11: Step 2 of the incremental migration strategy. The main element is the Service Component that provides service interfaces with all relevant validations, but not the business logic. [19]

after the commit of the transaction in the new system and as a result "this makes the new core system the master node in the parallel operations." [19].

In addition, as depicted in figure 4.12, external messages are now processed by both systems in parallel. Not depicted in the figure, external messages can now also be produced and sent by the new AODB. However, to minimize the impact, this step was deferred to a later date to provide the ability to do a production verification by comparing the messages generated by the old as well as the new system.

At this point the ability to turn off the legacy system completely depends largely on downstream functionality and features. In this case the last feature set consisted mainly of accounting (data extraction to SAP) and statistics (data extraction to a data warehouse) functionality. In addition, more time can be given to dependent systems for making the switch to the new core AODB.

Enterprise Architecture

The IT landscape and infrastructure of an airport has to fulfil many standard enterprise tasks. However, its core competencies can be split into four areas as depicted in figure 4.13: Planning, Operations, Passengers (and Information) and Accounting. As an airport grows so does its IT develop. A good and flexible AODB however, is always at the centre. Ideally the AODB can grow with the airport and its expanding requirements. This defines two core architectural design aspects of an AODB. First, the AODB has to handle both imprecise, possibly long term planning data and mission critical, near real-time information and represent each accordingly. Second it must be able to handle all the features to achieve the core data itself or receive information from one or more third party systems.

The four quadrants, from top left to bottom right also represent the flow and actuality of information and the long term life cycle of a single flight.

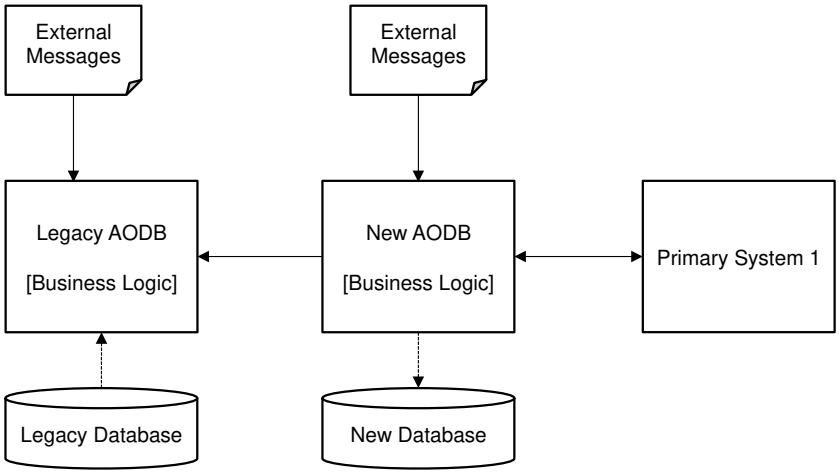

Figure 4.12: Step 3 of the incremental migration strategy. The two components of Step 2 are now merged together and provide a fully functional AODB. The legacy system is updated asynchronously to keep in sync [19].

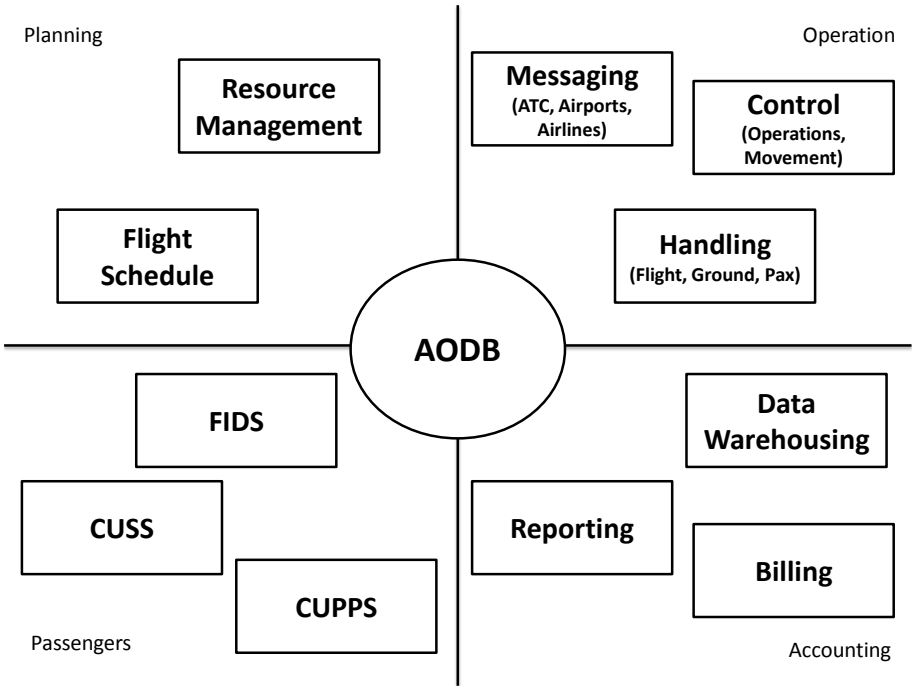

Figure 4.13: A high level airport enterprise architecture.

In the **Planning** phase the flight is typically announced by the airline. This is called the seasonal flight plan - frequently modified in detail, but in bulk done semi-annually. Based on this flight plan information resources (like check-in desks, positions, gates, baggage carousels or ground/pax handling personnel) are planned accordingly. A change in the flight plan of course renders most of the other planning obsolete, which is the main reason why the remaining planning is done only some days in advance. Once the day of the flight arrives, the **Operation** phase begins. Messages are sent between ATC, airports and airlines. Operations and movement control make sure the aircraft of the flight actually gets to the intended position in time and the different handling agents make sure it is dispatched properly. From the **Passenger's** perspective the flight information display system (FIDS) guides them through check-in, terminal, gate through the airport. IATA standardized systems like common use self-service (CUSS) terminals and common use passenger processing systems (CUPPS) aid in processing the passengers. Once the flight and its passengers and cargo are successfully processed the **Accounting** phase begins. Airports make their money (besides duty free shops and restaurants) by charging airlines for passengers and baggage handled, touch downs or departures, position and gate occupations and many additional services. All this fine granular data has to be aggregated for accounting and billing. In addition, the data has to be constantly analysed to find potential optimizations to enable the airport to process more flights and passengers in shorter turn-around times and minimize delays.

Two examples can illustrate further:

First is the planning of gates and positions. On many smaller airports this is done in an ad hoc manner, the aircraft arrives and airport operation control (AOC) appoints it with a position. If the position is later needed for a different aircraft, the previous one might be towed. No formalized planning is done. However, as the airport grows this becomes highly infeasible and unmanageable without tool support. Like in this case study, a separate tool is used to automatically plan and assign positions for the upcoming days (two to three days in advance). Manual changes are only done in unexpected situations like aircraft malfunctions. The tool had advanced graphical support to visualize the position occupations of aircraft on the time line. The AODB itself however could still be used to assign an aircraft a planned position (without temporal information) and to manage the current position occupation for all positions. This feature was also essential in cases when the resource management tool was down for maintenance.

The second example is about estimated time of arrival (ETA). This information is actually an educated aggregate of many different sources of information. It starts as the planned arrival time and is subsequently replaced by a first estimate sent by most airports once the flight has departed. This information is again frequently replaced by air traffic control (ATC) data as the aircraft travels through the different air spaces. More precise information is then available once the national ATC hands over the flight to the tower of the airport (10 miles out) which has a dedicated queue for incoming flights and together with the flight path can give a precise estimate as to when the flight should arrive. The last information is received from the ground radar once the aircraft is approaching the runway. The AODB has to understand and correctly prioritize this kind of information flow. If there is more precise data available it cannot replace it with lower quality data. For example on small airports, the departure message is still sent manually, after the flight has taken off. This means it will easily arrive after a more specific estimate from ATC that has taken over soon after the departure of the aircraft and sends messages automatically.

What both examples show is how an AODB has to evolve with the airport it is running on. As more specific systems are acquired to help in specific planning and resource allocation tasks as well as sensors providing live information about the current tactical traffic image the AODB will have to handle the increasing stream of data and prioritize and pre-process accordingly in order not to overwhelm the end user.

Figure 4.14: Legacy System user interface

Technical Aspects

Application Design

The basic design followed the classic three-tiered web application design. The database was accessed from a persistence layer, business services were decoupled using an inversion of control container and exposed both via REST-ful web services and message beans. The user interface was done using client side rendering technology and used the same REST services as were used for integration with other systems.

The business services and transactions were designed with a strong focus on the main functionality. All functions followed the same pattern. After syntactic and semantic input validation the necessary calculations and database modifications were performed. This was followed by downstream calculations and event generation. Cross cutting concerns were as far as possible handled by aspects and configured via annotation mechanisms. This included transaction and error handling, authorization and authorization based result filtering as well as logging and tracing.

The design and implementation of the user interfaces was one of the key challenges in this project. Users were used to functional, minimalistic interfaces, the ability to work only with a keyboard (including navigation) and of course lightning fast response times characteristic for mainframe based terminal systems. To translate this into a web application while retaining these characteristics was a non-trivial task. Client side rendering and input validation using JavaScript was one of the key success factors in this space. The web interfaces were typically designed as minimalistic and close to the VT100 terminal interfaces as possible. “Elements such as find-as-you-type or drag-and-drop were added as additional input methods [...]. This strategy was referred to as sugar on top.” [19]. All user groups were involved during the analysis and design phase of the user interfaces with communication based on contextual inquiries at the work environment of the prospective users, workshops, wire frame prototypes and finally early user acceptance tests. The transition from legacy system to web interface via a wire frame prototype is illustrated by figures 4.14, 4.15 and 4.16.

Messaging

With more than 90% of all transactions being triggered by automated messaging receiving and sending messages was one of the key requirements for the new system. All messages were en-

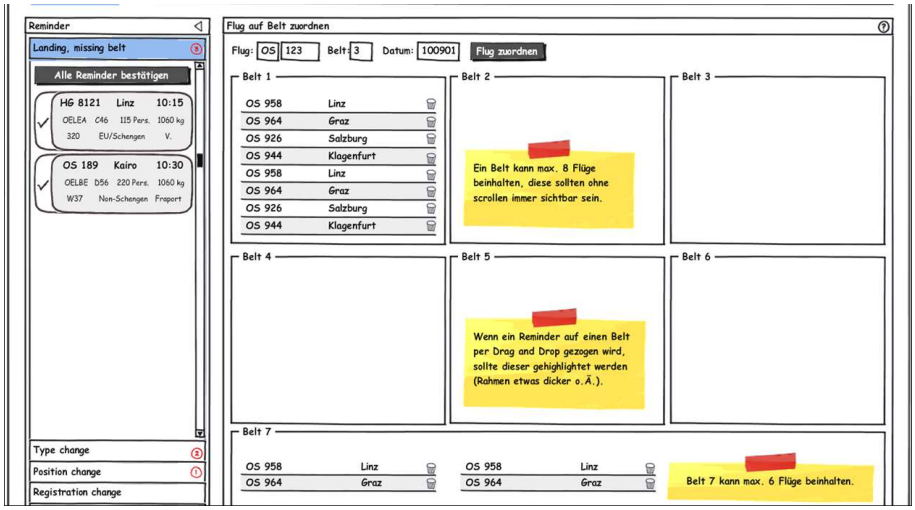

Figure 4.15: A mock up of the planned user interface including notifications on the left hand side

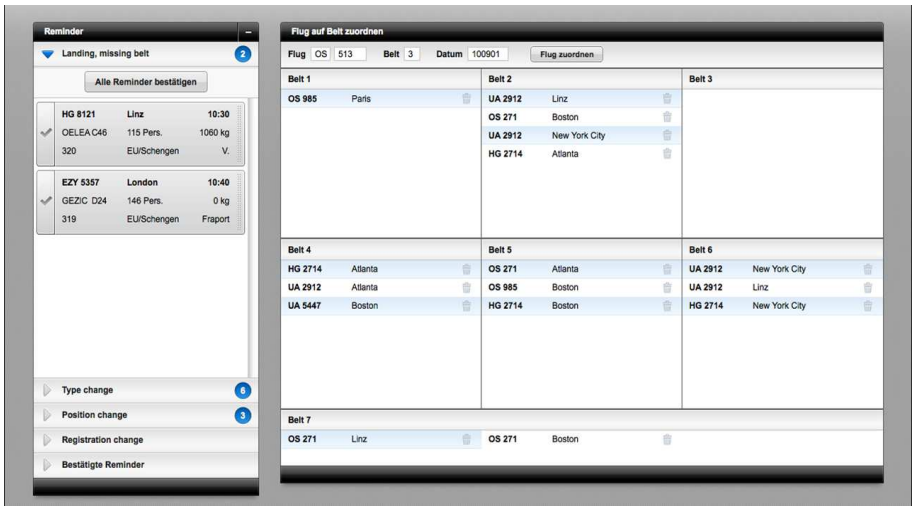

Figure 4.16: The final implementation of the user interface including notifications on the left hand side

veloped, routed and filtered by a message handling application and provided or received via Java Messaging System (JMS) Queues and Topics. For the standardized IATA messages custom parsers and validations were used before calling the AODB business services to trigger the necessary changes. For sending messages a template engine was used to transform the content into the necessary format. Internally the airport used a more modern, XML based format to communicate flight data changes between systems. As most messages of a single type had to be processed in the order of receiving a master node election mechanism was used to ensure processing was only done on a single node. This included automatic fail over in case the current node was brought down or otherwise unable to further process messages.

Legacy System Synchronization

Most of today's mainframe based systems in one way or another support the interaction with other systems through web services or messaging queues. However, these features frequently require additional licenses or even upgrades to newer versions - something that can be highly infeasible. In this case study a manual synchronization mechanism was built based on the VT100 protocol.

This allowed the developers to execute any transaction and read its results that was available as a VT100 user interface.

From a technical side the solution was realized by building the exact input sequence that is normally entered through the keyboard, including control characters and then serializing it accordingly. This sequence is then sent to the dedicated TCP port and the resulting screen is read back and de-serialized to analyse the result and communicate it to the caller. In most cases the result was limited to success or error codes, but in some cases whole screens were scraped and analysed to actually read data from the mainframe. The performance impact of executing the actual transaction was minimal in most cases, however the limiting factors for applying this solution were the cost of setting up a connection (offset by pooling and reusing the connections) and the amount of total available connections. Especially in the beginning of the project as most users were still connected to the host, this meant that at times fewer than 10 connections were available to the new system.

There were two possibilities to communicate with the mainframe. The version used primarily in Step 2 (as described in section 4.3.2) was the synchronous calling. This meant that the caller on the new system actually had to wait until the transaction was finished. Errors could thus be communicated to the end user. On the other hand, in later stages, it was increasingly relevant to try a best effort synchronization without affecting user perceptible performance of the transactions in the new system. For these cases the necessary synchronizations were only registered for execution during the transaction processing and then executed asynchronously after a successful commit. This way a failing or long lasting transaction on the mainframe could not affect the user. However, this also meant that failing transactions had to be monitored and necessary data corrections be performed manually by operations.

Eventing and Notifications

One of the major paradigm shifts was towards an event driven design replacing most of the legacy systems batch processing. Basically the host had two major timed transactions, one running every minute, the second every five minutes. Both performed a multitude of tasks like pushing data to the flight interface, producing notifications, updating information and sending outbound messages. This had the positive effect of higher performance in the actual transactions, as they included less business logic and an automatic retry with the subsequent batch run, but at the same time the negative effect of a single block of code with a large amount of different, ever growing responsibilities. This resulted in a situation where the batches sometimes took longer than their window allowed (1 or 5 minutes respectively) and therefore functionality had to be moved back to a bigger window delaying its execution and the availability of corresponding information.

In the new system, every change to a flight triggered a corresponding event. These events were used for a multitude of purposes. Most important was the notification of all other airport systems of the change. However, the events were used as well to generate user notifications and decouple functionality.

The user notification mechanism was another field of generous innovation. The legacy system had a hard coded mechanism to generate user notifications in certain scenarios. They either indicated an unusual or important event like the last minute change of an aircraft which requires a user to evaluate position and gate planning or might require different equipment for loading. Or they are used to notify a user of an event that has not taken place, like a flight that has just landed, but does not have a position or baggage belt assigned, in other words, has no place to go. These “missing” notifications were generated repeatedly until the situation would be resolved. Resolving the situation would automatically resolve the corresponding notifications as well. Unlike the legacy system, the notification mechanism in the new system was completely decoupled from the business logic. It can be configured which notifications are generated based on which events. This

configuration, stored in the database, can be modified at runtime. A similar approach was chosen for the delivery of notifications to end users' "inboxes" (for an example see figure 4.16). Even the data that each notification would display was evaluated through a scripting environment (expression language). This enabled the configuration of arbitrary new notifications for any event that would occur during the life cycle of a flight with minimal effort. This type of change was regularly blocked in the legacy system due to the complexity and wide-ranging impact.

Automated Integration Testing

One of the most crucial factors for the success of any highly automated system is the availability of a comprehensive and effective test suite. This was realized in this case by including automated black box system tests as part of the regular development cycle and consisted of two major parts. First the test setup was done so the tests could be run completely headless and in memory. This included an in-memory database as well as JMS application. Based on these parameters the whole application was started (without the user interface) and a sequence of messages and service interactions could be used to simulate arbitrary scenarios. The second building block was the conception of an expressive language for test data generation that enabled each test case to setup (and discard) its own set of test items (most notably flights) in exactly the initial state it needed.

All of the tests were executed on every build of the continuous integration system. At the end of the project more than 2500 test cases covered the code base and could be executed by every developer in less than ten minutes. This provided a high level of confidence for developers.

The testing infrastructure also proved highly useful in recreating complex production scenarios and simulating the course of events to reconstruct difficult to trace errors. It was therefore an invaluable tool in the weeks following the switch to be the new master AODB.

4.3.3 Experience

Verification

Verification of the resulting data in the new AODB was one of the core challenges in this case study. In general validating the correctness of the migrated application in terms of "it does what the old system has done before" seems to be one of major unsolved challenges in the field of software renovation [18, 19, 108, 110].

Apart from the usual quality assurance activities (unit and integration testing, manual black box and system tests, load & performance tests) special effort has been undertaken to enable parallel testing. The basic idea of this kind of tests is to run a test stage with the new version of the software in parallel to the current production system. At the start of each testing cycle the data between test and production has to be identical - ideally it is started with a complete copy of the production database. At the end of the testing cycle the two databases are compared to find any differences. Based on the analysis any behavioural differences can be eliminated (or explained and discarded).

This approach was applied in two different scenarios at the airport case study. First it was applied for the passive flight component after the modification as described in Step 1 of section 4.3.2. The component was modified to replace a state engine saving an initial state and a series of subsequent changes which was then used to calculate the current state with a system saving the current state and performing modifications on it through services. While this was done as a large refactoring, in the end it meant little of the production code base remained. The advantage in this case was the passive nature of the flight core. As it was only fed by one stream of input, this stream could easily be diverted from production to a test stage (or even a local developer machine). After setup no manual interaction was needed. The results of these tests by far exceeded expectations. It

was equivalent of running 800 new test cases per day. While it highlighted some errors in regular functionalities, the main benefit was in the many corner cases (as the relanding already described in a previous section). In the end it even provided functional input for business analysis by indicating scenarios that were not considered previously.

The second and more complex parallel test scenario was used during and after Step 2 of the migration strategy. This included manual input from actual users and was as a consequence more complex to setup and execute. For these "inputs testers were located next to operational airport personnel and manually redoing their actual inputs in the test stage." [19]. This was mainly possible due to two reasons: First of all, the amount of manual transaction was and is fairly small with less than 10% of the total transaction volume. Second, the individual user groups were all located on site and very cooperative, supporting the effort and taking the time to explain what they had just done and why. For larger volumes of manual user input in other scenarios, a variation of this strategy would have been necessary utilizing some form of capture & replay mechanism to further automate the process.

Code Reviews

Closely related to verification through testing methodologies as discussed in the previous section is the application of code reviews. As already mentioned the software was developed using a SCRUM-like agile development process. However, these frameworks strongly focus on automated, mainly static source code analysis to spot potential errors during continuous integration. While this is a valuable tool to detect syntactic errors and all kinds of "code smells" [51] it is not sufficient to detect many semantic errors and also does not cover additional files like XML-based configuration. The decision was therefore taken to perform manual code reviews in the form of a Delta-Continuous-Review (DCR) [16, 17] strategy.

The exact methodology, described in more detail in [18], allowed manual code reviews to be continuously performed in a distributed, asynchronous manner based on deltas or differentials to the previously existing code base. It was integrated into the regular development life cycle (sprint) and consequently was an established element of each feature development. For each feature (or work package) the developers (work was frequently done in pairs) performed the necessary changes which were linked to the feature using the source code management systems commit hooks and the task tracking tool (depicted in figure 4.17). After the developers were finished, a randomly chosen developer not previously involved with the feature takes the review task, aggregates all changes affecting the feature and reviews them using his IDEs compare editor. The reviewer's comments are then saved in the review task and the original authors are notified to perform the necessary rework.

A total of 114 code reviews were conducted during the course of the project. All of the developers performed at least a couple of reviews. The usefulness and effectiveness (base lined against standard IEEE-1028 team-walkthrough code reviews) was evaluated by the developers after the project. The results, published in [18], show a highly positive perception due to the low overhead and beneficial side effects on code understanding and collective code ownership. However, the formal effectiveness of 30 (percent of total issues found during code reviews) is below values found in literature (about 57% [78]). This stands in contrast to a clearly positive perception by all developers involved, which unanimously prefer the chosen approach to formal walkthroughs. In the author's view, especially compared to other migration projects, this clearly indicates a workable compromise to establish a review culture in fields where reviews are not formally or legally required and might otherwise be omitted completely.

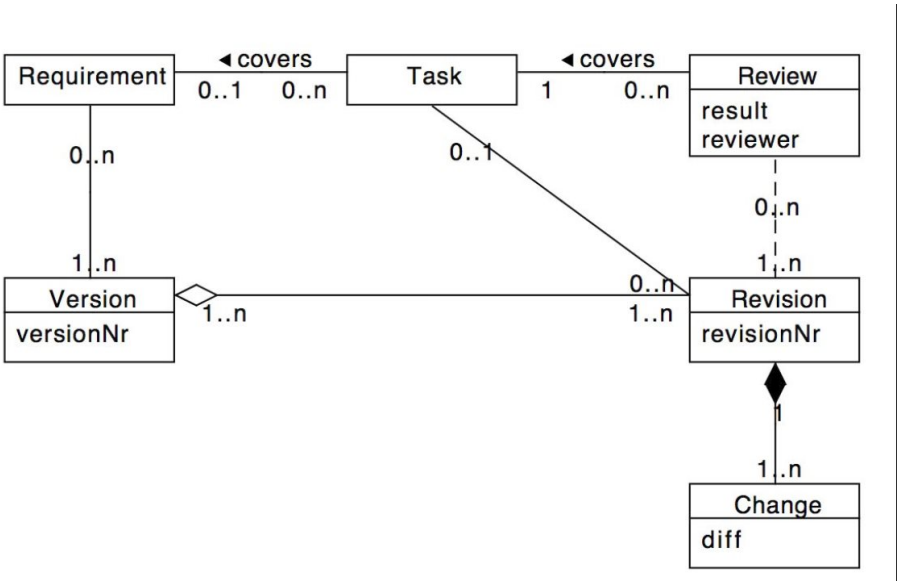

Figure 4.17: Data model for reviews for linking between entities is supported by the development environment services [18].

