



universität
wien

MASTERARBEIT

Titel der Masterarbeit

„Development of a Model for the Migration of Legacy
Architectures“

Verfasser:

Bakk.rer.soc.oec. Hannes Böhm

angestrebter Akademischer Grad

Diplom-Ingenieur (DI)

Wien, 25. März 2011

Studienkennzahl lt. Studienblatt:

A 066 926

Studienrichtung lt. Studienblatt:

Wirtschaftsinformatik

Betreuer:

Univ.-Prof. Dipl.-Ing. DDr. Gerald Quirchmayr

Schriftliche Versicherung

Ich versichere, dass ich die Masterarbeit selbstständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe. Diese Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer Prüfungsbehörde vorgelegt.

Wien, am 25. März 2011

Danksagungen

Die Fertigstellung dieser Arbeit stellt auch gleichzeitig das Ende meines Studiums dar – ein guter Zeitpunkt Danke zu sagen. In erster Linie gebührt dieser Dank meinen Eltern, die mir dieses Studium und alle damit verbundenen Erfahrungen ermöglicht haben. Ihr seid mir gemeinsam mit meinen Geschwistern ein unersetzlicher Rückhalt!

Ich möchte mich an dieser Stelle auch bei allen Freunden und Kollegen bedanken, die mich durch mein Studium begleitet haben. Weiters danke ich Herrn Mag. Robert Zehetmayer für seinen wertvollen Beitrag zu dieser Arbeit durch sein Expertenwissen im Bereich IT Migrationen. Besonderer Dank geht abschließend an Herrn Prof. Gerald Quirchmayr für die fachliche Unterstützung während meines Studiums und beim Erstellen dieser Masterarbeit.

TABLE OF CONTENTS

1. MOTIVATION	7
2. ABSTRACT	8
PART A – ANALYSIS AND DISCUSSION OF THE FOUNDATIONS ON WHICH THE CONTRIBUTION OF THIS MASTER THESIS IS BASED	9
3. WHAT DRIVES THE MIGRATION OF IT ARCHITECTURES?	9
4. LEGACY LANGUAGES, SYSTEMS AND ARCHITECTURES	10
4.1. What actually is legacy?	10
4.2. Typical examples of legacy systems	11
4.2.1. File based architecture	11
4.2.2. Database centric architecture	12
4.2.2.1. Hierarchical databases	12
4.2.2.2. Network databases	13
4.2.2.3. Relational databases	13
4.2.3. Transactions	13
4.2.4. Teleprocessing monitor architecture	14
4.3. Legacy languages	14
4.3.1. Traditional legacy languages	14
4.3.1.1. COBOL	15
4.3.1.2. C	15
4.3.1.3. Fortran	16
4.3.2. Future legacy languages?	16
4.3.2.1. Volatility index	16
4.3.3. Lessons learned	19
4.4. Common Anti-Patterns within Legacy Systems	20
4.4.1. Anti-Patterns in legacy systems	20
4.4.2. Anti-patterns of Enterprise Architecture	23
4.5. Typical Concepts to deal with Legacy Architectures	25
4.5.1. Enterprise Application Integration (EAI)	25
4.5.2. Message Oriented Middleware (MOM)	27
5. MIGRATION OF LEGACY SYSTEMS	28
5.1. Lehman's Laws of Software Evolution	28
5.1. Possibilities of evolving legacy systems	30
5.1.1. Maintenance	32
5.1.2. Modernization	32
5.1.2.1. White-Box Modernization	32
5.1.2.2. Black-Box Modernization	32
5.1.3. Replacement	32

5.2. Discussion of Terms in the Field of Software Engineering	33
5.3. Software Reengineering	34
5.3.1. Retargeting	34
5.3.2. Revamping	34
5.3.3. Commercial Components	35
5.3.4. Source Code Translation	35
5.3.5. Code Reduction	36
5.3.6. Functional Transformation	36
5.4. Software Migration	37
5.4.1. Types of Technical Migrations	38
5.4.2. Migration aspects	38
5.4.3. Migration Strategy	38
5.4.3.1. Re-implementation	39
5.4.3.2. Conversion	39
5.4.3.3. Encapsulation (Wrapping)	39
5.5. Migration in the Field of Software Enhancement Projects	39
5.5.1. Extension Project	40
5.5.2. Maintenance Projects	40
5.5.3. Migration Projects	40
5.5.4. Integration Projects	40
5.6. Reasons for Migrations	40
5.6.1. Aging	40
5.6.2. Cost	41
6. LEGACY APPLICATION REENGINEERING STRATEGIES	42
6.1. Historical	42
6.1.1. Ignore	43
6.1.2. Big Bang / Cold Turkey Migration	43
6.1.3. Integrate	44
6.1.4. Data Warehouse	44
6.1.5. Incremental / evolutionary / gradual Migration	44
6.1.5.1. Chicken Little Migration	44
6.1.5.2. Butterfly-Migration	45
6.1.5.3. Renaissance Approach	45
6.1.5.4. Capsule-oriented Reverse Engineering Method	45
6.1.5.5. Reengineering Factory	46
6.1.5.6. Service Migration and Reuse Technique	46
7. CURRENT ARCHITECTURES AND TRENDS	46
7.1. Current Architectures	47
7.1.1. Multichannel architecture	47
7.1.2. J2EE – Java Connector Architecture	48
7.1.3. .NET	50
7.1.4. Service Oriented Architecture (SOA)	50
7.2. Current Trends	52
7.2.1. Cloud Computing	53
7.2.2. Web and Enterprise 2.0	54
7.2.3. Ubiquitous computing	56
PART B: TOWARDS A SOLUTION	57

8. HUMAN FACTOR ASPECTS	57
8.1. Management	58
8.2. System Administrator	60
8.3. User	60
9. BASIS FOR RISK CONTROL	63
9.1. Benefits of IT Systems	64
9.2. Portfolio Analysis	65
9.3. Incremental Approach	66
9.4. Reference Process	67
9.5. Risk Management	67
10. THE COST BENEFIT ISSUE	68
10.1. Payback Period Rule	68
10.2. Return On Investment (ROI)	69
10.3. Business Case / Cost Benefit Analysis	69
11. THE REFERENCE MIGRATION PROCESS	72
11.1. Phases of the Reference Migration Process	73
12. ENTERPRISE ARCHITECTURE	80
12.1. TOGAF	81
12.2. TOGAF Architecture Development Method	83
13. REMIP AND TOGAF – A COMBINED VIEW	85
14. EXPECTED BENEFITS	89
15. THE REFERENCE MIGRATION MODEL	90
16. CHECK AGAINST PRACTITIONERS' REQUIREMENTS	92
17. SUMMARY AND CONCLUSION	93

1. Motivation

Information and communication technology (ICT) is a vital part for organisations, not only for those whose business model is based on ICT but also where ICT is supporting business functions. But ICT has always faced a lot of changes. In order to see that one just has to take a look at the history of programming languages that were developed ever since (ORM10). Or one can turn to the well known Gordon Moore's law predicting that the number of transistors placed on an integrated circuit would double every two years. A law, that is still valid, and is now supposed to stay valid until the year 2015 (CNET05). Of course not all of the languages made it into production systems of companies, but that is not even necessary as only the changes in paradigms and their respective architectural impact would already have been enough for an ever changing IT environment.

After all, it is still the business requirements that have to be supported by ICT and they are – also due to the new opportunities that ICT offers – changing fast! This results in a huge challenge for organizations to keep up with the pace of change and to stay competitive – a task, which is often impeded by the IT infrastructure already in place within a company.

Because unfortunately it is never as easy as just unplugging a machine and plugging in the new one or replacing one piece of software to be ready to go ahead. IT infrastructures in organizations have grown to complex systems that need careful attention especially when we speak about the necessary migrations of such structures. The goal of this thesis is it to look into this matter in a holistic way and offer a combined process model based on already existing approaches and insights. For that various perspectives have to be taken into consideration to achieve a holistic overview of the involved challenges. Furthermore the interrelations between those findings will be pointed out to then being able to suggest generic strategies and a model for the migration of IT infrastructures, which aim to minimize the involved risk.

2. Abstract

This paper is proposing a holistic process model for the migration of legacy infrastructures. To achieve this objective it is therefore necessary to investigate and define what legacy architectures and their related problems are. After that the established migration approaches to address the legacy challenge and an overview of current and ongoing architectures and trends form a broad enough headstone to then go on discussing which methodologies and means are available to facilitate a smooth migration. These are spanning from the human factor via risk control to economic matters related to system transformation projects. Finally the migration reference process ReMiP and the enterprise architecture framework TOGAF are introduced and combined to consolidate the presented approaches. The closing chapters compare the expected benefits with the requirements of a practitioner, propose a reference migration model and finally summarise the proposed approach.

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Erstellung eines holistischen Modells für Migrationen von Alt(last)systemen in der IT. Dafür ist es notwendig zuerst festzustellen was sogenannte Legacy Systeme überhaupt ausmacht und welche Probleme diese mit sich bringen. Danach werden bestehende Migrationsarten und -vorgehensweisen erörtert und aufgezeigt welche Architekturen und Trends momentan auf dem Markt vorhanden sind. Um die im letzten Abschnitt der Arbeit vorgeschlagene Migrationsstrategie zu unterstützen werden Methoden zur Risikokontrolle, zur Einbeziehung von Stakeholdern und zur wirtschaftlichen Begründbarkeit eingeführt. Schließlich werden der Referenzprozess ReMiP und TOGAF als Framework für Enterprise Architecture vorgestellt und eine Kombination der beiden vorgeschlagen und in einem Referenzmodell beschrieben. Die erwarteten Vorteile durch die in dieser Arbeit diskutierte kombinierte Anwendung werden von einem externen Experten aus dem Arbeitsfeld von IT Migrationen nach ihrer Anwendbarkeit beurteilt.

PART A – ANALYSIS AND DISCUSSION OF THE FOUNDATIONS ON WHICH THE CONTRIBUTION OF THIS MASTER THESIS IS BASED

3. What drives the migration of IT architectures?

The migration of IT infrastructures is usually dealt with in dedicated projects; especially if the topic of evolving legacy systems has been neglected for quite a while in an organisation. It is a common procedure – and the basis of appropriate risk management - to find out about the impacts a project will have on the company because especially in the case of the migration of IT systems the far-reaching consequences are easily underestimated. Table 3.1 shows interrelations in business and IT architectures together with their corresponding drivers and objectives. This helps to understand how closely related both domains are.

Area	1960-70	1980-90	1990+	1995+	2000+	2010+
Market imperative	market shares	efficiency	decentralisation	customer retention	real time enterprise	service oriented enterprise
Architecture	mainframe	module	client-server	application server	SOA	SOA
Objective	economy of scales	business process reengineering	business applications	customer retention	disentanglement	service economy
Driver	status quo, no scalability	falling CPU-costs	PC and networks	www	web services	semantic services

TABLE 3.1 OVERVIEW OF BUSINESS AND IT ARCHITECTURES (DIMA07)

Table 3.1 (DIMA07) shows for example that a mainframe was the technical solution for being able to make use of the economy of scales whereas during the nineties of the last century the emergence of PCs and networks provided a means for more decentralisation. It is hard to tell whether in all these cases the market imperative was dictating how information systems architectures should look like or whether technological improvement only made certain market imperatives possible. In general it is only important to keep these strong interrelations in mind when migration projects take place. But apart from this more global point of view there are also interrelations within a very organisation that need to be unfolded before launching a migration project. (WIUL02) therefore formulates that holistic thinking is essential and will address issues like a lack of project coordination from an enterprise-wide project

perspective or the tendency to gravitate toward quick and dirty solutions to complex problems like executives mandating oversimplified and unfeasible solutions. And for example (DIMA07) proposes that for the evaluation and the development of big software systems also psychology, sociology and system theory will be needed. The subsequent chapters now follow these propositions whereas Part A is focusing on determining what legacy architecture is comprised of, how it is currently addressed and what possible current target architectures and trends should be considered. Part B then is introducing how the challenge of legacy infrastructures can be addressed and proposes a solution on existing frameworks and approaches.

4. Legacy languages, systems and architectures

The term “legacy” turns out to be a highly emotional yet in practice quite narrowly defined term. It is easy to for example think about legacy infrastructure as mainframes running COBOL. And though this of course holds true in most cases there are several defining characteristics. The aim in this chapter is therefore to show what can be called legacy and why it is such a big challenge.

4.1. What actually is legacy?

It is often the case that old IT systems in companies are called as legacy systems – which in most cases will be true – and due to non-adaptability some legacy systems have already extinct and the ones still being around are those who at least could be customised to some useful degree over the decades. But there are some defining characteristics. (DIMA10) defines a legacy system as a system with social and technological influence that contains legacy software. Whereas legacy software is business critical software that cannot or can hardly be modified. As introduced in the last chapter it is therefore necessary to put such a system into the context of the respective organisation with whom the system has been evolving in parallel with over the time. As this evolution often doesn’t follow a certain planned concept the term of an accidental architecture is also quite commonly used (LILA99). Many big IT systems in major companies originally date back 30 years from now but are not at all the same anymore as back then. They generally incorporate not only old software but have become a mixture between aging software and more current one because they had to adapt to changes of the business model and are therefore way more complex as originally intended. As a result we speak then of the petrification of an information

system (BRST95) where repeated waves of hacking take place in order to modify and adapt the software and will lead them from the former flexible to a more and more inflexible architecture (LAKE00). And that is already bringing up another problem as within that time span the changes regarding business rules, processes or exceptions to such a system have been programmed by different generations of software engineers who may on the one hand not be employed in the company anymore or who can – if even – only tell about their own work but not about the adaptations of engineers before them. (LILA99) raises the question whether legacy systems therefore can be more seen as a company's asset or its liability. The answer follows as them being both at the same time. That is because the software concerned has been well debugged over the years and as the business and systems processes have grown and evolved together during the time the software not only supports the explicitly understood business processes but also incorporates so-called tacit knowledge that reflects the organisation. This asset viewpoint has the liability counterpart of the significant maintenance costs together with the risk that important and even business critical knowledge might get lost in a replacement effort.

4.2. Typical examples of legacy systems

The above-mentioned legacy issues appear in different typical variations and representations of tiers and mechanisms. The following sections describe which different types of data storage and which kinds of processing logic usually can be found. But it is also important to understand that the mere existence of one or more of these examples within a system do not necessarily already make up a legacy system. This matter will be addressed in later chapters.

4.2.1. File based architecture

In this group all data exchange is organised through files. Starting in the beginning with quite small applications that would only access a single file in batch mode to sort data or look up certain values. The values are therefore stored in column like structures. Every single entry in those file is called a record that is either “equipped” with key or not. Usually COBOL – which will be discussed later on – is used for processing those files and also features commands for key based files. In this original set-up everything worked out well but as the systems have grown problems emerged. Because extension either meant running multiple instances of an application on the same file or letting different applications access the same files. In

the first case it must be assured that parallel instructions of the different instances are serialised which unfortunately leads to longer runtime because the second program has to wait for the first program to release the lock of the file. A second solution is introducing transaction as the means of processing and setting up a transaction monitor that will handle those transactions avoiding inconsistencies and ensuring serialisation. The second case additionally poses the problem that different program logics usually ask for different data structures. That led on one hand to generating redundant copies of the original records for different applications and on the other to an increasingly poor quality of data because – if even available – data types could only be set for key values but not for the rest of the record.

4.2.2. Database centric architecture

After saving data into files there came the idea of using databases for that matter. A database consists of a logical data model that describes the semantic of the domain and a database management system (DBMS) providing the infrastructure for the database and the features of administrating and accessing the data. DBMS are application-neutral which means that they can be used by different applications. A thing that all databases have in common is that they already implement serialisation and resource control and due to transaction protocols already show good protection against data failures. We distinguish three different types of databases:

4.2.2.1. Hierarchical databases

The model for a hierarchical database is a tree structure that offers a parent-child relationship between a data segment, eg. an employee or a task of a company, whereas a parent can have zero to many children. The major inconvenience with this representation of course lies in the lack of freedom to create many-to-many relationships between data fields of the database. So it is legitimate to assign zero to many tasks to an employee but it would for example not be possible to describe that two employees of a company are working on the same task.¹ Access to a hierarchical database is usually done by searching a key in the segment and then read the segments sequentially. The entry point for the access is always the root element.

¹ Actually it is possible via introducing each others key in the related records but it's not very practical.

4.2.2.2. Network databases

Network databases – also called CODASYL (Conference on Data Systems Language) break up this strict structure mentioned above and can therefore represent orthogonal relationships apart from hierarchies too. The task from the example before that is saved in a network database could now be assigned to more than just one employee. The entry point for an access can be each segment of the database and it is also possible to read segments backwards.

4.2.2.3. Relational databases

The idea of relational databases goes back to Codd who in 1970 came up with the following characteristics that databases should have

No more hierarchies – the different segments, called relations, are all on the same level and connected via foreign keys

An own data type schema – independent from applications and mandatory during runtime the data type schema is structural part of the data model

Set theory and set operations – hierarchical and network databases are planned for single access where one segment is read. A relational database offers the possibility of using set operators that will allow to read a set of segments which are called relations in this case

So a relational database is comprised by a set of relations, also called tables, and their relationships to each other. In contrast to the so far mentioned database types and due to performance reasons there is a differentiation between batch and online applications accessing the database. Relational databases use indices to establish the connection between key values and the physical memory address of the corresponding tuple that represents a relation filled with concrete values or a row in a table. A rule of thumb says that the more indices exist the faster data can be read and the less indices there are the faster tuples can be modified.

4.2.3. Transactions

Other than in the real world where transactions are used for exchange of goods and money, transactions within a software system represent modifications most common in databases. Whereas all modifications of a transaction will be accepted or denied to ensure that different instances of an application can use the database as if they were the only ones accessing it. To make that possible transactions have certain guaranteed characteristics referred to as ACID characteristics

Atomicity – the transaction is either carried out completely or not at all

Consistency – a transaction leads the database from one consistent state into another obeying all structural constraints

Isolation – every temporary consistent or inconsistent state is not visible for other transactions

Durability – the result of a transaction will persist independent from hard- or software failures

4.2.4. Teleprocessing monitor architecture

The teleprocessing monitor architecture is used for managing a big number of terminals seemingly at the same time. Therefore the input of the different terminals is buffered in a message queue and will be processed sequentially. TP- systems are re-entrant which means that the applications are stateless and need a database to re-establish the last state. The applications are subroutines that are called from the TP-Monitor and are running in the same address space, which can cause the whole TP-Monitor to crash if they themselves crash. The application then takes the data out of the message queue, processes it and writes the data into an output queue. The TP-Monitor is in charge for the communication between the terminals or clients with the application.

4.3. Legacy languages

When the last chapter was aiming at data storage possibilities and actions based upon them, this chapter sets its focus on the involved programming languages. After a look at some of the traditional legacy languages it is even more interesting and important to generally find out what constitutes a legacy language.

4.3.1. Traditional legacy languages

As mentioned in the very beginning there is a vast variety of programming languages, which are usually grouped into four generations.

Starting with the First-Generation Languages (1GL), which only offered close to hardware, binary programming, there followed the Second-Generation Languages (2GL) using symbolic addresses and specific to a particular processor family like assembler. Third-Generation Languages (3GL) are also called high-level languages that refine 2GLS and emphasize more to be programmer friendly and let the computer take care of the more non-essential details such as memory management.

Representatives of 3GLs are classic languages such as Pascal, COBOL, Fortran or PL/I but also modern languages such as Java, C ++ or C#. Fourth-Generation Languages (4GL) form another additional abstraction level and usually special purpose programming languages like SQL for databases or analysing and reporting languages as ABAP. Fifth-Generation Languages (5GL) finally are logic programming languages to solve problems by giving constraints. Prolog would be an example for a 5GL.

Although legacy software is usually better described through its age and maintainability there are typical languages in which legacy code is written.

4.3.1.1. COBOL

COBOL (Common Business Oriented Language) is a programming language domain specific for the use in business applications and was introduced to be better human readable as Assembler. Featuring the essential functionality for the development of business applications it was (and still is) very popular and prevalent though regularly pronounced dead. “There are over 220 billion lines of COBOL in existence, a figure which equates to around 80% of the world’s actively used code.” (KEMI09) So it is pretty obvious that COBOL will accompany us for quite awhile, together with its characteristic of not being able to separate variable declarations from type declarations, its lack of structure in general and its global variables. All of which makes given COBOL code hard to decipher.

4.3.1.2. C

The programming language C is a widely used language for technical software and system development as in Unix, which is because of the possibility to work with memory addresses and links. That is also a reason for C being quite error prone and the most attractive target for security attacks such as buffer overflows. Another reason that makes it hard to deal with C is that the language of the pre processor is agnostic to the grammar of C. The header files help to produce better-structured code because the programmer can take care of definitions and constants separately but due to the different language grammar it is also easy to misuse and create hard-to-decipher header code.²

² What really is possible one can see in the results of the The International Obfuscated C Code Contest on <http://www.ioccc.org/>

4.3.1.3. Fortran

Fortran – former named as FORTRAN (Formular Translation) is the first high level programming language and has since the development by John Backus undergone several extensions that added it up to its latest version Fortran 2008 with different new language constructs, changed it to work with modern computer architectures and made it object oriented. Problems in reading Fortran again arise from global variables but also from redefining variables implicitly in the COMMON blocks in Fortran.

4.3.2. Future legacy languages?

Having learned what the current legacy languages typically are, it of course is interesting which of the current languages will be our future legacy languages. A bold answer would be that every language in use nowadays will become a legacy language in the future. But a more detailed answer lies in having a look at the volatility of a programming languages but also information systems on the whole.

4.3.2.1. Volatility index

Volatility is the risk of a system to be sensitive to changes. A related term would be stability. The more stable a system is the less it is influenced by small changes and therefore has a lower volatility. Stability in this matter means the long-term stability of the business process logic. So the system should for example be able to cope with short-term changes in user interfaces or adding new attributes to a system. The Volatility index V is defined as shown in Fig. 4.1. (DIMA10)

$$V = \frac{\sum C_{structuralchange}}{\sum C_{structuralchange} + \sum C_{physicalchange} + \sum C_{userinterfacechange}}$$

FIG. 4.1 FORMULA FOR THE VOLATILITY OF A SYSTEM (DIMA10)

The Volatility index measures the cost C of the respective change. The index is described in an indirect way because its basis is the cost for restoring and assuring the workability of the whole system, because a structural change usually also creates the need for physical and user interface change.

$C_{structuralchange}$ are cost for changes in the business process logic, so to speak the core of the system. They are also called costs of deep structural changes.

$C_{physicalchange}$ is the effort in terms of money for introducing new attributes or for example database systems without a fundamental change in the business process logic.

$C_{userinterfacechange}$ represent the cost for changes in the user interface and comprises also very small changes to the system.

The Volatility Index is primarily measured through cost. Empirical findings also proof though that it can also be measured as a function shown in Fig. 4.2.

$$V = \mathcal{F}(t_{programminglanguage}, t_{systemage}, s_{linesofcode})$$

FIG. 4.2 EMPIRICAL VOLATILITY INDEX FUNCTION (DIMA10)

$t_{programminglanguage}$ represents the age of the used programming language which indicated as follows – Fig. 4.3.