4.3.4 Alternative Outcomes

New Interaction Design

As previously mentioned the user interfaces were largely designed with a functional, minimalistic mind set focusing on a one functionality/transaction per interface paradigm which were then grouped and organized into workspace closely resembling the individual user groups. Only isolated features were allowed to be designed with some "sugar-on-top". This was definitely appropriate for the one-to-one strategy behind the actual migration as well as to minimize the effort necessary both for interaction design and end user training or documentation.

However, the technical design of the application with web service in the back end and a lightweight detached user interface based on client side rendering would have allowed to bring the user interface into the 21st century. This vision is based on a couple of basic ideas:

Querying is done via a generic interface allowing all kinds of queries based on the characteristics and meta data of a flight. However, by analysing the most popular queries per user group an optimized query interface could be built providing preconfigured queries (e.g. all incoming flights arriving in less than an hour). Taking it one step further, the most popular queries could be aggregated on information screens which automatically refresh avoiding the user input all together. By caching the respective query results this could also have positive performance side effects.

In addition, querying and input screens are currently separate. A flight can be searched, the user has to remember the flight identifier, go to the respective input screen and enter the data to be updated (e.g. the block off time). With simple pull down menus and placing of the contextually right functionalities, these interface switches could be avoided completely. This is especially relevant for screens like the current position occupation showing exactly the aircraft (and flights) currently eligible for block off.

The strict validations typically limit the amount of flights/flight identifiers available for certain operations to a hand full. To continue the block off example, the flight has to be at a gate or on a bus boarding position and the gate has to be in state "closed". Even without the second limitation the number of possible flights is limited to the number of positions (less than 100 in this case

study). These flights could be provided as recommendations to the currently selected operation to save the user from having to enter the respective flight identifier.

The user notifications, especially the missing notifications typically indicate to the end user that a specific action has to be taken for this flight. To continue the block off example, if no block off has been entered in a certain time period after the planned departure time (and there is no delay) a missing notification is created. Based on this the user then needs to perform the missing action (or another compensatory action). By providing the necessary actions as part of the notification the context switches could be avoided. This approach has been prototyped on the screen shown in figure 4.16. The notifications on the left hand side show flights that have landed, but no baggage carousel assigned. These can be dragged and dropped on one of the carousels displayed on the right to assign the flight.

The tactical traffic image is called an image, but unfortunately not visualized as one. Creating a comprehensive graphical representation of the ever changing state of an airport would be a great challenge for a group of user interface designers, especially with today's high resolution screens at their disposition. In conjunction with ever improving sensor data this could mean a near real time simulation of the whole airport, a possibility that could amongst many other benefits greatly improve security in low visibility environments. Taking it one step further, augmented reality technology could be used to display relevant information close to the (physical) aircraft, like the destination position and number of passengers for an aircraft currently taxiing from the runway.

While the AODB aggregates a lot of information, it is focused strongly on the tactical traffic image. This means that a lot of information already accumulating during the regular operations that might be helpful for the end user might not be available. However, the user interface should not be limited by this factor and aggregate information from multiple sources besides the AODB. This could include systems like CCTV, facility management data or any other system running in the IT landscape of a large airport. Figure 4.18 shows a mock up of a gate screen. It accumulates a couple of the ideas discussed in this section. Apart from the current gate state, a CCTV image shows the aircraft just arriving. The upcoming planned flights are shown as well including possible delays. In addition, the right hand side indicates the facility state with some planned repairs during the night as well as the ground handling user currently preparing to board the passengers.

4.4 Case: A Highly Loose Quagmire

Universities all over the world can be considered the cradle of computing. Therefore not surprisingly they have been among the first to build and adopt information systems to help with the daily administrative chores. In recent years the term Campus Management Systems (CaMS) has emerged [59] to describe these systems. Many of these systems have been and are still around for more than forty years. However, in the last ten to fifteen years two major, disruptive changes have pushed the legacy systems to their limits.

First the arrival of the Internet and with it the possibilities of e-Learning have dramatically increased the user base of CaMS from a couple of hundred or thousand employees to tens or hundreds of thousands users by including all students and later, to some degree, the public. It is now standard for students to register for courses and exams, pay fees, organize their time table, upload exercises, download materials, check exam results or communicate with their teachers and peers online.

Second, at least in Europe, the all-encompassing introduction of new curricula caused by the European Unions Bologna process has forced many universities to break up established structures

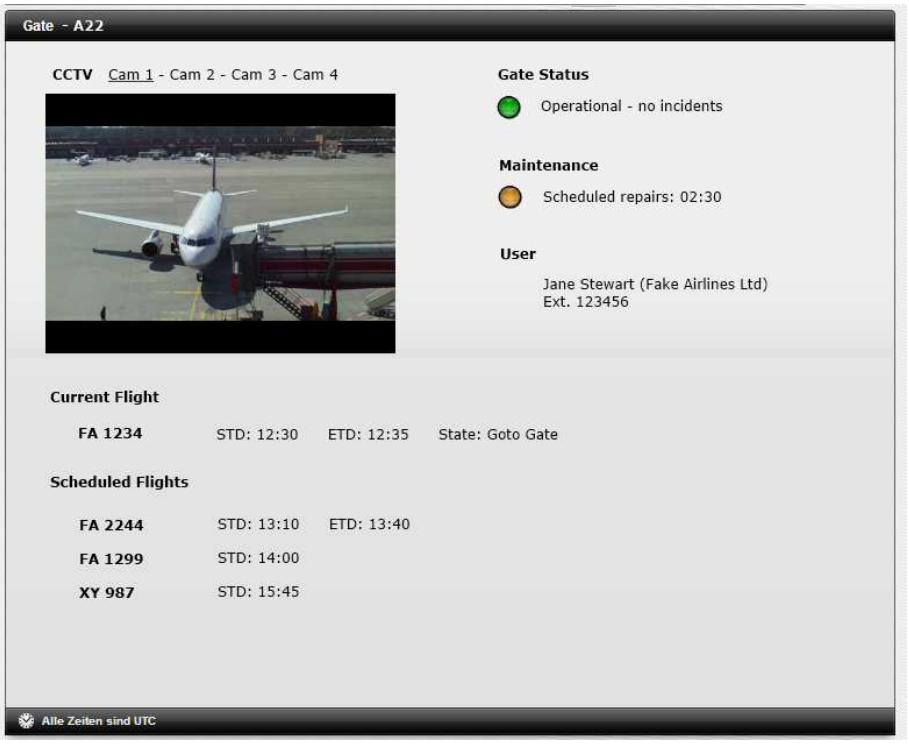

Figure 4.18: A mock up of a possible aggregated gate workspace

and procedures. However, regional and institutional specifics have so far prevented the emergence of a successful suite of standard products [22].

This case will present the complete renovation of an existing, highly heterogeneous CaMS. A more general, software engineering centric version of this case has already been presented in [20]. This case study will also use the pseudonym HISS established then. The university in question is one of the largest universities of the country with more than 30 000 active students and approximately 4500 staff.

4.4.1 Status / Analysis

In early 2007 the decision was taken by the university board to replace the ageing quagmire that was barely being held together by a team of veteran engineers with a modern system representative of the good reputation of the house. The final catalyst for this outcome was the decision by the European Union to harmonize studies across the continent according to the Anglo-American Bachelor and Masters model [59].

History

The history of the IT system dates back to the year 1968 when it was introduced to help with both student and staff administration. The largely batch oriented system was written in COBOL and running on a mainframe. Most of the business logic was hidden in the nightly batch runs. Data was stored in files. Innovative at the time, the system was not only used by the university described in this case study but also provided to several other local universities.

In the mid 80s with the arrival of relational database systems (RDBMS) the persistent data of the IT system was migrated to an external database running Oracle. Furthermore, a data access layer was

provided to enable access for the legacy programs. However, neither the structure nor the quality of the contained data was improved during this migration. In addition, barely any of the newly acquired database management features were put to use. The legacy database did not make use of primary or foreign keys or any kind of constraints. This was mainly due to the bad quality of the pre-migration data that prevented the (easy) implementation of any kind of constraint. Essentially the task was put off for a long planned but never implemented data quality improvement project.

At about the same time new and extended applications were written using a vendor specific rich client application library to enable window based user interfaces. This introduced a second language of choice, PL/SQL for implementing business logic. However, most programs were still limited to simple data entry with the user interfaces providing a thin layer above the database. Almost no server side logic (except a number of triggers and database functions) was implemented.

A first attempt to a web enabled system also catering to students was made in the late 90s. The system developed with an external partner was focused on newsgroups and discussions but also provided the possibility for publishing information about courses and lectures. Students could for the first time register for certain courses online. However, after an initial popularity surge the acceptance of the system did not reach the expected levels and dropped until the project was shut down in the early 2000s. A series of surveys indicated that the amount of effort necessary for the staff was too high and the resulting benefits for the students was too low.

Learning from previous mistakes the university again took matters in their own hands and adopted a then popular web framework to extend the existing information system for the web. The new system was able to access and build on the existing information (primarily student and lecture data) and accordingly provided the full curriculum from the beginning. Features were then added incrementally in close cooperation with student unions thus better reflecting the needs of the end users. Some of the legacy host applications were also migrated to the new system. However, both student and staff administration remained as terminal applications until the last day of the legacy application.

Organization

The organization of the university, as far as the renovation project is concerned is surprisingly straight forward. The project team consisted of two individual teams. The first team is responsible for project management, business analysis and requirements engineering. It is mainly staffed with university employees specialized in software engineering practices. The second team is responsible for development and testing and staffed by members of the internal IT department. It includes a number of developers of the legacy applications.

The teams are jointly headed by a product manager and an enterprise architect, supervised by the project lead who is staffed by the director of the internal IT department. Together they are reporting to the steering committee (approx. monthly, sometimes bi-weekly) which again has to report to the board of the university (quarterly).

The initial project plan was structuring the project into three main phases, each with a planned duration of about one year [74].

- Phase I: Preparing the migration involves iteratively refining the target architecture and reverse engineering process. The main focus however is the recovery and consolidation of the business processes.

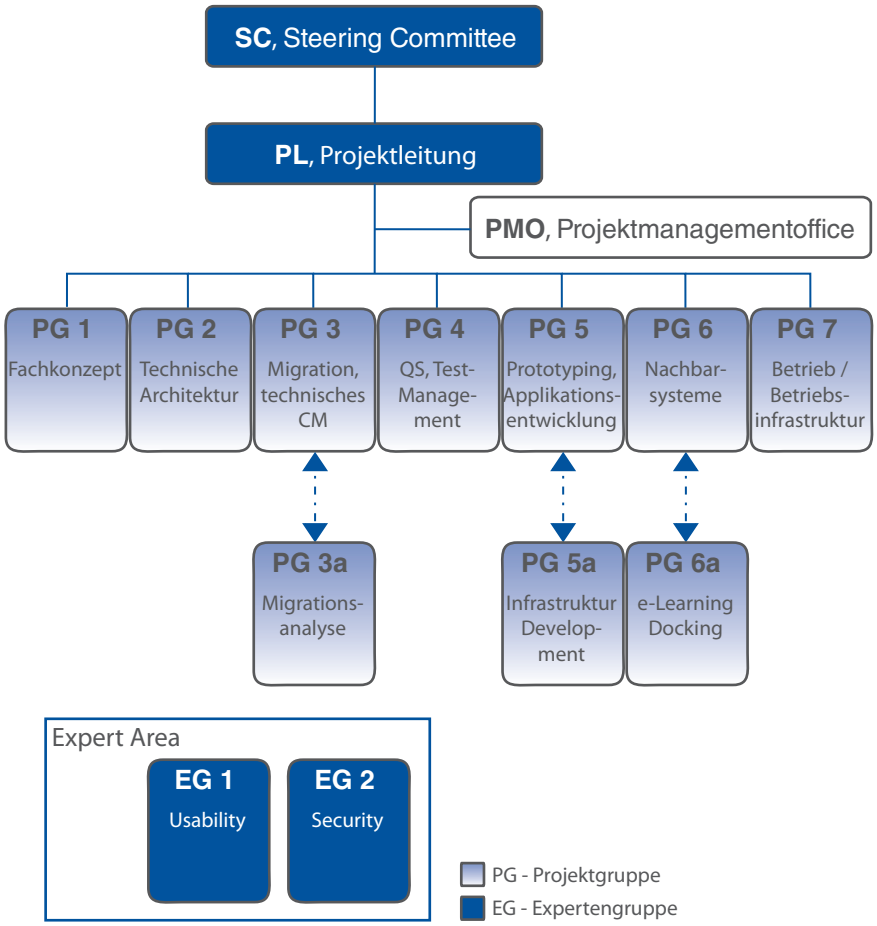

Figure 4.19: The university CaMS migration project organization [20].

- Phase II: Starts the migration of the first legacy systems building on the knowledge retrieved in the previous phase. In addition, the technology stack and architecture as well as the supporting processes (release, end user training and rollout) are finalized.
- Phase III: Focuses on bringing the IT system into full operation and replacing the remaining legacy systems. Furthermore the transition towards operation and maintenance by the university IT department is prepared.

Enterprise Architecture

As depicted in the left hand side of figure 4.20 the legacy architecture shows a fragmented landscape of historically grown, isolated applications. The heterogeneous nature of the used technologies and a significant overlap in functionality and data make maintaining and operating the legacy systems tedious and error prone. Implementing larger changes is a costly and high risk endeavour mostly avoided by management.

The main, overarching part of the university information system consists of two logical fragments. The core legacy system built mostly in COBOL and used throughout the university mostly by administrative staff. In addition, the web enabled part of the legacy system is used primarily by students and lecturers with a strong focus on the educational aspects at a university. It strongly depends on the base data of the core legacy system (e.g. lecturers, rooms, etc...). A third, somewhat decoupled, central system is the master data database. It offers a consolidated view of personal

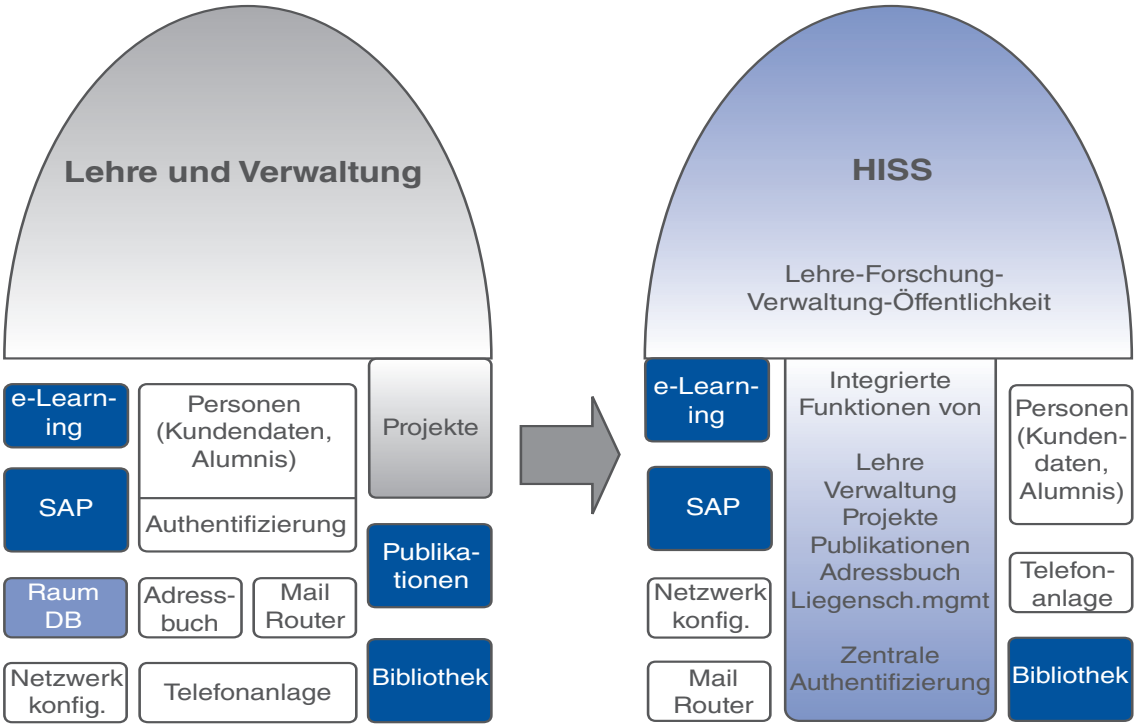

Figure 4.20: The university CaMS architecture transition [20].

master data for both students and employees. However, it highly depends on the correct data being present in the core legacy IS.

Around these systems are a series of established information systems that will not be in the scope for replacement/reengineering but need to be integrated to uphold and improve the overall enterprise architecture. Most notably this includes the Enterprise Resource Planning (ERP) systems, an established e-Learning platform with a strong need of close integration with the education modules of the new system and a library management system. Another specialized system that will be replaced in parallel to the effort described in this case is the phone system. This project will transition the phone setup towards voice over IP technology and in that course will again have a strong need for direct integration with master data.

4.4.2 Solution

A general overview of the solution as the case HISS in software engineering (without focus on migration or renovation) is also given by Grechenig et al. in [20, 103]. One of the key focuses there is the elicitation of requirements from a combination of analysing the legacy system and stakeholder workshops. In the following section the focus is on the target architecture on the one hand side and the actual progression of the project on the other.

Target Architecture

As shown in the right hand side of figure 4.20 the target architecture aims for both functional and technical consolidation. A total of 11 different legacy systems will be migrated within the scope of the project. This will eliminate a large amount of redundancies and integration points and result in a full featured campus management system (CaMS) as defined by Bick et al. [22] as well as

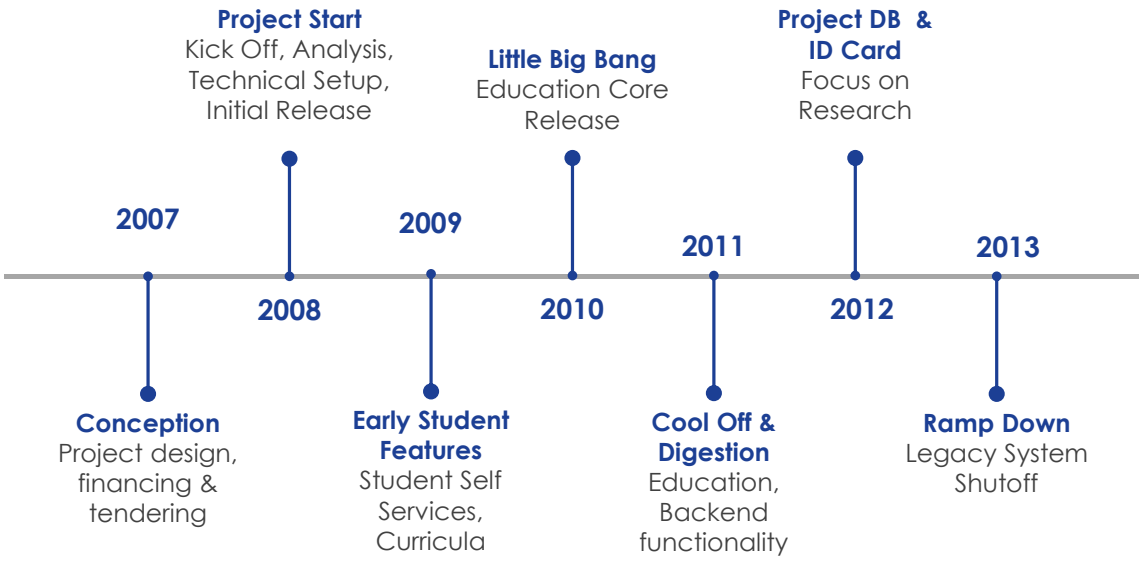

Figure 4.21: The timeline of the university case.

Spitta et al. in [113, 114]. In addition, clear system boundaries will help to distinguish it from existing solutions in the field of ERP, e-Learning and library management.

The functionally uniform solution will be complemented by a technological consolidation on the platform "Ruby on Rails". As this part of the planned target architecture was significantly revised later on, it suffices to briefly note the original intentions at this point. To avoid the formation of a monolithic structure the design called for a separation in multiple runtime components as well as proper modularization.

The target architecture will be established in line with the general approach in an incremental fashion. This allows for a continuous evolution of both the functional and technical aspects of the original design. A very good summary of the evolution of the target architecture can be found in [59].

The project

The project soon deviated significantly from the three phases depicted in the initial plan. This had both organisational as well as technical reasons which we will cover in the subsequent sections. In this section we will try to depict a high level, but actual timeline of the project following the schema depicted in Figure 4.21.

Conception

As mentioned in the introduction the project was authorized by the university board in late 2017. In this phase the project structure was established and team was formed. Everything was prepared for an intensive start in the new year.

Project Start

The early stage of the project was the focus on master data consolidation. Under the title of a unified address book numerous sources for data were consolidated and a new single source of truth

was established [118]. One of the longest standing issues in the data architecture of the university was corrected by no longer duplicating data between student data sets and employee data sets. In other words, a student who becomes an employee, a very common scenario with plenty of short term tutoring jobs, is now only represented once in the system. This was a major achievement both from a data consistency and a governance point of view. However, on the technical side it was increasingly obvious that the architecture and tool stack would need a significant overhaul, to scale to the full feature set.

Early Student Features

With the gradual establishment of the extended architecture a growing number of supportive student services were introduced. This enabled the students to complete administrative tasks online without heading to the physical student offices at the university. For the first time in the project there was the chance to have a positive impact on the daily life of the students and as a result gain more visibility and traction. Students could print certificates and various confirmations and register and get approval for bachelor and master theses.

A second significant feature set was introduced with the digitalization of the curriculum and the possibility to digitally process completed studies [21]. For the first time a paper intensive and long running (weeks to months) process was supported end to end by the IT systems. While a new feature as such it established one of the key success factors for the subsequent stages, the data model for all curricula of the university. This was a never before achieved feat as so far paper based curricula allowed for almost unlimited amounts of freedom in designing them.

Little Big Bang

Building on the early features and foundations, the core education features (course management, exam management, lab management) were released with the beginning of the university year (early fall) [117]. This was the only possible cut-over date between the new and the old system for these core features. Any delay in releasing these features would have had wide ranging consequences with a high risk of project cancellation. With the eventual success of this release the new system was firmly established as the primary IT system for all education purposes at the university.

Cool Off & Digestion

The Herculean effort of bringing the core education system to production combined with an intensive stabilisation phase in its wake required a certain degree of cool off [119, 120]. Apart from tweaking and optimizing features based on the feedback and learning from the initial year of operation, the focus gradually shifted from education to the areas of research and administration. A series of new subsystems were created to support these areas.

Project Database & ID Card

The final two big chapters of the project were completed. The project database supports the complete project life cycle for all types of projects implemented at the university including funding related features. With the introduction of a new ID card students and employees finally left the age of paper based IDs and stickers in them behind them. The new innovative card was also used in a newly built part of the university as a key to accessing restricted areas and resources. In addition, further administrative features were completed in preparation for the final ramp down of the legacy system.

Ramp Down

With the shut down of the main legacy system now imminent the remaining smaller features (and missing parts) had to be replaced or eliminated. 40 years of legacy system history therefore come to an end. The project team and setup were subsequently dissolved and the further development and maintenance handed over to the internal IT department.

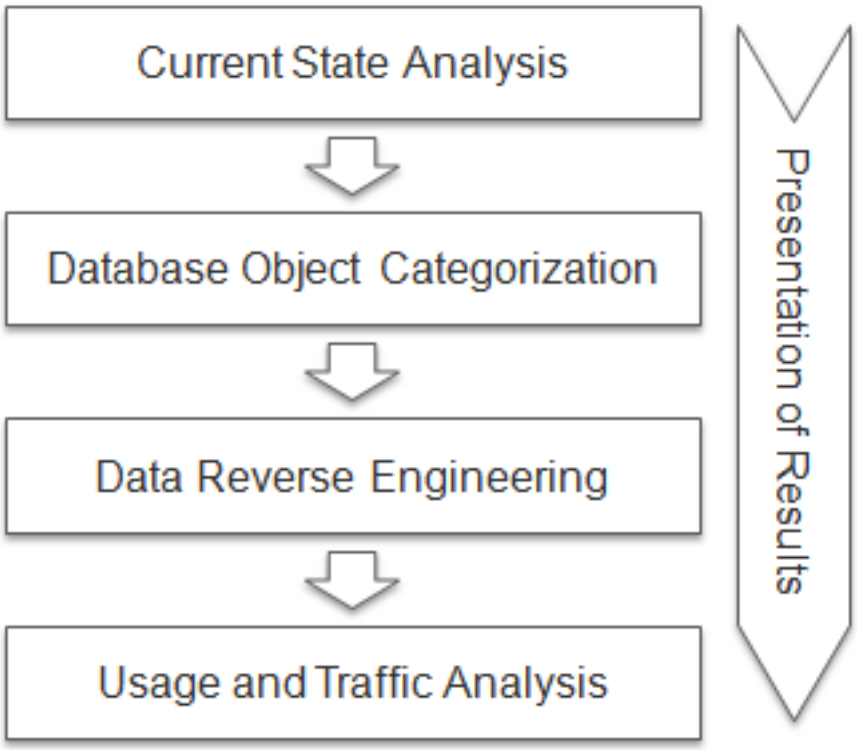

Figure 4.22: The process for reverse engineering the databases of the legacy CaMS.

4.4.3 Experience

Database Reverse Engineering

The legacy databases were the most valuable asset of the legacy system in this project. Learning from the existing database structure and data was therefore a key source of ground truth information during the project. Based on the work of Premerlani [95], Blaha [26] and Henrard [65] the author [116] created and implemented a process to redocument the database and provide this information to the different stakeholder groups.

Four problem areas were identified. Outdated or completely missing documentation forces the stakeholder to go back to the database to obtain information about the schema as well as the contained data itself [25]. Missing Normalization has a major impact on data quality. Data model deprecation unnecessarily increases the amount of work to be done during the migration and creates noise distracting from the actually valuable assets inside the database. Finally a missing correlation between the database structure and data to the responsible application or applications increases the risk of unwanted side effects.

The resulting process, as depicted in figure 4.22 can be executed iteratively to incrementally cover additional areas of large databases or to reflect improvements in the legacy database.

During the current state analysis CASE tools were used to extract the source physical schema from a clone of the production database. A total of 346 database tables containing more than 5600 columns were identified through this step. Some minimal documentation in the form of comments stored with these database objects were extracted as well.

Subsequently during database object categorization all identified objects were categorized into business relevant, technical and implementation specific and deprecated assets. Only the former would be considered for migration.

Data Reverse Engineering as described by Aiken [2] is the next step, focusing on extracting knowledge out of the data assets themselves. Basic information like value ranges or minimum and maximum length can again be extracted using CASE tools. More sophisticated analysis like the elicitation of implicit foreign keys [61] or data duplication are still predominantly manual tasks.

The final step of usage and traffic analysis enables to validate the results of previous steps by highlighting the usage patterns of individual database objects. Objects not accessed during longer periods of time indicate unused assets whereas the frequency of usage can help predict the future load for specific features.

Technological Evolution

As mentioned in the previous sections, during the early stages of the realization of the new project the impact of the initial technological choices became obvious. There were significant difficulties in adapting the technology to the overall needs of the project and especially in scaling the development effort to ensure the necessary throughput for successfully completing the project in time. This risk was identified in a timely manner and extensively discussed in the responsible boards. In all there were three possible options

- Complete transition to a new technology. This decision was a tough sell. In projects spending public money even more than private industry projects this would have been close to admitting project failure. While a long term transition was never completely ruled out, it was also never a part of a strategic decision.
- Sticking with the technology choice. The other extreme, this decision would have meant doubling down on the technology, investing significant resources in its adaptation, most likely bringing in outside help and expertise. However, it would have also meant training a high number of developers in a technology where they have absolutely no previous experience.
- Establishing a second technological pillar. This solution was the logical compromise. It allows to keep existing investments and retaining a team already productive in this technology. At the same time, building on an existing prototype, a new technology could be established without too much disruption.

With the first two options being politically infeasible, most effort went into deciding the strategy and technical details of the third approach. The resulting solution was transparent to the end user, being fully integrated at the user interface layer. Underneath, however, two distinct applications were running the two pillars. While this introduced additional technical complexity, it also had some major benefits. In conformance with Conway's Law [39] the two separate systems with separate teams behind them had a visible effect on the overall design of these systems. This effect ensured that the main component built in the first technology, the master data administration, an absolutely crucial component to the overall system, was encapsulated in a clean way and provided well defined interfaces to the outside world, both the other parts of the new system as well as third party systems. Data ownership, for the first time in history, was technically enforced. The remaining system was, although homogeneous from a technical perspective, structured in a similar way and teams organised accordingly. Over time, new modules and responsibilities were added

on both sides, with features that needed additional encapsulation (like IP telephony integration) frequently going to the initial stack.

In retrospective this evolution highlights two key points: For one, technical decisions, although being proven wrong during the project do not necessarily have a fatal impact on it, as long as the organization as a whole reacts in time and appropriately. In other words: projects do not fail for technical reasons. Secondly, a solution that is politically and socially workable can have a significant positive effect on the design and overall technical qualities of the system.

The Last Mile

One of the observed difficulties is the completion of the last mile towards a complete decommissioning of the legacy systems. Up to the phase called Little Big Bang (see section 4.4.2) there was a clear and strong focus on migrating legacy features and replacing the legacy systems. This was the proving point for the project. Achieving this milestone fully legitimized the project and established the new solution. Politically bringing live this part of the system was a point of no return. Killing the project silently afterwards was no longer an option. However important the success of this milestone by reducing the pressure to deliver on the team, it also took away a large amount of high level management attention.

The effect of this situation can be seen by the focus of subsequent releases in this case. It almost immediately shifted from replacing the legacy functionality to required new functionality. This can be (partly) explained by the intensity of the previous effort, binding large amounts of resources and focus. Reasonable to some degree and in the nature of competing priorities and interests, it presents a significant challenge to the legacy migration effort. The success of the little big bang translates into the realization of a good portion of the promised (if project promotion was realistic) tangible benefits like a modern, user-friendly interface or the ability to realize long-standing feature requests. However, a lot of the less prominent benefits as the reduced cost of operation frequently only come about once the system has been completely decommissioned. With the pain points being reduced to maybe less vocal (or heard) stakeholder groups (as the operations department frequently seems to be), the last necessary steps are prioritized lower and as a result dragged out.

The downside of this delay is a negative effect on the realization of the full set of benefits as promised by the migration project. In addition, unplanned and often unmapped complexity in the enterprise IT landscape occurs as a result. Little islands of remaining functionality with business value remain. As with pockets of ember after a large fire these can, over time, flare up emerging as a new blaze of problems at unexpected times and places. The effort for putting out the last pockets can seem disproportionately large, but it is equally important to realize the full potential and to be able to start new projects without initial, hidden debts.

4.4.4 Alternative Outcomes

A major question mark that needs additional consideration in this case is the implications of a failure at the point of the little big bang release (see section 4.4.2). As mentioned the one fact carved in stone was the release date of this release. It was either then, or a full academic year later, as a transition during the academic year was considered to be highly infeasible and failure even more disruptive.

From a distance and in retrospect there might have been different ulterior motives. Leaving high level politics aside to focus on the aspects in the influence of the project team: Foremost the given strategy, provided it does not fail is a brute force way of moving forward. Once the whole

university has started using a new system for a couple of days (or weeks) going back is de facto impossible. Without an explicitly designed and implemented way back, which did not exist, the old system was thus not revivable after a short period of time. This establishes facts. Furthermore this strategy enables great flexibility. Most of the temporary transaction data does not have to be migrated. This allows for the new model and system to be non backward compatible and as a consequence deviate in functionality and technical design. On the other hand it also removes a validation step that ensures to some degree all functionality is present and usable. In addition, allowing data to be partially dropped during the migration opens a window for data loss. Lastly the approach is practically not testable. Unless the whole university plays "start of the academic year" at once, the dynamics of the beginning of an academic year are very hard to emulate. From a risk management perspective the chosen approach is difficult to justify. The arguments are largely in line with that of any big bang approach. The risks and potential rewards are high. The fact that surviving the introduction of the new system is counted as success is a prominent indicator.

An alternative outcome would therefore be an incremental approach to migrating the educational core of the system. To be able to design and develop this, the release would have most likely had to be postponed by a year. The key features of an incremental release would add the possibility to individually select courses or fields of study for partial migration. The former is primarily needed for testability. Courses can then be migrated during the academic year and be validated by actual users in their day to day work. This fine granularity allows for intensive and targeted testing of edge cases. The latter might have been a valid strategy for incrementally transitioning the university to the new system. This might have meant for some of the students and most of the faculty a period of simultaneously using both systems, but enabled the chance to control the speed of role out and at the same time incorporate the incoming user feedback. The possibility to migrate during the academic year would have, on the other hand, opened up a completely different time line. It would have eliminated the strict once-a-year window for migration and replaced it by the possibility to roll out the new system over the course of weeks or maybe months.

Taking this concept one step further an even more fine-grained and user driven scenario would have been to give the lecturers a choice of where they are conducting the courses. This would have resulted in a high pressure towards the project team to sell this to their users. There need to be clear incentives in terms of additional features, greater usability or better presentation of information to lure early adopters. Through word of mouth, a considerable force in a university environment, positive feedback would have reinforced a user-driven transition to the new system.

For several reasons, like statistics and accounting, the core data elements would have to be written back to the legacy system in this scenario, as these use cases need a full set of data to produce results. Building on this and taking it one step further, this would have provided at least a basic mechanism for stepping back to the old system in case of severe problems. As a result one of the key risks of the big bang approach, reaching the point of no return, would have been mitigated, almost as a side effect.

Additional risks arise in this scenario as well. Most notably the risk of a temporary solution becoming permanent. In a worst case scenario the acceptance of the new system is not good enough for the remaining stakeholders on the legacy system to follow, leaving part of the university on the new and part on the old system. However, due to the additional possibilities of feedback-cycles and the intensified user-centric testing possible with this approach, this seems unlikely. In addition, writing back into the legacy system inherently carries risk. A faulty new system now has the chance to not only break on its own but also bleed the faultiness into the legacy system.

All in all, this alternative would have been a possibility to greatly reduce the frictions of the introduction of a new system, to better bring the different stakeholders on board and ultimately to

reduce the risk of complete project failure, which was very present during the execution of the big bang.

5 Cautious Abstractions & Generalisations

In Chapter 4 we have presented four cases which were evaluated. These observations highlight the diverse nature of legacy system migration efforts. As already discussed in Chapter 2 this imposes severe constraints on the scientific and industrial communities to elicit reliable processes and tools. Rather than consolidating the knowledge and experience from these cases in a lowest common denominator based methodology we proceed more cautiously: This chapter summarizes the attempt to abstract and generalize in three core areas. Each observation is underpinned by examples from individual cases and designed to build a bridge to subsequent theses and areas of future work.

The generalisations are structured in a bottom up fashion. We start by examining comparatively low level technical experiences, continue on the still technical level of architecture and IT strategy and conclude with observations on an organizational level. Business strategy as the highest potential driving factor has been explicitly excluded from the scope of this research. Figure 5.1 shows a graphical mapping of the topics to their groups and the relative positioning to each other. Business strategy is shown here for completeness, but as a category does not have any assigned topics. The horizontal axis can be considered to be reaching from practical to conceptional where as the vertical axis is used to illustrate the positioning within the group. For example the placement of "IT Strategy & EA" can be read as being on the organisational side of the architectural group. The topic of "Automated Transformation" on the one hand is a highly practical matter with "Tailored Development Process" on the other hand being of a conceptual nature.

For better readability most titles of the individual experiences have been abbreviated:

- Technical Experiences & Observations
  - Cleanup in the Legacy System: 5.1.1 "Cleanup in the legacy system"
  - Elicit Unused Assets: 5.1.2 "Elicitation of unused assets"
  - Automated Transformation: 5.1.3 "The value of automated code transformation"
  - Leading System: 5.1.4 "Always have a leading system"
- Architectural & IT Strategy Experiences & Observations
  - IT Strategy & EA: 5.2.1 "IT Strategy and Enterprise Architecture should be in place before deciding on the target architecture"
  - SOA & Microservice Myths: 5.2.2 "Looking behind the SOA and Microservice Myths"
  - Build one to throw away: 5.2.3 "Build one to throw it away"
  - Repeatable Approach: 5.2.4 "Towards a repeatable approach for understanding a legacy system"
- Organizational Experiences & Observations
  - Strong Architect: 5.3.1 "Enable architectural decision making"

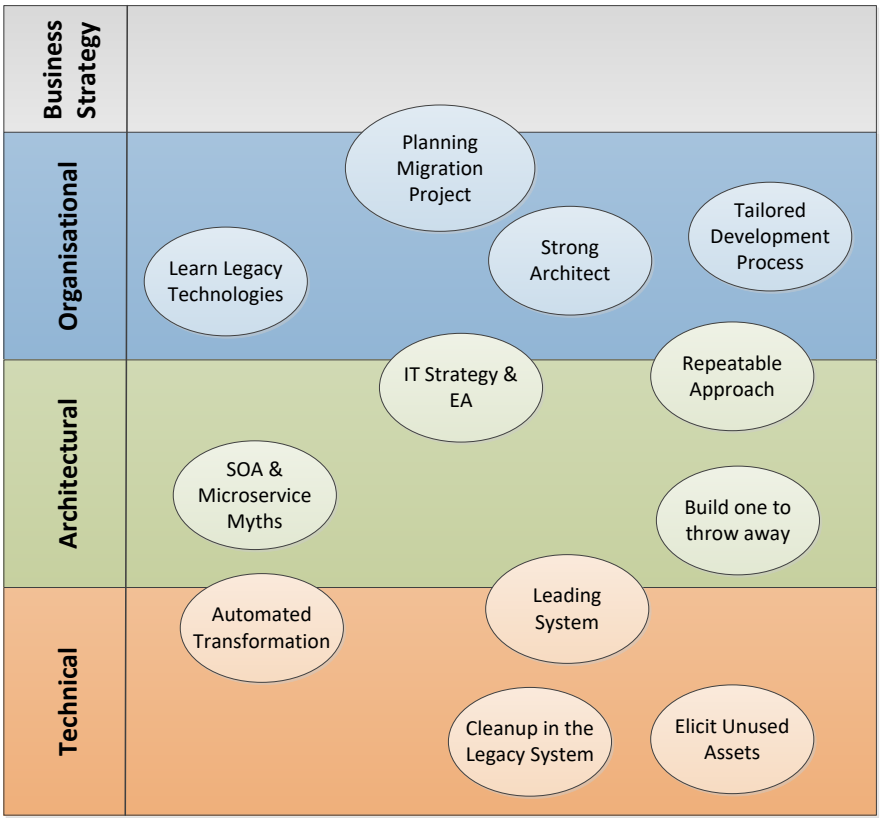

Figure 5.1: The mapping of the cautious abstractions & generalisations with respect to their grouping in levels and practical or conceptual nature.

- Tailored Development Process: 5.3.2 “Established software development process”
- Learn Legacy Technologies: 5.3.3 “Learn legacy programming languages (again)”
- Planning Migration Project: 5.3.4 “Acknowledge software renovation project and plan accordingly”

5.1 Technical Experiences & Observations

5.1.1 Cleanup in the legacy system

Once an organization has committed itself to the renovation of a legacy system, it tends to resist any type of change. From a management as well as a developer or maintainer perspective this seems like a logical conclusion. After all, why invest in a product that will be obsolete in the near future and why risk breaking something in a system that nobody really dares to touch? However, the experience from our case studies clearly show this strategy to be problematic or even dangerous for the overall effort.

Data migration is a significant portion of the overall effort of a system renovation with a high level of complexity. As a transformation from the “old” to the “new” data model is needed, analysing the existing data in detail is a key step towards a successful execution. This will inevitably lead to

the discovery of gaps that have to be filled in order to complete the migration. Simple examples include datatype transformations (e.g. VARCHAR to NUMBER or DATE) and optional vs. non-optional fields. In general, all cases where the new data model needs more information or requires a stricter format (equalling more information as well) need additional attention. Any gap that can be filled in the legacy system will simplify and speed up the migration to the renovated system.

A specialization of the scenario described above is process instance migration. Tool vendors will urge to avoid any type of instance migration. In practice, any three of the following migration scenarios are likely to be encountered (in the order of likelihood): instance migration over process definition versions, instance migration over process engine versions and instance migration over process engines. As described in Section 4.1 and published in [115] the main cleanup that can be achieved in the legacy system is the elimination of process instances using old process definitions, as well as zombie (or stuck) and failed process instances. This does not mean that these process instances should be simply removed, but it might be much easier to restart them on a current process definition in the old engine to be subsequently migrated to the new engine. This has similar effects as pre-emptive code refactoring has. It makes a subsequent change/task much easier to accomplish.

In more complex cases a functional modification of the legacy system can be necessary as well. This is the case, when the legacy system functionality produces data that the renovated system cannot deal with at all and that cannot be handled during the data migration itself. A good example for such a situation can be found in our AODB case study (section 4.3). The legacy system handles a so called relanding, when a flight unexpectedly returns to the airport of origin, by modifying existing data on the outbound flight. This means that an outbound flight all of a sudden has to be treated like an inbound flight handling, for example, touch down times. In the legacy system, this resulted in a series of workarounds in a number of places and a significant amount of exceptional situations that were not properly (or at all) handled by the system - e.g. when changing the aircraft. For the new system, the plan was to properly create a new set of planned flights for the relanded and the new outbound flight. This enables the system to handle all situations correctly, even in extreme cases like repeated relandings. However, this resulted in a mismatch between flight data during parallel operations of the legacy and the new system, breaking the existing correlation mechanisms. The decision to perform this change in the legacy system was by far the biggest functional change in years, resulting in significant opposition from management. In the end “the high change effort was rated lower than the risk of maintaining the 2-flight model in the new system” [19]. In the end however it can be seen as one of the key success factors for the project and retrospectively a core reason for the failure of previous efforts.

The case for pushing changes in the legacy system is based on four main arguments, all of which revolve around minimizing the impact on the ongoing renovation project:

First and foremost there is the availability of resources. The ongoing renovation effort is setup as a project with assigned resources and tasks and a clear long term goal. The maintenance team for the legacy application on the other hand is part of the line organization specifically designed to handle problems as they appear. In practice this should translate to a mechanism where changes to the existing system can be realized within a limited time frame. For the renovation project this has two major benefits: The resources of the project are not bound by the cleanup and they do not have to handle a special case as part of the transformation resulting in a reduction of the total required effort and therefore an earlier completion. In addition, the cleanup can be performed before the completion of the renovation effort reducing both the complexity and the processing time of bringing the new system into production.

Second low data quality in the target system forces developers to elaborate workarounds to mitigate any problems. This can range from simple fixes in validation and presentation code to complex

pre or post processing strategies. A frequently encountered pattern is a distinction during input validation between newly created and modified existing data sets allowing users to save changes to data in a constellation not valid. The main argument behind this strategy is the user's inability to resolve all or some of the issues that can be found in legacy data. However, as a side effect this strategy frequently allows users to work their way around existing restrictions, even in cases that were not initially targeted by the solution, resulting in the long run in an overall lower data quality.

Third, as already indicated in the previous argument, any low quality data that makes it into the renovated system is likely to remain on this level of quality. This means that one of the core goals of a renovation effort, the improvement of the quality of data, is compromised even before the new system comes into operation. In addition, by weakening both the validations in the software and the constraints in the data model nothing can prevent the entry of new data with equally low quality or the degradation of higher quality data.

Fourth, cleanup or restructuring (Sellink et al. [105]) prepares a legacy system for potential reuse. They point out that in order to enable reuse, a certain module or functionality needs to

- provide remotely accessible interfaces,
- insulate the program from its environment,
- isolate individual functions from one another, and
- prevent undesired side-effects.

The amount of actual cleanup that can be done in the legacy system again depends on a number of factors. The quality of the legacy system and its data is the most significant one. If it were good enough, the amount of cleanup necessary would be insignificant, however it would probably mean, that no renovation would be necessary or economically feasible at all. Consequently the availability of resources to execute the cleanup is frequently the limiting factor as low resources for system maintenance are amongst the key drivers for legacy system migration. Last legacy systems by definition significantly resist modification. Any cleanup effort is ultimately a change to the legacy system, which stakeholders might not want or dare to risk.

5.1.2 Elicitation of unused assets

In the previous section we have addressed the possibility and need to perform cleanup in the legacy system before the actual transformation to reduce both the complexity and the necessary effort. However, an even greater potential for reducing the scope of the system renovation effort is the elicitation of unused assets in a legacy system. Already Brodie and Stonebraker state in [32]: "[...] first consider radically reducing the functions and data [...]. Much legacy data and many functions may be entirely unnecessary [...]". The golden rule in this abstraction is the point of time when the elicitation happens. Ideally the unused asset is discovered as such during the early requirement analysis and planning phases of a renovation projects. In the worst case the obsolete feature is identified as such after its migration or even completion of the project resulting in extra effort to remove it from the new code base. For incremental approaches the first question before starting the reengineering effort on any asset should be "Is it still needed?".