Language	Value
Assembler	1
COBOL	2
Fortran	2
4Gl	3
Java	4
C++	4

FIG. 5.3 VOLATILITY OF PROGRAMMING LANGUAGES (DIMA10)

Fig. 4.3 shows that older programming languages are classified by lower numbers, e.g. Assembler shows age value 1, and younger ones are classified by higher numbers, e.g. C++ shows age value 4.

$t_{systemage}$ indicates the average age of the applications within a system.

$t_{linesofcode}$ tells about the size of a system by counting the lines of code of the whole system.

Also a linear relationship was found described in Fig. 4.4 whereas the correction parameter ϵ turned out as being small.

$$V \approx c_0 + c_1 t_{programminglanguage} + c_2 t_{systemage} + c_3 s_{linesofcode} + \epsilon$$

FIG. 4.4 LINEAR EMPIRICAL RELATIONSHIP OF THE VOLATILITY INDEX (DIMA10)

The size of a system $t_{linesofcode}$ correlates with the amount of nodes of a system and given a more or less homogenous environment $t_{programminglanguage}$ and $t_{systemage}$ can be considered constant. That leads to an approximation of the empirical Volatility Index described as shown in Fig. 4.5.

$$V \approx c_0^* + c_3^* n_{nodes}$$

FIG. 4.5 VOLATILITY INDEX APPROXIMATION (DIMA10)

The term “nodes” comes from the approach of representing enterprise architectures through graphs whereas the edges between the nodes are service-calls towards nodes. Nodes can therefore be seen as applications. So in this approximation the volatility of the system increases through the number of applications.

The first way of calculating the index describes quite clearly that the volatility of a system depends on how many deep structural changes occur to it whereas the latter definition of volatility makes it obvious that volatility is strongly related to the size of a system. The more applications, the higher the Volatility index.

If a system reaches certain value of volatility the cost for maintenance will increase dramatically. A way to keep the volatility index lower is to choose a programming language with a lower Volatility index. JAVA for example has a lower Volatility index than COBOL because it has a higher degree of semantic information through libraries that facilitate user interface development and also physical change. But even deep structural change is supported better which leads to a higher numerator in the equation of Fig. 4.1 but at the same time this is compensated through the simultaneous increase of the denominator. The result is that the Volatility index will stay low for a longer time. Fig. 4.6 shows the comparison of COBOL and JAVA in their Volatility Index.

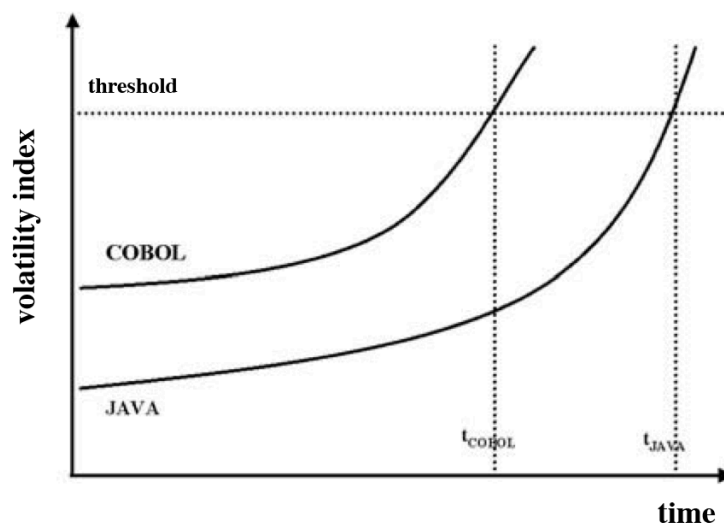


FIG. 4.6 VOLATILITY INDEX DIAGRAM FOR COBOL AND JAVA (DIMA10)

COBOL reaches the point of excessive maintenance cost way earlier than JAVA does. But also JAVA systems will sooner or later hit that the volatility threshold where their maintenance cost will explode. Software in general follows a volatility curve presented in Fig. 4.7.

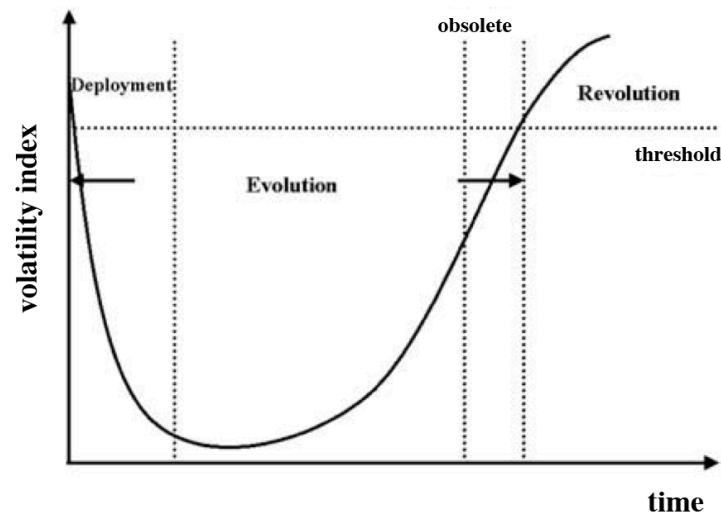


FIG. 4.7 VOLATILITY INDEX DIAGRAM FOR SOFTWARE (DIMA10)

In the deployment phase of new software usually bigger structural changes take place until the software fully provides the required and planned functionality. That leads to a high Volatility index. Afterwards the typical evolution phase starts until in the end it will reach a phase where the software becomes obsolete and has to be replaced in a revolution phase. A large enterprise architecture of course consists of many separate software packages that will undergo this lifecycle which leads to small peaks in the overall Volatility of the whole EA. Still, in the long run the volatility curve of the EA will follow the same schema as shown in Fig. 4.7. The topic of software evolution will be discussed in chapter 5.

4.3.3. Lessons learned

On the one hand code will always become legacy at some point and on the other hand even the least volatile language will still become legacy if we fail to understand how to properly use the language or better the paradigm and architecture behind the language. This means for example that an object-oriented programming language of course will only show the desired behaviour if the programmer uses it in the right way like realising encapsulation and modularity. Talking about a suggested architecture, e.g. using JAVA in the context of a J2EE architectural design, the developer but also the enterprise architect of course have to understand the architecture and have to have a clue how it has to be implemented.

4.4. Common Anti-Patterns within Legacy Systems

The chapters so far only showed examples and so to say the bricks that build legacy systems. And as of course they can and do constitute part of the problem they are not abstract enough to describe the problems of hard-to-adapt systems. The following two chapters therefore take a look behind the scenes and discuss what leads to undesired, petrified systems on the one hand and which typical traps for software architects there are on the other hand. Because although good enterprise and software architecture is there to maximise adaptability and flexibility, the described weaknesses in the architecture just make the situation worse.

Both viewpoints are described as anti-patterns whereas anti-patterns in contrast to patterns show how undesired effects are building up, what these negative effects are and how they can be avoided.

4.4.1. Anti-Patterns in legacy systems

This first group of anti-patterns now address how systems end up being inflexible and far from being as adaptive as they actually should be. (LAKE00) found six of such in legacy systems.

Anti-pattern	Brief description of the problem
Ball and Chain	lack of portability to other platforms because software being too strong tied down to the operating system
Tower of Babel	lack of inter-operation due to programming languages that can't communicate with each other
Monolithicity	lack of componentization and reuse
Gold in them thar Hills	lack of understanding of the underlying system because of the tacit knowledge that is scattered in the software and therefore bad adaptability to changes in business processes
Tight Coupling	lack of loose coupling makes systems heavily depending on each other and sensitive to changes in each other hindering maintainability, adaptability and re-use
Code Pollution	the problem-domain-oriented code is usually wrapped in guard-code that leads to tight coupling to the system. This

	guard-code pollutes the code space and hinders maintainability, adaptability, understandability, and reusability.
--	---

TABLE 4.1 OVERVIEW OF ANTI-PATTERNS IN LEGACY SYSTEMS (LAKE00)

Each of these anti-patterns – see Table 4.1 – counters a desired pattern that should avoid the undesired effects described above. Whereas the first three presented patterns will mainly address the issues of portability and interoperability and form an appropriate interim solution but are not helping to make the system more adaptive. They represent the so-called external view of the system.

1. Portability adapter

To solve the problem of software being too tied down to operating system's services a portability adapter, in (LAKE00) called the Fundamental Service Layer (FSL), should prevent direct calls to the operating system and therefore offering a single and consistent interface improving the re-hosting of a system to different platforms.

2. Babel Fish

Quite often ad-hoc one-to-one connectors are used for letting applications written in different programming languages inter-operate. Over time an organisation usually ends up with uncontrolled growth of these bridges that create a complex system that is hard to maintain. A single middleware solution, called Babel Fish, is the solution to only provide one bridge to each legacy sub-system.

3. Virtual Componentization

This pattern follows the idea that components within a system are easier to understand and re-use as monolithic sub-system. But although componentization is the objective it can hardly (at least not facing a reasonable level of risk) be achieved in one step starting from a monolithic architecture. The virtual componentization starts with the definition of desired components and their interfaces and after that follows by implementing a Façade just on top of the legacy systems. Later on real components can replace the virtual ones.

The following three patterns now build the internal view of the system.

4. Gold Mining

In this approach Business Action Theory (BAT) of the so-called Language Action Perspective (LAP)³ is used to find out information about the domain the information

³ http://www.fsc.yorku.ca/york/istheory/wiki/index.php/Language_action_perspective

system participates in. (LAKE00) proposes to set up three different workshops. The Ontology Workshop is used for establishing a shared understanding of the business process and it is itself further divided into three workshops. The gold mining workshop tries to find out as much as possible about the tacit knowledge and to describe it explicitly. A main means for this is intensive code inspection. In the next workshops, the process reconstruction workshops, this gained knowledge is refined and reconstructed. This means that terms used in the system are for example changed to more appropriate business ones and that in general the processes found in the gold mining workshops are described more from the stockholder's view than the legacy system's one. This description of the current state of business processes is used as the input for the ontology refactoring workshops in which a common understanding of the goals and the problem to achieve those goals is established. Through discussions solutions should be found and negotiated that form the new business ontology. The then following requirement workshops use this ontology to negotiate the requirements for the new system by defining how the system can and will support the business processes. During negotiations both developers and stakeholders are present to ensure that the contemplated future system is feasible in realisation and on the other hand not solely technology-centred. The developers then go on to hold development workshops in which they negotiate designs and prototypes based on the before identified requirements. Other requirement workshops will still take place to evaluate prototypes and adapt requirements.

5. Implicit Invocation

When methods are calling each other in an explicit way they become sensitive to changes in the methods they are relying on. This can lead to unwanted behaviour in case one of the methods changes and makes it harder to proof that for example a patch in one part of the system does not have ramifications on other parts of the system. And the more explicit invocations are used the more likely that such side effects on other parts of the software can occur. To achieve implicit invocation brokers for components are required that will manage events that each component is interested in. This de-coupling leads to a greater freedom of adaptability, maintainability and reusability.

6. Explicit protocol reflection

Interfaces of components indicate in which events a component is interested. As there is often a time constraint of when a component is interested in which events

guard code is introduced to prevent such things like a lock on a component that already is locked by another event. This guard code is usually not kept in one single place and therefore pollutes the code and the situation gets worse as the component grows. As a replacement state charts can be used that describe in which state the component is interested in which events.

4.4.2. Anti-patterns of Enterprise Architecture

The former chapter describes anti-patterns in an existing system and deals with the problem on a more technical level. In this section right now captures the anti-patterns in higher abstraction level; in the implementation of enterprise architectures (EA) (AMSC10) and (DIMA05). These examples endanger the benefits of an EA like better alignment of business, IT architectures or more specific the information system architecture.

30.000 feet and climbing – the EA has been defined in an abstract and generic way so that development lacks information to even realise it at all. In order to prevent this anti-pattern a reference architecture should be in place and particular proof-of-concept projects should be realised.

Stuck in the weeds – this can somehow be seen as the counterpart of the before mentioned anti-pattern. It describes an EA that is far too detailed and comes from the architect's need to control everything or the idea that a perfect model leads to a perfect result. But the result usually is late projects and a technology that is already outdated, when the projects finally finish. Just as before a reference architecture and a proof-of-concept project help along with a strict separation of roles into the architect and the designer of the applications.

Ivory Tower Architecture – this case is related to the first anti-pattern, as it describes an EA that looks good as a model, but is completely impractical or does not really consider the environment, in which it will be deployed, and the business requirements. Reasons why an EA can end up like this usually lie in the fact that the EA-team does not pay attention to feedback and imposes their model onto the developers. Again a reference architecture, a proof-of-concept-project and clear EA team-management can help to prevent this undesired situation.

Strive for perfection – in this example one prototype follows another but in the end no roll-out of the actual EA is ever achieved. The reason for this behaviour of an EA team is that prototypes, even if they are all together successful and working, provide

an excuse if things should go wrong with one prototype, because it is just a prototype. Rigid project management and clear approval of deliverables should provide a solution against this.

Bleeding edge – if the EA team always tries out new technologies that are not mature and stable enough to deploy or / and too little knowledge and skills exist within the organisation for this certain technology EA projects can end up being an example of the bleeding edge anti-pattern. Knowing the skills within the company and sticking to what is really needed to support business will help preventing this problem.

Technology above all – especially in highly technology-oriented organisations it can happen that enterprise architects tend to believe IT is the business driver instead of the business enabler.⁴ Therefore special attention should be paid that IT follows the actual business requirements.

Gold plating – this “approach” leads to the situation that the effort put into the EA is way bigger than it would need to be. This means that the architecture becomes “overbuilt” by adding “nice-to-have” features that are not really necessary for what the system is supposed to do. Again the EA team should be reminded to stick to the requirements.

Silver-Bullet – the Silver-Bullet anti-pattern is probably the widest spread of all. It describes the view that the right tool is capable to solve (even complex) problems immediately. An assumption that is always wrong as complex problems in software systems always have come from organisational problems too. A tool will then support the solution as soon as the cause for the problem is analysed and identified. This anti-pattern is one of the most dangerous as the notion of a Silver-Bullet solution is fed by vendors of software products that will not get tired to assure that their latest product described through the latest buzzword will solve all problems. Well-known examples are EAI or the introduction of ERP systems that by far could not keep their promises of integration or simplification, but even worse caused damage to the systems they were introduced to. And the damage can go as far as causing a company bankruptcy (ERCH09).

⁴ It can happen though that technology like applications for handhelds e.g. the iPad and other tablets can turn out to be business drivers too.

Apart from these anti-patterns there is one more insight that an enterprise architect should have heard about.

Conway's Law – in 1968 Melvin Conway discovered a relation (COME68) between the organisation of organisation structures and the design of their products. *“Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure.”* It is easy to guess that companies with big and inflexible organisational structures will tend to produce software or will establish an EA showing the same undesired attributes. So also the organisation of the project team can already have impact on the system to be developed.

All the presented anti-patterns should help to show on the one hand how easy it is to go astray within EA but on the other hand they provide a good means to reflect and review the work of an enterprise architect.

4.5. Typical Concepts to deal with Legacy Architectures

The so far discussed topics were describing what we are dealing with when we talk about legacy and which reasons and traps there are for something to be called a legacy. And although the anti-patterns already propose punctual solutions for the detected problems the challenge is to address the problems system-wide. Vendors long sold the following approaches as silver bullets and promised that legacy systems can be “tamed” through them. But although they are eligible and useful and can even be part of a migration they do not replace a migration and an overall EA strategy. These most famous but quite arguable approaches will now be discussed together with their shortfalls.

4.5.1. Enterprise Application Integration (EAI)

The term EAI generally is used for the integration of heterogeneous applications within and extension to IT-systems of an organisation. The aim of EAI is to loosely couple the applications and sub systems concerned. The challenge lies in the integration of the syntax, the semantic and the business processes. We distinguish three different kinds of EAI types:

Data-integration – Data-integration either means connecting applications by letting them exchange data or shared usage of databases by separate applications. A main problem poses the different data models for different programs because of different

data types or constraints within those data types. Also data quality is major issue for data-integration.

Business-Object-Integration – To achieve integration through business objects uniform object models for each business object have to be defined and described through its class, interfaces and exceptions.

Business-Process-Integration – Through business-process-integration completely separated and heterogenic applications are integrated by defining business process rules for the execution of a concrete workflow.

EAI solutions offer three kinds of services to address the above-mentioned types of integration - interface services, transformation services, and business process services. Whereas the interface services are used for data-integration, transformation services build the basis for business-object-integration and at last business process services are the vehicle for business-process-integration.

EAI in general stands for a reasonable priced effort to integrate functionality from a legacy system without having too much of an impact on the legacy system (WIUL02). Through EAI stovepipe applications can be combined to support business processes better. Because of this it has been widely accepted as a way to deal with legacy applications but it has drawbacks especially if implemented without a proper EA in mind. The general problem is that the applications that should be integrated through EAI usually are not based on public but on proprietary standards. Web services based on SOAP will maybe provide a good means for implementing EAI but at the time EAI is still struggling with typical problems of integration matters. Just as seen in the development of programs EAI led to applications that are linked directly and indirectly, resulting in a so-called spaghetti architecture as shown in Fig. 4.8.

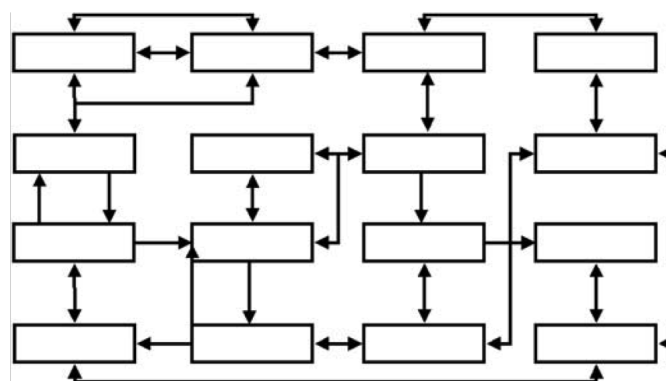


FIG. 4.8 SPAGHETTI ARCHITECTURE OF EAI (DIMA05)

As business processes were changing the requirements for the EAI changed too and new interfaces had to be established between applications. Doing so the flexibility of

the whole system decreased dramatically as again more and more dependencies between applications were created and no one could be assured that by changing code in one place would not lead to a changed behaviour in another function of the system. And as with every complex system the cost for maintenance exploded. An additional reason for this uncontrolled growth also can be found in too complex interfaces within the EAI so that developers tended to create their own new interface instead of taking the time trying to understand the existing ones. The effect was a lack of re-use of interfaces. Needless to say a system like that does not provide scalability – neither in hard- nor software matters.

If EAI is realised through web services it will become useful but nowadays in general EAI should still be seen more as an intermediate than a long-term solution that should be carefully implemented following an EA. If this is not the case, organisations will end up with spaghetti architectures that rather complicate a migration than facilitate it.

4.5.2. Message Oriented Middleware (MOM)

MOM represents a loosely coupled and asynchronous solution to let applications remotely communicate through messages and via a message queue manager. Other than with remote procedure calls (RPC) where applications need to know the application that they want to reach, applications using MOM are indirectly connected. Instead of the particular application names the sending applications just sends a message to the message queue manager using logical queue names that then help the message queue manager to decide to which queue a message will be routed. The receiving application will then take the message from this queue, his input queue, and process it. Depending on the degree of certainty about the deliverance there will be a fast best-effort quality of service (QoS), a slower but more reliable guaranteed QoS or a transactional QoS that will ensure complete delivery or failure of parts of messages. For MOM it also not a pre-requisite that the “called” application has to be running at the time the message is sent. The message queue manager either will just put the message in the appropriate queue and the application will receive it from there when started again or it will start the application if needed.

But in the end MOM suffers from similar shortcomings as EAI. As standalone solution outside from an EA they usually just add yet another component to the system and will not deliver the necessary performance in large systems.

5. Migration of legacy systems

After this thorough look into legacy systems and the already first hints on how to meet their challenges we now turn to the proposed solution of migrating IT systems. But the term of migration itself will be discussed after pointing out why migrations are necessary.

5.1. Lehman's Laws of Software Evolution

As already mentioned in the discussion of the volatility index in chapter 4.3.2.1 software evolution is a key theory behind the necessity of maintaining and migrating IT systems because it describes how and why software systems change. Lehman and Belady (LEBE85) found reasons for the change behaviour of software and proposed the so-called Lehman Laws of Software Evolution as shown in Table 5.1, whereas an e-type system represents real world software⁵.

No.	Brief Name	Law
I 1974	Continuing Change	<i>E</i> -type systems must be continually adapted else they become progressively less satisfactory.
II 1974	Increasing Complexity	As an <i>E</i> -type system evolves its complexity increases unless work is done to maintain or reduce it.
III 1974	Self Regulation	<i>E</i> -type system evolution process is self regulating with distribution of product and process measures close to normal.
IV 1980	Conservation of Organisational Stability (invariant work rate)	The average effective global activity rate in an evolving <i>E</i> -type system is invariant over product lifetime.
V 1980	Conservation of Familiarity	As an <i>E</i> -type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behaviour [leh80a] to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.
VI 1980	Continuing Growth	The functional content of <i>E</i> -type systems must be continually increased to maintain user satisfaction over their lifetime.
VII 1996	Declining Quality	The quality of <i>E</i> -type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.
VIII 1996	Feedback System (first stated 1974, formalised as law 1996)	<i>E</i> -type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

TABLE 5.1 LEHMAN LAWS (LEBE85)

The first law is establishing that continuing change must take place in a system to ensure that the system stays satisfactory. This is because a system needs to reflect the changes in its environment to not become less useful. A side effect is that through such modifications in the software its environment will also be changing restarting change in the system all over.

⁵ The whole universe, all its properties and everything that happens in it comprise the real world.

The second law states that in changing the system structure will deteriorate leading to increasing complexity. Therefore steps must be taken to maintain and reduce complexity.

The third law suggests that evolution in IT systems is based on self-regulation. This means that attributes like size, time between releases and the number of reported errors will be constant for each system release. The reason for this behaviour lies in the size of the system. A growing system will at some point enter a stage of inertia as its size and therefore complexity per se inhibits vulnerability for faults degrading the system's functionality. Moreover large systems are usually created by large organisations inhibiting bureaucracy steps for each change request. The change rate is therefore dependant on the system size and a company's decision-making process.

The fourth law states that the rate of development for a large system is approximately invariant over its lifetime and not depending on its resources for development. A reason for this behaviour is that large software development teams are creating an overhead spent on communication leading to less productive output – also described in (BROO75).

The fifth law again deals with a constant incremental change over the lifetime of a system. In this case the necessity for it comes from people concerned by the system – spanning from developers over marketers to users – must stay familiar with the system and its functionality.

The sixth law states that a system must keep on adding functionality to satisfy its users leading to continuing growth over lifetime.

The seventh law suggests that quality within systems will be declining if rigorous maintenance toward operational environment changes is neglected.