Small unused assets like code fragments or procedures can be elicited easily or even automatically as part of compiler optimizations [27]. In today's programming languages integrated development environments will indicate these to the developer or compilers or interpreters will perform dead code elimination as a way to optimize the executions of the program. However, when the granularity of the unused asset grows, it becomes increasingly hard to detect unused artifacts or features

and requires techniques like accessibility analysis or data flow analysis [9]. This could be whole input screens (or significant portions thereof), services or transactions, modules or even applications of a system. In none of these cases the irrelevance of the artifact can be obvious, given the right amount of organizational and functional complexity and size.

The following four strategies can be applied to elicit unused assets. The suitable combination depends on the available resources as well as the type of renovation effort. It is important to note that these strategies can also be applied to identify possible areas for improvement in the legacy system, as discussed in the previous section.

Business analysis based: During the analysis phase the main focus is to identify as many requirements and features out of the legacy system as possible. This rightly includes features that might be unused, as the decision not to include a certain feature in the target system should always be an educated and well documented one. A proven approach to find unused features is to match features with at least one sponsoring user group, key user or other stakeholder. This has two main benefits. One the one hand a defined point of contact is established for any question raised during a later detailed analysis or development of that feature. On the other hand, features that cannot be assigned to any stakeholder might not have enough business value to justify its existence in the renovated system.

Code reading based: During detailed analysis of specific features code readings can be applied to align business expectations with the business rules actually implemented in the legacy system. This allows to identify feature requests as such and to differentiate between them and existing functionality. Also code readings are an absolute necessity for validating assumptions raised out of the following two strategies.

Transaction log / log analysis based: The types of log files available from the legacy system vary widely depending on the technologies used. A host based system might provide logs of all transactions or programs executed while a web based system might provide access logs for its services or user interfaces and a two-tiered application might have a comprehensive database log. These logs are frequently used to analyse the load of the existing system and accordingly estimate the expected load of the new system. However, another type of information is easily overlooked. The fact that a certain transaction, user interface or service is never mentioned in these logs is an excellent indicator for rarely or never used functionality. The confidence in this type of analysis is highly dependent on the time frame that is provided for analysis. While a couple of weeks of log files will easily miss quarterly or annually needed functionality, longer periods including sensitive dates (beginning of a quarter, beginning of a year/fiscal year) can greatly boost the reliability of the extracted information. For the airport case study “Usage statistics of the last 3 months were used to support the code reading by identifying sections that are never called” [19]. It is therefore recommended to start log analysis as soon as possible and keep performing it until the legacy system is actually shut off.

Database analysis based: The database provides another source of ground truth to the reengineering team. It shows which data is actually accumulated and in which combination. It gives excellent indicators to scrutinize the requirements as communicated by the business and reminding them of special cases that might not yet be covered in the specification. Database logs can provide information about which tables are actually accessed whereas the data inside the tables can provide further indicators as to the usage patterns of individual columns. For example empty tables are a good indicator of an unused feature (or temporary processing space), empty or constant value columns on the other hand are of similar value as they indicate a design for cases that might never occur in real operations [116].

5.1.3 The value of automated code transformation

In 2010 Terekhov et al. [122] observed in "The realities of language conversions" that managers sitting on a large pile of legacy code are susceptible to "silver bullet" [45] solutions recommending the easy and quick automated translation of legacy code, however thin the proof for actual existence and sustainability [109] of such a solution might be. He called this "name magic" and concludes with "Easy conversion is an oxymoron", seeing "much work must be done before language converters give satisfactory results on real-world code."

A decade and a half later this still seems to be fully applicable and is consequently recommended reading for anybody considering automated code translation. Most case studies of this thesis have touched the topic of automated transformation, including the quest for a silver bullet on the one side and the "technological quackery of language conversion vendors" [122] on the other. As Seacord states in [104]: "Automatic translation [...] cannot significantly change the structure of the code. For example, automatically translating COBOL to Java often results in code that looks like COBOL written in the Java programming language". An observation very much consistent with ours. This does not mean that (semi-)automated, tool supported approaches should be discarded without proper consideration, but is a plea for careful assessment of realistically possible benefits versus inherent risks and side effects.

The first example is from the airport case study. As already indicated in section 4.3, the customer had previously tried (and failed) to migrate away from the mainframe several times. One of the attempts involved the automated source code transformation from COBOL to Java. The details of this approach have been published by Sneed with the following concluding remarks: "Whether the converted JAVA code can be readily maintained remains to be seen." ... "Ensuring functional equivalence of the newly generated JAVA components with the original COBOL programs remains as the greatest barrier to making an economical and quick migration. Overcoming this barrier remains as a challenge to the software reengineering community" [108]. In a later publication the same author acknowledges the necessity for a manual reimplementation [111]. Having completed the system renovation successfully as described in the case study, to the author's knowledge not a single line of the automatically transformed code has ever been run in production. Neither has there been any significant use during the work on the case study (e.g. for understanding the legacy system). The primary reason for this situation was the fact that the converted part of the legacy system was not useful as a standalone component. To bring it into production would have required significant efforts to integrate it both with the remaining legacy system and the other currently running software. Not less significant was the fact that only a small portion of the translated code ("all 104 Java packages could be compiled and four were functionally tested"[108]) was properly tested leaving a high level of uncertainty for possible later users of the code. Effectively this means that the shipped code only met the "quality attribute" of "it compiles!". Even less useful was the code for understanding the legacy system. Code reading was an integral part of the successful migration. However, the translation had multiplied the size of the code base by a factor of three with no real enhancements concerning the readability compared to the original. In fact an early assessment concluded that COBOL skills were necessary to properly understand either of the code bases. It was in reality easier to read and especially understand the currently running legacy COBOL code. This had the additional benefit of covering all changes performed since the transformation.

The second example was the case at the ministry (section 4.2) in the late stages of the planning and bidding phase of the original tender. As previously highlighted test runs were done using a single program with around 4 KLOC of source code. The main findings after a quick (and not optimized) translation were sobering for all sides. In total two departments of the organization as well as two independent, external experts were asked to review the resulting code, which was not

executable due to a number of gaps in the current state of the transformer used. This stopped any attempts to perform functional testing. Apart from this most prominently the size of the code base increased tenfold. Other problem areas included missing capabilities of the transformer for widely used program constructs (as the code was based on an older standard), brute force dealing with go-to statements, incomprehensibly long single lines of code (up to more than 7000 characters) and no strategy for dealing with assembler subroutines. It was acknowledged from all sides that the prototype solution would have significant room for improvement and the output is as a result not completely suitable for judging the approach of automated code transformation as a whole. Two key findings were crucial at this stage. The people doing the transformation were openly surprised by the negative effects of the source code quality on the transformed code. Bad quality input does not only result in bad quality output, but in significantly worse quality output. Resulting from this is the finding that in order to have a chance of achieving a feasible transformation, significant investments into the source quality have to be made (similar to the arguments brought forward in section 5.1.1). Even then, the resulting code would still be hardly comprehensible to a programmer only proficient in the target language.

The third example involves what from some points of view might actually be seen as a success and involves the insurance case study (Section 4.1). The system in question was used to perform benefit calculations, a host based system running COBOL and accessed via a PowerBuilder front end. The transformation as such was done by a third party and few technical details were revealed, so the analysis focuses on the status quo several years after the migration. To mention the positive aspects: The back end system was successfully migrated from COBOL to Java and the calls to the mainframe transactions changed to web service calls. A comprehensive automated test suite ensured functional equivalence and provided safety against regressions for ongoing maintenance work. Corrective maintenance and minor changes were performed by a small team of developers. Nonetheless the legacy of the Cobol days loomed over every single aspect of the system. The most significant aspects thereof were:

- A full framework was needed for the code to run and behave similarly to the legacy COBOL code. This framework code was closed to the customer and only released at the time when the decision was made to no longer support the framework
- The framework code was tightly coupled with the application server and version in use. There was no upgrade path to the newer version of the same application server or to other application servers.
- The framework code modified system internals of the application server like the behaviour on certain transaction failures of the underlying database system (Therefore also coupling it with a single database vendor).
- To simulate COBOL-like behaviour most standard Java functionality could not be used. Instead, framework methods even for simplest tasks like checking for equality were provided. (Example: `var.xpl_equals(var.getPbParameter().getIdVfa(), "G")`). The same was true for using basic data types, requiring the use of framework data types as `Text10` instead of `java.util.String`.
- The transformed services, albeit syntactic Java, were extremely complex to read and interpret. The transformation had followed an approach translating each COBOL transaction into a single Java service (plus data structure objects). This resulted in single services having more than 10k LOC.
- Although technically web services, the data was transported using a proprietary, binary format which required additional build steps to generate appropriate Java bindings. This

required an additional marshal step as part of every service, but was necessary to minimize the impact on the existing front end code.

- Every service/transaction was equipped with an equivalent variables object holding all variable state needed or modified by the service. Besides it was used to hold (temporary) execution and error state. This data holder typically contained hundreds of values and was passed through all methods and sub-method calls and required extensive mapping for service to service calls.
- The same data structures were exposed for integration with other systems. This had the effect that hundreds of variables were available to the developers with no indication as to which were required to properly call the legacy system.
- For functional and performance traceability a huge amount of logging and tracing code was inserted. This cumulated to an estimated 40% of the code base.

To summarize, yes, the system was running, but no, the system was not anywhere near the characteristics of a modern Java based application. This had severe implications for maintenance and the affected developers. The findings are in line with recent case studies like [84] stating the need for a deep understanding of the COBOL language and mainframe concepts as well as difficulties in recruiting Java developers to maintain automatically translated source code as primary obstacles towards successful long term maintenance. In addition, to the domain knowledge required to understand the system extensive specialized technical capabilities had to be acquired to be able to perform even the simplest tasks. In reality this meant that only the remaining developers of the legacy system were actually able to understand the system, as now proficiency in both the source and target language were required to perform maintenance work. In effect nobody dared to perform major changes to the system at all resulting in the transformed system still exhibiting all the major legacy system characteristics [32], except formally the modern programming language. How can it still be considered a success? From a financial and management perspective, some of the key characteristics did improve significantly:

- No mainframe was necessary to run the information system translating into significantly lower infrastructure costs.
- The cost of operation was significantly reduced due to the use of an open source, Linux based operating system and application server.
- The availability of developers was greatly improved and the cost of hiring reduced accordingly.

What all three examples from the different case studies show is the chances and risks of a source code transformation. In the end what it comes down to is a fairly simple statement. The primary benefits of transforming the source code are a relatively fast change of platforms and possibly homogenization of an enterprise IT landscape. If hardware and licensing costs are the primary concern, a change of platform without source code transformation should be considered first. A transformation to a common programming language like Java and C# might bring short term benefits of being able to run in the same environment as new applications, but will, in the end, increase the total cost of truly renovating the legacy system. Neither of the steps will bring an improvement of the negative legacy system characteristics of any information system.

5.1.4 Always have a leading system

One of the most important assets of any legacy system is the data managed and contained by it. Brodie and Stonebraker note "the fundamental value of a legacy IS is buried in its data" [32]. The legacy system might be envisioned as a dragon furiously guarding its most precious treasure. Some skeletons in front might be the remains of past modernization or migration attempts. The ownership of the data is made very clear. Any legacy system migration effort will put the data ownership in question sooner or later. Apart from the physical data migration that has to be performed, the transfer of ownership is a crucial step. Incremental migration further complicates this step [6].

Data ownership can either be assigned globally for a whole database (or set of databases). This is sometimes the only option for non-decomposable legacy systems. Or it can be assigned on a more granular component level. The important part is that data ownership for every element is always clearly defined. This is even more important, when (interim) copies of the data are created during the migration process. Data ownership does not have to be necessarily assigned to either the new or legacy system but might also go to a dedicated (database) gateway as defined by Brodie and Stonebraker [32]. This is necessary to ensure consistency over the data during the whole migration process.

The airport case (Section 4.3) is a prime example of a non-decomposable database asset. As described in [19], the switch of data ownership from the legacy to the new system was one of the key turning points in this effort. The fundamental behaviour of the target system changed at this point from first executing the transaction in the legacy system, then, if it succeeds, executing it in the target system to the exact opposite. In layman's terms this means that after the switch the rule "the legacy system is always right" changes to "the target system is always right".

What happens if neither the one nor the other rule can be applied to a dataset? The insurance case (Section 4.1) provides a suitable experience. Due to a number of reasons it was discovered during the migration process that a 3-way synchronization of master data would be necessary for a limited, yet extended time period. Two of the three source systems would at this point still be editing the same master data. The resulting gateway module was highly complex to develop and run. In addition, due to the nature of consistency problems caused by concurrency issues it was not able to guarantee consistency in all scenarios. Instead it was limited to detecting these issues and notifying users about them. These issues then had to be resolved manually by merging the two (or more) conflicting updates. A laborious task frequently involving the editing users to elaborate the final correct state. In the end, a whole team of engineers was busy in fixing errors and covering edge case after edge case for more than a year. Only when the situation was resolved by consolidating the writes in the target system, consequently again assigning clear data ownership, the maintenance and support effort was reduced to acceptable levels. The complexity of the gateway was then significantly reduced as its remaining task was just the propagation of changes from the target to the legacy systems. In retrospect, the only fact that allowed this approach (and maybe through this, the project as a whole) to survive was the fact that updates to master data are relatively infrequent (a couple of thousand per business day) and due to the nature of the legacy and target systems composition, conflicts rather unlikely. In any scenario where updates were more frequent or the chances of conflict higher this approach would have not been feasible to maintain over longer periods of time.

As Brodie and Stonebraker note "Update consistency across heterogeneous information systems is [...] problem with no general solution yet devised [...]" [32]. Alternative approaches and strategies are therefore needed. When planning the incremental steps of a legacy system migration, each step needs to be associated with the transfer of data ownership. Only when data ownership is clearly defined during all steps of the process, consistency can be ensured.

5.2 Architectural & IT Strategy Experiences & Observations

5.2.1 IT Strategy and Enterprise Architecture should be in place before deciding on the target architecture

This seemingly obvious statement turns out to be one of the most crucial deal breakers (or makers) in software renovation projects. Out of the case studies presented in this thesis only one had what could be called an established enterprise architecture and IT strategy, the airport (section 4.3). Both the insurance (section 4.1) and ministry (section 4.2) did have a commitment towards a Service Oriented Architecture (SOA) and Java as the primary environment of choice, but neither a clear view what this actually meant for the organization nor how to get there. The university 4.4 was one step further behind, not even sure about the technology and architecture of choice.

To start with the positive example: At the airport, the IT department was able to deliver small to medium software and integration projects on time in the target architecture. This included both individual application development as well as integration of COTS product. A driving factor behind this success was the established development process, as discussed in the previous section (5.3.2). The overall IT strategy was clear as well. There was a clear commitment for the principal technology stack of a lightweight application server in combination with open source development tools and frameworks for individual software. The use of COTS products in suitable areas was defined and accepted, as were the points and modes of integration. For the host migration project this meant a couple of key benefits: clearly defined contacts and boundaries, clearly defined processes for pushing the boundaries, an established functional infrastructure and a common language and understanding to communicate with other (customer) teams.

The weak link in this case was the enterprise architect, as detailed in section 5.3.1. From the target architecture perspective this allowed the individual teams to deviate from the standard further than actually necessary. As an example, it enabled a project to introduce a completely new user interface paradigm without any prior strategic consideration. In addition, it facilitated the fragmentation of responsibilities that should have been consolidated in one team or function. All in all this put the burden of maintaining a clean architecture in accordance with the defined strategic goals on the individual teams, a situation clearly favouring strong teams and individuals in key positions.

On the other side, the ministry and the insurance both took a similar approach. After decades of developing applications for host based systems they decided to go for a SOA based architecture (more on that in the following section 5.2.2). This was followed by a big tender to buy both suitable hardware and software products. Only then they were able to start developing software organized in large five plus year programs to modernize their IT landscape. At the height of each, these programs involved several hundred people working on a multitude of artifacts and functional aspects. However, neither of them actually had a proven architecture but merely a rough vision - in many places highly influenced by the promises of the sales departments of the big vendors fighting for the large contract. In the end this resulted in scaling out to large teams including tons of guiding paper work without even a small vertical integration of the new software design. The results reflected this situation, although the manifestations differed significantly.

While the outcome of the insurance case study will be reflected in some more detail in the following section (5.2.3), this section will focus on the (early) outcome of the ministry case study: One of the significant problems with the host based legacy systems at the ministry was the fact that similar (or identical) functionality was built by each core application. This resulted in different design decisions and usability between those applications. In addition, master data was held and managed by each application individually, resulting in redundancies, stale data and therefore a

mediocre customer experience. For example a name change after marriage might be reflected in one system administrating child care benefits but not in another for tax returns resulting in letters addressing the same person with different names from the same ministry.

This problem was to be addressed by developing standardized core components that provide common functionalities as well as a centralized master data management. This happened before any of the core business processes were brought to the new platform resulting in two major disadvantages. First there was little outside visibility for the amount of effort that was put into developing these components, leading into reduced end user interest and involvement and a catastrophic cost to (visible) benefit ratio for management. Second lacking a core business process to benchmark and validate the functionalities of the individual components the requirements were frequently based more on an abstract business analyst vision instead of real world end user needs. This combination lead to over-engineered components with a catastrophic end user experience and non-functional characteristics, resulting in a highly negative perception of the project output paired with too high delivery costs.

In a similar way the university case (section 4.4) took an interesting turn during project setup. Due to previous experience from an order of magnitudes smaller but successful project a technology decision was made towards a technologically popular and to some degree hyped at the time but overall still niche solution. The technology as such would not have been a problem. It had been proven (by others) to be scalable to the necessary extent. However, the technology imposed a series of threats to the project that were not considered at the time of decision. First of all, apart from the small group working on the previous project nobody at the organization had any working knowledge of the technology. Second the market of potential employees experienced in this technology and willing to work for a large governmental organization was diminishing, maybe at least also due to the agile and rapid prototyping promises of the technology not matching with the corporate culture. Third, while doable, scaling the technology in terms of modularization and developer comfort was a challenge which would have required significant upfront investment. In all the decision neither aligned with past nor with a current vision for a organization wide IT strategy.

What these examples show is the massive underestimation of the value and necessity of a solid enterprise architecture and IT strategy before diving into the renovation of mission critical in-house applications. While the lack thereof has not lead to any outright project cancellations in our case studies the impact was still massive and all of these projects were brought to the brink at one point or another. Even if it is not feasible or serious to try to calculate the exact monetary impact, the project delays caused by misguided architectural and design decisions due to the lack of an established strategy gradually accumulated up to years in each of the examples with corresponding costs.

A final word of caution: this abstraction does not recommend an approach where a target architecture is elaborated in detail on paper in isolation. However, in case the target architecture is not already proven (in terms of successfully applied in at least one sufficiently sized project) a clear declaration of a reference project with that responsibility is proposed. This has to include the necessary powers and room to build the foundations for a target architecture while delivering end user benefit at the same time. As the topic of the subsequent section will further elaborate this must include room for error as well as experiments.

However, when it comes to the enterprise architecture and high level IT strategy it is an absolute must to have the cornerstones in place. Otherwise the renovation project will lack the high level vision to find its place in the IT environment of an organization.

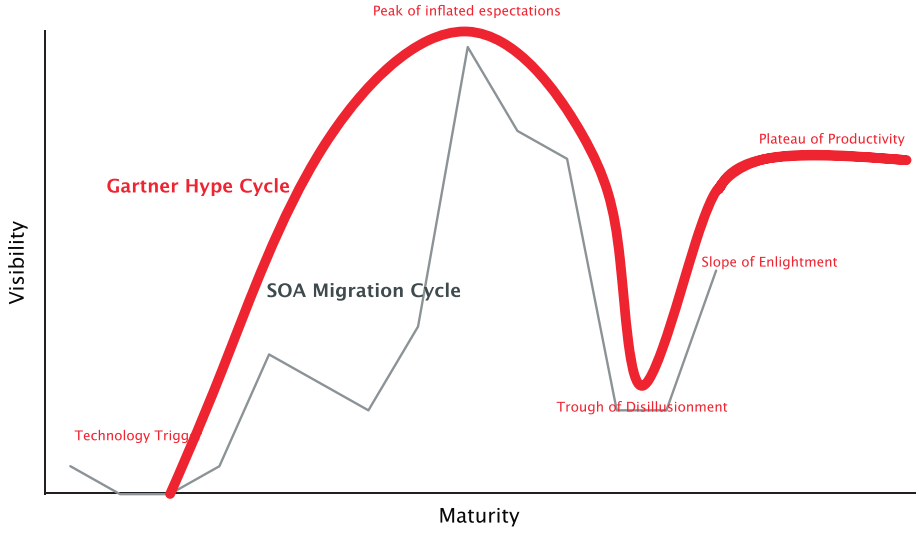

Figure 5.2: The SOA Migration Cycle overlayed by the Gartner Hype Cycle [99]

5.2.2 Looking behind the SOA and Microservice Myths

Two large scale trends can be easily identified when it comes to architectural styles of the last two decades. Service Oriented Architectures (SOA) dominated the 2000s and Microservices Architectures have been rapidly gaining ground. In terms of the Gartner Hype Cycle [56] SOA might be on the upper part of the "Slope to Enlightenment" whereas Microservices are near the "Peak of Inflated Expectations". Both were respectively heavily marketed with similar slogans and myths as the solution to almost all of software engineering's worries. If your existing code base is highly tangled and difficult to maintain, here is your solution. Quality problems, security issues will be addressed as well. The same goes for performance and scalability issues. Time to market and customer satisfaction targets can be met as well.

The relevance of this aspect is through the dominance of these topics in the space of (legacy system) migration. The topic changes from "migrating a legacy system" to "migration towards SOA or microservices" respectively. This is nicely illustrated by recent publications. In 2013 Lewis and Smith published a book collecting articles on the topic of "Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments" [82]. In 2015 Razavian and Lago [99] created a systematic literature review, illustrating the parallels between the Gartner Hype Cycle for SOA and the publication of papers on the topic of "SOA migration" (see Figure 5.2). More recently the focus has shifted to migration towards Microservices [35] or "the Cloud" [8, 70] or a combination of both [7]. This culminates in the topic of migrating from one hype to the next, as a systematic mapping study by Botto-Tobar et al. [30] shows. In combination all of this literature can be seen as an indicator how current architectural industry hypes can influence if not hijack the currently ongoing scientific discussion. This leaves little room and resources for other issues.

Out of the case studies the insurance case (see Section 4.1) provides the strongest example. As highlighted the initial project tender requested a full stack SOA landscape. At that time, neither a general migration plan, nor one towards SOA was in place. In fact the project was setup as a regular greenfield software development effort. A large single team was in charge of the whole application, from application server, over rule server and process server to front end and database. As predicted by Lehmans Laws [81] the result of the initial release was a distributed monolith. Logically as well as concerning runtime dependencies the application was structured and built like a monolith and as a result behaved as one. Components were not able to run, if some other component was not

available. All components had to be deployed simultaneously for every release. In addition, the solution was plagued by problems rooted in the distributed nature of its components. Performance was barely acceptable with end user requests taking multiple seconds. Consistency was a constant issue as failed transactions were not compensated properly across components. Traceability in case of errors was severely hindered due to the lack of suitable tokens as well as log aggregation tools. In the end the project was turned around by following a monolith first approach as proposed by Fowler [50]. Some runtime components were replaced completely, others were integrated more tightly, others again were broken out completely to be handled by separate teams. In addition, where distribution was necessary, robustness was gradually restored and traceability enabled. In the end the resulting architecture is nowhere near what SOA would have proposed.

This example shows how a hype around an architectural strategy paired with inexperience and management belief that technology and tools can by themselves solve the hard problems of previous experiences on the one hand and aggressive marketing on the other will result in serious trouble for most projects. Several years of project time on the brink of failure have then resulted in a "new system" that itself is closer to being a candidate for renovation than a basis for continuous development and change for the upcoming decades as previously envisioned.

As already discussed in Section 5.3.3 there is a mismatch between architect's and developer's perception and an organisation's or corporation's actual need. The hypes, especially around micro services and cloud, originate from the performance and requirements of today's internet giants. The maybe inconvenient fact is, that neither an insurance (Section 4.1) nor a government ministry (Section 4.2) nor an airport (Section 4.3) nor a university (Section 4.4) have the same needs as Alibaba, Google or Netflix do.

5.2.3 Build one to throw it away

"Where a new system concept or new technology is used, one has to build a system to throw away, for even the best planning is not so omniscient as to get it right the first time. Hence plan to throw one away; you will, anyhow." [33, 41].

As the discussion of Brooks' original statement in the 20th anniversary edition [33] shows, the statement should not be taken literally (any more). However, substituting "throw" with "refactor" and leaving out how much refactoring is allowed to still talk about the same code base, the quintessential message is still very much applicable: Expect not to get it (completely) right the first time. Significant software renovation projects lead to new architectural frontiers for the organization replacing one of its core IT systems.

In our case studies, we have seen two major incarnations of this phenomenon:

The airport case study (section 4.3) features a so called "prototype feature" in a plan to do an early vertical integration. This step was essential for the project team to understand the depth and complexity of a seemingly simple feature. With the face value of storing a timestamp and a string in the database - basic CRUD functionality - underestimating the necessary effort is easy. Under the hood the amount of validation and dependent functionality necessary to complete the feature became slowly apparent. Without going into too much functional detail discovering the amount of validations and dependent calculations hiding behind a simple transaction was like peeling layer after layer from an onion slowly revealing a highly cohesive core of crucial functionalities.

Leaving aside the equally important realizations this approach revealed for the business analysis perspective (an aspect already covered in detail in [19]), it left a series of lessons for the technical and planning perspectives. The most influential of them was the fact that the existing flight core, designed as a state machine and persisting only a series of state changes, was not suitable to handle

dependent calculations. Persisting the current state and completely refactoring the core towards a set of business services was the logical consequence. This also signifies (although its development lies outside the project scope) the much bigger iteration of "build one to throw it away". Had the previous project not tried the concept, a state machine might have seemed like a logical choice at the beginning of the project. In addition, the prototype revealed a couple of design issues in the database as well as the target platform that could later be addressed during the development.

The insurance case study (section 4.1) features the second major refactoring and an especially steep learning curve for everybody involved. Again little was actually thrown away, as functionally the first release was essentially on target. However, as already described in the previous section (5.2.1), the target architecture was only a vision at the time when development actually started. With a completely new tool and system landscape a series of questionable (although understandable) decisions led to catastrophic non-functional characteristics in the system. Both stability and performance were below par and retrospectively seen unacceptable, as was the speed of delivery.

However, the experience from this prototype (which found its way into production) led to an almost three year long technological consolidation process evolving the target architecture into a pragmatic but at the same time leading edge approach. During this time the user interface technology, the process engine [115] and the identity management system were replaced, the business rule engine was eliminated completely as was one of the two application server technologies. The technological transformation was accompanied with corresponding organizational changes to adapt the processes accordingly.

Extreme Programming [10], an agile development practice has a fitting term for formalizing what might seem like "trial and error" to a bystander: Spike [38]. In software renovation projects developers and architects can be confronted with problems that do not have an obvious (working) solution. By "spiking" the solution a minimal prototype is built to answer a specific question. It might be as easy as "Can we transform the data in table X to format Y?" or as complex as "Does the page scale to 10000 concurrent users?" The question "Can we build our system with the current target architecture?" should be asked (and answered!) at the beginning of any renovation project. Answering it highly depends on the maturity of the target stack and the organization's level of experience with it. However, the key point is being made by the following series of schematic graphs (see Figure 5.3) on the necessity and feasibility of upfront design.

For low complexity projects any previous design will do. For medium complexity projects a suitable design can be found with some design thought or based on previous experience. For sufficiently complex projects experimentation is necessary and therefore some room is required to allow for backtracking and learning on the go. Large scale legacy system migration efforts tend to fall in the third category, as obstacles tend to come up in unexpected areas. As Bach et al. describe in a case study on a large scale host to open systems migration: the problems were not, as initially expected, in the area of source code transformation, but in the added complexity due to the use of bridging mechanisms to support parallel operations of legacy and target platforms as well as the necessary rapid introduction of an unusually wide array of new technologies [6].

5.2.4 Towards a repeatable approach for understanding a legacy system

Reverse Engineering is a key task in most legacy system migration projects. Either for the purpose of re-documentation or design recovery or both [37]. However, this is not the first step or phase in the inception of a legacy system migration project. A highly unstructured process that happens before or during the inception phase of a project is a general learning phase about the system in question, its environment and also its history. This process might happen implicitly or spread out

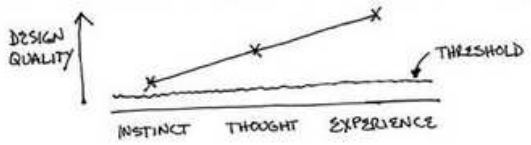

FIGURE 18. Any old design will do

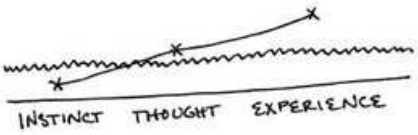

FIGURE 19. Some design thought or experience is necessary

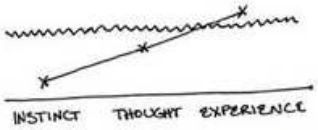

FIGURE 20. No amount of pure thought will suffice

Figure 5.3: Different projects require different levels of upfront design [10]

over a longer period of time if all stakeholders are already familiarized with a system. It might be more explicit on the other hand when (a team of) outsiders are brought in.

Seacord describes as part of his risk based modernization approach [104] some initial steps as "Identify Stakeholders" and "Understand requirements", before moving on to creating an actual business case for a proposed migration effort. While these steps are certainly part of a phase there is a softer aspect to this as well. This might be termed mapping the story of a system. Experienced architects and project managers will perform this step in a very early phase, frequently by talking to available stakeholders and contrasting with past experience. However, by leaving this phase in a very informal and unstructured context, as it is currently perceived, the risk of bias is high. On the one side bias is introduced by the person(s) forming the mental model of a system's story. This bias is mostly based on existing areas of expertise and past learnings. On the other side bias is also introduced by stakeholders being interviewed. A newly introduced project lead for a migration project who has just gone through this phase himself might have a very different perception of the system than a long time maintainer. However, this mental model which in this stage will consist largely of assumptions based on previous experiences and analogies as well as anecdotal evidence can have great influence on the subsequent steps of project inception and setup.

The airport case (Section 4.3) can serve as an example of how assumptions can have great effect on later developments. One of the key architectural influences was the anecdotal fact that messages concerning the flight might come in out of order, e.g. the information about an aircraft's take off would be received before the message about it leaving its parking position. This would severely inhibit message processing as it would either require deferred processing or temporary status transitions. Under this premise a highly complex information processing architecture featuring a state machine capable of back tracking was designed and partially completed at an early stage. However, as the final migration project progressed, data analysis showed that out of order messaging is

at best a one off, extremely rare event. As with other error scenarios (e.g. the loss of a message) it was therefore covered in the final architecture with a manual compensation mechanism. In the process the architecture and design of the message processing was greatly simplified.

The following collection of artifacts might serve as an initial guideline as to which information can be retrieved in the early stages and what the main motivation is behind it. While certainly not exhaustive it should serve as a starting point for tailoring to a specific setting/system.

- Stakeholders: Users, Management, Developers & Maintainers, QA, Architects, Operations
- System history and key milestones: the initial genesis of the system, major enhancements or modifications, past modernization efforts
- Current architecture: operations perspective, integration perspective, component/modularization, user perspective, staging model
- Current processes: maintenance or development, change requests, quality assurance, releases and deployment, operations, support, training
- Supportive Infrastructure: requirements management, source code management, change and defect tracking (ticketing), test cases, build and deployment

Identifying all relevant stakeholders and their major interests is the first step towards obtaining the full picture. It defines the group of people that need to be interviewed. It is important to note that in some instances past stakeholders will be a valuable (if available) source of information. For example the project managers of the last round of modification might not be an active stakeholder in the system any more, but might be able to provide some insights about the experience of dealing with the system. The perspective of someone who is no longer directly involved with the system on a day to day basis might also help to eliminate a bias introduced by recent events or current pain points.

The system history is the time line to the story behind a legacy system. The goal is less completeness or a high degree of detail, but more a focus on the pivotal events during the life span of the system in question. It should answer questions like where the fault lines first erupted or why an architectural trait originated. As an example the migration of the database away from a legacy, file based system to a relational database should feature prominently as should the initial web enabling of a legacy IS. A minor feature release to ensure regulatory compliance on the other hand might not be worth mentioning, as long as it was not disruptive to the nature of the system.

Obtaining the different views of the current architecture is an aspect that might be already an early stage of a later reverse engineering redocumentation effort. It should not replace this effort but obtain what is currently available. The fact that for example no consolidated information regarding the integration architecture is available, is an important piece of the whole story in itself. The result might accordingly be more of a roadmap of future reverse engineering tasks than a detailed picture.

Mapping the stakeholders to current processes is the next step towards understanding the inner workings of the legacy systems ecosystem. As with architecture, complete re-documentation is not the goal, an understanding of where gaps might be and where working procedures are established is. The meta information about how the organization perceives the currently established processes is equally important. The findings will be important when deciding whether a new development methodology needs to be established or if it is feasible to build on existing practices to minimize disruption.

Information about supportive infrastructure is key when it comes to technical setup of the project and team. As with established processes the availability of modern development infrastructure or lack thereof will be a good indicator as to how disruptive the legacy system migration will be on the IT organization. In addition, it creates a map of where to find the existing legacy artifacts. It therefore defines the starting points for subsequent reverse engineering efforts. Most importantly it should answer the question where the source code of the system can be found.

Collecting the information and packaging it in a short biographic story of the legacy system to be addressed in an upcoming migration effort will ensure that everybody has a minimal set of common knowledge about this system. It allows people coming on board to better relate to the system and existing stakeholder groups. It allows these stakeholders to convey their story. It establishes a common base line and as a result counters misconceptions and misunderstandings.

5.3 Organizational Experiences & Observations

5.3.1 Enable architectural decision making

As already indicated in section 2.1.11 the Enterprise Architect is a vital role in any organization, if properly staffed and positioned. However, throughout the case studies in this thesis there was a consistent lack, both in management's understanding of the role and the current incumbents. That is, if the role was filled at all or even existed within the formal structure of an organization. The enterprise architects who were encountered uniformly seemed to be near perfect incarnations of Laurence J. Peters' "Peter Principle" [93], stating that people tend to rise to their "level of incompetence" in the hierarchy of an organization.

The first example to demonstrate the effects of a weakly positioned enterprise architect is the airport case study. In this case the position formally existed and was defined relatively closely to the definition used in this thesis. However, two major drawbacks prevented the position to be effective. First of all in the organizational structure the position was based in the department responsible for the development of individual software. This effectively meant that the enterprise architect was positioned at least three levels below the CIO, preventing adequate powers, responsibilities and consequently wages. Furthermore it reduced the de facto influence to the one department and shielded all other departments (COTS, ERP and operations) from any possible influence. In reality the enterprise architect was limited to the fields of individual software development and some system integration agendas. The second equally major drawback was the fact that the position was filled with the most senior (in terms of age) employee of said department and as a result is the point when it is necessary to come back to the Peter Principle. The effective output of the position was therefore limited to vague guidelines with enough room for interpretation to avoid any form of confrontation. As an example, the guideline for communication between systems could be reduced to the following sentence: "The synchronous communication between systems has to happen based on SOAP web services, except when another technology is more appropriate."

This problematic situation was temporarily mitigated by the program and project managers tasked with the transition of the airport IT infrastructure in anticipation of the new terminal opening. Especially the program manager played a vital role as the first level of escalation between the ongoing projects. However, while this approach was successful in the short to medium term because the program was of the highest strategic importance at that time, the long term effects can only slowly be assessed. From the author's point of view several key areas were neglected: Most prominently the integration architecture between the core airport system and a series of new COTS products necessary for the new terminal was done on an ad hoc basis, leading to a series of point to point connections and dependencies. Technology decisions and necessary upgrades were

either postponed or left to the individual projects leading to unnecessarily heterogeneous system designs and solutions. In addition, several projects of strategic, but not operational importance were chronically sidelined diminishing their business value. For example the introduction of a standardized identity management solution was repeatedly delayed stopping a long planned single-sign on solution from being implemented.

The second example, based on our insurance case study, shows a different scenario with similar effects. Unlike the first example, the role of an enterprise architect formally only existed within the described project, but not on an organizational level. The role was thus effectively limited to an application or system architect, without any influence or even insight over operations and the maintenance of existing applications. Again the staffing of the position was another problematic aspect, in this case primarily caused by the organization's policy to fill core positions only with in-house applicants combined with the fact that no technologically and structurally similar project had ever been done in this organization.

In combination with the lack of a proper IT Strategy (see section 2.1.10) this situation created a de facto vacuum between the director of IT, the project managers and the application managers. Without a centrally positioned body to establish clear rules and boundaries for the individual applications, the power of the respective managers over their domain was de facto unlimited. This seems to be one of the core motivations to avoid relinquishing control. The medium to long term negative effects of this arrangement however are one of the core disruptive factors in this organization. Basically all applications act and operate as if in complete isolation. Any point of integration is accordingly reduced to a compromise influenced mainly by the power structures and not by rational and technical considerations and as a result almost by definition not conforming to any best practices that might exist. This is only topped by the complete lack of effective guidelines from service management and availability to non-functional and security considerations. Due to the lack of a guiding framework the resulting IT landscape of the organization is highly heterogeneous, complex to understand and maintain and thus equipped with meager functional and non-functional attributes, leading to an unavoidable high cost to benefit rating.

The third example is based on the university case. As in the insurance case, the architect was positioned within the team with no corresponding role on an organizational level. Apart from the role of architect he was also the development team lead. A very capable developer the situation can best be described as him falling victim to the "Second System Effect" [33, 42], a scenario that could and should have been avoided with a more senior and experience technical role overseeing the project and factoring in organization-wide and long-term strategic requirements.

What these two examples show and the abstraction is trying to illustrate is the medium to long term effect that ad hoc, unstructured and unguided decision making has on an enterprise architecture level. While the gap is frequently filled by other roles on a case by case basis, these roles in reality represent the interests of their project, program or application inevitably leading to a weakening of the strategic orientation of the enterprise.

5.3.2 Established software development process

The time up to around the mid 90s was mostly dominated by sequential software development approaches, most prominently featuring the waterfall model [75]. By the end of the 90s and early 2000s first iterative approaches like the Rational Unified Process (RUP) were adopted [77]. In 2001 Kent Beck et al. published the "Agile Manifesto" [11] establishing the foundations of today's agile practices. Most popular among them is Scrum [126]. Another popular contender is Kanban as originally defined by David J. Anderson [3] tracing back to efforts of Poppendieck and Poppendieck [94] with ideas borrowed from Lean Manufacturing practices adapted to the field of

software engineering. When reading research about the state of agile practices, this is the picture being commonly painted.

However, with respect to the case studies and their respective sectors and industries we can observe a different viewpoint. While a widespread transition to agile and lean practices can be observed across the industry, out of the four different cases only one (AODB, Section 4.3) had made significant efforts and progress towards adapting agile practices. The other three would be best described between waterfall and ad hoc software development processes. This is not necessarily saying that software development is happening in an unstructured way, but can be better characterised as the corporate culture being reflected - in essence, once again, an application of Lehman's Law [81].

A typical "development process" for the legacy system would look similar to the following: Somewhere in the business organization someone writes down a new or changed requirement. This is then propagated upwards to management and at some level handed over to IT where it is again handed down until it lands on some developer's desk. In some organizations this official path is mirrored by an in-official shadow path of direct communication between the customer and the developer. In some organizations the inofficial path is the only one. This may result in scenarios where realizing changes to the legacy system only actually works if the right customer talks to the right developer. However, the involved stakeholders (excluding management) actually perceive this approach as the "way things work best". Little to no overhead in communication and potentially very rapid time to market. These are similar characteristics as agile and lean practices have been claiming. So from the perspective of a legacy system developer introducing agile practices is adding overhead without promising much improvement.

The point here is not about whether or not introducing lean or agile or any other for that matter software development methodology. A whole area of research is dedicated to finding suitable processes and tailoring them to the individual needs of a project or organization. It is up to the individual organization to decide on and introduce a suitable development methodology. Two observations that need to be highlighted in the context of large scale legacy system migration are:

1. Legacy system migration efforts have unique requirements when it comes to software development methodology.
2. Attempting technological and organisational transformation in the same step puts massive stress on all stakeholders.

The first item is an area where the author sees an urgent need for further research. While there is plenty of research for methods and approaches on how to plan and execute a legacy system migration, the aspect of actually developing software in these scenarios is treated more like an afterthought. Storytest-Driven Migration (STDM) as proposed by Abbattista [1] might count as an exception.

The second item is a key obstacle in the author's perception. Especially for legacy system developers and their customers, but in general also for the organization as a whole, transformation to modern software development methodologies is a major undertaking. In some aspects it is quite similar in size and risks to a legacy system migration. It could consequently be named a legacy process migration and should definitely be a project on its own. The key point here being, whether the decision is to maintain, formalize and maybe optimize the established development process or to introduce a completely novel approach, the actual implementation should not happen in parallel to a system migration effort.

5.3.3 Learn legacy programming languages (again)

In the UK and Australasia, the programming language Java is, by far, the most common programming language taught in computer science (CS) courses, with Pyhton and the C family close contenders [107], although there are significant regional differences. No such research could be found for Austria, the country of the case studies in this theses. The perceived prevalence of Java in the author's home region is even greater. However, this view could be biased by a strong focus on Java at Vienna University of Technology.

Out of the organizations and projects described in the case studies, only one (airport, Section 4.3, Java) had previously established teams of programmers and internal knowhow in any of these top contender languages being taught at universities. The languages of choice for the business critical systems of these organisations: COBOL and PL/I. In fact, worldwide, COBOL still dominates in terms of the business that it runs. At an estimated 220 billion lines of code programs written in COBOL are still responsible for 95% of all ATM transactions [64], countless retirement plans and a large percentage of all government transactions [83]. Most of it running on traditional mainframes. This topic is widely discussed in popular business media like CIO Magazine [101], Computerworld [88, 123] or American Banker [40] as well as online discussion channels like Quora [4] and Blogs [5] drawing hundreds of comments. However, little to no scientific publications can be found describing this topic in general or discussing ways to address it. A call by Lindoo [83] in 2014 to reintroduce COBOL in university curricula has gone without citation since and itself does not use a single scientific reference. Of course conferences like ICSME treat subtopics on program comprehension, automated knowledge extraction and automated program translation - however little of this is suitable to attack the situation at the scale it presents itself today.

Paired with the knowledge that the average COBOL developer is in the age bracket of 45 to 55 years [64], the question that needs to be asked is "Who Will Maintain It?" [36]. Fleishman has written a good summary of the topic in 2018 for Increment [48], noting that this, unlike the Y2K crisis, is not one with a hard approaching deadline, starving it from the attention of CIOs, even when "their own" magazine is proclaiming "Why it's time to learn COBOL" [101].

This industry is facing a double challenge. On the one hand the baby-boomers are increasingly leaving the work force, on the other hand young developers are shying away from learning the skills to take their place [88]. One of the fears might be having to maintain these legacy applications for the remainder of their professional careers. The author observes, especially among young colleagues, a strong gravitation towards innovative, consumer facing mobile centric applications. There is a drive to be part of the next Facebook, Twitter or Uber and similar disruptive Startups, using or even better inventing the next programming trends and paradigms, ideally becoming rich along the way. Albeit understandable, it neglects a large portion of the overall IT market. While some of these organizations might become victims of disruptive upstarts, most will be here to stay, most prominently among them, the government sector.

This leaves the playing field to established commercial technology and service provides like IBM, Microfocus, Accenture or Unisys. These organizations train thousands of employees to provide their clients with the maintenance services and technologies they require to keep up the status quo. While this closes part of the gap in providing manpower, it does not address the overarching issue of where to go next. A critical, impartial and unbiased approach is therefore urgently needed to come up with ideas that go further than the current "silver bullet" approaches like replatforming and automated source code translation (see Section 5.1.3). An impartial and commercially unbiased approach will, in the author's view, require deeper involvement by academics and their institutions. One way to get started in this direction is to start teaching COBOL and similar languages and incorporate legacy system related topics into the CS curriculum more deeply.

On the industry side one key observation was how organizations dealt with legacy system developers (if they were still around). The usual approach seems to be to stick them into trainings for modern programming languages and afterwards incorporate them into development teams (typically with an average developer age approximately half of theirs). While this might theoretically be a suitable approach for those individuals interested in pursuing this path, it was rarely observed in practice. This approach also neglects the fact that legacy systems need to be maintained during the whole, potentially years long transition period. Even afterwards the source code and data of the legacy system might be the only source for some types of past information. It furthermore leaves a huge potential untapped: the enormous experience, domain and technical knowledge of long time employees. So instead of forcing them down a path which they might not want, an approach is needed that positively involves them in the transition and gives them the chance to pass on some of their knowledge to younger colleagues.