The eights and last law describes the evolution process as multi-level, multi-loop and multi-agent driven because organisational and societal – even more than - technical factors are influencing this global evolution process within the whole software development process. These influences include many individuals at different usage levels and concern different parts of the system.

As already mentioned in chapter 4.3.2 looking at the evolution of EA also helps to understand how software becomes legacy, whereas evolution of software means continuous, quasi-infinitesimal, incremental changes in software (DIMA05) or can be

seen as the process from creating to shutting down the system passing through the phases of development, maintenance, migration and shutdown (WERE08). Software evolution is comprised by maintenance, software enhancement by adding functionality and lifecycle-enablement, which sums up all activities that make maintenance and software enhancement even possible. This could for example be that developers have to make themselves familiar with a certain used technology. The revolution of a system is on the other hand the counterpart of evolution and is characterised through a catastrophic and spontaneous change of the EA. Such a change can happen because of the introduction of a new architectural principle or a new development paradigm and is visible in the non-continuous discontinuity of metrics such as entropy – compare Fig. 5.1.

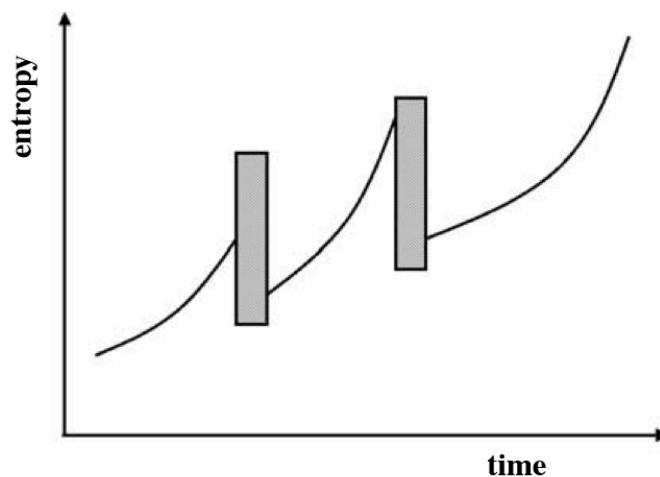


FIG. 5.1 NON-CONTINUOUS DISCONTINUITY OF ENTROPY CURVE OF A SYSTEM (DIMA05)

The continuously increasing entropy curve of a system that is based on evolutionary actions gets interrupted through a revolution that is taking place in the system, whereas entropy is a metric that describes the degree of dependencies or coupling and orthogonal disjunction or cohesion in a system. Maintaining or evolution in general leads to higher coupling and lower cohesion within a system and as we already know from chapter 4.4.1 to undesired side effects like blurring architecture, design and modularisation of a system (WERE08). Migration of legacy systems would represent a revolution to the system, as will be discussed in chapter 5.4.

5.1. Possibilities of evolving legacy systems

After having established why evolution in software system takes place it is now time to describe how evolution takes place. There are three types of evolving a system

that will occur within and influence the information system life cycle as shown in Fig. 5.2 (SEPL03).

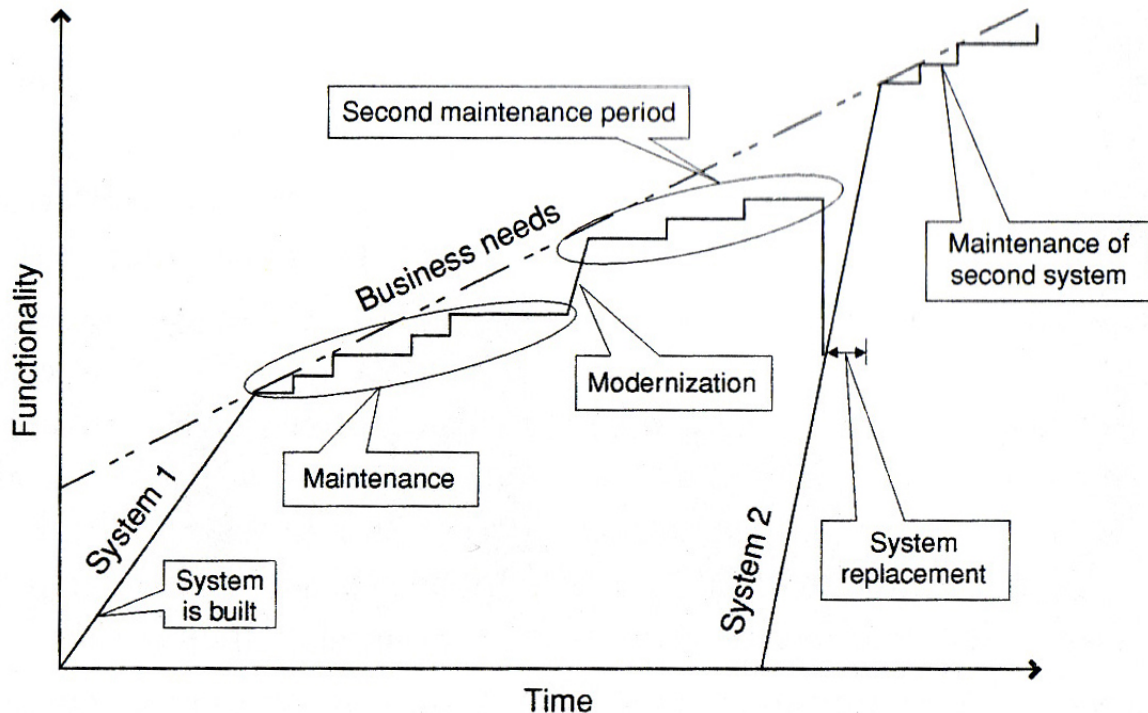


FIG. 5.2 INFORMATION SYSTEM LIFE CYCLE (SEPL03)

After a system is built it (ideally) meets the needs of the organisation for some time and goes through maintenance to support the changing requirements. Sooner or later it will fall short on keeping up with the pace of change request and will need to undergo a bigger adaptation effort and a so-called modernisation⁶ of the system will be applied. After that there usually will follow phases of maintenance again but at some point it won't be possible to evolve the system any further and in the end it will be replaced. Of course this schematic life cycle can also be extended by more modernisation phases. The decision for finding a strategy has to be based on assessments of the value of the system itself and to the company and on the effect that each chosen procedure will have on both. How the term migration fits into this picture will be discussed in 5.4. But first the terms of the software life cycle are defined in more detail and will be compared other existing terms outside of this taxonomy.

⁶ In [WERE08] modernisation would be part of the maintenance phase in the life cycle, as there maintenance is defined as the phase between deployment and shutdown of the system.

5.1.1. Maintenance

Maintenance can be seen as a process of small improvements or corrections to a system whereas its actual structure is not changed. It contributes to the evolution of the system but is limited, as it can't introduce new technology in these small, incremental and iterative steps. Also maintenance becomes more expensive over the time the system exists and grows and will more and more struggle with the anti-pattern of "tight coupling" as mentioned in chapter 4.4.2.

5.1.2. Modernization

Modernization reaches further than maintenance and will quite likely involve bigger structural changes, functional enhancements and modification of software attributes. Modernization still preserves the tacit knowledge of the legacy system and can be grouped into white-box modernization or black-box modernization depending how much knowledge about the system to be modernized will be gained.

5.1.2.1. *White-Box Modernization*

Similar to white-box testing of software this type of modernization is done knowing the internal behaviour of the system. Therefore it is necessary to gain that knowledge in a process of program understanding through Reverse Engineering and design recovery (CHCR90) – as discussed in chapter 5.3. This step is followed by system or code restructuring measures that will improve maintainability and / or performance.

5.1.2.2. *Black-Box Modernization*

Black-box modernization on the other hand is done through examination of the external behaviour provided by the system interfaces and is less time-consuming as program understanding. With the knowledge about the interfaces black-box modernization aims at wrapping the functionality of the legacy system and introduces an additional software layer that will provide suitable interfaces to modern systems. Unfortunately solely understanding the inputs and outputs of the legacy system's interfaces won't be enough in most of the cases and so some deeper knowledge of the internal behaviour will be needed too.

5.1.3. Replacement

If maintenance and modernization fail to evolve the system as desired or are too costly rebuilding the system from scratch is the only alternative. This approach is resource intensive and holds the most risk in it.

5.2. Discussion of Terms in the Field of Software Engineering

Before looking at the terms in the field of evolving software systems it is helpful to get an overview. For this reason Fig. 5.3 (CHCR90) depicts the terms and relationships between them. The three abstraction levels of an application or system requirements, design and implementation are introduced to explain that different actions effect different abstraction levels. From left to right logical, implementation-independent designs to physical implementation levels are presented. Forward engineering for example is always leading from a higher to a lower abstraction level. So starting with the requirements for software the first forward engineering step leads to the design and the next step leads to the implementation of software. Reverse engineering on the contrary leads from lower to higher abstraction levels. Through reverse engineering efforts it is possible to get from an implementation of a program to the design level and conclude about the requirements such as business rules. The diagram illustrates the relationships between Requirements, Design, and Implementation across four levels of abstraction, showing forward engineering, reverse engineering, design recovery, reengineering (renovation), and restructuring/redocumentation.

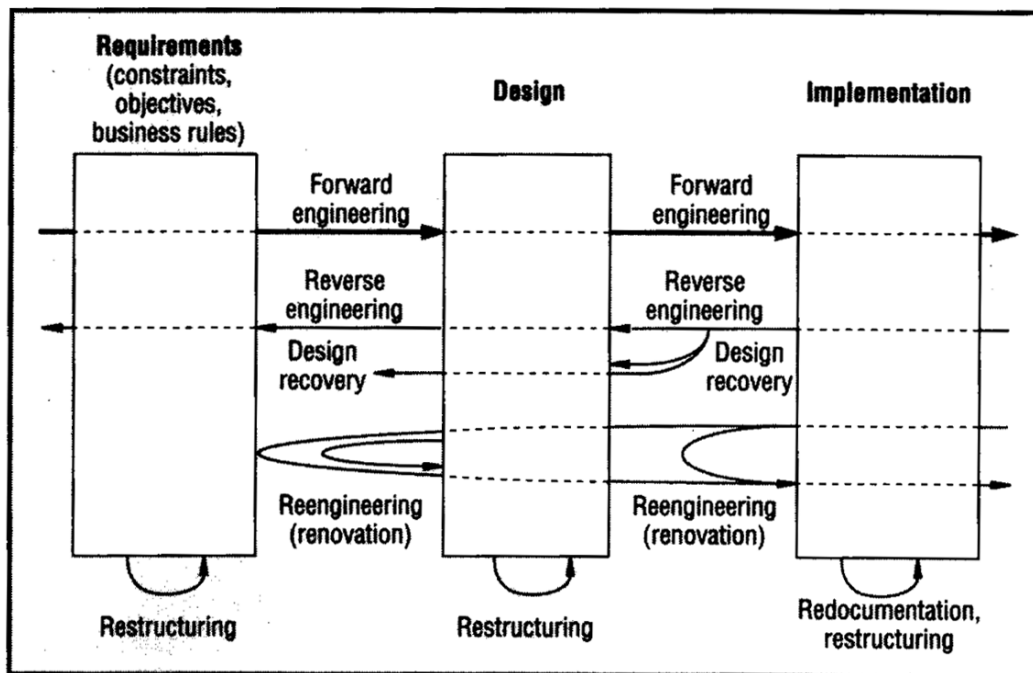


FIG. 5.3 TERMS IN SOFTWARE ENGINEERING (CHCR90)

Design recovery and re-documentation are subsets of reverse engineering. Design recovery re-establishes design abstractions using a combination of code and existing design documentation, personal experience and general knowledge about the domain. Whereas re-documentation results in a semantically equivalent representation on the same abstraction level and tries to recover documentation of for example an application by generating a diagram from the code.

Restructuring is leading one representation form to another at the same level of abstraction but is not changing the external behaviour of the system. Differences to reengineering are that quite often knowledge about the structure without understanding the meaning of the system is sufficient for restructuring and that modifications following new requirements do not normally take place. But of course restructuring makes the need for modification quite often visible. A special and prominent case of restructuring is refactoring that leads code to an object-oriented representation. For further details look up (METO04) and (FOWL04).

The missing term of reengineering will be dealt with in the

5.3. Software Reengineering

Software reengineering examines and alters design and implementation of a system in a new form. It involves reverse and forward engineering to first find out in a more abstract description how the system works and then to implement the planned changes usually also considering new requirements.

Reengineering is the systematic transformation of an existing system into a new form to realise quality improvements in operation, system capability, functionality, performance or evolvability at a lower cost, schedule or risk to the customer (Tilley 95 p. 3). This means that the aim of software reengineering is to improve an existing system to achieve a better return on invest (ROI) than through a replacement because it is not as risky and cost-intensive.

(SEPL03) describes the following types of software reengineering.

5.3.1. Retargeting

Retargeting changes the hardware platform of a legacy system to a new platform, like moving away from an operating system that is not longer supported to a more current OS and faster hardware. Through this approach an organisation can usually take advantage of more powerful, current platforms and achieve a reduction of cost for maintaining and running the new system which should also be easier to evolve. Retargeting can also be defined as a porting of a system – see chapter 5.4.

5.3.2. Revamping

One talks about revamping if only the user interface (UI) of a legacy system is replaced. In many cases this means getting rid of the “green screens” of host applications and replacing them with a graphical user interface (GUI). In this

approach improvements in the usability can be achieved by combining functions of different existing systems maybe presented in a browser based web-interface. A special case of revamping is screen scraping that wraps one or more old UIs into a single new GUI or UI. This can also be used to build an application program interface (API) that can be used in other systems to extract data from the system that is wrapped like that. But screen scraping itself is black-box technique and does not change or improve the underlying system and usually is slower than accessing the legacy system via its original UI. It can be useful though as an intermediate step in a bigger migration project to start designing the new (G)UI of the future system.

5.3.3. Commercial Components

Using so called commercial off the shelf (COTS) components or systems will try to replace legacy components or the whole system whereas the main idea is that the in-house source code and maintenance cost should be reduced. But one should not forget about the necessary cost for extracting legacy code and data, nor the effort of customisation or licensing and additional training. COTS software can be grouped into infrastructure and functional components. DBMSs, application servers or middleware products are examples for infrastructure components and feature advantages like better robustness, security, better performance, scalability and usability. They usually implement standard interfaces, which are easier to integrate. Their biggest challenge is though to extract the functionality – quite often spread out in the system – from the legacy system and replace it by an infrastructure COTS component. Functional components cover single domains like finance, human resource or are compound with others to an enterprise resource planning (ERP). They also offer robust functions with good performance but as they are usually designed after a process that is defined by the vendor it definitely takes customisation and some business process reengineering (BPR) which can turn out to be expensive and time-consuming. Which is an aspect not to be underestimated.

5.3.4. Source Code Translation

In a source code translation the source code of the legacy system is translated into a modern target language. This can be useful when moving to a new platform – be it to a different operating system or a system with a completely different architecture – where the language of the legacy system is not available anymore. Although this sounds like an easy task especially given that there are automatic translation

programs for this task the benefits of such efforts is quite often overestimated and the costs on the other hand underestimated. In general one can say if the difference between the languages is not too big – e.g. from C++ to JAVA, from one COBOL version to another – automated source code translation can help to evolve the legacy system. But if the translation takes place between languages that follow a different paradigm e.g. a procedural language like COBOL and an object-oriented language like JAVA – see refactoring in chapter 5.2 – translations can be risky and lead to unwanted structure or behaviour, e.g. large class structures, variables and methods are declared as public and generally look like COBOL written in JAVA for example.⁷

5.3.5. Code Reduction

Code reduction aims to eliminate functionality in the code that is no longer necessary or to identify redundant blocks of code and rewrite them into reusable subroutines. Code reduction is of special interest when the legacy source code for example will be ported to a different platform or language, as it does not make sense to translate code that is not longer necessary. The savings achieved in code reduction can be made quite obvious if a third party is responsible for the code translation, because they usually charge a certain amount of money for every line of code (LOC). One possible way to manage code reduction therefore is to reduce the number of LOCs (and the corresponding cost) to a wanted extent. But although the principle of code reduction is quite easy and obvious it has risks, because a deep understanding of the dependencies has to be established to avoid unexpected and unwanted ramifications. That is why, although there are tools to automate the extraction of business rules and the identification of similar code segments that can help to find similarities, the rewriting of subroutines has to be done manually. In combination with the goal of reducing LOCs and therefore costs one can start to rewrite redundant blocks of code with big similarities first and drilling down to code blocks that are less similar. The risk that the rewrite has side effects on other parts of the system therefore of course increases.

5.3.6. Functional Transformation

Program structure improvement, program modularisation and data reengineering are called functional transformation. Structure improvement has to consider structural

⁷ Source code translation is very similar to restructuring as mentioned in chapter 5.2 and therefore not really a valid technique of reengineering.

flaws first and has to choose an alternative one to set up rules how the original code has to be modified. Getting rid of GOTOs and simplifying complex condition statements are examples of structural improvements that can be done automatically or manually. Program modularisation tries to reorganise related parts of a system into modules to eliminate redundant code, optimize interactions between modules and establish well-defined interfaces. And in the modularised code is not only easier to maintain but also to migrate. Finally data reengineering deals with the problem of data degradation and data inconsistencies by revisiting the storage, organisation and format of legacy data.

5.4. Software Migration

(DIMA05) describes migration as the opposite of a system development from scratch. Migration is seen as a whole process: *“In nearly all cases existing systems and data stores have to be taken into consideration because the information within them often represent a big asset of companies. That is why they have to be taken over into the new system in a controlled and risk aware manner.”*⁸ *“(Software) Migrations are technical transformations following clear definition of requirements.”*⁹. Together with the schematic representation of Fig. 5.4 (GIWI05) stresses that existing software is to be migrated only following the functional requirements of the legacy system adding non-functional requirements from stakeholders. The latter origins from the fact that the legacy system is not sufficiently able to adapt to changed business processes, functional extension or external requirements e.g. changes in applicable laws.

⁸ Translated from German

⁹ Again translated from German

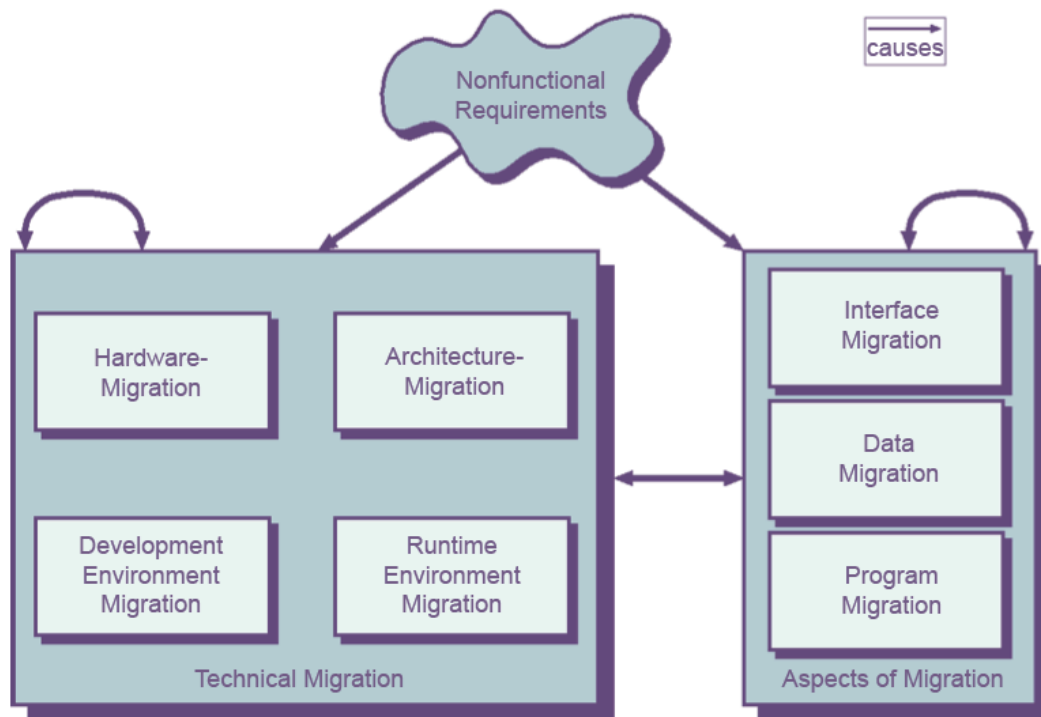


FIG. 5.4 TYPES OF MIGRATIONS AND MIGRATION ASPECTS (GIWI05)

5.4.1. Types of Technical Migrations

Regarding technical view there are different types of migrations - as shown in Fig. 5.4. Hardware migrations deal with the change of underlying hardware set-up as moving from a mainframe to a client/server environment. One layer above that we speak of migrations of runtime environments. This can be for example a change of operating system or DBMS. Migrations of the next abstraction levels either concern the development environment e.g. change of programming language or architecture of software e.g. migration from monolithic to a multi-layered architecture

5.4.2. Migration aspects

Depending on the goal of the migration different aspects can be concerned. User interface migration for example focuses on the migration of components that enable interaction between user and system, whereas data migration aims at the stored legacy data and its data structures and schemas. And finally program migrations transform the business logic of a program into a new environment.

5.4.3. Migration Strategy

As migrations are different from case to case there are different strategies to handle the challenges of a given legacy system. Which one is finally applied heavily depends on the context and the objective of such a project. In contrast to the

historical reengineering strategies discussed in chapter 6 (SNWH10) defines three groups of strategies that are actually dealing with what to do with legacy software.

5.4.3.1. *Re-implementation*

A re-implementation seeks to reuse all information from the legacy system and leads to a recoding of the old system – compare replacement in chapter 5.1.3 – preserving its architecture. Only small changes e.g. regarding data structure or structure of the code will take place. That is because the “new” database and programming language might urge for such adaptation. Reverse engineering plays a major role to obtain the necessary insights into the legacy code.

5.4.3.2. *Conversion*

The concept of conversion concentrates on an automated transformation of data and applications into the new environment. To achieve a high level of automation the legacy system must be fully understood – compare white box modernisation chapter 5.1.2.1 – in order to be able to develop the necessary tools. Because only if the transformation can take place automated to a high extend it will be financially viable. (SEPL03)

5.4.3.3. *Encapsulation (Wrapping)*

In this case of black box modernisation – compare chapter 5.1.2.2 – the legacy system is encapsulated through software wrappers, which allow to leave the old code untouched and just establish and provide interfaces to access functionality and data. As only input and output analysis of the existing system is needed it represents a fast and cost saving approach but neglects to improve code quality, which is one of the main drivers for migrations – see 5.6. (SNWH10) lists encapsulation or wrapping of legacy applications as migration, which stands in contrast to listings of (UMAR97) and (KSNW04) where integration is being seen separately.

5.5. Migration in the Field of Software Enhancement Projects

The following section describes so-called enhancement projects that all have one thing in common. Regarding the effort and the challenge they call for realisation as a project as they are exceeding the possibilities of the day-to-day business of the IT department.

5.5.1. Extension Project

Extension projects are adding functionality to a system. It is comparable to a new development taking place under constraints of the system already in place. So for example new components need to follow the already existing architectural specifications. The same goes for programming language or techniques and data models.