A positive example of this can be found in the case airport (Section 4.3), where the remaining legacy system developers were treated like internal consultants and could be accessed by requirements engineers, developers and testers alike to tap into their wealth of knowledge. They patiently explained existing solutions and their genesis and were a valuable resource in evaluating potential new solutions to avoid repeating past mistakes. The university case study (Section 4.4) on the other hand retrained a large portion of the legacy system developers in the new programming language and frameworks. Few of them actually wanted this, none of them knew what it would entail. This had several side effects. For one, the lead developer was bogged down and frequently held up with helping the insufficiently trained colleagues. Instead of leading a capable team of developers and solving the major technical obstacles, he was forced to perform training on the job and to correct problems inevitably caused by inexperienced team members. Naturally both progress and team spirit suffered. An additional obstacle was the ongoing maintenance of the existing system. Even those team members motivated to learn the new technologies were frequently distracted or even pulled out to perform emergency as well as routine maintenance tasks. In addition, they were needed as experts for the legacy system, which again cut into their actual development time. In the end performing three jobs (developer, maintainer and expert source of knowledge) overburdened all but one of them, resulting in a gradual exodus of most team members from the organization and therefore ultimately a great loss of knowledge.

5.3.4 Acknowledge software renovation project and plan accordingly

... and not as a greenfield software development project.

One of the first steps in starting a legacy system migration project is actually acknowledging it as one. This might seem obvious, if the conception of the project started with the realization that the current system is actually a legacy system and that it needs to be renovated or replaced to accommodate future needs. This scenario is not the problem. Projects perceived or dressed up as greenfield (or maybe brownfield) projects are. Brodie and Stonebraker [32] already state as the number one problem with the "Cold Turkey" approach, that "A better system must be promised". As technical debt, legacy migration projects that do not introduce additional business value through new or extended functionality are a hard sell. The return on investment (ROI) of transformation projects is even harder to calculate than for regular IT projects. In addition, executives are seemingly biased towards easier or cleaner solutions in the form of new (off the shelf) products. Ulrich calls this "the Transformation ROI Dilemma" [124].

Ignoring or denying the existence of legacy assets has severe drawbacks. The legacy system and its maintainers are being ignored as a valuable source of information and experience. This risks repeating past mistakes and missing functions. The data processed by the legacy system is ignored while designing and implementing the new data structures. This risks difficulties in a subsequent

data migration effort as well as the risks mentioned for the previous point. It also might result in a disconnect between the specification of the new features and the realities "on the ground" of established business processes. The major risk here is a lack of acceptance by the most important stakeholder group - the prospective users of the new system.

The airport case (Section 4.3) on the one hand can be regarded as a positive example. The migration of the legacy system was planned as one straight from the beginning. In addition, some major ground rules were established and can be retrospectively regarded as major success factors. New Features and enhancements were strictly forbidden in favour of a one-for-one rewrite, with the exception of unavoidable changes due to a changing environment. An incremental approach was mandated to mitigate the risk of a big bang integration. The maintainers of the legacy system received roles in the project and were accordingly available as an invaluable source of information. Extensive log analysis of the legacy system was performed to uncover hidden functionality. The business processes were documented on the ground, visiting and involving the actual users of the system.

Both the university (Section 4.4) and insurance (Section 4.1) cases on the other hand can be regarded as counter-acting observations. Neither of the projects were sold as legacy system transformation effort. Both had to promise significant additional business value and combine it with ambitious time lines to obtain approval for the project. On a high level, legacy issues were acknowledged, but in the day to day project work this was not properly reflected. Both mostly ignored the legacy systems as a source of information. In very similar functional fields (managing personal master data) the following scenarios developed:

In the university case, the development of the master data management was used to establish the new technology stack and architecture. The focus was heavily on developing the new assets and producing a clean and powerful data model for the university's master data. However, this resulted in a disregard for one of the major challenges at hand. Master data in the legacy IS was actually dispersed among several systems and data quality was varying greatly. After functional completion with the module, data migration was attempted, revealing several problems. Duplication was a major issue due to the fact that the legacy system had different data sets for employees and students. No joint unique identifier existed to link those. Furthermore historic data was lacking many of the attributes now required for data entry and was not normalized. In the end the data migration and consolidation effort consumed more time and resources than the actual development of the initial system. During the course of that period the system had to undergo major modifications, breaking many of the original designs and constraints, at the end exhibiting similar characteristics as a legacy system itself. This entailed a project delay of more than a year and was a key driver behind the subsequent review of the chosen approach. The learnings from this effort helped the team to turn the ailing project around and, after painful restructuring and a long and hard crunch time to deliver a project success in the end. However, to this day, the master data system continues to exist as a major debt in the new system landscape.

The insurance case had a very similar scenario with a master data system developed "greenfield" disregarding the legacy systems both as a source of information as well as a future consumer of the data. Work on data migration was started only after the new system was actually in production covering a one-off use case, resulting in an increase in "legacy" data source from two to three. The migration subsequently turned into major rework and as a result delays in project execution. Again, data quality and duplication were major issues. On top in this case one of the legacy systems remained open to data entry, resulting in the problem already described in Section 5.1.4. The resulting three way synchronisation and gateway mechanism had to include facilities for conflict detection and (manual) resolution. Daily error lists kept a whole team of developers and users busy over months until the legacy system was finally turned into a passive consumer of the data.

This observation can be regarded as the culmination of the whole chapter. Most of the aspects highlighted here interact with each other and in combination result in the previously discussed situations. The basic ideas and approaches necessary to tackle the technical problem as such have been around for a while and are well published [32, 104, 124]. Simply pointing a finger at a project team claiming they did not do their homework would however severely trivialize the situation. In fact the complex socio-technical environment seems to encourage the occurrence by favouring new features and silver bullet solutions over technical realities. Visions are discussed and presented extensively, legacy assets on the other hand seem like an inconvenient afterthought. Instead of asking "where do we want to go?" as a first question it might be a more suitable approach to first find out "where are we going to start?".

Seacord opens his case study "The Beast" with a citation from Fables by Jean de La Fontaine "He told me never to sell the bear's skin before one has killed the beast" (originally in French) [104]. It seems this piece of wisdom needs to be taken one step further transforming it into "... before one has found the beast".

6 Towards a Topology for the Field of Legacy System Migration

The Cautious Abstractions & Generalisations introduced in the previous chapter and illustrated in Figure 5.1 discuss a wide array of points on legacy system migration (LSM). Figure 6.1 puts them into a more general context and highlights relationships. Grouping the results points to two important realizations. First of all, the distinction between technical, architectural and organizational observations is rather weak. A number of items can be regarded as potentially fitting to two of the categories, which has been illustrated by placing them at the border. Second of all, grouping the items into closely related clusters reveals a set of five topics that will be discussed in the following paragraphs. Each topic will culminate in a (bold) crunch point and some notes on perceived challenges ahead. The resulting mapping can be regarded as an initial topology for the field of legacy system migration. The topics represent areas of research that have been partially explored as part of this thesis utilizing both the presented cases and existing literature as a basis. The crunch points can be regarded as the findings of more specific and localized explorations. The level of abstraction has been chosen to represent the middle ground between extremely high on the one hand, where it is easy to assemble a set of over-generalized rules into a rulebook and too low on the other hand where findings are too specific for wider applicability.

6.1 Best Practices

A series of "best practices" provides a foundation of low hanging fruit and easy takeaways. These best practices are easily applicable patterns designed to facilitate the actual implementation of the migration project. A key aspect is to actively engage with the legacy system early on in the project. This approach has two major benefits. First of all, it forces the project members to familiarize themselves with the system at hand and to get to the point where it is generally acknowledged that retiring the system actually means getting to know it intimately. Second, the effect of this interaction is taking away the fear of being bitten by the legacy "beast", as an initial step towards taming it. In the end, this should lead to the capability of being able to clearly distinguish between ancient myth and actual fact when it comes to reasoning about the system. All of this is necessary in order to be able to narrow down the scope of the migration to what is absolutely necessary, to experiment and most importantly to be able to quickly answer questions based on facts from the only ground truth, the legacy system. This is a time and resource intensive task, but there is no way around it as the legacy system represents the most important and frequently only source of information. This results in the following realization:

Crunch Point 1: A successful Legacy System Migration effort needs access to and the knowledge embedded within the legacy system and both the capability and the empowerment to make modifications to it.

The best practices highlighted in this thesis only represent a fraction of what is out there. However, unlike in object orientation [54] or enterprise integration [66] no established and widely acknowledged pattern language exists. Apart from the obvious need for such a collection of patterns to help engineers through legacy system migration, it more importantly leaves a gap in terms of a common language amongst researchers. This lack of a commonly accepted base line makes it

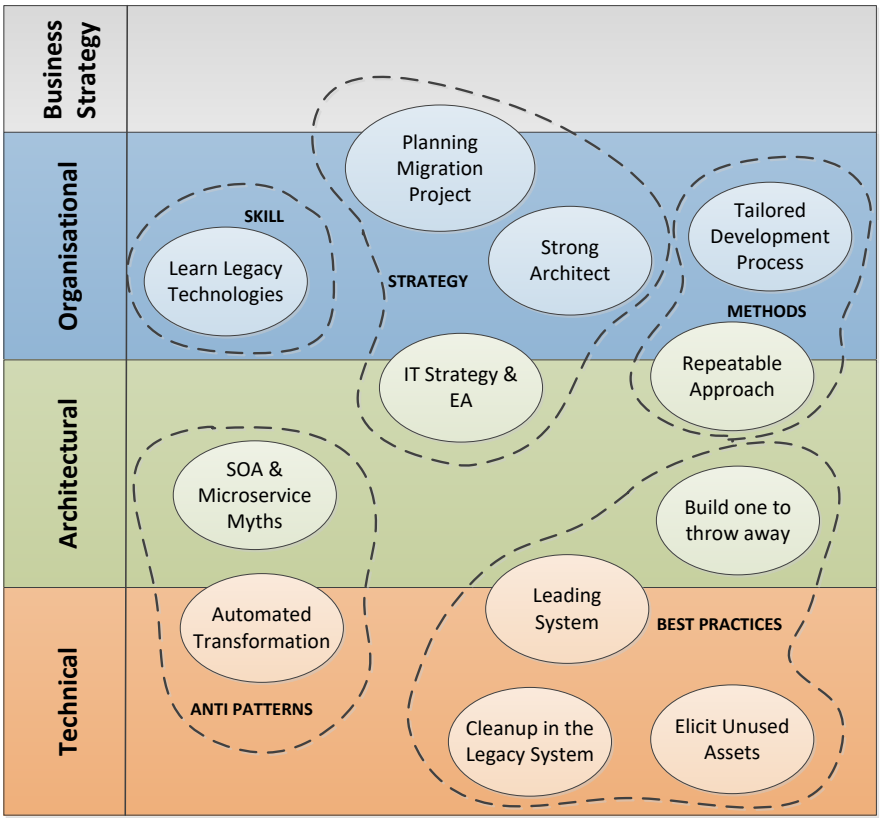

Figure 6.1: Legacy System Migration Topology: Grouping the cautious abstractions & generalisations into related topics to highlight relationships.

harder for researchers and practitioners to classify problems as well as solutions and leaves them without a common point of reference.

The identification and classification of legacy systems is another major challenge. The attempts to classify the units of investigation in Chapter 3 are unfortunately not based on a commonly accepted standard. In essence the problem already starts by an overly generic definition of "legacy system" (see also Section 2.1.1). However, simple but at the same time essential characteristics like a homogeneous monolithic or heterogeneous distributed nature of a system are not captured. All too often it seems that the term legacy system is reduced to the stereotype of a monolithic application written in an early third generation programming language. As demonstrated by the cases Insurance (Section 4.1) and University (Section 4.4) this is frequently not what is found in practice.

6.2 Anti Patterns

In contrast to the "best practices" the topic of "anti patterns" highlights areas with a problematic tendency towards the silver bullet syndrome. This by no means is an argument that these practices do not have their place. There are certain use cases where the application does make sense and is capable of delivering unique benefits. However, the observations show that benefits are frequently greatly exaggerated and the risks systematically downplayed. This is especially true in the context

of specialised providers pitching their services. Naming and describing anti-patterns is an effective way to provide the reasoning for a set of patterns, as demonstrated by Lauder et al. in the context of legacy systems [79]. This leads to the second Crunch Point, which can be regarded as the anti-thesis to Crunch Point 1.

Crunch Point 2: Any attempt on migrating a legacy system without understanding its design and the underlying sources of complexity will result in a reincarnation of the same legacy system with only slight technical variations, but the same fundamental characteristics.

Publication bias represents a major challenge in this context. Academics and probably to an even higher degree practitioners from industry routinely refrain from publicly talking about their failures and the reasons behind them, although it seems commonly accepted that reengineering projects have a disproportionately high failure rate [15]. As an example in this crunch point, the observation behind automated source code transformation (Section 5.1.3) can be used. Once the decision has been made and a significant amount of effort put into an automated transformation there is a high tendency towards selling the outcome, however disappointing, as a success. Often just the idea of a truly failed project would be political suicide inside an organization. This leaves very little room for critical out of the box thinking or investment into post mortem analysis efforts, both preconditions for learning from a mistake.

6.3 Skill

The topic of "skill" focuses on the argument around a lack of programming skill matching the technologies used to build legacy systems. This can be seen as emblematic for the whole range of legacy system migration projects. It is true for all common roles in a software project. Business Analysts need specialised skills and tools to understand legacy systems, Testers need specific test methods to ensure functional compatibility between the source and the target system and Project Managers need specialised approaches and metrics to account for the specifics of a migration project. For all stakeholders the specific skills need to be taught and practised. After all experience is the key to mastering any skill.

Crunch Point 3: Performing a Legacy System Migration requires a unique, specialised set of skills for all involved stakeholders. However, even more so than in regular software projects, no amount of pure skill or effort will suffice if it is not matched with correlating experience.

In the introduction to his last book summarizing forty years of IT project experience Sneed writes that "it is surprising how little the young generation of computer scientists know about the evolution of computer science" [111] (freely translated from German). Universities frequently argue, that they are not meant to teach skills that can be easily acquired on the one hand and are quickly outdated on the other, but they want to support the students in understanding the concepts behind a set of problems and how to solve them on a conceptual level. This argument is frequently brought in the discussion about which (modern) programming language to teach. The same argument should be made for legacy languages like COBOL or PL/I. It is true, university is not the place to learn specific languages just because industry still has large piles of systems written in these languages, but it is essential, that students understand the concepts behind these languages and how they came into being. Students should be equipped to learn any legacy language the same way as they should be equipped to master any modern or emerging one.

The challenge however is not limited to programming languages. The concept translates to process and methodology as well. Students need to learn how to approach legacy systems and to use associated techniques and tools. Reverse engineering a system is a skill that any computer scientist should have, irrespective of later job descriptions. Methods and approaches, although

still immature as highlighted in the next section, should be incorporated as well. Altogether, the requirement is to better prepare students for the world full of IT legacy that is waiting for them after graduation.

This leads to a third challenge, a mismatch of expectations between young developers and enterprises as their potential employers. Apart from the formal education developers primarily learn and read about new and "cool" technologies and approaches by the big internet companies (like FAANG: Facebook, Apple, Amazon, Netflix, Google). However only comparatively few of them land a job in those (or similar) companies. This mismatch results in a hard landing in reality when realizing that the corporate environment at a local bank or insurance company might not match the laid back or ambitious atmosphere as propagated by the FAANGs HR departments and are definitely not part of the "move fast, break things" family. On the one hand the challenge lies in managing expectations of future generations of software engineers accordingly. Maintenance and evolution of a core insurance system is different from "inventing the next big thing". While it is certainly an attainable goal to be part of a disruptive undertaking, expectations must be managed to achieve an understanding that this by definition cannot be the norm. Companies on the other hand have to acknowledge the unavoidable change in culture which comes with their new employees and their expectations. Seeing and accepting software development as a creative process that is supported by all measures that foster creativity is a great initial step. Supporting continuous learning and providing the chance to contribute to open source community efforts can help offset the monotony of keeping alive enterprise (legacy) systems and at the same time readies the employees for the inevitable point in time when their system is no longer around.

6.4 Methods

The topic of "methods" discusses the necessity to adapt established approaches to the specifics of a legacy system migration project. The two key points in this respect are the need for established methodologies prior to the commencement of the migration effort on the one hand and on the other hand the requirement to adjust them to the unique challenges presented by this type of project.

Crunch Point 4: Tailoring established methodologies to suit the needs of a legacy system migration project is essential. Fanatically sticking to textbook fundamentals will lead to catastrophe.

There are two main challenges related to this topic. The first one is the need to integrate requirements for handling both the effects of software ageing and the final decommissioning of a system into the initial design and concept and to smooth the transition between implementation and maintenance. The first step in this is finally acknowledging that most of the money spent on a software is spent in these phases [81]. Maintenance and evolution still happen in a very ad hoc and unstructured manner, in many cases reduced to the bug fixing and the implementation of new features with no budget for an actual evolution of the underlying technology [97]. A key point is figuring out how to effectively sell reengineering projects to the business. Continuously neglecting or postponing these efforts radically shortens the lifespan of a software system.

The second is tailoring of existing software development processes to the unique challenges of legacy system migration efforts, an observation in line with recent publications as for example Model-Driven Software Migration [52] or recent systematic literature reviews on software migration [106]. This represents a clear gap to reality as virtually every large industry effort does include at least some modernization or migration component. A common set of methods and frameworks that spans the fields of software engineering and legacy system migration needs to be developed in order to address the currently observable shortcomings. How can a legacy system be represented as a stakeholder? How do agile development methodologies fare in migration scenarios? There are only few reports of this being done in practice, like [1]. Requirements engineering and

reverse engineering are not easily interchangeable, yet lead to similar artifacts that are used for subsequent forward engineering. It starts with identifying legacy artefacts as assets that provide valuable input to a software project and continues with strategies to properly approach these assets and subsequently get the most out of them.

6.5 Strategy

Finally, the topic of "strategy" is about making sure the individual legacy system migration effort is embedded in a sufficiently complete and long term big picture. This means on the one hand to lay out the necessary strategic roadmap to give everyone a common vision to aspire to. On the other hand this implies to establish the structures for decision making according to the previously laid out goals, but also to acknowledge the need for continuous evaluation and evolution of the package as a whole. Basically this is the level where LSM efforts are aligned with overarching business strategies and efforts like business process reengineering (Section 2.1.5).

Crunch Point 5: A high level IT strategy and a matching structure are necessary to provide a common understanding of purpose and goals. The trinity of line, project and mentorship has to be mobilised at every level to sustain technical, organisational and political direction and promotion.

The first IT systems were created to support the digitalisation of the enterprise. Instead of paper/file based handling, centrally managed data is updated by one stakeholder and available immediately to everyone with access to the system. Aggregated data is used to produce comprehensive reports that give a clear picture about the current state of affairs to decision makers. With the pressure of globalization web enabling enterprise IT systems was necessary to make them available to all locations worldwide. Today, more and more industries are disrupted by start ups leveraging state of the art technology as well as methodology and not bogged down by a mountain of legacy. Incumbents naturally have great difficulty to keep up. Agile and lean transformation of the enterprise appears to be commonly motivated by this realization.

The challenge in this context is the fact that it is not or at least no longer enough to replace a legacy IT system with an extensible and maintainable counterpart based on modern technology. On the one hand rapid adjustment to a changing environment must be possible. This means that the organization as a whole needs to be able to change business processes or even rip out and replace parts of the systems as quickly as possible. On the other hand the specific problem must be clearly understood to address it properly. Recognising the nature of the challenges ahead will help to avoid ill advised technical decisions or hasty orientation towards a silver bullet hype like yesterday's SOA or today's Microservices. Overall this goal is only achievable if a mixed team comprised of all affected stakeholder groups is ready to tackle the problem without organizational or technical obstacles in their way. The abstractions in section 5.2.1 "IT Strategy and Enterprise Architecture should be in place before deciding on the target architecture", section 5.3.1 "Enable architectural decision making" and section 5.3.2 "Established software development process" support this crunch point and give more insight.

7 Conclusion

The research field of legacy system migration is not new by any means. Standard works like the book by Brodie and Stonebraker [32] as well as the taxonomy by Chikofsky and Cross [37] were established in the 90s and still hold up well. The literature was later complemented with the intensive learning phase from the Y2K frenzy. Examples include the books by Ulrich [124] and Seacord [104]. A Roadmap for the upcoming decades was suggested by Bennett and Rajlich [13]. They conclude that "Software evolution needs to be addressed as a business issue as well as a technology issue, and therefore is fundamentally interdisciplinary." and furthermore state "Too much focus at present is on the technology, not on the end-user." The following chapter will scrutinize if academia and industry have heeded this advice.

In this thesis we first selected and then presented four units of investigation. Each case represents an industrial instantiation of a legacy system migration (LSM) with unique characteristics and features resulting in a distinct set of lessons learned for future migration efforts.

The Insurance case (Section 4.1) at the first glance looks more like a digitalization than a legacy system migration effort due to its highly heterogeneous nature. On the one hand this can be regarded as true on a macro-scale, on the other hand this is likely the most typical scenario of all. In reality legacy systems and newly planned ones seldom match one to one as migration happens as part of a larger scale architectural transformation. However, any large digitalization effort will be faced with a multitude of challenges imposed by existing assets. The field of legacy system modernization provides the tools to address these challenges and the means to make an educated decision whether to wrap or to rebuild, whether to homogenize platforms or embrace a heterogeneous IT architecture as propagated by Microservice architectures or whether to employ gateways or adapters. All of these decisions have to be made at a time of massive disruption to the way business is done and have to be aligned with the necessary organisational transformations to create a viable timeline to achieve the overarching goal of digital transformation. At the second glance this case has it all: automated code transformation, replatforming, three-way-synchronising gateways, wrapped legacy systems, recently written and therefore young legacy systems and an ever mounting pressure to finally and fundamentally change much more rapidly that in past decades.

The Ministry case (Section 4.2) is an outlier in the sense that it is an example of a rehosting effort. The author himself has argued in the previous section that this by itself is not a permanent solution to the challenges presented by a large legacy system. However, this case illustrates a scenario where the solution does address the immediate problem of exuberant costs and therefore was accepted as an intermediate solution. In addition, it presents a valuable lesson about the lure of silver bullet automated code transformation and how to counteract on it. In the end, the intermediate step taken there means that this case should be revisited in the future. After all it still represents a legacy system and therefore an opportunity to learn about the (long term) effects of rehosting on the one side and subsequent steps to a final solution on the other.

The Airport (Section 4.3) case most closely represents a classic legacy system migration. It was planned and executed as an almost one hundred percent one to one redevelopment. It represents the final, successful step after a series of failed trials, again including automated code transformation. It also holds almost textbook grade examples on the dangers, but also benefits of an incremental migration strategy. It can furthermore serve as an example showing that investment in the legacy system up to the last minute of its productive operation does pay off in terms of facilitating the

migration itself and consequently improving the final quality of the reengineered system. Finally it is crucial to note that this also shows the importance of having the strategic and organisational structures in place before tackling the technical challenges.