5.5.2. Maintenance Projects

If an adaption of a system concerns more than half of the code base it is called a maintenance project. Existing code will be structured or refactored for improving structure and quality. The difference to usual maintenance is that a dedicated team undertakes this one combined step for the whole system instead of many small maintenance steps in different components.

5.5.3. Migration Projects

Similar to the descriptions in chapter 5.4 a migration project transforms software from an old into a new environment. This can lead to one or a combination of the four mentioned types of migrations. Once again migration is a sole technical transformations that does not add functionality.

5.5.4. Integration Projects

Just as maintenance projects they are seen as a one-time effort to integrate one system into another. Integration projects can show both technical and functional measures like encapsulation of code into a component with a corresponding interface or the introduction of ontologies for easier data exchange within a company.

5.6. Reasons for Migrations

So what are the reasons that we end up with systems that need to be migrated? Aging of software and maintenance cost comprise the main reasons.

5.6.1. Aging

At the first glance one might claim that software representing a valid mathematical algorithm cannot wear out but the cost involved with aging software proofs otherwise. There are two reasons for that (PARN94):

1. “Lack of movement” or inability of adapting to changed requirements
2. “Ignorant surgery” or side effects of changes in software

Ad 1.

Besides changing business requirements expectations of users have changed through the years and will be changing in the future too. So even assuming that a legacy system still would be able to do the job, it would be regarded worn-out if it would be running a command line interface instead of a graphical interface as end users are used from current operating systems. New software will therefore always replace old one if the advantages – including subjective ones – exceed cost for training and migration. And of course especially the subjective advantages are well fed through marketing activities of software vendors to sell new releases and updates.

Ad 2.

It is pretty obvious that changes need to take place to keep a system up-to-date with the business requirements but as already mentioned changes – especially applied incorrectly regarding the original concept or EA idea – lead to higher complexity and degradation of software. Degradation can mean a slow or limited behaviour of the system which in (GRKI08) is described as partial failure or service failures that either lead to an incorrect and no service at all. In combination with the fact that developers rather add code instead changing existing code a system ends up with a multiple of lines of code up to 10 – 20 times as many. At some point requirements then will be arising faster than the software is able to adapt and this creates a maintenance backlog, which together with steadily increasing data makes the situation even worse.

5.6.2. Cost

The average cost for maintenance of an EA can be schematically described through Fig. 5.5.

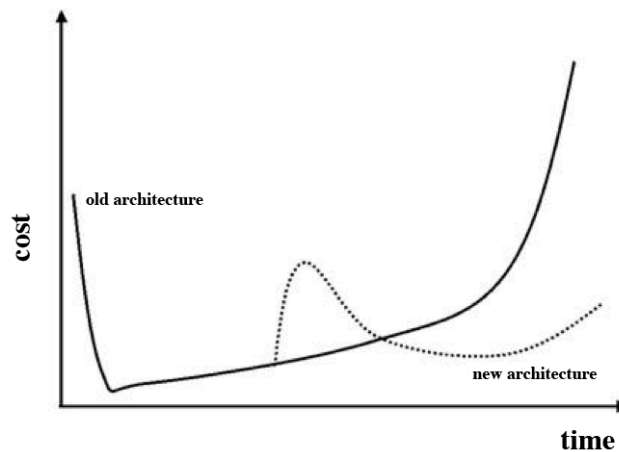


FIG. 5.5 MAINTENANCE COST DEVELOPMENT OF AN EA (DIMA05)

During the introduction of an EA maintenance cost will be quite high due to lifecycle-enablement effort. That is because developers usually need to get to know the new EA and will need training. The before-mentioned aging of the EA will lead to increasing cost until a point in time where they will “explode” exponentially. This would be the perfect time to introduce a new EA that will of course more or less show the same picture but if planned out well under a different time scale.

6. Legacy Application Reengineering Strategies

This chapter gives an overview about existing reengineering strategies in the area of legacy applications whereas the focus lies on incremental strategies.

6.1. Historical

A look at the history of systems reengineering reveals that the problem is not new to the still young history of information technology. Although it took some time that managers realised that there is a gap evolving between business needs and the largely static information architectures. And of course everybody was looking for a silver bullet solution to the problem field promising a simple solution to a complex task. In the early eighties of the last century fourth-generation languages (4GL) were meant to displace legacy systems. With the 4GLs end-users would be able to simply program new systems whenever needed without the need for dedicated programmers anymore. A promise, that could not be kept. Next in line of simple solutions was Computer-Aided Software Engineering (CASE) during the late eighties and early nineties. CASE-tools should make it possible to generate systems from business models, but although the idea itself was promising, the tools and

methodologies behind the CASE-idea failed to deliver. Another attempt arising during the early nineties tried to rewrite all mainframe applications in C and move them to the workstation with the prediction that mainframes would be completely replaced in the mid-nineties. A prediction – as we know now – that never came true. But also there was the underestimated problem of extracting process knowledge and data from the complex mainframe applications, which pretty much put a monkey wrench into the works of rewriting those systems. The so far most promising approach is the EAI approach, which is regarded as a partial or interim solution. Its concept is simple and non-invasive to the legacy system and can be applied straightforwardly. But as it is quite simple and admittedly smart to just connect applications via APIs or middleware these solutions proliferated and experienced quick fixes and non-coordinated projects lacking or ignoring a strategic integration plan. So in the end however sophisticated middleware and other components might get, they will add yet another layer of legacy software if they are taken for something else than an interim solution (WIUL02).

Quite usually migrations took and unfortunately take place like the before-mentioned scenario. (UMAR97) gives a general overview of migration possibilities whereas these options differ in describing what to do e.g. integration or introduction of a data warehouse and how to do e.g. big bang and incremental migrations that focus more on the procedure of the whole approach.

6.1.1. Ignore

This strategy would mean discarding the legacy application for all future developments. An approach that will not be practicable in most of the cases especially because if not for the business logic the data kept in the legacy application is usually of interest for future usage.

6.1.2. Big Bang / Cold Turkey Migration

The Big Bang or Cold Turkey Approach stands for attempting to rewrite the legacy system from scratch. The target system then should be realised by using modern software techniques and modern hardware environments. (BRST95) In general such an enterprise – especially for big legacy systems – is bound to fail because business, its requirements and also technology never stand still causing the system to be outdated when it finally is re-implemented.

6.1.3. Integrate

The aim of the integration approach is to consolidate legacy software into the current and future applications by so-called access in place. As already presented under the name of EAI in chapter 4.5.1 integration is only advisably as an interim solution. Because even if integration needs with other host applications are regarded low and a few clients only access the legacy application, there is the inherent risk that the overview of the direct and indirect links established between legacy and current systems. Again the most important thing is to establish a proper EA to handle all the connections in a planned and controlled manner.

6.1.4. Data Warehouse

A data warehouse aims at using heavily accessed, operational data that is used in day-to-day business for business analysis and report generation. For this reason a so-called “shadow” system, the data warehouse, is established. All the interesting operational data will be downloaded into this warehouse and build the basis for decision support applications.

6.1.5. Incremental / evolutionary / gradual Migration

This last category is aiming at re-architecting and transitioning a legacy system gradually. All these strategies are characterised by the fact that the existing and the target system are co-existing during the migration and usually gateways are necessary to synchronise them. The following subsections present examples for gradual migrations.

6.1.5.1. *Chicken Little Migration*

The Chicken Little Migration represents an approach in eleven small, consecutive steps in which the old system is migrated into the new. Both systems are coexisting during the whole project. They are synchronised through coordination components and gateways. Because of the small increments the project risk is decreased because after each step requirements and conditions can be re-evaluated.

Which gateway solution will be taken is depending on the tiers of the legacy system. The range goes from decomposable via semi-decomposable to non-decomposable systems whereas only the first follows the model-view-controller (MVC) paradigm. The second type of system is only realised through two tiers that are either combining the view and the controller or the controller and the model. Non-

decomposable systems are monolithic systems that are only represented through one single tier. (BRST95)

6.1.5.2. *Butterfly-Migration*

The Butterfly-Methodology aims to avoid the establishment of gateways to synchronise the old and the new system, as a higher number of gateways adds complexity and therefore also risk. Only the so-called Data Access Allocator (DAA) acts as a gateway encapsulating the legacy data and establishing temporary data stores that incrementally can be transformed – by the so-called Chrysalizer – into the data structure of the new system. During the whole time of the project only the legacy system stays operative and only after a small amount of data is left, the legacy system will be cut over to the new system which in the meantime could already be tested with the already migrated data. (ICEC97)

6.1.5.3. *Renaissance Approach*

The Renaissance Approach (WAIA99) includes different types of migration steps. Source code transformation is dealing with the automated translation of the source code into the planned language. Followed by a step of program and system restructuring, whereas the latter is aiming at cleansing of architecture e.g. separation of access- and business logic or splitting up components that are too big. After that data reengineering takes place in which the structure of the data store is being analysed and adapted and ready to be filled in the next step of data conversion. Through reverse engineering based on the modified system the necessary software structure is obtained to facilitate the last step of system integration which incorporates the migrated applications and data into the new environment. The Renaissance Methodology therefore suggests and focuses – besides the use of UML for reengineering purposes – on the integration into an EAI-architecture based on CORBA, XML or WSDL.

6.1.5.4. *Capsule-oriented Reverse Engineering Method*

The Capsule-oriented Reverse Engineering Method (COREM) is dealing with the migration of procedural legacy code into object-oriented code. This shift in programming paradigms is met through a combination of top-down and bottom-up object-oriented analysis. On the one hand the original requirements are examined to get to the top-down object-oriented model and on the other the existing source code

is statically analysed and step-by-step the second – this time bottom-up – model is derived. After that both models are compared, aligned and manually merge together. The result is an object-oriented documentation of the system that can be used to re-implement the system. (GAKL93)

6.1.5.5. *Reengineering Factory*

The Reengineering Factory is – despite the name – actually dealing with migrations and tries to automate the transformation of a legacy system into a new environment comparable to a process in a factory. It tries to take advantage of the fact that in migration projects 70 to 75 per cent of the tasks are similar. These tasks are the candidates for tool-supported automation, whereas the tools and the created organisational environment build something like a factory for mass-production. All tasks that cannot be automated will be carried out manually and are possible and actually meant for outsourcing. Test-driven quality management ensures that the migration takes places showing the same functionality as the existing system whereas a stronger focus lies on testing the error-prone manual tasks involving outsourcing. The approach needs a quite big effort up-front as the clearly formulated migration tasks need comprehensive project preparation and analysis of both the legacy and the future system. (BORC95)

6.1.5.6. *Service Migration and Reuse Technique*

The Service Migration and Reuse Technique (SMART) is based on the Risk-Managed Migration (RMM) approach of (SEPL03) and is – besides the risk of an erroneous transformation – also focusing on performance, acceptance, integration and maintenance risks for the future system. (LMSS08)

7. Current Architectures and Trends

By now only the necessity for migrating systems has been discussed. But which are the target solutions a migration is supposed to take us? There is no universally valid answer to this question! An appropriate target architecture is just too heavily depending on an organisation's context that it is not possible to generalise¹⁰. The only advice at this point is again going into the of a sound implementation of enterprise architecture. Only then it is possible to follow an integrated strategy for the

¹⁰ Doing that would actually suggest again a non-existing silver bullet solution!

whole business and deciding which technologies and architecture will be chosen. But to decide which architecture or trend to follow it is of course necessary to find out about the current situation in the field. Having this in mind this chapter right now points out which promising approaches are currently out there and describes them briefly¹¹.

7.1. Current Architectures

Different than the presented legacy languages the current languages also suggest a certain architecture at the same time. In the case of JAVA and C# it is even that the related architecture, so J2EE and .NET, existed before the language. Apart from these there are also general architectural developments noticeable (DIMA10).

7.1.1. Multichannel architecture

A still ongoing trend in IT-environments is the proliferation of end devices. Especially the presentation layer experiences fast changes as one can see in Table 7.1.

Layer	Alteration time	Stability
Functional objects	5 – 10 years	Systematic
Functional Processes	1,5 – 3 years	Quasi-systematic
Channel-adaptors	3 – 24 months	Opportunistic
Presentation	1 – 100 days	Hyperopportunistic

TABLE 7.1 LAYERS OF AN IT SYSTEM (DIMA10)

Table also reveals the stability of the different layers and how fast they are changing. There we see that business objects are only slowly changing in a frequency from 5 up to 10 years whereas the duration of the presentation layer is changing fast in a frequency from 1 to 100 days only.

Current topologies usually look like Fig. 7.1 that leads to many interfaces and components and a high complexity in extending and maintaining the system. It is therefore important to decrease this complexity and also try to gain a business advantage of meeting the hyper-opportunistic character of the presentation layer.

¹¹ The subsequent chapters are needless to say a snapshot of current architectures and trends and do only represent promising examples from the author's point of view gained through the research for his paper.

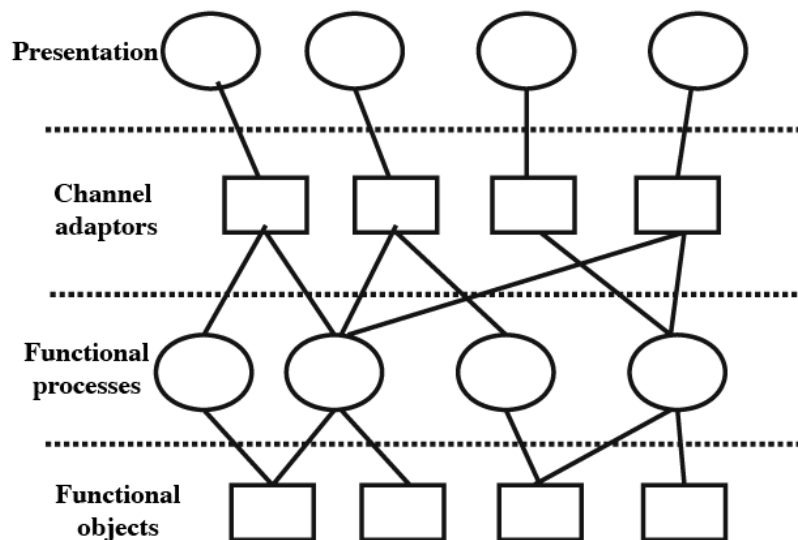


FIG. 7.1 EXAMPLE FOR CURRENT SYSTEM TOPOLOGIES (DIMA10)

A possible way to achieve that is to represent data in a canonical internal format so that the business logic can work with the data regardless from which input it originates and on the other hand can produce results in this canonical format too, not having to care about on which presentation end device it will be displayed.

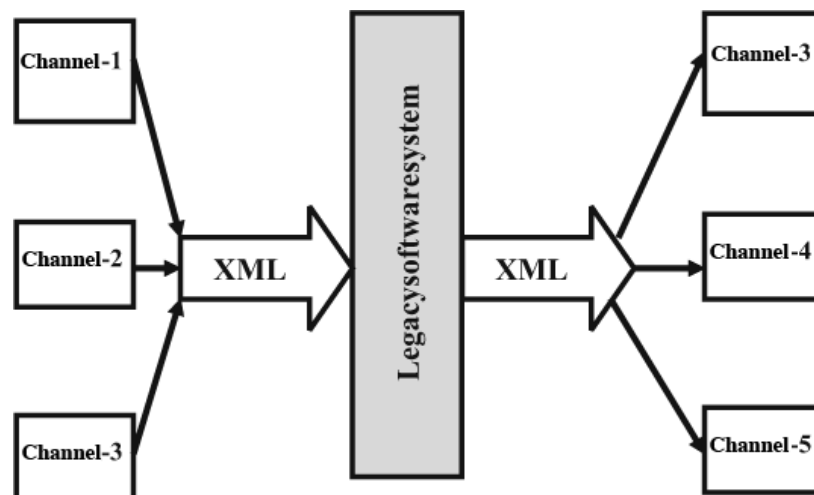


FIG. 7.2 XML AS CANONICAL INTERNAL FORMAT (DIMA10)

XML for its mentioned versatility is an appropriate representation in this matter.

7.1.2. J2EE – Java Connector Architecture

The origin for the Java 2 Enterprise Edition¹² lie in the development of the underlying language JAVA, that was at first meant to control kitchen devices and for that reason was paying big attention to be portable between devices. During the rise of the internet Sun recognised the opportunity and extended the language further for client

¹² <http://www.oracle.com/technetwork/java/javaee/overview/index.html>

side applications programming and moving into the server application area starting with a database connectivity via Java Database Connectivity (JDBC). J2EE itself is an integrated platform for developing portable server side software components and although a reference implementation in JAVA exists JAVA is not necessarily the language in which it has to be implemented. J2EE represents a distributed application server environment providing means to realise business logic in the application layer. Main components are the runtime infrastructure and interfaces and roles for the development of applications. Like that J2EE defines a separation of applications and runtime environment, which enables the latter to provide the applications with the underlying operating system services in a standardised way. The Model View Control Pattern (MVC-Pattern) is recommended for the realisation of a client.

Finally the Java Connector Architecture (JCA) as shown in Fig. 6.2 is in place to establish connections between J2EE components and “outside” applications like databases, ERP systems or also transaction monitors.

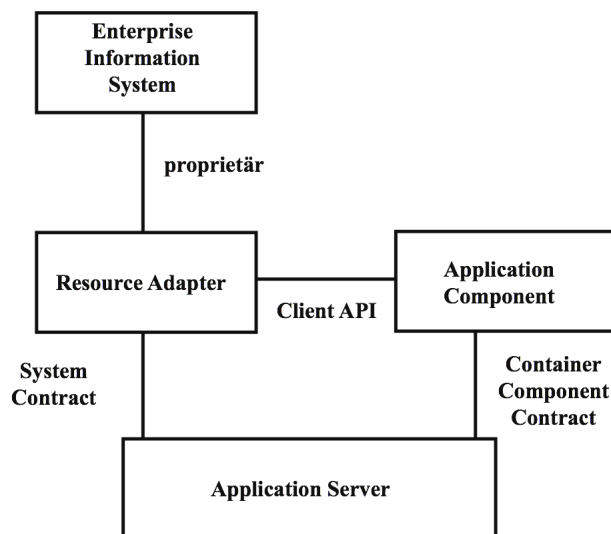


FIG. 7.3 JAVA CONNECTOR ARCHITECTURE (SUN03)

In Fig. 7.3 the developer of the propriety EIS needs to provide also a resource adapter with a client API and fulfilling the system contract with the application server that defines connection, transactions and security handling between them. Through the client API the resource adapter and the application component are able to communicate and finally the container component contract ensures the communication between server and component. With JCA a producer of an EIS only needs to deliver a resource adapter for each application server instead of developing an interface for every program or component working with the EIS.

7.1.3. .NET

Microsoft presented its .NET ¹³ strategy in the year 2000 to gain share in the server market using XML as the key technology. The goal was to combine existing Microsoft components like database access, directory services or messaging to a bundle. Whereas these components communicate over the simple object access protocol (SOAP) that relies on the hyper text transfer protocol (HTTP) and XML, which allows a heterogeneous infrastructure on server and client.

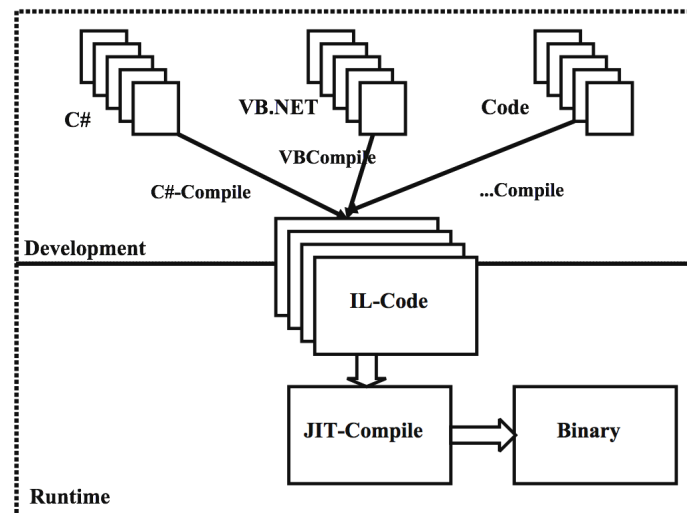


FIG. 7.4 RELATION BETWEEN CIL AND OTHER LANGUAGES (WEST02)

Fig. 7.4 shows that development can take place with different languages that are translated into the Common Intermediate Language (CIL), the former Microsoft Intermediate Language (MSIL), which is strictly typified and platform-independent. The respective code is translated into so-called assemblies and therefore allows interoperability because they are all translated into the same language. The CIL-code consists – beside the executable code – also of metadata of the applications and therefore don't need external resources as for example the registry of the OS. The common language runtime (CLR) finally acts like a virtual machine that executes and controls the assemblies and provides services like garbage collection and interoperability with not-.NET-applications.

7.1.4. Service Oriented Architecture (SOA)

The term SOA was first mentioned in 1996 in a research note of the IT research and advisory company Gartner Inc. ¹⁴ and describes a system that is fully consisting of

¹³ <http://www.microsoft.com/net/>

¹⁴ <http://www.gartner.com/>

services interacting via the same public protocol and realising the three roles of a provider, a consumer and a broker. In general SOA is more of a paradigm than a mere IT infrastructure. The main goal is to achieve far-reaching re-usability by encapsulating functionality into services. These services can then be combined through orchestration to deliver a more complex service. E.g. if a system provides a service like a database query and a service for calculations one can orchestrate those to be able read a number value from a table, do calculations upon this value and write it back into the same table. This admittedly very simplified example should make clear that two quite basic services can be easily combined to create a more meaningful service as in the mentioned case could be changing the stock of a product. The approach itself is quite easy to understand but the realisation within an organisation's IT demands a strong integration of all components to enable a cost-efficient orchestration. Furthermore it effects the whole organisation and its business processes as full advantage of the flexibility can only be taken if also the organisation itself acts like a service oriented enterprise (SOE). SOA can be seen as an evolution or a special case of a component architecture to achieve further loose coupling, re-usability, high scalability and abstraction whereas a service can be described as a hybrid between an object and a component. An object is usually divided into smaller sections than a service and a component on the other hand is coarser. In general SOA demands more abstraction than the architectures so far, as only the functionality of a service is important but not its implementation or the necessary transport mechanisms.

Web services

Web services are more a by-product of the Internet than of SOA. Early web services can be more seen as direct calls of services over a web service protocol. So in general the difference can be stated as SOA is an idea for an architecture, and web services follow a certain protocol shown in Fig. 7.5 ¹⁵.

¹⁵ <http://www.w3.org/TR/2002/WD-ws-arch-20021114/#stacks>

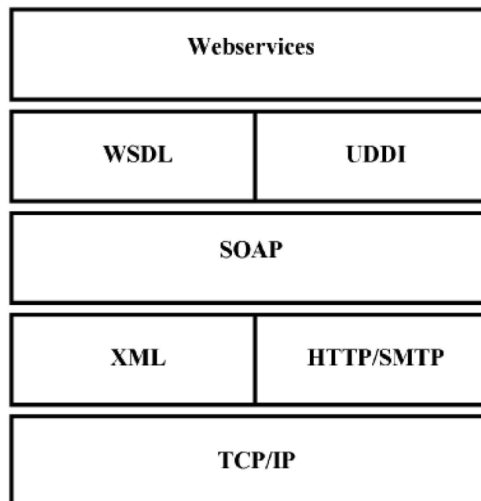


FIG. 7.5 WEB SERVICES PROTOCOL STACK ¹⁶

Web services themselves provide a functionality that is described in web services description language (WSDL) via indicating their interfaces, data types, protocols and necessary information for accessing the services. The universal description, discovery and integration (UDDI) was the former registry for registering and locating web services worldwide, but was not taken on as widely as expected and was discontinued in early 2006 (SOA05). The protocol SOAP is used to transfer messages between web services and it uses XML, HTTP or SMTP and even further TCP/IP for the representation and transport of the messages.

7.2. Current Trends

After having a look at the current architectures an enterprise architect or anyone trying to set up a migration project should of course always keep in mind what trends are currently out there. This is because trends can be enabler for more competitiveness through cost reduction, better customer service and therefore better customer satisfaction. But also within an organisation trends can be used to either become more efficient or raise motivation – as will be mentioned in chapter 8.3 - through implementation of current trends. On the other hand there is of course always the danger that some trends are just turn out to be fast vanishing hypes. The following examples have already taken on and seem to prevail in the future.

¹⁶ <http://www.w3.org/TR/2002/WD-ws-arch-20021114/#stacks>

7.2.1. Cloud Computing

One of the big buzzwords in today's IT landscape is the term "Cloud Computing". And as with every new term there are quite different opinions on what the "Cloud" should look like. (ARAR10) therefore try to find clarifying definitions and distinguish between Cloud providers and Software as a Service (SaaS) providers. The first group is defined by providing infrastructures and platforms for organisations through a network, which means that they are providing the hardware and systems software in data centres reaching from storage to different kind of servers, be it database, application, mail or mere web servers. Depending on whether this service is made available to the general public or internally in an organisation it, we distinguish between public and private cloud. But in both cases 3 aspects need to be fulfilled to speak of cloud computing:

1. "infinite" scalability
2. up-front cost-savings
3. pay-as-you-go billing

Ad 1.

An essential advantage of cloud computing rather than in-house hosting of hard- and software is the fact that through the massive resources of a data centre the user of the cloud should be provided "infinite" scalability. This means that the cloud provider should be able to quickly deliver even big load surges as can happen on public websites for example. Like that a cloud computing user can eliminate the need for far ahead provisioning.

Ad 2.

The second characteristic is that through the use of cloud computing companies are not forced to up-front commitment in hardware before actually needed. Through the cloud they can satisfy the need of resources exactly at the time they are really occurring. For example an advantage very attractive to start-ups with limited financing.

Ad 3.

Through paying only the resources that were actually in use cloud providers are selling hardware in a pay-as-you-go manner and help converting capital expenses to operating expenses (CapEx to OpEx).

On the other hand Software as a Service (SaaS) providers – which exist actually longer than cloud computing – offer different kinds of services like application services, e.g. office application or even whole desktop services through the Internet.

The relationship between cloud provider and SaaS provider is illustrated in Fig. 7.6.

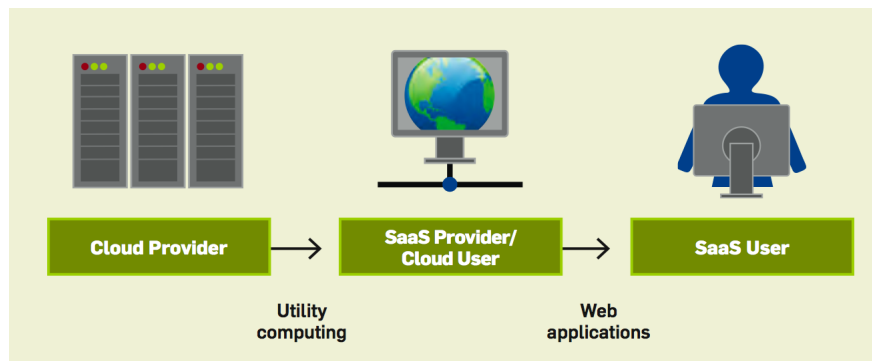


FIG. 7.6 SAAS PARTICIPANTS AND THEIR RELATIONSHIPS (ARAR10)

So a cloud user can be for example a SaaS provider which itself in turn provides web applications to a SaaS user. But of course a SaaS provider can himself be a cloud provider hosting its web applications in its own data centre. And on the other hand a cloud user must not definitely be a SaaS provider but a company using utility computing.

7.2.2. Web and Enterprise 2.0

The term Enterprise 2.0 was first mentioned by (MCAF06) and describes how Web 2.0 techniques, like for example blogs or wikis, can be used within an organisation. In contrast to the before existing and common approaches like using channels, e.g. e-mail and person-to-person instant messaging, which enable information creation and distribution by anyone but only allows a low degree of commonality as the information is visible to a limited group of consumers. Consumers in the case of e-mail for example would be all the recipients of a particular mail. The second communication category is platforms like intranets and corporate portals. In this case content is created by a small group but visible to many or even all depending on the permissions of the intranet or portal. The following group of Web 2.0 technologies – the so-called SLATES – help to lead to a Enterprise 2.0 where content can be generated by every employee and will also appear visible to everybody within the organisation¹⁷.

¹⁷ Of course restrictions and permissions on what is viewable to whom still can be put in place.

Search

One of the most important things in an information platform, like an intranet, is that the information that is available can also be retrieved.

Links

Links help to structure information and help to improve search mechanisms as most important information and pages will be linked to the most. This of course asks for the before-mentioned possibility of many people being able to even create links.

Authoring

Through authoring staff's implicit knowledge can be made explicit, eg. in wikis or blogs, and therefore shared and collaboratively worked on.

Tags

Tags provide employees with a means to categorise content and helps to establish information structures and relationships that people actually use. This so-called folksonomy¹⁸ could be also be used to visualise patterns and processes in knowledge work if users are able to see other users' tags, as it is used on tagging sites like del.icio.us¹⁹

Extensions

On the basis of tagging categorising and pattern matching of users' interests can be compared and recommendations can be inferred. Examples for this mechanism can be found on Amazon²⁰ or Stumbleupon²¹.

Signals

Finally signals help to users to stay updated. Because of the huge amount of information one can lose overview quite quickly, but by subscribing to a signal, like Really Simple Syndication (RSS) employees can for example stay updated to their favourite blogs.

But of course there is more to the conversion into an Enterprise 2.0 than just setting up an intranet providing the SLATES-functionalities. Requirements are a receptive culture within an organisation to establish these new types of collaboration, a common platform instead of silo platform, eg. a small amount of wikis accessible by a bigger group of employees instead of many wikis accessible by only small groups to ensure that searching, linking or signalling can work properly. The rollout is

¹⁸ <http://vanderwal.net/folksonomy.html>

¹⁹ www.del.icio.us

²⁰ www.amazon.com

²¹ www.stumbleupon.com

suggested to take place informally and without too much up-front pressure on users to use it. Instead management should try to encourage a few groups to start blogging and creating wikis that will raise the other employee's interest. But of course managerial support will be needed to encourage people further to use the Enterprise 2.0 technology. Managers need to take action within wikis and blogs too and usually set up structures of wikis for example. Like that users will be able to react to the content and structure and will start improving or making comments.

But altogether it can be said that corporate culture must be able and willing to deal with Enterprise 2.0. This includes for example the existing etiquette within a company that will be reflected also in its so-called netiquette²². On the other hand it is important to see that Enterprise 2.0 empowers users to speak their minds to a bigger audience, a means that was usually only reserved for management in the intranet. Therefore users will have it easier to express dissatisfaction and negativity that poses a challenge for management how to deal with breach of etiquette or with dissatisfaction. As a permanent intervention will prevent people from using Enterprise 2.0 technologies but on the other a clear message has to be made about what is tolerable and what not. Given these challenges it is obvious that not all organisations will be able to equally make use of the opportunities of an Enterprise 2.0.

7.2.3. Ubiquitous computing

We might still be far away from the world of ubiquitous computing as (MAWE91) was describing it, but near field communication devices such as Radio Frequency Identification (RFID) tags in products even like clothes, with our mobile devices like laptops, tablet PCs and of course cell phones – though today usually better referred to as smart phones we are continuing the journey to a world of ubiquitous computing. A world in which for example different than a virtual reality we also have the possibility to add information to our reality leading to an augmented reality. And for economic matters it does not really matter that the computer as being a visible part of our surrounding is not yet reality. It is enough that we now already can think of new usage of mobile devices such as using tablets for trade representatives being on-site of our customers. Like that the agent will be provided with accurate information about the customer and will be able to deliver really personal and therefore better customer service. A new vision within organisation is for example also letting employees taking

²² The etiquette that is obeyed in online mediums, like wiki, blogs, e-mails etc.

their laptops or smart phones with them to directly work in their used computer environment in the company. This bring-you-own-device-mentality would not have been thinkable a few years ago.

Looking at architectures based on mainframes it is quite obvious that amendments must take place to live up to the expectations of ubiquitous computing.

PART B: TOWARDS A SOLUTION

The chapters so far established a comprehensive starting point for the following chapters. But knowing what the term legacy in IT matters really describes and how migration is defined in comparison to software engineering and re-engineering is a vital prerequisite of common understanding and part of the before mentioned holistic view on the whole topic. From this chapter onwards the focus is more shifting to how to implement and set-up the often-referred migration project and the proposal to frame it with the more long-term approach of enterprise architecture. The subsequent chapters are dealing with the human influence on projects via the presentation of risk control instruments to the more monetary perspective. Important techniques and tools will be discussed to develop a risk minimising concept. Having these means in place finally the more procedural points of view of migrations can be discussed and enriched to finally develop a reference migration model.

8. Human Factor Aspects

The migration of legacy systems is apparently a complex technical issue which can lead to the opinion that the human factor in such projects might not be as influencing. This chapter tries to bring back to mind what usually everybody who took part in a project in general will have experienced already. People can throw the monkey wrench into a project's plans. Social sciences and psychology provide a big number of theories to describe human behaviour towards change, acceptance and impacts of introduction and use of technology or about motivation of individuals. In this chapter different appropriate theories are taken into consideration to describe which influence and factors should be given a thought. Because reaching from executives stopping support and funding via discontent administrators to users not accepting or using the changed system in a right way, those groups can become showstoppers in migration projects. That is why these three main groups of stakeholders and participants in a

migration project have been picked to point out and strategies for each group will be presented.

8.1. Management

On the one hand there is the expectations of top management, which need to be addressed. And the reason for that is quite obvious; if your project fails to make sense to your sponsor, you can pretty much consider your project as cancelled. Of course the major argument for the migration will be presented in the business case that will be mentioned in chapter 9 but in addition (JESI07) suggests that top management needs to be involved to undertake sponsorship and ownership throughout the project. The key to this desired support is participation of the top management to show where the problems in the current situation lie and how the migration project will contribute to solve these problems. A means to reach this goal is called the problem mapping technique. Whereas its value also lies in the fact that the ones responsible for investigating the legacy system have to strongly focus on the customer's problems and needs rather than solely reflecting on solutions. So besides examining the current IT system interviews with representative employees and the management for strategic and economic aspects should take place. The findings in the interviews will be grouped into "problems" or "needs", "causes", "consequences" and "ideas for solutions". Together with the reflection whether the findings really originated from an interview or was more of a supposition of the interviewer that leads to really focus on the problems with their causes and consequences. After the revised results of the interviews have been written down they can be prioritized and discussed with top management to finally set up the exact definitions for the migration project. Like that the top management is participating in reviewing the system, giving comments on consultant's suppositions and helping to reformulate and prioritize the key arguments and features for / of the future IT system. This creates a feeling of ownership for the management and helps to understand the reasons for a migration as they then share the same overview of identified problems and business needs. Thus top management will be more likely to support the project in an appropriate way and will appear as the "sponsor". One of the three roles that (WIUL02) strongly suggests to guarantee a successful project. Whereas the "sponsor" addressed through the before-mentioned strategy should be able to fund on executive level and to convince other executives to support across the borders of the IT department. The next role is the so-called "champion" on

management level who understands and is willing to promote the concept of transformation in order to initiate the migration strategy and to obtain sponsorship. But there should be more than just one champion to ensure that if one of them retires from the project there are others to take over the place and fill the gap. The last of the roles is the so-called evangelist, a highly respected technician that helps to fortify the project by convincing others on a more technical level – see chapter 8.2.

Furthermore all individuals at the executive level or at management level may function as change agents that are responsible for the change strategy and its implementation. They create the vision, identify the need for and implement the change. In order to act as such a change agent they should show certain leadership skills as recognised in a perception based survey in organisational change by (GIGI09). These skills were perceived as being most important for change effectiveness in leadership and leading to successful change projects.

Coaching

Coaching takes place in the form of training, counselling, confronting and mentoring and gives leaders feedback and the possibility to see a situation from a new perspective. Through coaching a leader establishes synergistic relationships with their subordinates, which inspires them to remain optimistic and to enhance cooperation among each other aiming for better results.

Communicating

Leaders should communicate clearly and without ambivalence. It should be made clear where the risks of holding on to the status quo lie and what the prospective benefits of the planned change are. Enthusiasm should be spread in frequent communications but should not become unrealistic optimism. Employees should find an environment created by their leaders for realistic, truthful discussions including the scope of change and also negative aspects of the implementation.

Motivating

A motivating manager is able to persuade and influence his employees to act following a common direction. A skilled leader therefore should be able to organise, communicate effectively, address employees' questions, generate creative ideas, prioritise ideas, direct personnel practices, plan employees' actions, commit employees to action and provide follow-up to overcome motivational problems.

Building teams

The main key skills for building teams are effectiveness in managing teams and structuring work groups to facilitate collaboration. The latter can be achieved through open communications, coactive leadership and clear definition of roles and work directives.

Other skills like rewarding employees' performance and involving employees into decisions affecting their well-being and organisational performance were found to be less significant. Of course nevertheless these leadership skills have a positive influence on personnel to prepare for organisational change.

8.2. System Administrator

System administrators also form a special group of participants in a migration project as – of course depending of how big the change to the existing set-up of the IT infrastructure will be – they are on the one hand an important source for gaining knowledge about the existing system as well as they usually will be affected the most by the planned changes. Even further one can usually witness that system administrators have somehow a feeling of ownership for the legacy system and therefore need to be taken into special consideration and be involved into planning of the migration project. Because *when adopters are externally persuaded to buy into the vision of an outside expert, they tend to demonstrate inertia and resistance, ...* (STSA09). Inertia and resistance in this case means that they will try to keep the legacy system up and running for as long as possible. Acting like that they can undermine the project. (VEMO03)

But that would be one of the least favourable attitudes as the system administrator is most useful as an evangelist – mentioned in chapter 8.1. With his high knowledge in technology and the respect he gets from others for his knowledge the system administrator helps to argument for the change and will help to creative a positive atmosphere towards the planned migration.

8.3. User

Finally also the users are an important group of human influence to a migration project. They are usually the ones that will be affected the most by the replacement of a legacy system. In this section theories addressing how people experience change and introduction of innovations will be discussed.

Diffusion theory can for example be used to understand how people deal with change in general but also change within an IT system, as such a change can be seen as an introduction of an innovation. *Diffusion is the process by which an innovation is communicated through certain channels over time among the members of a social system.* (ROGE03) Through diffusion theory a rollout can be planned better as the phases of accepting change become clear. First of all it is important to know that the acceptance of change takes place in stages: awareness of the change, interest in the change, trial and the decision to continue or quit and in case of continuing the adoption of the change into one's life.

In quite all cases of migrations users will be dealing with a change in their working routine. This can be for example a changed business process, a new application or interface of an application. So first of all awareness of the change has to be raised. For this first phase a channel of mass communication is suited best. Within a company emails to all employees and an enterprise-wide appearance at the intranet would be a good means to spread the news of the change. From this stage on interpersonal communication plays a bigger role in the process of diffusion of innovations. Through such channels as person-to-person communication and blogs with the possibility to comment interest in the change for each user should be raised. In the case of a migration the then following phases are different as in research field of diffusion of innovations because within an organisation users will need to work with the new application and will not have the choice of not adopting it. But nevertheless users can be quite creative to circumvent an application they do not like or they are using it in a wrong way – again two scenarios that would be very unfavourable. So in the next phase of trying out the change within the system users should be able to give feedback. This will for example take place when hands-on training sessions start (ROGE03). Through good training and an environment of discussion about the new application or user interface employees should be convinced to use it in the right way and help to lead the whole undertaking to success. As already mentioned before also the last phase of adopting the change or not is not really a decision that lies in the hands of the user but again the more they accept it the more likely the system runs as expected.

Another interesting field of theories for migration are those dealing with the acceptance and use of technology. As there are plenty of theories and influencing factors out there in academia, (VEMO03) formulated a unified view stating that there

are four direct determinants of user acceptance and usage behaviour. As shown in Fig. 8.1 (VEMO03) those four are performance expectancy, effort expectancy, social influence and facilitating conditions. Furthermore there are key moderators that have influence on the determinants: gender, age, experience and voluntariness of use.

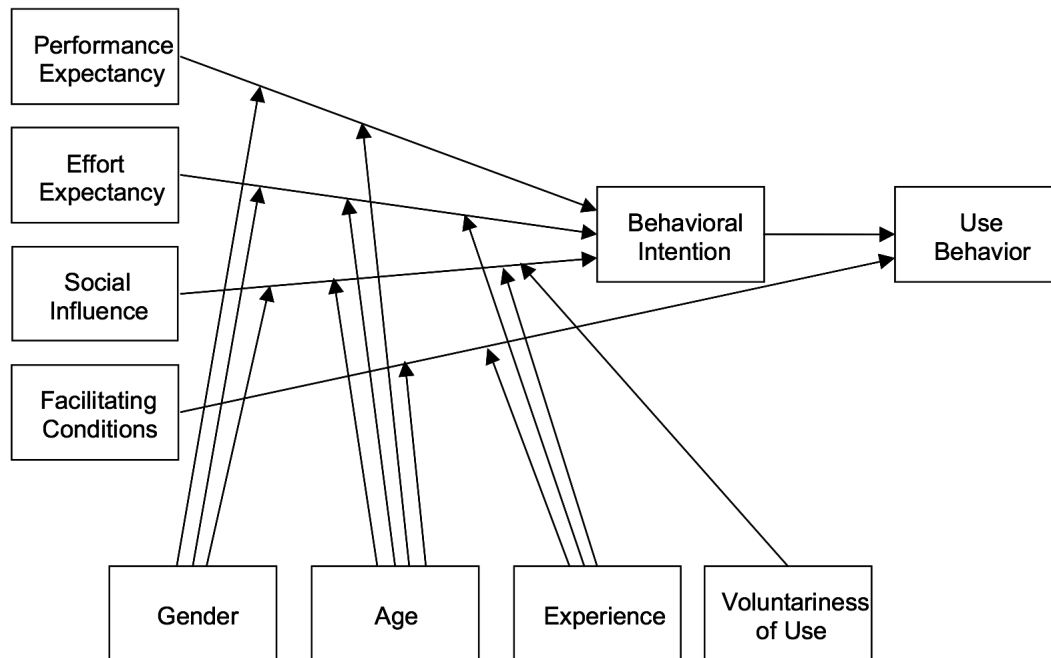


FIG. 8.1 ACCEPTANCE AND USE OF TECHNOLOGY (UTAUT) (VEMO03)

Performance expectancy describes the degree of believe of an individual that using the system will help to achieve advantage in job performance. Besides job performance this also includes extrinsic motivation like payment and promotions. The moderators gender and age in this case mean that this determinant is stronger among young co-workers and men. Performance expectancy is the strongest predictor of intention in voluntary and mandatory usage contexts.

The degree of ease associated with the use of the system is defined as the effort expectancy and only significant in the first time period of usage. After periods of extended and sustained usage it is becoming less and less significant until it is a non-significant determinant. That is because during introduction an innovation is perceived as being relatively difficult to understand and use. This effect is influenced by the moderators gender, age and experience, which shows that women, older workers and employees with limited experience are effected more strongly. Again also effort expectancy is significant in both usage contexts – voluntary and mandatory.

Social influence describes how much an individual perceives that important others believe he or she should use the new system. Subjective norm and image within a group of people that are important to an individual to use or to not use the new system also describe social influence. Gender, age, voluntariness and experience moderate this determinant and show that women, older workers, workers with limited experience all under conditions of mandatory use are more effected. Similar to effort expectancy the social influence is only significant in early stages of usages until individuals increase their experience with the system and are less reliant on social influence for their individual intention. Different than the other determinants social influence is non-significant in voluntary but becomes significant in mandatory contexts.

Facilitating conditions are the degree of belief of an individual that an infrastructure to support use of the system exists. This infrastructure can be organisational or technical. It encompasses the feeling of having control over using the system – also referred to as perceived behavioural control – and the feeling that the system is compatible with one's work style and work. As the before-mentioned determinants performance and effort expectancy are similar to and at some points overlapping with the determinant facilitating conditions the influence of the latter on behavioural intention is non-significant but is significant to use behaviour as shown in Fig. . The moderators age and experience impact the influence of facilitating conditions such that older workers and in particular those with increasing experience are effected the most. This for example means that in both cases – voluntary and mandatory – older workers that are increasing their experience with a new system are first more likely to be overcharged and rely on facilitating conditions. When they start gaining experience more ways of finding help and assistance are found. That underscores the direct positive influence for a sustained usage.

Addressing all three stakeholders groups as described should facilitate organisational change and decrease the risk of stakeholder not to contribute to the project.

9. Basis for Risk Control

Especially for big projects like a migration project risk control is inevitable. Mentioning the size of such projects it is first of all necessary to find out “how big” the migration effort will be on the whole or in other words which parts of the existing will be

concerned. But as it is usually in their nature to be a big effort it is also a vital task to communicate the importance of the system and therefore the migration. Only if clear benefits can be presented stakeholders will be willing to support the project – be it on launching or keeping it running in rough times. All this means that the legacy system should be inspected first.

9.1. Benefits of IT Systems

In general it is important to find out which strategy is pursued within a company. Is it either a strategy of differentiation or of low cost leadership? The former focuses on differentiating someone's organisation from other through quality, innovation, service, etc, whereas the latter concentrates more on reducing cost for the company. This will result in different questions for measuring the IT system. But which are now the benefit criteria for IT systems and how can they be measured? In general it is important to distinguish between tangible and intangible benefits, which means that there is a group of benefits that have a monetary effect – tangible benefits – and there is the group of intangible benefits, which have only an indirect impact on either cost or revenue. Examples for tangible measurements are the cost for maintenance, the availability of the system, the development cost for new software or productivity in several areas e.g. how many orders can be handled within an hour. User friendliness, improved structure, increased security and stability, future orientation of the system, compliance, improved corporate image or improved employee morale are on the other hand regarded as intangible, which do not affect organisation's bottom line profit (RMSI03).