The University case (Section 4.4) illustrates how far legacy systems can be taken, resulting in being termed with the highly descriptive phrase “Highly Loose Quagmire”. It demonstrates two important points. For one it shows that the functionality represented by the code of a legacy system can deteriorate through the effects of software ageing to a point where there is virtually nothing salvageable left, yet the legacy system still somehow stays operational and of value to the organization. Moreover, it highlights the fact that the most and ultimately the only remaining valuable asset of a legacy system will always be the data handled by it. Above all this case emphasizes that nobody, not even a university equipped with highly capable computer scientists and software engineers is immune to the inevitable and ultimately fatal effects of software decay.

Aspects from all four cases were published in scientific literature. In the Ministry case “Challenges in re-platforming mixed language PL/I and COBOL IS to an open systems platform” [130] discussed some of the main technical obstacles of performing the re-platforming step. For the Insurance case “A Tiered Approach Towards an Incremental BPEL to BPMN 2.0 Migration” [115] discussed the specifics of migrating process instances from a legacy process engine to a modern, highly integrated counterpart. In the context of the Airport case the focus of scientific publication was on the incremental migration strategy [19] and the applied code review methodology [18]. The University case published a series of online resources making transparent the project’s progress as well as obstacles [21, 74, 117–120]. In addition, a specific focus was set on the reverse engineering of the legacy system data sources in “Digging deep: Software reengineering supported by database reverse engineering of a system with 30+ years of legacy” [116]. Furthermore the cases Airport and Insurance are used as real world examples in teaching lectures on “Advanced Software Engineering” and “Software Maintenance and Evolution” at the Vienna University of Technology.

We continue by elaborating three groups of “Cautious Abstractions & Generalisations”, which are extracted from and argued on the basis of the previously presented units of investigation. This restrained approach is necessary to be able to avoid jumping to premature conclusions and to cover the intended wide range of topics from technical over architectural to organisational topics. The strategy can be compared to writing a serial of position papers instead of full blown journal articles. However, each generalisation has the aspiration of delivering a message viable on its own and the potential to be developed into a standalone publication by itself. As such, each point can be regarded as potential future work and therefore an open research question.

Concerning “Technical Experiences & Observations” the initial focus is on preparatory measures to perform the initial groundwork necessary for getting started with a migration effort. “Cleanup in the legacy system” argues that all steps that can be taken to improve the quality characteristics of the legacy system prior to the migration will improve the chances of success by eliminating barriers and technical debt. “Elicitation of unused assets” complements the prior section with a focus on artifacts no longer in use. Both of these abstractions counteract the general tendency to avoid touching the legacy system at all cost and have the goal of reducing the problem space for a migration as much as possible. A different technical aspect is discussed in “The value of automated code transformation”. People looking for a silver bullet are magically drawn towards automated code transformation. While this approach certainly has its place, it has been observed that the risks in many cases heavily outweigh the benefits. It can therefore be considered an urgent warning to treat this topic with utmost caution, especially when being sold as a solution to all the problems imposed by the legacy system in question. The last section, “Always have a leading system”, focuses on a strategic design problem manifesting in technical pains and therefore also serves as the connecting passage to the subsequent discussion on architectural experiences. Many

solutions with the primary goal of avoiding a "Cold Turkey" approach resort to synchronisation mechanisms (or gateways in the terms of Brodie and Stonebraker) that distribute changes to data between different systems. If there is not a completely clear separation of data responsibility between the players, this leads to a series of issues in the area of consistency and determinism.

The "Architectural & IT Strategy Experiences & Observations" increase the measure of abstraction by addressing topics on a strategic design level. As a start "IT Strategy and Enterprise Architecture should be in place before deciding on the target architecture" argues that while a focus on tackling the legacy issues facing an organisation is an important measure it must be embedded in and aligned with a high level strategic goal as a guiding principle. It is important to note that this is not sufficiently covered by deciding to migrate towards either SOA or Microservices. These two hypes are discussed in the subsequent section, "Looking behind the SOA and Microservice Myths". The key point of this experience is that, although a lot of research is currently targeted on migrating towards these two target architectural concepts, there is little evidence suggesting that these approaches are a good fit to a proportionate amount of legacy system migration efforts. "Build one to throw it away" makes the case for prototyping and critical reevaluation of architectural and design decisions. Experience shows that a lot of this happens in an accidental way and the conclusions are drawn too late or in a worst case scenario never. The latter typically happens with the argument that too much has been invested already to make a fundamental change at this point in time. The second group of generalisations concludes with a suggested strategy to "Towards a repeatable approach for understanding a legacy system". This builds on the realization that little is available in terms of methods and techniques towards approaching an unknown legacy system. The biographic approach outlined is geared towards a high level, holistic understanding without getting ahead of a potentially more comprehensive reverse engineering effort.

The chapter is rounded off with "Organizational Experiences & Observations". As a first step we argue for the "Enable architectural decision making". From our experience we notice a striking lack of empowerment, if such a role exists at all. The most visible result of this is a merging of managerial roles with technical decision making which leads to an observed tendency towards short sighted and mainly short term benefit oriented decision making piling on initially less perceivable technical debt. The second step follows similar lines making the case for ensuring an "Established software development process". In short, the replacement of a core legacy IT system of an organization is just not the place for experiments on methods and processes. Any established process will have to be tailored for the specific needs of such a project as most development processes are not geared towards a legacy oriented project and thus needs to be well established in advance. We continue with a different topic by making the case that "Learn legacy programming languages (again)".The main arguments are that on the one hand this is a task for universities and other educational institutions to bring legacy technologies and their concepts back on the curriculum, on the other hand a task for the industry to make roles that include dealing with legacy systems significantly more attractive. The concluding observation is a call to "Acknowledge software renovation project and plan accordingly". It builds on most of the previously handled topics and highlights the pervasiveness of projects with a significant legacy portion in an enterprise setting. We just can not go on acting as if we were dealing predominantly with greenfield projects, when these are in fact unicorns.

The "Cautious Abstractions & Generalisations" are subsequently interpreted further by grouping them into categories and therefore creating a topology of the field of LSM in Chapter 6. This topology represents the main result of this thesis. It provides means for navigating the subject of legacy system migration and highlights connections between topics and their relationships. The five crunch points, one for each category, highlight the learnings in a very condensed and slightly provocative form. They can be regarded as beacons providing orientation in the uncharted territories of the legacy system migration landscape.

7.1 Future Work

The cases selected in this case study do have a certain bias towards the public sector. This might have been influenced by a strong presence of the government sector at this location as well as macro-timing in the time lines of the projects. Several further cases did not make the final cut, but do have the potential to deliver further evidence for strengthening as well as theoretically falsifying the findings and abstractions presented in this thesis. These potential future cases include sectors like health, banking, insurance, an international organization, transportation, retail and manufacturing. They include several international cases to address any possible location bias. These stories from the industry sector urgently need to be told to highlight current and upcoming challenges. These challenges include shortcomings in methodologies, processes and strategies as well as tools. For both academia and industry a growing collection of cases, benchmarking and refining a selection of reasonable abstractions (as done in chapter 1) that can be used to construct and validate a set of theses should be a solid foundation for increased exchange of ideas and collaborative research.

These additional cases should be complemented by quantitative research to better gauge the current state of the industry as well as its trajectory. As highlighted in Section 5.3.3 there is still a giant amount of software running on what is now considered legacy hardware. However, these are rough, mostly global estimates and questions on topics like if the size of the code base is actually growing or shrinking are left unanswered. In addition, software written in a legacy language does not automatically mean it exhibits the characteristics of a legacy system (section 2.1.1). I do suspect that the picture presents itself very differently depending on both the industry as well as region being investigated. Quantifying the install base of an industry including how much of it is considered to be legacy and what technologies these legacy systems are based on will therefore be an important step in adequately being able to address more specific challenges and research questions.

A recurring theme throughout this thesis is the necessity for a stronger cooperation between industry and academia. This is hardly a novel realization: “We end up writing papers that are read by our fellow researchers but not many others.” [92]. These two sides have to move closer together and acknowledge the invaluable assets each can bring to the table. Academic results need sooner and especially wider validation in an industrial setting. Academic education urgently requires a stronger focus on maintenance, evolution and migration. Razavian et al. identify a gap between academia und industry where the former has exhibited a focus on reverse engineering based migration approaches, whereas the latter prefers and applies forward-engineering approaches, primarily eliciting knowledge from stakeholders [99]. Every student needs to be aware that what is expecting him in industry is most likely very far away from a greenfield setting. Software Maintenance and Software Reengineering still does not receive nearly enough room in current curricula [47, 96]. Industry on the other hand urgently needs to open up to academia and facilitate collaborations, even if this means an initial investment of time and resources with uncertain outcome. This is what research is about. Analysing and publishing failures gives everybody the chance to learn from mistakes. Finally bridge builders are required that are able to navigate both worlds and translate between them while doing so. This means establishing platforms like conferences and journals that actively focus on bringing academia and industry together to move past the isolated industry track on the one side and pure sales events on the other.

Model Driven Engineering (MDE) approaches in reverse and re-engineering as well as knowledge extraction feature prominently in legacy system migration research [49, 52, 53, 127, 129]. However, due to the selection of cases and the limited scope of this thesis it was not possible to accommodate this topic adequately, only brushing it in Sections 2.4.3, 2.4.5 and 4.1. In addition, the widespread practical applicability seems to be in question [29]. As with other approaches, like the

discussed topics of automatic source code transformation (section 5.1.3) and SOA / MicroServices (section 5.2.2), there is a certain tendency towards the silver bullet syndrome. Furthermore MDE is frequently combined with SOA / MicroService as well as automated code transformation approaches (for example [139]) combining both risks and benefits. Therefore, additional cases that utilize such methodologies accompanied by a critical appraisal is needed to show merits as well as limitations of these approaches. The famous words of Alfred Korzybski might serve as a guiding principle: "A map is not the territory it represents, but, if correct, it has a similar structure to the territory, which accounts for its usefulness." [76].

7.2 Final Remarks

In the introductory section of this thesis (Section 1) the ambition to contrast industrial and academic best practices is laid out. The resulting Cautious Abstractions & Generalisations give an indication of both gaps and commonalities. However, they also map the extent of the field and therefore problem space.

Legacy system migration is a multi-faceted challenge. It reaches far beyond mastering technical obstacles which could be addressed by the application of a series of technical patterns. Recipes applicable in practice can by definition only work for a small subset of the problem space, due to the highly diverse nature of what can constitute a legacy system. Even if the problem actually matches, success is highly dependent on a number of critical preconditions being in place, ranging from strategic alignment over established software development methodology to a clear architectural target picture. Last but not least, experience is key. Having successfully executed software projects is not enough. Embracing people with previous exposure to migrating legacy systems and especially including the last remaining people with knowledge of the legacy system at hand are crucial. For an industry practitioner this means reaching out to academia to make use of the latest developments in the field, for a researcher in academia this means approaching industry.

One final thought that needs to be put out there: It is true that the problems facing industrial legacy system migration cannot easily be synthesised in an academic laboratory setting. However, computer science is definitely not the first discipline to face that problem. Maybe medicine can serve as an example. After all, neither diseases nor the ageing of a body (amongst other scenarios) can be properly simulated and there is simply no replacement for practising on real human bodies. This is why the field of medicine has resorted to the possibilities of voluntary donations. This is a call to the industry to think about similar mechanisms for their legacy systems. Instead of burying them in the graves of corporate long term archiving, put them out there for the benefit of science and for progression in the (academic) field. Give them to universities, give them to students or even better bring them to public domain under a permissive open source license. As in medicine this definitely has legal and ethical concerns that need to be properly taken care of especially when it comes to the data handled by a system and as in medicine this will not be possible in every case. Unlike in medicine every donation can be reused and shared infinite times, to try to reproduce results, to try new ideas or to serve as proof that a new method or technique was tested on a non-trivial example.

I am of the strong personal opinion that in the field of legacy system migration academia and industry are bound to collaborate in a much more tightly integrated manner. Academia needs those real world cases with decades of evolution that just cannot be simulated. Industry needs a strong partner that can abstract across different settings and innovate without the pressure of an imminent eruption of our collective software volcano [46].

In this chapter I have summarized the findings of this thesis and outlined a possible path to move forward. My personal takeaways are the fundamental realization that the challenges lying ahead

will last more than a lifetime and that perseverance, patience and long term personal investment will be key to success, both in an industrial and an academic setting. There are few realistic quick wins and no "silver bullets". Supply of research material should be endless as the whole world is jointly working on producing tomorrow's legacy systems. What remains true of any large legacy system migration project is that we initially never know how to find the way out. We learn the most and best in the middle of the densest jungle. Only after suffering the pain we can see the light.

I would like to close with a one of the "Frequently Forgotten Facts" [57] by Robert L. Glass, an expert on failed projects and author of the infamous book "Software Runaways" [58]:

> M5. Most software development tasks and software maintenance tasks are the same—except for the additional maintenance task of "understanding the existing product." This task is the dominant maintenance activity, consuming roughly 30 percent of maintenance time. So, you could claim that maintenance is more difficult than development.

If this is true for maintenance, what conclusion can be drawn for legacy system migration as the ultimate sustainable form of software retirement?

Bibliography

References

[1] Fabio Abbattista, Alessandro Bianchi and Filippo Lanubile. 'A Storytest-Driven Approach to the Migration of Legacy Systems'. In: *Agile Processes in Software Engineering and Extreme Programming*. Ed. by Pekka Abrahamsson, Michele Marchesi and Frank Maurer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 149–154. ISBN: 978-3-642-01853-4.

[2] P. Aiken. *Data Reverse Engineering: Slaying the Legacy Dragon*. McGraw-Hill, 1995.

[3] David J. Anderson. *Kanban: successful evolutionary change for your technology business*. E-Book. Blue Hole Press, 2010. ISBN: 978-0-9845214-2-9.

[4] Travis Atkin. *How many COBOL programmers are left in the US?* 2015. URL: https://www.quora.com/How-many-COBOL-programmers-are-left-in-the-US (visited on 11/10/2018).

[5] Jeff Atwood. *COBOL: Everywhere and Nowhere*. 2009. URL: https://blog.codinghorror.com/cobol-everywhere-and-nowhere/ (visited on 11/10/2018).

[6] Johannes Bach and Martin Schulze. 'Das Debeka-Projekt MiKe – Migration der Debeka-Kernanwendungen von Bull/GCOS8 auf AIX'. In: *Software archeology and the handbook of software architecture*. Ed. by Rainer Gimnich et al. Bonn: Gesellschaft für Informatik e. V., 2008, pp. 21–33.

[7] Armin Balalaie, Abbas Heydarnoori and Pooyan Jamshidi. 'Migrating to Cloud-Native Architectures Using Microservices: An Experience Report'. In: *Advances in Service-Oriented and Cloud Computing*. Ed. by Antonio Celesti and Philipp Leitner. Cham: Springer International Publishing, 2016, pp. 201–215. ISBN: 978-3-319-33313-7.

[8] A. Balobaid and D. Debnath. 'An Empirical Study of Different Cloud Migration Techniques'. In: *2017 IEEE International Conference on Smart Cloud (SmartCloud)*. Nov. 2017, pp. 60–65. DOI: 10.1109/SmartCloud.2017.16.

[9] Camila Bastos, Paulo Afonso Junior and Heitor Costa. 'Detection Techniques of Dead Code: Systematic Literature Review'. In: *Proceedings of the XII Brazilian Symposium on Information Systems on Brazilian Symposium on Information Systems: Information Systems in the Cloud Computing Era - Volume 1*. SBSI 2016. Florianopolis, Santa Catarina, Brazil: Brazilian Computer Society, 2016, 34:255–34:262. ISBN: 978-85-7669-317-8. URL: http://dl.acm.org/citation.cfm?id=3021955.3021998.

[10] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2Nd Edition)*. Addison-Wesley Professional, 2004. ISBN: 0321278658.

[11] Kent Beck et al. *Manifesto for Agile Software Development*. 2001. URL: http://agilemanifesto.org/ (visited on 07/01/2017).

[12] Keith Bennett. 'Legacy Systems: Coping with Success'. In: *IEEE Software* 12.1 (1995), pp. 19–23.

[13] Keith H. Bennett and Václav T. Rajlich. 'Software Maintenance and Evolution: A Roadmap'. In: *Proceedings of the Conference on The Future of Software Engineering*. ICSE '00. Limerick, Ireland: ACM, 2000, pp. 73–87. ISBN: 1-58113-253-0. DOI: 10.1145/336512.336534. URL: http://doi.acm.org/10.1145/336512.336534.

[14] John Bergey et al. *Options Analysis for Reengineering (OAR): Issues and Conceptual Approach*. Tech. rep. CMU/SEI-99-TN-014. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=13319.

[15] John Bergey et al. *Why Reengineering Projects Fail*. Tech. rep. CMU/SEI-99-TR-010 ESC-TR-99-010. Pittsburg, PA, USA: Software Engineering Institute, Carnegie Mellon, 1999.

[16] Mario Bernhart, Andreas Mauczka and Thomas Grechenig. 'Adopting Code Reviews for Agile Software Development'. In: *Agile Conference (AGILE), 2010*. Aug. 2010, pp. 44–47. DOI: 10.1109/AGILE.2010.18.

[17] Mario Bernhart et al. 'A Task-Based Code Review Process and Tool to Comply with the DO-278/ED-109 Standard for Air Traffic Managment Software Development: An Industrial Case Study'. In: *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*. Nov. 2011, pp. 182–187. DOI: 10.1109/HASE.2011.54.

[18] Mario Bernhart et al. 'Applying Continuous Code Reviews in Airport Operations Software.' In: *QSIC*. 2012, pp. 214–219.

[19] Mario Bernhart et al. 'Incremental reengineering and migration of a 40 year old airport operations system.' In: *ICSM*. 2012, pp. 503–510.

[20] Mario Bernhart et al. 'Softwaretechnik - Mit Fallbeispielen aus realen Entwicklungsprojekten - Fallbeispiele'. German. In: 1st ed. München: Pearson Studium, 2009. Chap. 3, pp. 102–138. URL: http://www.inso.tuwien.ac.at/publications/softwaretechnik/.

[21] Andreas Böhacker et al. *TISS – Neue Workflows für Studienabschlüsse und den Zeugnisdruck*. 2009. URL: https://www.zid.tuwien.ac.at/zidline/zl19/tiss_planen_der_strassen_und_roden_im_dickicht/ (visited on 29/11/2018).

[22] Markus Bick, Thomas Grechenig and Thorsten Spitta. 'Campus-Management-Systeme - Vom Projekt zum Produkt.' German. In: *Vom Projekt zum Produkt*. Ed. by Wolfram Pietsch and Benedikt Krams. Vol. 178. LNI. GI, 2010, pp. 61–78. ISBN: 978-3-88579-272-7. URL: http://dblp.uni-trier.de/db/conf/wimaw/wimaw2010.html#BickGS10.

[23] Ted J. Biggerstaff. 'Design Recovery for Maintenance and Reuse'. In: *Computer* 22.7 (July 1989), pp. 36–49. ISSN: 0018-9162. DOI: 10.1109/2.30731. URL: https://doi.org/10.1109/2.30731.

[24] Jesús Bisbal et al. 'Legacy Information Systems: Issues and Directions'. In: *IEEE Softw.* 16.5 (Sept. 1999), pp. 103–111. ISSN: 0740-7459. DOI: 10.1109/52.795108. URL: http://dx.doi.org/10.1109/52.795108.

[25] Michael R. Blaha. 'Dimensions of Database Reverse Engineering'. In: *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*. Washington, DC, USA: IEEE Computer Society, 1997, p. 176. ISBN: 0-8186-8162-4.

[26] Michael R. Blaha. 'On Reverse Engineering of Vendor Databases'. In: *WCRE '98: Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*. Washington, DC, USA: IEEE Computer Society, 1998, p. 183. ISBN: 0-8186-8967-6.

[27] Rastislav Bodík and Rajiv Gupta. 'Partial Dead Code Elimination Using Slicing Transformations'. In: *SIGPLAN Not.* 32.5 (May 1997), pp. 159–170. ISSN: 0362-1340. DOI: 10.1145/258916.258930. URL: http://doi.acm.org/10.1145/258916.258930.

[28] C. Bojinca. *How to Become an It Architect*. Artech House professional development and technology management library. Artech House Publishers, 2016. ISBN: 9781630814342. URL: https://books.google.com/books?id=NY6uDgAAQBAJ.

[29] Miguel Botto-Tobar, Javier Gonzalez-Huerta and Emilio Insfran. 'Are model-driven techniques used as a means to migrate SOA applications to cloud computing?' In: vol. 1. Apr. 2014, pp. 208–213.

[30] Miguel Botto-Tobar et al. 'Migrating SOA Applications to Cloud: A Systematic Mapping Study'. In: *Technologies and Innovation*. Ed. by Rafael Valencia-García et al. Cham: Springer International Publishing, 2017, pp. 3–16. ISBN: 978-3-319-67283-0.

[31] Michael L. Brodie. 'The Promise of Distributed Computing and the Challenges of Legacy Systems.' In: *BNCOD*. Ed. by Peter M. D. Gray and Robert J. Lucas. Vol. 618. Lecture Notes in Computer Science. Springer, 1992, pp. 1–28. ISBN: 3-540-55693-1.

[32] Michael L. Brodie and Michael Stonebraker. *Migrating legacy systems: gateways, interfaces & the incremental approach*. The Morgan Kaufmann series in data management systems. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995. ISBN: 1-55860-330-1.

[33] Frederick P. Brooks Jr. *The Mythical Man-month (Anniversary Ed.)* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-83595-9.

[34] Gerardo CanforaHarman and Massimiliano Di Penta. 'New Frontiers of Reverse Engineering'. In: *FOSE '07: 2007 Future of Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 326–341. ISBN: 0-7695-2829-5. DOI: http://dx.doi.org/10.1109/FOSE.2007.15.

[35] Andrés Carrasco, Brent van Bladel and Serge Demeyer. 'Migrating Towards Microservices: Migration and Architecture Smells'. In: *Proceedings of the 2Nd International Workshop on Refactoring*. IWoR 2018. Montpellier, France: ACM, 2018, pp. 1–6. ISBN: 978-1-4503-5974-0. DOI: 10.1145/3242163.3242164. URL: http://doi.acm.org/10.1145/3242163.3242164.

[36] David Cassel. *COBOL Is Everywhere. Who Will Maintain It?* 2017. URL: https://thenewstack.io/cobol-everywhere-will-maintain/ (visited on 11/10/2018).

[37] Elliot J. Chikofsky and James H. Cross II. 'Reverse Engineering and Design Recovery: A Taxonomy'. In: *IEEE Softw.* 7.1 (Jan. 1990), pp. 13–17. ISSN: 0740-7459. DOI: 10.1109/52.43044. URL: http://dx.doi.org/10.1109/52.43044.