But still intangible characteristics can be quantified and measured too. For example users can be asked about user friendliness and developers about the future orientation of the system. Improved structure can be measured by counting the amount of components, modules, ... and their respective gateways. Added functionality can for example be measured through a mere count of new functionality. Data rationalisation can be expressed in the redundant entries that have been removed. Some of the measurement techniques are definitely depending on subjective opinions and findings but they nevertheless give insight into the system.

So as one can see there are several ways of measuring information systems, but the biggest problem is just that they are hardly implemented in organisations. If that is the case then a migration is actually a perfect time to introduce them.

Whether a migration needs to take place or not often depends on some of the criteria being knock-out criteria. It is quite easy to think of tangible factors as K.O.-criteria because if maintenance and development cost for new functionality / software is exploding or the availability is far from satisfying action needs to be taken. But intangible characteristics can appear as triggers to those K.O.-criteria as compliance issues like changes in respective laws, the Y2K-bug or also a change of currency will reveal the above stated problems that can be well expressed in money respectively cost.

9.2. Portfolio Analysis

The portfolio analysis of (WAIA99) is a perfect means to find out which and how parts or whole legacy system should be treated and helps to decide in which order systems should be migrated. Like that the risk can be identified and a strategy for each risk can be developed. It is a similar approach as the portfolio analysis in the product life cycle from consulting companies such as Boston Consulting Group or McKinsey for planning on product and product lines. In this case of the portfolio analysis graph there are two dimensions used to identify action steps for legacy systems or parts of it. On one axis the technical quality of the system will be measured and on the second axis the business value will be expressed. Whereas the technical quality is comprised of the frequency of new releases, ease of making changes, hardware and software reliability, organisational infrastructure, system performance, accuracy, ease of operation, availability of training and number of vendor related tools and hardware and the business quality indicates how important the system or application is to the organisation. It describes the contribution to profit, level of usage, number of business goals satisfied, system value, user satisfaction and the value of the information stored.

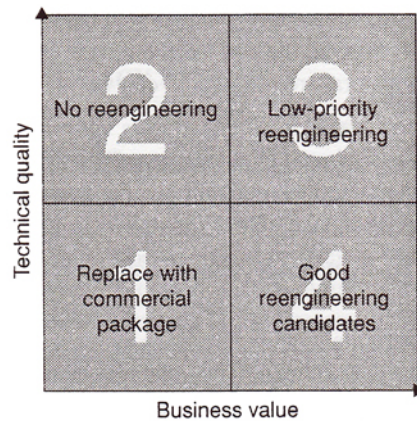


FIG. 9.1 PORTFOLIO ANALYSIS OF LEGACY SYSTEMS (WAIA99)

The graph in Fig. 9.1 shows how each system and application can then be evaluated against the above-described criteria and will end up in one of the quadrants. Each quadrant leads to a different strategy for evolution.

Quadrant 1:

Systems in this quadrant only deliver low business value and therefore are usually not critical or representing core competencies of an organisation. Together with their need of improvement regarding the technical quality they make perfect candidates for a replacement with COTS as they usually are used for payroll, human resources or similar services, which are already covered quite well by commercial packages.

Quadrant 2:

Applications placed in this quadrant should not need any further attention, as the technical quality is enough for the business processes in the moment of the analysis.

Quadrant 3:

Evolution of systems with a high level of technical quality and important business value should guarantee that the applications adapt to technical and process changes.

Quadrant 4:

In the fourth quadrant one will find the most promising systems for a migration or a replacement as their low technical quality is endangering the delivered high business value.

9.3. Incremental Approach

So besides finding out about how which parts of the legacy system should be treated an incremental approach in general is capable of controlling risk better by dividing the whole effort into smaller steps and therefore decreasing the complexity of the project. Increments per se help to eliminate uncertainties from one iteration to another

because following iterations can go back and use the results of previous iterations. Furthermore availability during the project will be improved because the down time of the system will be shorter than in comparison to a big bang approach (SEPL03).

An additional advantage regarding controlling risk is also that increments lead to faster results and ensure that the result itself still contributes something even if the whole project is discontinued. So all results and all knowledge gained can be for example at least used for maintenance reasons or smaller restructuring projects.

9.4. Reference Process

Both before-mentioned techniques are valuable contributions for controlling risk, but another advantage discussed in this work will be presented in chapter 11. A reference process is a generic examination of a complex effort such as a migration projects. It can be seen as a kind of guideline for essential decisions in migrating legacy systems. This is because it considers existing theoretical and also best-practice approaches and combines them on a more abstract level. Like that a reference process can be seen as a more sophisticated bulletin list that guides the migration. It breaks down the complex task into phases, actions and milestones helping, that important questions and dependencies are addressed properly and not overlooked.

9.5. Risk Management

In the end of course all these inputs and enablers for risk control have to be addressed by proper risk management. (PMIP08) proposes risk management as an integral part of its project management body of knowledge. It is suggested to align a projects risk management plan with the organisation's risk management policies. An important input for the mentioned plan is also the stockholder's risk tolerance to be able to adjust the handling of risk towards these tolerance levels. Of course defined roles and responsibilities are a key factor to risk management too and ensure that somebody monitors each risk. Subsequently the possible risks and their corresponding triggers need to be assessed on the basis of existing risk categories, document reviews of the project planning material and an assumption analysis. The risks, which are identified in this process, need then to be ranked and prioritised regarding their probability and impact. Therefore first the impact on the project goals will be qualified on a very low to a very high or numerical impact scale with e.g. three further impact levels in between the extremes. Second techniques like a decision tree

analysis or a simulation will help to quantify the risk with certain cost. After all that a decision needs to be made whether to try to avoid, transfer, mitigate or accept each risk. These decisions then form the basis for the so-called risk response plan and help establishing which the residual risk and secondary risks are. And last but not least all risk management activities are an important input for contractual agreements e.g. with insurances, suppliers or business partners and for the contingency planning in regard to time and money as a buffer.

10. The Cost Benefit Issue

Talking about risk and finding out about which parts of the legacy system have to be treated how, is a very important topic and necessary to gain an overview but in the end it all comes down to fund-raising for the project on management level. Decision makers need arguments to base their decisions on and three suitable vehicles for gaining that information are calculations of the payback period rule, the return on investment and business cases – also called cost-benefit analysis.

10.1. Payback Period Rule

The payback period rule is used to find out in which period, in this case years, the investment is amortised. This helps to get a feeling for the monetary benefits of a planned migration and how long it takes until the project – so to say – repays for itself. Of course in general the shorter this time span is or more precisely the less periods it takes, the better. The formula for the payback period rule can be seen in Fig. 10.1.

$$\text{Payback Period} = \frac{\text{Initial Investment}}{\text{Annual Return}}$$

FIG. 10.1 PAYBACK PERIOD RULE (DIAL05)

The initial investment in this case is represented by the cost of the migration and as the annual return cost savings in maintenance and in future developments can be taken into account.²³ If the annual return should not be the same during time it would also be possible to accumulate the annual returns until they are equal or have exceeded the initial investment. The period in which this is achieved represents the

²³ In order to represent the time value of the returns a correction of the annual return can be achieved by applying a weighted average cost of capital discount.

result. The migration cost is depending on the complexity and size of the legacy system and the chosen migration strategy and the annual return is a matter of estimations but (CAPE06) claims that maintenance cost can be considerably cut down to half of its development cost if the system is well structured and well tested and maintenance takes place following the architectural constraints. Software development companies on level 3 or higher on the capability maturity model (CMM) scale are believed to achieve such results.²⁴

10.2. Return On Investment (ROI)

The ROI shows the relation between the achieved profit or loss and the money invested for the migration. Calculating the ROI follows the formula shown in Fig. 10.2.

$$\text{Migration-ROI} = \frac{(\text{Utility Value} - \text{Migration Cost})}{\text{Migration Cost}}$$

FIG. 10.2 RETURN ON INVESTMENT FORMULA (SEPL03)

The utility value can be estimated through the cost for a new implementation of the legacy system, whereas a fully debugged line of production software is quoted to take an effort of cost in the range of 18 USD to 45 USD. (SMIT10) But it is not the whole system that is considered for this estimation as it was found that only 30 per cent of the code is containing business logic. So 70 per cent of the lines of code do not go into the calculation of the utility value. As we will see in chapter 11 it will be necessary to calculate the ROI for different scenarios concerning different strategies to be able to decide which strategy to choose. But in general the ROI has to be positive, showing a significant advantage that will be gained through a migration project. If that is the case the set up of a business case will finally give a good overview for decision makers.

10.3. Business Case / Cost Benefit Analysis

A business case should sum up why a migration project is necessary and how it can be justified. It should contain at least the following aspects:

Problem statement

The problem statement should give a specific description of the current situation and create a unified view of the inadequacies, inefficiencies and weaknesses in the

²⁴ <http://www.sei.cmu.edu/cmmi/>

current system and help to avoid space for (undesired) interpretation. Readers of the business case must be shown the negative consequences if the problem would not be addressed. This could either be a loss of money, decreased efficiency or a lack of quality of the services a system delivers. It should also be made clear how the migration project is different from other ongoing projects or the day-to-day business.

Solution

After the presentation of the problem the solution should be described whereas special attention should be drawn to that it is at an appropriate level of details. In general it should be made clear what will be migrated and how, e.g. that the solution consists of moving a system from a mainframe system to a web-enabled client-server architecture in incremental development and deployment steps. The description of the solution should also consist of a time schedule and a cost estimate. Moreover assumptions must be documented to give a clear and full image of the project. Assumptions could for example be that the organisation is not split up during the project by selling a certain division of the organisation or that the staff in relevant areas is not cut down.

Risks

Every project faces risks and especially in a migration project there are a number of risks that will appear over and over again, e.g. side effects on the current system or limitations in the resulting system that cannot be foreseen at the time the business case has been set up. As these risks exist it is very important to mention and describe them to on the one hand gain credibility for business case being close to reality and to set decision makers' expectations early. Only if management is informed about the risks and of course appropriate counter measures, it can take a valid decision and can also not be surprised if one risk will strike during the project, maybe even affecting the outcome of the whole project.

Costs and benefits

In the end it of course all comes down to costs and benefits, which should quantify how much the solution costs also taking into consideration risk as a cost factor. The same procedure has to be exercised for the benefits. Both, costs and benefits will be either expressed in terms of money or other measurements – as presented in chapter 9.1. The assumed person-days for the whole project for example can usually be calculated but improved quality will be hard to express in money.

A possible cost / benefit analysis can for example look like Table 12.1 taken from the field of software performance engineering (WISM03).

SPE Cost/Benefit Worksheet			
One-Time Costs	\$	Cost Avoidance	\$
Tools		Refactoring	
Performance Modeling Tool		Lost Revenue	
Load Driver		Customer Support	
Measurement Toolset		Liability	
Workstation(s)		Hardware/Licensing	
Training			
In-House Training (Developers)			
Performance Engineer(s)			
Consulting/Mentoring			
Total One-Time Costs		Total Cost Avoidance	
Recurring Costs (Annual)	\$	Intangible Benefits	
Software Maintenance (Tools)		Improved Corporate Image	
Salaries (Including Benefits)		Enhanced Customer Relations	
Performance Manager		Improved Employee Morale	
Performance Analyst			
Continuing Education			
Total Recurring Costs			

TABLE 10.1 COST / BENEFIT ANALYSIS EXAMPLE (WISM03)

Table 10.1 shows that the costs can be divided into one-time costs and recurring costs that will appear in this case on an annual basis. On the right hand side of the table the benefits are presented as cost avoidance and intangible benefits. In this example cost avoidance is central argument of the business case. But (REIF02) presents all in all four arguments as dimensions of improvement. The dimensions are increasing productivity, reducing time to market, avoiding and reducing cost and improving quality. Regarding benefits that can be represented in terms of money cost reduction and cost avoidance are most important dimensions whereas cost reduction relates to actions undertaken at the present moment leading to an immediate decrease of cost and cost avoidance refers to a decrease of cost in the future by the actions taken in the present. Especially because high cost occur at the beginning of a migration project, e.g. equipment, redesign of the system, ... cost avoidance benefits should be identified and added to the cost / benefit analysis.

In general different scenarios of the business case should be elaborated. Usually this would mean a pessimistic, an anticipated and an optimistic scenario. An approach that – if worked out appropriately - helps to understand that the optimistic scenario is possible but the anticipated results will be the most realistic ones whereas the pessimistic view makes decision makers aware that it is still possible that even the anticipated scenario – although not quite likely – might not hold. Over time a re-evaluation of the business case should take place to check whether the assumptions have changed.

11. The Reference Migration Process

Cost-benefit analysis and risk control should be integral parts of a migration process. But as mentioned in earlier chapters there are plenty of views and aspects that have to be considered to ensure that such an effort is not going to fail. Chapter 5 describes that there are already many approaches in both the academia and business world that are providing useful references for the broad topic of migration of legacy systems. And although in general the migrations are quite comparable to the process of software engineering and have a lot in common (SNWH10) found the existing design models from the waterfall to agile development missing the restrictions that a migration project has to deal with. In case of a new development of software there are usually by far not as many constraints narrowing down the freedom, as it is the case for a migration. And although there always have been recommendations for a cautious, phased migration strategy (LAKE00) or (WIUL02). For a long time there was a lack of standardisation and necessary abstraction in such suggestions. (SNWH10) finally took up the task for trying to find a way to represent this existing knowledge in form of a Reference Migration Process (ReMiP) that is also inspired by the Rational Unified Process (RUP)²⁵.

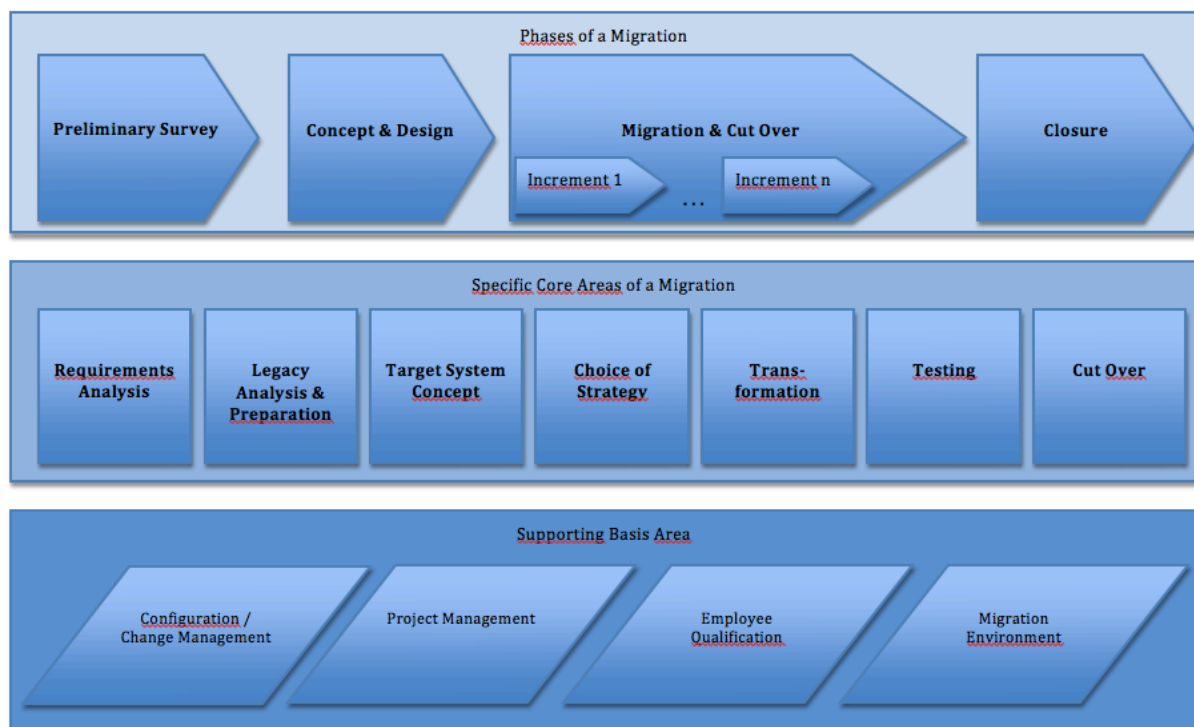


FIG. 11.1 REFERENCE MIGRATION PROCESS AND ITS VIEWS (SNWH10)

²⁵ <http://www-01.ibm.com/software/awdtools/rup/>

ReMiP is grouped into three views – compare Fig. 11.1 –describing the phases of a migration, the specific core areas of a migration and the supporting areas of a project.

11.1. Phases of the Reference Migration Process

The phases of a migration are as follows

Preliminary survey

In this phase analysing and collecting information that is necessary for the development of a solid migration strategy set up the project. The milestone for the preliminary survey is to define the goals of the project and to suggest a tentative course of action.

Concept and design

Based on the preliminary survey and under consideration of business requirements, of given resources and of the technical environment a binding master plan is developed in which migration strategy, target system and target system environment are described in detail. The master plan has to be evaluated and accepted by the stakeholders.

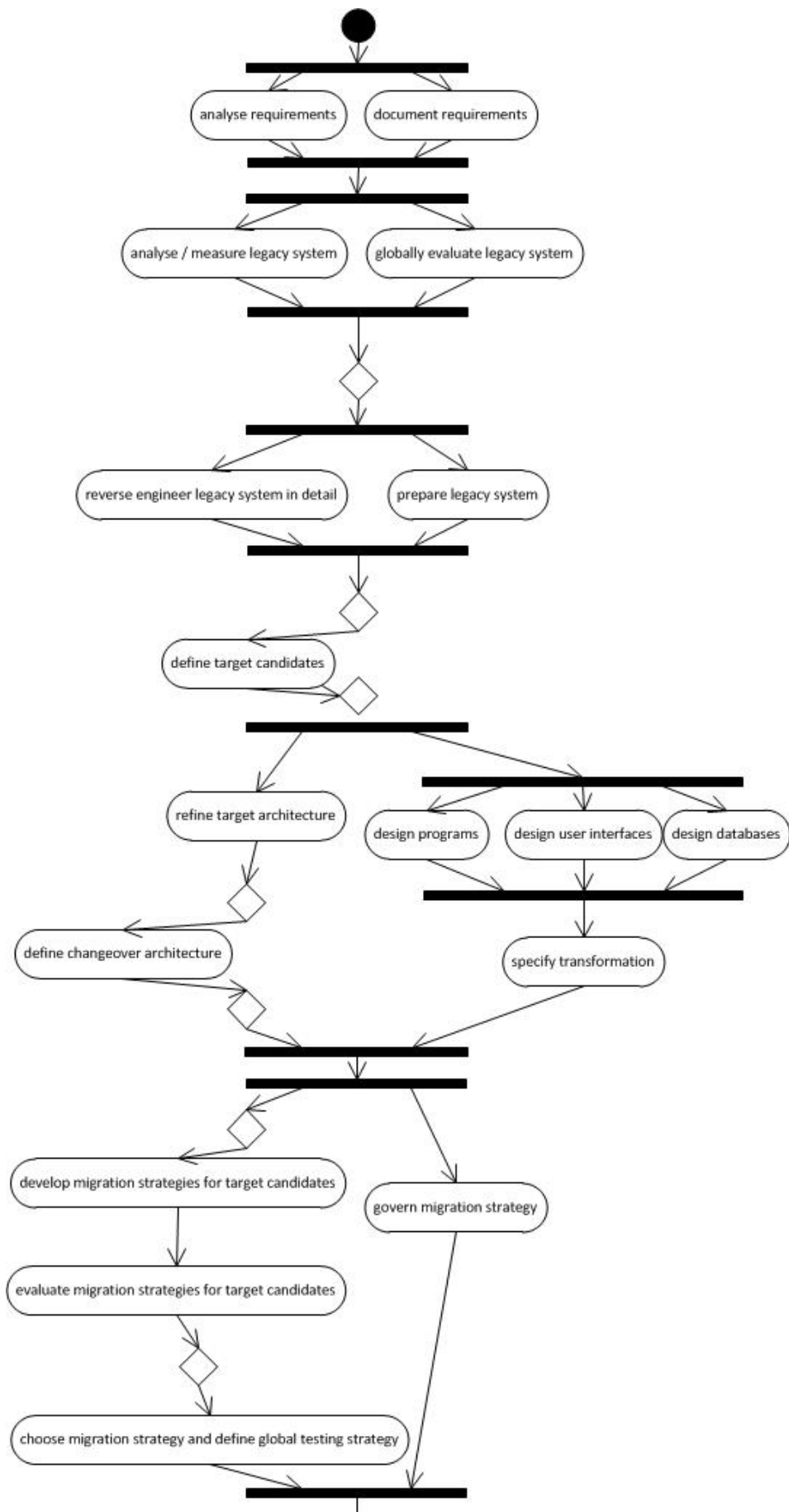
Migration and cut over

During this phase the system is transformed, tested and delivered in several increments.²⁶ After all increments are executed the milestone of the migrated system-release is achieved. That is only the case if the whole system has successfully been transformed, tested and installed in its target environment.

Closure

Closure involves checking and verifying the resulting system, user acceptance and a post calculation including a target/actual comparison. The last milestone for the project is the final documentation also including the before-mentioned verification.

²⁶ ReMiP would also be suitable for a big bang migration, which needs to be mentioned but is not of interest in this thesis though.



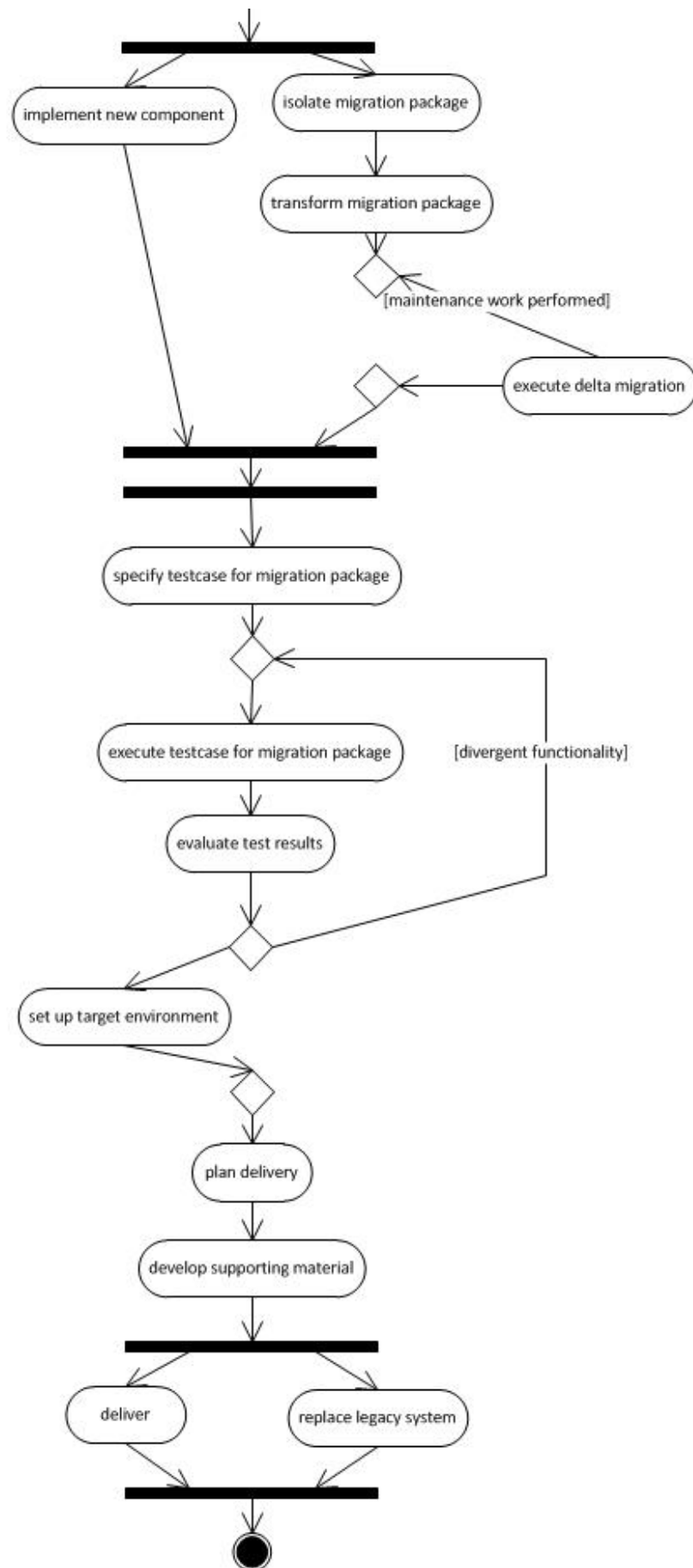


FIG. 11.2 ACTIVITIES OF THE REMIP (SNWH10)

Fig. 11.2 shows the ReMiP and its activities as an UML-activity diagram. The subsections of the process are described in more detail as follows:

Requirements analysis

Before a migration can take place the requirements need to be established. Therefore existing software and data has to be analysed in an – where possible – automated manner. The measured values will help to make statements about the project regarding goals, size, cost, final inspection criteria and functional and non-functional requirements. Although as already mentioned before the functional requirements are staying the same as in the legacy system and the focus lies more on the non-functional requirements as system quality, architecture, programming language, user interfaces etc. At that time techniques and approaches presented in chapter 8 will then help to find out about stakeholder views and wishes. When all the requirements are collected temporary components like gateways, wrappers and tools have to be defined and – if existing – appropriate commercial tools can be chosen. Having all this in place documentation has to record all the requirements including functional requirements gained through reverse engineering of the legacy (system in sufficient detail for the project).

Legacy analysis and preparation

To come up with a reliable plan for a migration the system first has to be analysed and measured. It therefore is important to inventory and evaluate all existing and running software components within the legacy system and to categorise them into programs, libraries, scripts, files and directories and databases. Evaluation in this matter means to describe the quality and complexity of all findings to have a means for calculating the project cost. Where possible existing documentation of the legacy system and people – if at hand – that know the legacy system should be interviewed. Following the guidelines in chapter 9.2 it has to be decided whether a legacy system should be migrated at all or which parts of it should be included in the actual project. So it is also possible that the system's quality or re-usability are not satisfying leading to a replacement from scratch or the use of COTS instead. After taking these decisions the legacy system is undergoing reverse engineering, whereas it is depending on the size of code, the amount of components, the chosen migration strategy and the available knowledge of experts or existing documentation to which extent reverse engineering techniques and tools should be used. The appropriate

trade-off between knowing it all, which is expensive and takes time, and a blind migration, which is faster but more risky, has to be found. But in any case it is important to detect which parts of the legacy system must undergo restructuring to facilitate a smoother transformation.²⁷ The goal of restructuring is to facilitate the definition of independent manageable migration packages that are needed for an incremental approach. Like that each increment will transform packages ensuring the functionality of the whole system during the project. And of course the documentation has to be updated after restructuring and it should already be defined which packages should be migrated simultaneously.

Target system concept

The goal of the system transformation is to retain as much functionality as possible from the legacy system, but to also ensure that the new system conforms to the overall EA strategy. Before being able to migrate it is important to get familiar with possible target technologies and to design different possible target system architectures and to evaluate them. The most suitable architecture will then be chosen and refined as needed for each increment. It is not necessary to have the whole target architecture in place in this early stage, because unplanned changes can and will appear during each increment due to new findings in the legacy system or due to changed requirements from stakeholders. As long as the whole migration has not fully taken place there will be a need for a transitional architecture that describes the necessary gateways. The challenge for the design of the new programs is to make use of the new chosen technology and to still preserve existing knowledge and processes. For the interfaces of the system-to-be usability is the most important thing to achieve and finally databases usually have to be transformed retaining their data and data structure, if it is not a data migration effort. After these design phases the actual transformation has to be defined to describe how the legacy will be transformed to the target system for each increment. Depending on how specific these rules can be established it will be possible to automate some of the transformations – e.g. transform data types from the old language to the target language. Having all this in place requirements for the transitions environment (hardware and software and the necessary configuration) and still open, smaller preparing restructuring needs become clear.

²⁷ In some cases it can even be possible that an own restructuring project must take place before the actual migration project.

Choice of strategy

A global migration strategy is usually comprised by various transformation strategies for each target candidate whereas strategies regarding (SNWH10) are reimplementation, conversion and encapsulation. All of those can take place in either a big bang or incremental manner. Regarding the cut-over of the systems again a cold-turkey approach or an incremental one can – depending on the decision for each target candidate – be applied. In case of the incremental version it is possible that already cut-over packages will replace the legacy package and be operative at once or that they are part of a second redundant system beside the legacy one. If an incremental approach is possible a definition of the potential package building approach has to be elaborated and the defined packages then ranked regarding their priority. In both steps it is very important to see the dependencies between the packages and to find out about the necessary restructuring requirements. All possible migration strategies for target candidates then need to be evaluated and checked whether they are realisable within the given constraints of the system. After pre-selecting appropriate strategies it is now time to choose a strategy following evaluation regarding complexity, cost and risk on the one hand and a cost / benefit analysis on the other hand. Each strategy therefore needs to be refined to establish the necessary input data regarding risk and finance matters. Based on these results the final decision which migration strategy to choose is made and documented to serve as input for the concrete transformation and project planning.

To support an incremental way of planning the migration the identified strategies including newfound risk, dependencies have to be documented. This ensures to fully take advantage of the incremental approach of planning because when requirements or even target technologies should change during these first phases one can go back to data that already is documented and start again from there.

Transformation

During the transformation phase the defined migration packages are being transformed into its target structure according to their migration strategy. Conversion tools, out sourcing of the transformation and in-house implementation are the available means for the transformation. But first of all the package to be migrated have to be isolated into the conversion environment to ensure that on-going maintenance is not interfering with the migration. Of course if legacy system maintenance is allowed – a decision, that is different for each package – and taking

place during the migration of a package it has to follow certain principles to be able to reproduce it in a delta migration. Delta in this case means the difference between the already transformed package and the legacy system including the changes through maintenance. It is also possible, that an implementation of necessary gateways of re-implementation of software components that are not suitable for a transformation will take place.

Testing

Testing ensures that the migrated system in its target environment shows the same functionality as the legacy system. It is by far the biggest and most important part of a migration and usually takes more than half of the work of the whole project. First of all planning and preparing the tests is a crucial task and a global testing strategy should already be defined after establishing the global migration strategy. In this step it is important to define goals, testing environment, involved roles and the necessary test data. Furthermore – as testing is such a big effort – automated tests should be put in place where ever possible. Having the global testing strategy established the testing specification will be refined for each migration package. Main means of testing are so-called regression tests that are used in combination with test data in a test suite. The tests will be run on both the legacy and the target system. A successful transformation is only achieved when the results match 100 per cent. In any other case the transformation needs to be revised and again tested until a full match is achieved.

Cut over

The final phase deals with the cut over of the system and is already defined in the global migration strategy. It includes installing, configuring and testing the target environment. Just like for every other step in the ReMiP a plan for the cut over has to be defined which describes in detail how each migration package will be cut over and when and how the related legacy component will be displaced. After successfully testing the package the so-called migration release installation instructions will be documented and all necessary documents for end user and persons responsible for maintenance have to be generated. If user interfaces were changed during the migration especially end user documentation is a valuable source for user training. After that the actual cut over of a migration release can take place following the cut over plan. This can for example involve for example that temporary gateways need to

be installed first. Final acceptance testing is performed by the actual users of the target system and already takes place in the target environment.

The final step of the migration process is then displacing the legacy system by each migration release and de-installing related temporary gateways until the whole legacy system is successfully migrated. After archiving the legacy components the project can be closed.

12. Enterprise Architecture

“The ability to *change* systems is a more important goal than the ability to build them *ab initio*.” (LFAN03) And migration is one important part what “change” in these quotes is all about and especially the last chapter took an elaborate look at this topic. But another essential part to achieve and adaptable and flexible IT system is to have a mechanism in place that facilitates the alignment of business and its IT systems.

The therefore needed overview can be provided by an enterprise architecture (EA), that captures the essentials of the business, IT and its evolution (MALA09). The EA should deliver an overview of the organisation without going too deep into detail of implementations. For example quoting all the tasks of a sub process to a high-level order process would be too much information. From a technology point of view it would be too much to explain – given an object oriented environment – to enumerate all e.g. JAVA-objects of the system. In many cases a migration project will not have the luxury to find an EA already in place but can help to establish EA within an organisation as it is aiming at the same goals such as flexibility, adaptability and efficiency of the system and therefore the whole business. So it becomes quite obvious that the migration of IT infrastructures and EA have a big overlapping because – as described in Fig. 12.1 (RUPA10) – the application, infrastructure and technology architecture form a major part of an EA.

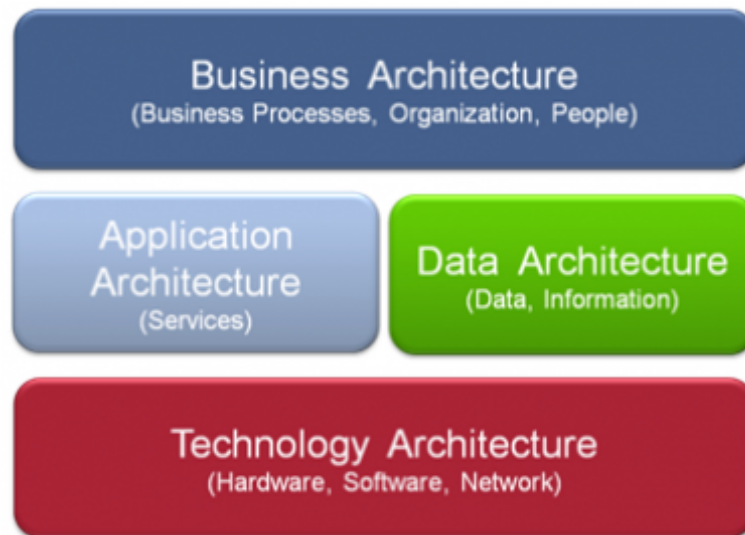


FIG. 12.1 COMPONENTS OF AN EA (RUPA10)

For establishing an EA there are quite a few frameworks ready to use. Starting from the Zachman Enterprise Architecture Framework to The Open Group Architecture Framework (TOGAF) and the US Federal Enterprise Architecture Framework (FEAF) or Department of Defence Architecture Framework (DoDAF) ranging to a further development as the Extended Enterprise Architecture Framework (E2AF) just to mention the most known ones.

12.1. TOGAF

For the use of this paper TOGAF has been chosen to exemplify how EA works and to show how EA helps in the context of migrations. TOGAF is comprised by the following four components that will only be discussed shortly in order to give an overview.

Business Architecture

The goal in this section is to represent the structure of an organisation through describing their stakeholders, business processes, business functions and rules, products, business units and business objects and the relationships between them. Whereas for example a business object could be an order within a company and describe what is going into the header or the content parts of such an order. Later on an information object of an application described in the application architecture can refer to a business object.

Application Architecture

This part of the TOGAF framework describes the landscape of applications, interfaces between those applications or users and the company's information

objects together with their relationships between them. As described earlier information objects are the specific data of business objects and will link to the appropriate pendant in the business process architecture.

Data Architecture

The data architecture describes the metadata and the data models of an enterprise's data structures. Moreover it depicts the association between the data and the principles of how it is organised and managed in the context of the application language. Together with the application architecture they are also referred to as information system architecture.

Technology Architecture

The technology architecture should describe how the applications are implemented and on which infrastructure elements they are deployed. Preferably this happens in a so-called Configuration Management Database (CMDB) as introduced in the IT Infrastructure Library (ITIL)²⁸. The standards and technologies in use have to be pointed out and it should define the reference architectures e.g. whether a single tier or multi-tier architecture is in place and which software frameworks e.g. .Net and Off-The-Shelf-Software e.g. ERP systems are employed.

Regarding the granularity of information kept in the models and descriptions of the above named sub-architectures it heavily depends what is needed for the migration project. For the general EAM usage it is always important to decide the balance between necessary detail to take managerial decisions and the effort to maintain actuality. The latter is essential to have an accurate image of the company and to get real analyse results. In the end a poorly maintained EA that is not up-to-date is nearly as good as now EA at all.

And as can be seen in Fig. 4.1 of course an IT strategy should determine the long-term goals of IT in general. On the other hand the business strategy comprises the IT strategy and defines where the journey of the company will be going in the long run.

The outline in Fig. 4.1 and EA in general shows pretty clear how all the four different views – that is what each sub-architecture in the end is – are very much interrelated and should not be addressed in a separate way. So if a company for example just simply wants to replace a machine it might not know which applications and then in the end business processes it affects if there is no adequate information available as

²⁸ <http://www.itiil-officialsite.com/>

EA provides it. But of course also if the business process changes it is necessary to know where the corresponding information objects are located in the various applications to make the appropriate changes in the software. In individual cases such as single maintenance tasks it might not be too big of a problem to find out which lines of code have to be extended or changed to achieve the desired behaviour and therefore the impression could be received that EA is yet another documentation overhead. But when whole parts of the system are to be modified it becomes complex quite fast. And as proofed by research at Warwick Business School already done in 2007 it showed that in the retail industry, where for example Amazon would be one of the biggest players, downtime cost are estimated as 350.000 GBP per hour (REYN07). So it is quite understandable that nowadays nobody can risk system downtime just because of some trial and error maintenance or modernization and trial and error it actually is if one is not paying attention to the possible effects that a change in code might have on other applications or a business process as a whole.

12.2. TOGAF Architecture Development Method

The Architecture Development Method is at the heart of TOGAF and describes a method for the introduction and development of an EA which is supported by a common description language and architectural artefacts in the Enterprise Continuum of TOGAF and a resource basis offering processes to govern EA, architecture principles and necessary skills of roles within the whole process. It is comprised by eight phases that are started by a preliminary phase of deciding which framework and which architecture principles to choose and defining the responsibilities for activities within the whole process. All phases need to consider the business requirements following the business and IT strategy. Fig. 12.2 (OPGR09) depicts the method, which is incremental within each phase and also incremental as the last phase is followed by the first again. This leads to an ongoing discussion about EA throughout its lifetime and assures to follow the requirements of EA.

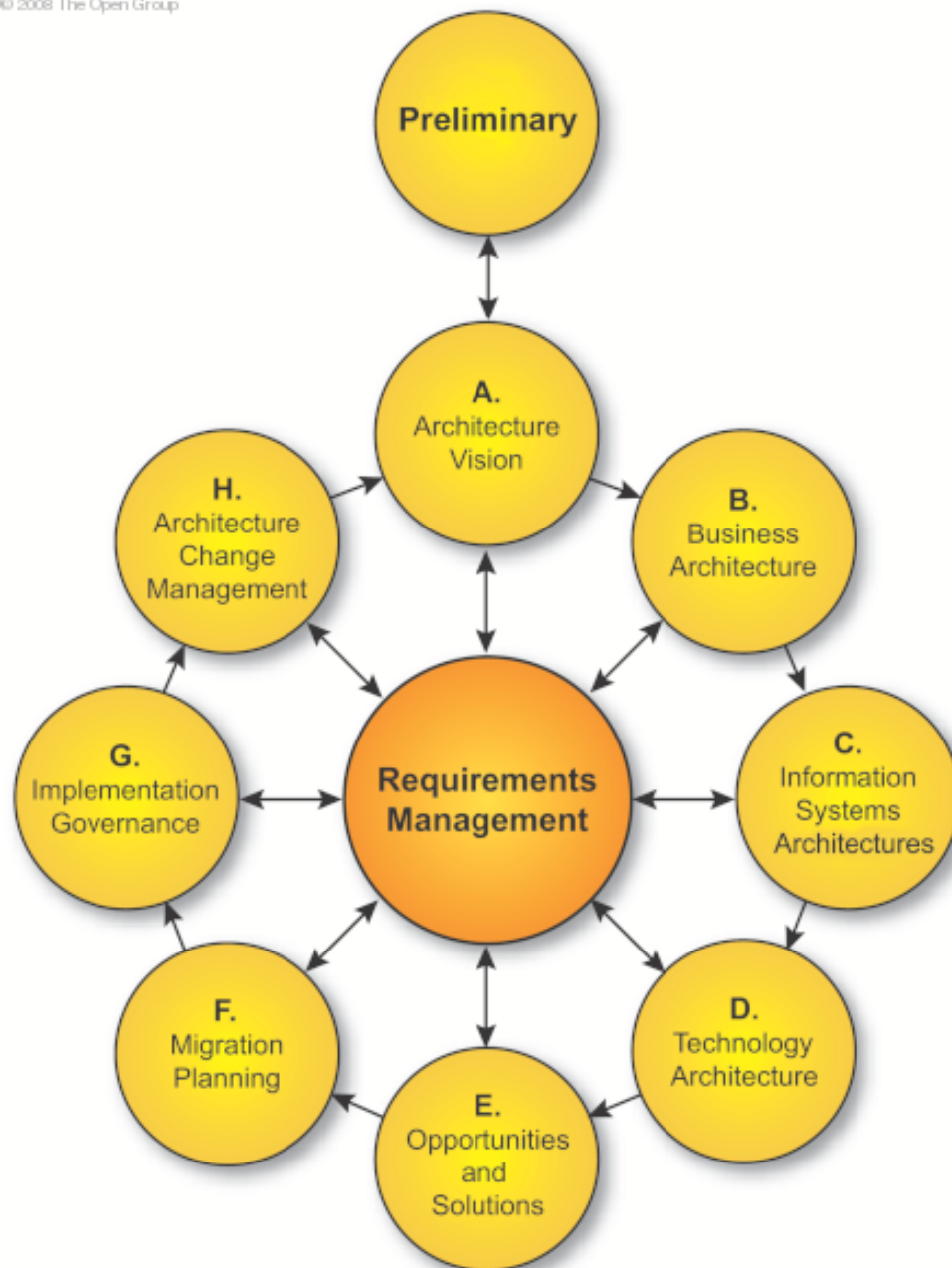


FIG.12.2 ARCHITECTURE DEVELOPMENT METHOD (ADM)²⁹

In phase A the architectural vision needs to be established. Therefore it is important to find out about the stakeholders and their strategic goals. Together with the definition of architecture principles the scope of the planned EA should be elaborated. In the second stage, phase B, the existing EA is investigated and the future-EA needs to be defined. Products and services are described and the difference between as-is-EA and to-be-EA is determined. Tools and techniques are agreed on to support presentations for stakeholders. A common means for modelling

²⁹ <http://www.opengroup.org/architecture/togaf9-doc/arch/>

are for example UML use-case-diagrams and class diagrams. Having this in place an architecture model will be developed and architecture patterns and building blocks chosen. Non-functional criteria, like performance, cost and volume will also be discussed. The phase Information Systems Architectures focuses on the development of the information system architecture, including a prototype, and a review of the features of the architecture. After that, the architecture should be completed and gaps should be determined. Phase D concentrates on designing the technology architecture to form the basis for the implementation. The Opportunities and Solutions stage evaluates the different elaborated target architectures in terms of costs and benefits and chooses the most suitable one. Deliverables of this stage are an implementation and a migration plan. Phase F then establishes the dependencies between other ongoing projects and activities and should predict costs, benefits and risks to be able to decide whether the migration can and should even take place. Documentation and a road map for the migration are the deliverables of the activities in this stage. The subsequent phase deals with recommendations of the implementation project and builds up the architecture contract defining implementation and development processes. Finally the Architecture Change Management stage should ensure a smooth migration to the new architecture and applies change management processes. Most important in this phase is the decision to update or adapt the architecture in place and therefore to trigger another iteration of the whole ADM. Following this method ensures that EA is applying necessary changes when they occur.

13. ReMiP and TOGAF – a combined view

ReMiP provides a very good means to a single migration effort but it stops after the migration. Any further migrations are more or less taking place disconnected from the previous one and will not follow a strategic approach of governing IT as such. Already (WIUL02) suggests that a migration effort should be aiming at establishing central unit for IT architecture but is not considering EA as an appropriate means for that. This is why this paper suggests establishing a combination of TOGAF and ReMiP to gain benefit from both frameworks and to consistently establish an organisation of change. Fig. 13.1 depicts where TOGAF shows similarities to ReMiP and also where it can deliver input for it.

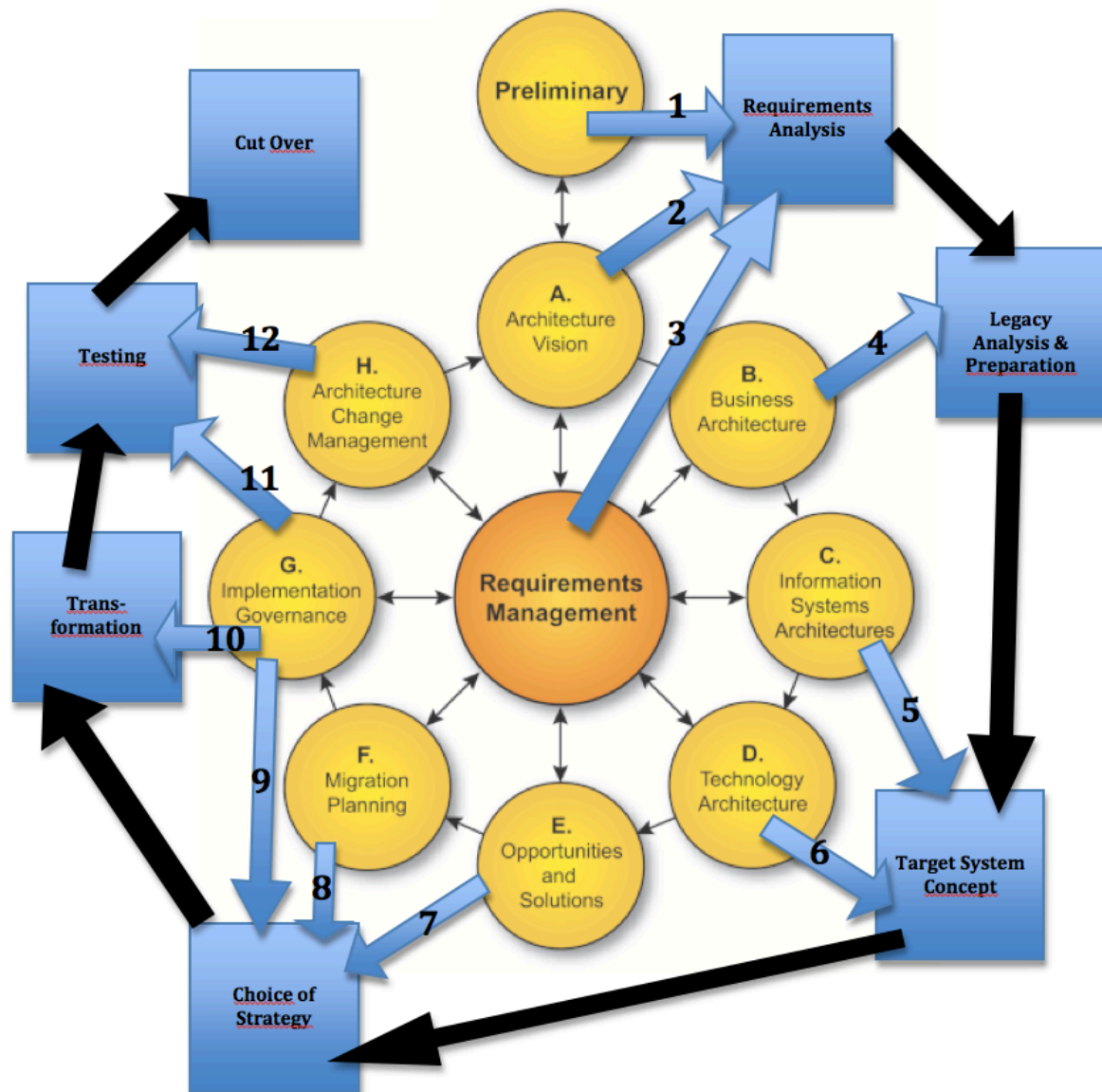


FIG. 13.1 COMBINED VIEW OF TOGAF AND REMIP

The figure is set-up through overlapping the TOGAF framework in yellow with the corresponding parts of the ReMiP in blue. Like that a clear overview can be illustrated. The black arrows of the ReMiP follow the procedural sequence in which they are already connected in the original process description. The necessary connections between the two approaches is realised through the numbered blue arrows, which will each be discussed subsequently. In general the requirements management of TOGAF transitorily influences all process steps of ReMiP through their affiliated TOGAF phases.