[38] Alistair Cockburn. *Spike Described*. 2014. URL: http://wiki.c2.com/?SpikeDescribed (visited on 24/10/2018).

[39] Melvin E. Conway. 'How Do Committees Invent?' In: *Datamation* (Apr. 1968). URL: http://www.melconway.com/research/committees.html.

[40] Penny Crosman. *Wanted at Banks: Young Tech Pros with Old-Tech Smarts*. 2014. URL: https://www.americanbanker.com/news/wanted-at-banks-young-tech-pros-with-old-tech-smarts (visited on 11/10/2018).

[41] Ward Cunningham. *Plan To Throw One Away*. 2014. URL: http://wiki.c2.com/?PlanToThrowOneAway (visited on 24/10/2018).

[42] Ward Cunningham. *Second System Effect*. 2014. URL: http://wiki.c2.com/?SecondSystemEffect (visited on 24/10/2018).

[43] Ward Cunningham. 'The WyCash Portfolio Management System'. In: *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*. OOPSLA '92. Vancouver, British Columbia, Canada: ACM, 1992, pp. 29–30. ISBN: 0-89791-610-7. DOI: 10.1145/157709.157715. URL: http://doi.acm.org/10.1145/157709.157715.

[44] K.H. Davis and P.H. Alken. 'Data reverse engineering: a historical survey'. In: *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on* (2000), pp. 70–78. DOI: 10.1109/WCRE.2000.891454.

[45] Tom DeMarco and Lister Timothy. *peopleware - productive projects and teams*. 1987.

[46] Arie van Deursen, Paul Klint and Chris Verhoef. 'Research Issues in the Renovation of Legacy Systems'. English. In: *Fundamental Approaches to Software Engineering*. Ed. by Jean-Pierre Finance. Vol. 1577. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, pp. 1–21. ISBN: 978-3-540-65718-7. DOI: 10.1007/978-3-540-49020-3_1. URL: http://dx.doi.org/10.1007/978-3-540-49020-3_1.

[47] Mohammad El-Ramly. 'Experience in Teaching a Software Reengineering Course'. In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: ACM, 2006, pp. 699–702. ISBN: 1-59593-375-1. DOI: 10.1145/1134285.1134395. URL: http://doi.acm.org/10.1145/1134285.1134395.

[48] Glenn Fleishman. *It's COBOL all the way down*. 2018. URL: https://increment.com/programming-languages/cobol-all-the-way-down/ (visited on 11/10/2018).

[49] Franck Fleurey et al. 'Model-Driven Engineering for Software Migration in a Large Industrial Context'. In: *Model Driven Engineering Languages and Systems*. Ed. by Gregor Engels et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 482–497. ISBN: 978-3-540-75209-7.

[50] Martin Fowler. *Monolith First*. 2015. URL: https://martinfowler.com/bliki/MonolithFirst.html (visited on 23/10/2018).

[51] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-48567-2.

[52] Andreas Fuhr, Tassilo Horn and Andreas Winter. 'A.: Model-Driven Software Migration'. In: *In SE 2010, LNI 159:69–80*. 2010.

[53] Andreas Fuhr et al. 'Model-driven software migration into service-oriented architectures'. In: *Computer Science - Research and Development* 28.1 (Feb. 2013), pp. 65–84. ISSN: 1865-2042. DOI: 10.1007/s00450-011-0183-z. URL: https://doi.org/10.1007/s00450-011-0183-z.

[54] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[55] A. Sivagnana Ganesan and T. Chithralekha. 'A Survey on Survey of Migration of Legacy Systems'. In: *Proceedings of the International Conference on Informatics and Analytics*. ICIA-16. Pondicherry, India: ACM, 2016, 72:1–72:10. ISBN: 978-1-4503-4756-3. DOI: 10.1145/2980258.2980409. URL: http://doi.acm.org/10.1145/2980258.2980409.

[56] *Gartner Hype Cycle*. 2018. URL: https://www.gartner.com/technology/research/hype-cycles/ (visited on 11/10/2018).

[57] Robert L. Glass. 'Frequently Forgotten Fundamental Facts About Software Engineering'. In: *IEEE Softw.* 18.3 (May 2001), pp. 112–111. ISSN: 0740-7459. DOI: 10.1109/MS.2001.922739. URL: http://dx.doi.org/10.1109/MS.2001.922739.

[58] Robert L. Glass. *Software Runaways: Monumental Software Disasters*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998. ISBN: 0-13-673443-X.

[59] Thomas Grechenig et al. 'Entwicklung und Betrieb eines Campus-Management-Systems - Aspekte zur Nachhaltigkeit am Beispiel TISS'. German. In: ed. by Hans Brandt-Pook. Vol. P-209. Software Management 2012. Bielefeld: GI e.V., 2012, pp. 135–152. ISBN: 978-3-88579-603-9.

[60] Jean-Luc Hainaut et al. 'The Nature of Data Reverse Engineering'. English. In: *Proc. of Data Reverse Engineering Workshop 2000 (DRE'2000)*. 2000.

[61] Jean-Luc Hainaut et al. *The Nature of Data Reverse Engineering*. Institut d'Informatique, University of Namur. 2002.

[62] Michael Hammer and James Champy. 'Reengineering the corporation: A manifesto for business revolution'. In: *Business Horizons* 36.5 (1993), pp. 90–91. URL: https://EconPapers.repec.org/RePEc:eee:bushor:v:36:y:1993:i:5:p:90-91.

[63] Van Haren. *TOGAF Version 9.1*. 10th. Van Haren Publishing, 2011. ISBN: 9087536798, 9789087536794.

[64] Travis Hartman. *COBOL Blues*. 2017. URL: http://fingfx.thomsonreuters.com/gfx/rngs/USA-BANKS-COBOL/010040KH18J/index.html (visited on 11/10/2018).

[65] J. Henrard et al. 'Strategies for data reengineering'. In: *Procceedings of the Ninth Working Conference on Reverse Engineering 2002, (WCRE '02)*. 2002, pp. 211 –220.

[66] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321200683.

[67] IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE, 1990. DOI: 10.1109/IEEESTD.1990.101064.

[68] Anca Daniela Ionita et al. *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*. 1st. Hershey, PA, USA: IGI Global, 2012. ISBN: 1466624884, 9781466624887.

[69] Ivar Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004. ISBN: 0201403471.

[70] P. Jamshidi, A. Ahmad and C. Pahl. 'Cloud Migration Research: A Systematic Review'. In: *IEEE Transactions on Cloud Computing* 1.2 (July 2013), pp. 142–157. ISSN: 2168-7161. DOI: 10.1109/TCC.2013.10.

[71] Volker Johanning. *IT-Strategie: Optimale Ausrichtung der IT an das Business in 7 Schritten*. German. Springer Fachmedien Wiesbaden, 2014. ISBN: 9783658020491. URL: https://books.google.com/books?id=L-6bBQAAQBAJ.

[72] Ravi Khadka, Amir Saeidi and Andrei Idu. *Legacy to SOA Evolution : A Systematic Literature Review*. Tech. rep. 2012.

[73] Ravi Khadka et al. 'How Do Professionals Perceive Legacy Systems and Software Modernization?' In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 36–47. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568318. URL: http://doi.acm.org/10.1145/2568225.2568318.

[74] Wolfgang Kleinert et al. *The Making of TISS: Juni 2008*. 2008. URL: https://www.it.tuwien.ac.at/de/zidline/zl18/the_making_of_tiss_juni_2008/ (visited on 29/11/2018).

[75] R. Kneuper. 'Sixty Years of Software Development Life Cycle Models'. In: *IEEE Annals of the History of Computing* 39.3 (2017), pp. 41–54. ISSN: 1058-6180. DOI: 10.1109/MAHC.2017.3481346.

[76] A. Korzybski et al. *Science and Sanity: An Introduction to Non-Aristotelian Systems and General Semantics*. International non-Aristotelian library. International Non-Aristotelian Library Publishing Company, 1958. ISBN: 9780937298015. URL: https://books.google.de/books?id=KN5gvaDwrGcC.

[77] Philippe Kruchten. *The Rational Unified Process: An Introduction, Second Edition*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0201707101.

[78] Oliver Laitenberger and Jean-Marc DeBaud. 'An Encompassing Life Cycle Centric Survey of Software Inspection'. In: *J. Syst. Softw.* 50.1 (Jan. 2000), pp. 5–31. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(99)00073-4. URL: http://dx.doi.org/10.1016/S0164-1212(99)00073-4.

[79] A. Lauder and S. Kent. 'Legacy System Anti-Patterns and a Pattern-Oriented Migration Response'. English. In: *Systems Engineering for Business Process Change*. Ed. by Peter Henderson. Springer London, 2000, pp. 239–250. ISBN: 978-1-4471-1146-7. DOI: 10.1007/978-1-4471-0457-5_19. URL: http://dx.doi.org/10.1007/978-1-4471-0457-5_19.

[80] M. M. Lehman. 'On Understanding Laws, Evolution, and Conservation in the Large-program Life Cycle'. In: *J. Syst. Softw.* 1 (Sept. 1984), pp. 213–221. ISSN: 0164-1212. DOI: 10.1016/0164-1212(79)90022-0. URL: http://dx.doi.org/10.1016/0164-1212(79)90022-0.

[81] Meir M. Lehman. 'Programs, life cycles, and laws of software evolution'. In: *Proc. IEEE* 68.9 (Sept. 1980), pp. 1060–1076.

[82] Grace A. Lewis and Dennis B. Smith. *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*. Ed. by Anca Daniela Ionita, Marin Litoiu and Grace Lewis. IGI Global, Jan. 2013.

[83] Ed Lindoo. 'Bringing COBOL Back into the College IT Curriculum'. In: *J. Comput. Sci. Coll.* 30.2 (Dec. 2014), pp. 60–66. ISSN: 1937-4771. URL: http://dl.acm.org/citation.cfm?id=2667432.2667440.

[84] Alessandro De Marco, Valentin Iancu and Ira Asinofsky. 'COBOL to Java and Newspapers Still Get Delivered'. In: *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. 2018, pp. 583–586. DOI: 10.1109/ICSME.2018.00055. URL: https://doi.org/10.1109/ICSME.2018.00055.

[85] Andreas Martens, Matthias Book and Volker Gruhn. 'A Data Decomposition Method for Stepwise Migration of Complex Legacy Data'. In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '18. Gothenburg, Sweden: ACM, 2018, pp. 33–42. ISBN: 978-1-4503-5659-6. DOI: 10.1145/3183519.3183520. URL: http://doi.acm.org/10.1145/3183519.3183520.

[86] John A. McDermid and Keith H. Bennett. 'Software engineering research: A critical appraisal'. In: *IEEE Proceedings - Software* 146.4 (1999), pp. 179–186.

[87] Daniel Méndez Fernández et al. 'Artefacts in software engineering: a fundamental positioning'. In: *Software & Systems Modeling* (Jan. 2019). ISSN: 1619-1374. DOI: 10.1007/s10270-019-00714-3. URL: https://doi.org/10.1007/s10270-019-00714-3.

[88] Robert L. Mitchell. *The Cobol Brain Drain*. 2012. URL: https://www.computerworld.com/article/2504568/data-center/the-cobol-brain-drain.html (visited on 11/10/2018).

[89] Hausi A. Müller et al. ‘Reverse Engineering: A Roadmap’. In: *Proceedings of the Conference on The Future of Software Engineering*. ICSE ’00. Limerick, Ireland: ACM, 2000, pp. 47–60. ISBN: 1-58113-253-0. DOI: 10.1145/336512.336526. URL: http://doi.acm.org/10.1145/336512.336526.

[90] Stephan Murer, Bruno Bonati and Frank J. Furrer. *Managed Evolution; A Strategy for Very Large Information Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg: Berlin, Heidelberg, 2011. ISBN: 3642016324.

[91] M. Papazoglou. *Web services: principles and technology*. Pearson Education, 2008.

[92] David Lorge Parnas. ‘Software Aging’. In: *Proceedings of the 16th International Conference on Software Engineering*. ICSE ’94. Sorrento, Italy: IEEE Computer Society Press, 1994, pp. 279–287. ISBN: 0-8186-5855-X. URL: http://dl.acm.org/citation.cfm?id=257734.257788.

[93] Laurence J. Peter and Raymond Hull. *The Peter Principle : Why Things Always Go Wrong*. Harper Perennial, Sept. 1998. URL: http://www.worldcat.org/isbn/0688275443.

[94] Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison Wesley, 2003. ISBN: 0-321-15078-3.

[95] William J. Premerlani and Michael R. Blaha. ‘An approach for reverse engineering of relational databases’. In: *Commun. ACM* 37.5 (1994), 42–ff. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/175290.175293.

[96] R. Pérez-Castillo et al. ‘A teaching experience on software reengineering’. In: *2013 IEEE Global Engineering Education Conference (EDUCON)*. Mar. 2013, pp. 1284–1293. DOI: 10.1109/EduCon.2013.6530272.

[97] Václav Rajlich. ‘Software Evolution and Maintenance’. In: *Proceedings of the on Future of Software Engineering*. FOSE 2014. Hyderabad, India: ACM, 2014, pp. 133–144. ISBN: 978-1-4503-2865-4. DOI: 10.1145/2593882.2593893. URL: http://doi.acm.org/10.1145/2593882.2593893.

[98] Václav T. Rajlich and Keith H. Bennett. ‘A Staged Model for the Software Life Cycle’. In: *Computer* 33.7 (July 2000), pp. 66–71. ISSN: 0018-9162. DOI: 10.1109/2.869374. URL: http://dx.doi.org/10.1109/2.869374.

[99] Maryam Razavian and Patricia Lago. ‘A Systematic Literature Review on SOA Migration’. In: *J. Softw. Evol. Process* 27.5 (May 2015), pp. 337–372. ISSN: 2047-7473. DOI: 10.1002/smr.1712. URL: https://doi.org/10.1002/smr.1712.

[100] M. G. Rekoff. ‘On reverse engineering’. In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-15.2 (Mar. 1985), pp. 244–252. ISSN: 0018-9472. DOI: 10.1109/TSMC.1985.6313354.

[101] Paul Rubens. *Why it’s time to learn COBOL*. 2016. URL: https://www.cio.com/article/3050836/developer/why-its-time-to-learn-cobol.html (visited on 11/10/2018).

[102] Eduardo Santana De Almeida et al. *C.R.U.I.S.E: Component Reuse in Software Engineering*. Jan. 2007, p. 222.

[103] Wolfgang Schramm et al. ‘Softwaretechnik - Mit Fallbeispielen aus realen Entwicklungsprojekten - Implementierung’. German. In: 1st ed. München: Pearson Studium, 2009. Chap. 6, pp. 245–296. URL: http://www.inso.tuwien.ac.at/publications/softwaretechnik/.

[104] Robert C. Seacord, Daniel Plakosh and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321118847.

[105] Alex Sellink, Harry Sneed and Chris Verhoef. 'Restructuring of COBOL/CICS legacy systems'. In: *Science of Computer Programming* 45 (Sept. 2000). DOI: 10.1016/S0167-6423(02)00061-8.

[106] Muhammad Shoaib et al. 'Software Migration Frameworks for Software System Solutions: A Systematic Literature Review'. In: *International Journal of Advanced Computer Science and Applications* 8 (Jan. 2017). DOI: 10.14569/IJACSA.2017.081126.

[107] Simon et al. 'Language Choice in Introductory Programming Courses at Australasian and UK Universities'. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. SIGCSE '18. Baltimore, Maryland, USA: ACM, 2018, pp. 852–857. ISBN: 978-1-4503-5103-4. DOI: 10.1145/3159450.3159547. URL: http://doi.acm.org/10.1145/3159450.3159547.

[108] Harry M. Sneed. 'Migrating from COBOL to Java'. In: *2013 IEEE International Conference on Software Maintenance* 0 (2010), pp. 1–7. DOI: http://doi.ieeecomputersociety.org/10.1109/ICSM.2010.5609583.

[109] Harry M. Sneed. *Objektorientierte Softwaremigration [Object-Oriented Software Migration]*. German. Addison Wesley Longman, Bonn, Germany, 1998.

[110] Harry M. Sneed. 'Risks involved in reengineering projects'. In: *Reverse Engineering, 1999. Proceedings. Sixth Working Conference on*. Oct. 1999, pp. 204–211. DOI: 10.1109/WCRE.1999.806961.

[111] H.M. Sneed. *Endstation Wien: 45 Jahre Projekterfahrung in der deutschsprachigen IT-Welt*. Books on Demand, 2017. ISBN: 9783744883641. URL: https://books.google.at/books?id=_xZADwAAQBAJ.

[112] John Spaces. *8 Types of Legacy System*. 2015. URL: https://simplicable.com/new/legacy-systems (visited on 18/02/2019).

[113] Thorsten Spitta et al. 'Campus-Management-Systeme'. German. In: *Informatik Spektrum* 38.1 (2015), pp. 59–68. ISSN: 0170-6012.

[114] Thorsten Spitta et al. 'Campus-Management Systeme als Administrative Systeme (Campus Management Systems as Administrative Software Systems)'. German. In: *Bielefeld Working Papers in Economics and Management* 06-2014 (Apr. 2014). DOI: http://dx.doi.org/10.2139/ssrn.2433470. URL: http://ssrn.com/abstract=2433470.

[115] Stefan Strobl et al. 'A Tiered Approach Towards an Incremental BPEL to BPMN 2.0 Migration'. In: *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*. 2016, pp. 563–567. DOI: 10.1109/ICSME.2016.41. URL: https://doi.org/10.1109/ICSME.2016.41.

[116] Stefan Strobl et al. 'Digging deep: Software reengineering supported by database reverse engineering of a system with 30+ years of legacy.' In: *ICSM*. 2009, pp. 407–410.

[117] Monika Suppersberger, Andreas Knarek and Wolfgang Kleinert. *TISS News*. 2010. URL: https://www.zid.tuwien.ac.at/zidline/zl22/tissnews/ (visited on 29/11/2018).

[118] Monika Suppersberger et al. *TISS – Planen der Straßen und Roden im Dickicht*. 2009. URL: https://www.zid.tuwien.ac.at/zidline/zl19/tiss_planen_der_strassen_und_roden_im_dickicht/ (visited on 29/11/2018).

[119] Monika Suppersberger et al. *TISS Epistemologie I*. 2009. URL: https://www.zid.tuwien.ac.at/zidline/zl21/tiss/ (visited on 29/11/2018).

[120] Monika Suppersberger et al. *TISS Epistemologie II*. 2010. URL: https://www.zid.tuwien.ac.at/zidline/zl22/epistemologieii/ (visited on 29/11/2018).

[121] T. Tamai and Y. Torimitsu. 'Software lifetime and its evolution process over generations'. In: *Proceedings Conference on Software Maintenance 1992*. Nov. 1992, pp. 63–69. DOI: 10.1109/ICSM.1992.242557.

[122] A.A. Terekhov and C. Verhoef. 'The Realities of Language Conversions'. In: *IEEE Software* 17 (2000), pp. 111–124.

[123] Patrick Thibodeau. *Should universities offer Cobol classes?* 2013. URL: https://www.computerworld.com/article/2496360/it-careers/should-universities-offer-cobol-classes-.html (visited on 11/10/2018).

[124] William M. Ulrich. *Legacy Systems: Transformation Strategies*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002. ISBN: 013044927X.

[125] William M. Ulrich and Philip Newcomb. *Information Systems Transformation: Architecture-Driven Modernization Case Studies*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. ISBN: 0123749131, 9780123749130.

[126] Raoul Vallon et al. 'Systematic literature review on agile practices in global software development'. In: *Information and Software Technology* 96 (2018), pp. 161–180. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2017.12.004. URL: http://www.sciencedirect.com/science/article/pii/S0950584917302975.

[127] K Velmurugan and Maluk Mohamed. 'A Model driven Approach for Migrating from Legacy Software Systems to Web Services'. In: (Jan. 2012). DOI: 10.4018/978-1-4666-0155-0.ch011.

[128] J. M. Verner et al. 'Guidelines for industrially-based multiple case studies in software engineering'. In: *2009 Third International Conference on Research Challenges in Information Science*. Apr. 2009, pp. 313–324. DOI: 10.1109/RCIS.2009.5089295.

[129] Christian Wagner. 'Model-Driven Software Migration'. In: *Model-Driven Software Migration: A Methodology: Reengineering, Recovery and Modernization of Legacy Systems*. Wiesbaden: Springer Fachmedien Wiesbaden, 2014, pp. 67–105. ISBN: 978-3-658-05270-6. DOI: 10.1007/978-3-658-05270-6_3. URL: https://doi.org/10.1007/978-3-658-05270-6_3.

[130] Thomas Wagner et al. 'Challenges in re-platforming mixed language PL/I and COBOL IS to an open systems platform'. In: *2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, October 2-4, 2019, to appear*. 2019.

[131] Ian Warren. *The Renaissance of Legacy Systems: Method Support for Software-System Evolution*. Secaucus, NJ, USA: Springer-Verlag, 1999. ISBN: 1852330600.

[132] Nelson Weiderman, Dennis Smith and Scott Tilley. *Approaches to Legacy System Evolution*. Tech. rep. CMU/SEI-97-TR-014. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997. URL: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=12919.

[133] Niklaus Wirth. 'A Plea for Lean Software'. In: *Computer* 28.2 (Feb. 1995), pp. 64–68. ISSN: 0018-9162. DOI: 10.1109/2.348001. URL: https://doi.org/10.1109/2.348001.

[134] Bing Wu et al. 'Legacy System Migration: A Legacy Data Migration Engine'. In: *PROCEEDINGS OF THE 17TH INTERNATIONAL DATABASE CONFERENCE (DATASEM'97), PAGES 129– 138. CZECHOSLOVAK COMPUTER EXPERTS*. 1997, pp. 129–138.

[135] Bing Wu et al. 'The Butterfly Methodology: A Gateway-free Approach for Migrating Legacy Information Systems'. In: *Proceedings of the Third IEEE International Conference on Engineering of Complex Computer Systems*. ICECCS '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 200–. ISBN: 0-8186-8126-8. URL: http://dl.acm.org/citation.cfm?id=850931.852129.

[136] Robert K. Yin. *Case study research: design and methods*. 4th ed. Los Angeles: Sage Publications, 2009. ISBN: 978-1-4129-6099-1.

[137] J. Zachman. *CONCEPTS OF THE FRAMEWORK FOR ENTERPRISE ARCHITECTURE*. Tech. rep. Zachman International, 1997.

[138] J.A. Zachman. 'A framework for information systems architecture'. In: *IBM Systems Journal* 38.2.3 (1999), pp. 454–470. ISSN: 0018-8670. DOI: 10.1147/sj.382.0454.

[139] C. Zillmann et al. 'The SOAMIG Process Model in Industrial Applications'. In: *2011 15th European Conference on Software Maintenance and Reengineering*. Mar. 2011, pp. 339–342. DOI: 10.1109/CSMR.2011.48.