Connection 1

Already beginning in the preliminary phase the TOGAF ADM can provide input to the requirement analysis of ReMiP because it finds out which stakeholders are

concerned and involved in EA and therefore maybe also in activities of a migration process.

Connection 2

The already existing architecture principles and responsibilities from phase B influence the requirements of the migration and clarify who is responsible for which decision and input as the ReMiP is going ahead. But also the stakeholders' strategic goals and the establishment of the scope for the migration can be taken into account for migration process.

Connection 3

TOGAF is a requirements-driven approach. Therefore the requirements management for first of all establishing and then monitoring and adapting requirements is crucial and influencing the whole ADM. Quite obviously the inputs coming from this part first of all relate smoothly to the requirements analysis phase of ReMiP. But through its vital importance and position within EA it indirectly affects all other connected ReMiP stages too.

Connection 4

Phase B in ADM again delivers valuable preparatory work for ReMiP as both the existing and the future EA are discussed. Also the representation through UML models in EA is suitable as incoming information and helps to inventory and evaluate existing and planned architecture as it resembles documentation about the system as well. The documentation within EA might need further elaboration but in general the similarities in the prevalent trade-off between collecting everything in detail and gaining just an overview are going in the same direction. Finally the future EA influences and delivers information for the ReMiP target system concept too.

Connection 5

The information systems architecture stage provides the overall strategy that is needed to be able to decide which technologies come into question.

Connection 6

Results from investigating and studying these technologies already come from the technology architecture stage within ADM and help respectively preliminary decides which technology to choose for the target system concept.

Connection 7

Opportunities and solutions phase of TOGAF more or less replace the evaluation of different target architectures regarding cost and benefits and guiding the global migration strategy of ReMiP.

Connection 8

The choice of strategy in the migration process also needs to know about dependencies and involved risk. The migration planning stage of ADM delivers exactly this information.

Connection 9

As in phase G, implementation governance, already defines implementation and development processes. Through connection 7, 8 and 9 ReMiP can take on from there with choosing the appropriate global migration strategy following the road map for migration of TOGAF.

Connection 10

But implementation governance also influences the transformation itself through its process definition.

Connection 11

The same goes for testing processes, as they are also part of the whole implementation process of phase G.

Connection 12

Finally the architecture change management facilitates a smooth cut over to the new architecture and ReMiP ensures that the displacement of the old system follows along controlled activities.

The established connections in Fig. 13.1 should point out that there is quite a big overlapping the two approaches whereas especially TOGAF often delivers valuable input for the migration process. But it also shows pretty clear that ReMiP lacks a similar construct as phase H “Architecture Change Management” in TOGAF. Of course ReMiP satisfies the need for a single migration and provides a very helpful means for it, but only if a company also has an EA in place it will be able to establish a more integrated strategy for IT and the alignment of IT and business requirements. As already the introduction of an EA will be a big effort in a best-case scenario an EA should be established before considering a software migration following ReMiP, but as of course this can not be ensured because of the urgency of a migration at least

the retrieved documentation of the migration project can then contribute to the implementation of TOGAF.

14. Expected Benefits

Apart from ReMiP so far most publications in academia and industry have lacked a holistic view on the topic of migrations neglecting the already existing knowledge. Although migrations are not a new chapter to IT anymore and are concerning a broad range of organisations – if not even all – not much effort has been undertaken so far to elaborate methodologies such in classic software engineering with their prominent examples from the waterfall model up to agile approaches. The focus usually was lying on special case migrations e.g. towards a SOA or narrowed down certain aspects of information systems re-engineering. ReMiP is the first promising candidate trying to incorporate process model knowledge such as RUP and to abstract from special case approaches. It provides an already elaborate road map to a successful system transformation. But there is still a lack of integrating even more holistic methodologies as EA – in the case of this paper represented through TOGAF. This combined view of both frameworks helps though to create the long-term context in which migrations should take place. Because in the long run it is desirable and important to follow the IT strategy which is an essential part of an EA to ensure both the necessary agility and stability of an IT system, but also to facilitate maintenance or compliance to risk management and security policies. ReMiP itself is a good means for a single migration but it stops there and can ideally be extended through and combined with TOGAF to achieve a better alignment of business and IT strategy.

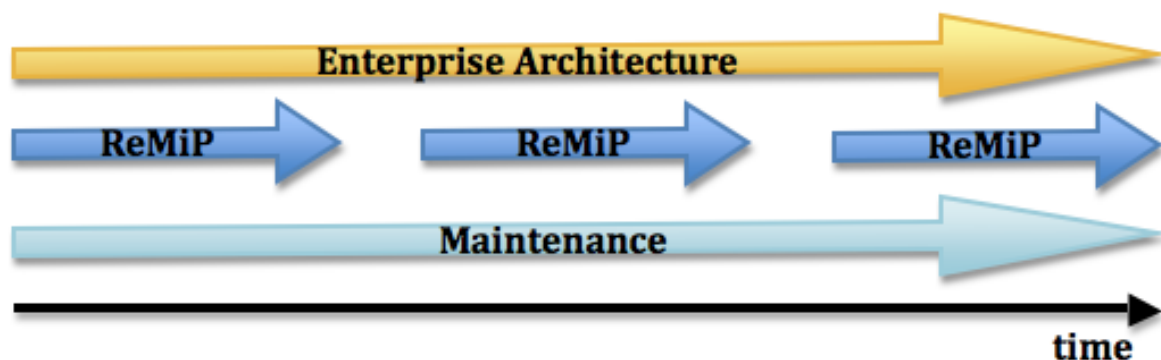


FIG. 14.1 COMPARISON OF SYSTEM CHANGING APPROACHES REGARDING TIME

Fig. 14.1 depicts in a simple way that put on a time scale a migration – as ReMiP guides it – is taking place within and focussing on a shorter time span than EA approaches like TOGAF. Furthermore a migration does not necessarily concern the whole IT infrastructure of a company and therefore maintenance work for other parts of the system will still take place at the same time as the migration takes place, but definitely occurs in between migration projects. So although ReMiP preserves important information like lessons learned and architectural decisions EA helps to also keep track of changes within IT architecture that take place between migration projects and build a good information and decision basis for another necessary execution of a migration.

In addition to the contributions of RUP and TOGAF towards project management e.g. defining roles and responsibilities for tasks this paper presents theories and techniques to address stakeholders and participants motivation and ensure their positive contributions. This is believed to lead to better project results on the whole. This work presents a holistic view on the topic of migrations of legacy systems by discussing the origins of the whole problem up to introducing and referencing important and helpful framework and techniques e.g. such as risk management guidelines from the PMI project management method.

15. The Reference Migration Model

So after mapping and pointing out the expected benefits this chapter incorporates both of them into an integrated procedural model. Following Fig. 14.1 in which the general overview and connection were established Fig. 15.1 now subsequently builds up an executable model. The model is inspired by the V-Model XT³⁰ (FHKS09) and incorporates the idea of increments on a high level rather than on a e.g. prototype level of systems or components. This abstraction leads to the desired behaviour that a migration decision is triggered through change management of an Enterprise Architecture framework e.g. like the presented TOGAF in this thesis.

³⁰ Official website of the V-Model: <http://www.v-modell-xt.de/>

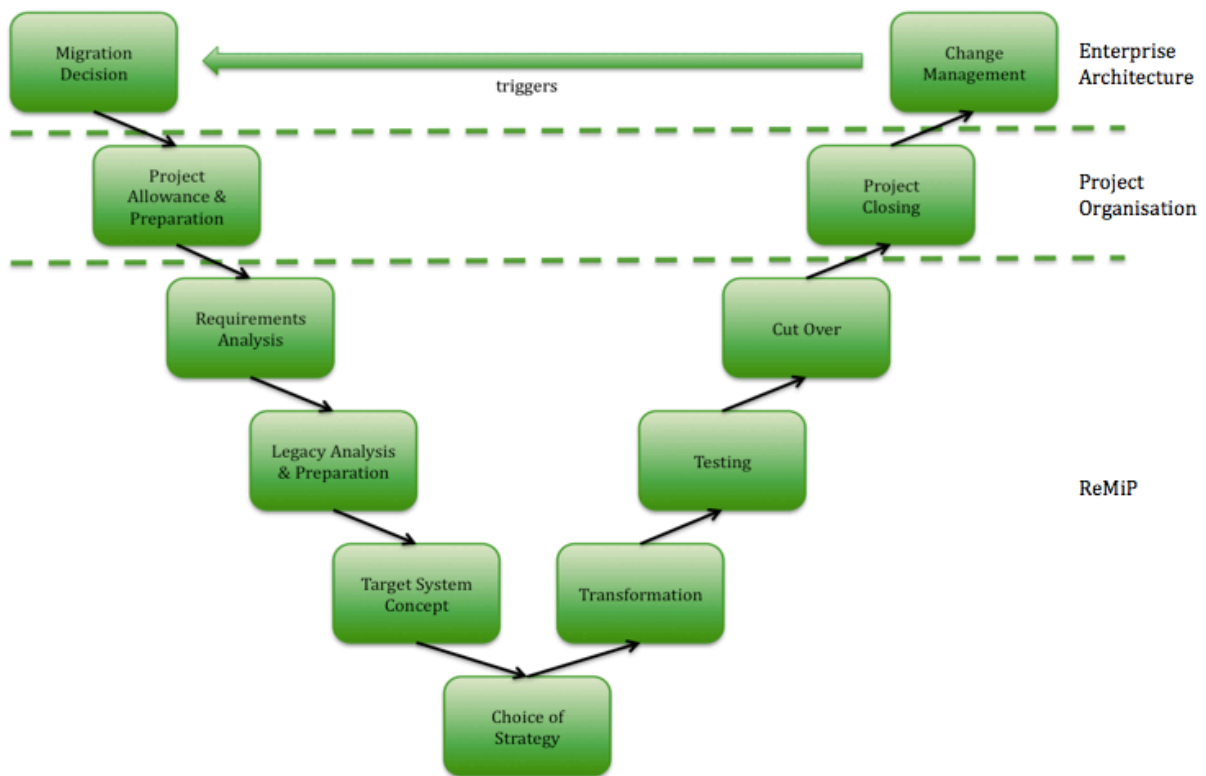


FIG. 15.1 REFERENCE MIGRATION MODEL

But although the model in Fig. 15.1 is inspired by the V-Model the only ideas adopted from it are the concepts of the Project Allowance and Preparation and Project Closure phase. Through these two phases the gap between EA and ReMiP can be bridged. The main body is formed by ReMiP – recognised as the most promising approach. So at the beginning the given information through an EA will lead to the question whether a migration should take place in order to fulfil the architecture vision of the EA. This especially refers to other system evolution possibilities presented as maintenance in chapter 5.1.1. So if maintenance is not capable of making the desired improvements anymore a migration should take place. All this considerations and steps are so far taking place in the EA area of the model as indicated through the first dotted separation line. With the more high level information – high level at least compared to the input needed for the actual transformation of the system – and the techniques presented in chapters 9 and 10 the necessary arguments will be presented to the management. Upon this information a decision whether a migration project takes place or not has to be made. Together with the final step of closing the project this forms the area of the project organisation that actually frames the following area of the ReMiP. Again, both areas are separated through dotted lines. Once more it has to be stressed that ReMiP is an incremental approach and the connecting arrows just indicate the overall direction of the ReMiP phases.

The three areas help to illustrate how the combined view of TOGAF and ReMiP presented in chapter 13 can be connected via the project organisation area because the two frameworks themselves lack to introduce phases like Project Allowance. But exactly the project organisation area connects the different timelines of EA and ReMiP in Fig. 14.1. The strategic motivation of EA is then capable of initiating a certain migration project consequently following the ReMiP guidelines. After the phases of ReMiP are successfully executed and finished the Project Closure phase then seamlessly leads over to the Change Management phase of TOGAF by taking in the necessary documentation gained through the migration and so updating existing documents of the EA.

16. Check Against Practitioners' Requirements

The proposed combined view of TOGAF and ReMiP as an approach to configure the migration of IT infrastructures was presented to a practitioners working at a well known consulting company with long expertise in the field of migrations. The results of the interview generally acknowledge the proposed proceeding but point out divergences and points of focus. Because although an incremental migration is one of the main drivers for a risk-minimised migration it shows that especially in the banking sector where custom developed host applications have to be migrated a big bang cut-over is prevalent despite the higher risk. This stems from the fact that especially the banking sector with a hardware architecture often starting from the size of 100 mainframes and several hundreds of interdependent applications represents a hard to decompose system where time and financial constraints limit the number of increments and lead to a favoured big bang cut-over with a strong emphasis on testing to ensure the accuracy of the new system and minimise risk. In the case the migration takes longer than two or three days – for example during a weekend – to cut over also manual supplements in data are considered to still ensure the availability of the system. The focus on the communication with the stakeholders has been acknowledged with a special reference to the problem of unclear responsibilities especially in architecture matters. This can lead to the unwanted situation of long response time in certain architecture decisions and to the hazard of decisions not conforming a global IT strategy. Regarding the expert's view the most crucial activities of the migration are those establishing the requirements, the choice of strategy, the actual transformation including testing and the final cutover. Of

course all other activities are important and need to be carried carefully too, but it is in these mentioned steps where many projects experience problems and delays.

General advices and remarks deal with the fact that in many cases political decisions will determine how the target system has to look like. This can either mean choosing a certain vendor but also a certain architecture. Moreover it is desirable to have an EA already in place rather than establishing it at the same time as executing the migration because it would go way beyond the scope of the migration. Nevertheless it is acknowledged that the knowledge – e.g. in form of the produced documentation – of the transformation deliver input for the introduction of an EA.

17. Summary and Conclusion

The outline of this thesis is comprised by integrating ideas and theories in the field of EA, insights about legacy software including the evolution of software with knowledge in place about how to address stakeholders involved in migration of IT systems and processes and tools involved. The presented procedural model finally depicts how the discussed frameworks and techniques of this paper can co-operate. This holistic view responds to the problem field in an appropriate manner that is able to minimise the involved risk and will lead to better results. So it is not necessary to reinvent the wheel in migrating legacy software but just to find an integrated way of using what is actually already at our hands. Doing so it should be possible to raise the quality of migration projects and therefore IT projects on the whole and to ensure that they more and more meet their stakeholder's expectations.

LIST OF FIGURES AND TABLES

TABLE 3.1 OVERVIEW OF BUSINESS AND IT ARCHITECTURES (DIMA07)	9
FIG. 4.1 FORMULA FOR THE VOLATILITY OF A SYSTEM (DIMA10)	16
FIG. 4.2 EMPIRICAL VOLATILITY INDEX FUNCTION (DIMA10)	17
FIG. 5.3 VOLATILITY OF PROGRAMMING LANGUAGES (DIMA10)	17
FIG. 4.4 LINEAR EMPIRICAL RELATIONSHIP OF THE VOLATILITY INDEX (DIMA10)	17
FIG. 4.5 VOLATILITY INDEX APPROXIMATION (DIMA10)	17
FIG. 4.6 VOLATILITY INDEX DIAGRAM FOR COBOL AND JAVA (DIMA10)	18
FIG. 4.7 VOLATILITY INDEX DIAGRAM FOR SOFTWARE (DIMA10)	19
TABLE 4.1 OVERVIEW OF ANTI-PATTERNS IN LEGACY SYSTEMS (LAKE00)	21
FIG. 4.8 SPAGHETTI ARCHITECTURE OF EAI (DIMA05)	26
TABLE 5.1 LEHMAN LAWS (LEBE85)	28
FIG. 5.1 NON-CONTINUOUS DISCONTINUITY OF ENTROPY CURVE OF A SYSTEM (DIMA05) ..	30
FIG. 5.2 INFORMATION SYSTEM LIFE CYCLE (SEPL03)	31
FIG. 5.3 TERMS IN SOFTWARE ENGINEERING (CHCR90)	33
FIG. 5.4 TYPES OF MIGRATIONS AND MIGRATION ASPECTS (GIWI05)	38
FIG. 5.5 MAINTENANCE COST DEVELOPMENT OF AN EA (DIMA05)	42
TABLE 7.1 LAYERS OF AN IT SYSTEM (DIMA10)	47
FIG. 7.1 EXAMPLE FOR CURRENT SYSTEM TOPOLOGIES (DIMA10)	48
FIG. 7.2 XML AS CANONICAL INTERNAL FORMAT (DIMA10)	48
FIG. 7.3 JAVA CONNECTOR ARCHITECTURE (SUN03)	49
FIG. 7.4 RELATION BETWEEN CIL AND OTHER LANGUAGES (WEST02)	50
FIG. 7.5 WEB SERVICES PROTOCOL STACK	52
FIG. 7.6 SAAS PARTICIPANTS AND THEIR RELATIONSHIPS (ARAR10)	54
FIG. 8.1 ACCEPTANCE AND USE OF TECHNOLOGY (UTAUT) (VEMO03)	62
FIG. 9.1 PORTFOLIO ANALYSIS OF LEGACY SYSTEMS (WAIA99)	66
FIG. 10.1 PAYBACK PERIOD RULE (DIAL05)	68
FIG. 10.2 RETURN ON INVESTMENT FORMULA (SEPL03)	69
TABLE 10.1 COST / BENEFIT ANALYSIS EXAMPLE (WISM03)	71
FIG. 11.1 REFERENCE MIGRATION PROCESS AND ITS VIEWS (SNWH10)	72
FIG. 11.2 ACTIVITIES OF THE REMIP (SNWH10)	75
FIG. 12.1 COMPONENTS OF AN EA (RUPA10)	81
FIG.12.2 ARCHITECTURE DEVELOPMENT METHOD (ADM)	84
FIG. 13.1 COMBINED VIEW OF TOGAF AND REMIP	86
FIG. 14.1 COMPARISON OF SYSTEM CHANGING APPROACHES REGARDING TIME	89
FIG. 15.1 REFERENCE MIGRATION MODEL	91

REFERENCE LIST

- [AMSC10] Ambler, S. W. (2010, November 29). *Agile Modeling*. Retrieved November 29, 2010 from <http://www.agilemodeling.com/essays/enterpriseModelingAntiPatterns.htm>
- [ARAR10] Michael Arbrust, A. F. (2010). A View Of Cloud Computing. *Communications of the ACM* , 53 (4).
- [BORC95] Borchers, J. (1995, September). Erfolgsmechanismen großer Reengineering-Maßnahmen. *Software-Entwicklung - Methoden, Werkzeuge, Erfahrungen '95* .
- [BROO75] Jr., F. P. (1975). *The mythical man-month*. USA: Addison-Wesley Publishing Company, Inc.
- [BRST95], M. B. (1995). *Migrating Legacy Systems: Gateways, Interfaces & The Incremental Approach*. California: Morgan Kaufmann Publishers.
- [CAPE06] Jones, C. (2006, February 14). <http://www.compaid.com/>. Retrieved February 08, 2010 from Computer Aid, Inc.: <http://www.compaid.com/caiinternet/ezone/capersjones-maintenance.pdf>
- [CHCR90] Chikofsky, E. J., & II, J. H. (1990). *Reverse Engineering and Design Recovery: A Taxonomy*. Auburn/Cambridge: Institute of Electrical and Electronics Engineers (IEEE).
- [CNET05] cnet.com. (2005, 04 19). *cnet.com*. (M. Kanellos, Editor, & M. Kanellos, Producer) Retrieved 08 05, 2010 from http://news.cnet.com/New-life-for-Moores-Law/2009-1006_3-5672485.html
- [COME68] Conway, M. E. (1968, April). How Do Committees Invent? *Datamation magazine* .
- [DIMA05] Masak, D. (2005). *Moderne Enterprise Architekturen*. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg.
- [DIMA07] Masak Dieter. (2007). *SOA? Serviceorientierung in Business und Software*. Berlin Heidelberg: Springer Verlag.
- [DIMA10], M. D. (2010). *Der Architekturreview - Vorgehensweise, Konzepte und Praktiken*. Berlin-Heidelberg: Springer-Verlag.
- [ERCH09] Chickowski, E. (2009, May 15). *IT Management - Baseline*. Retrieved November 30, 2010 from Projects Gone Wrong: <http://www.baselinemag.com/c/a/IT-Management/Projects-Gone-Wrong-234902/3/>
- [FOWL04] Fowler, M. (2004). *Refactoring: Improving the Design of Existing Programs*. Addison Wesley Longman, Inc.
- [GAKL93] Harald Gall, R. K. (1993). Capsule Oriented Reverse Engineering for Software Reuse. *Lecture Notes in Computer Science* , 717, pp. 418-433.

- [GIGI09] Ann Gilley, J. W. (2009). Organizational Change: Motivation, Communication, and Leadership Effectiveness. *Performance Improvement Quarterly* , 21 (4), 75-94.
- [GIWI05] Rainer Gimnich, A. W. (2005). Workflows der Software-Migration. Koblenz: Universität Koblenz.
- [GRKI08] Michael Grottke, R. M. (2008). The Fundamentals of Software Aging. *1st International Workshop on Software Aging and Rejuvenation/19th International Symposium on Software Reliability Engineering*. Seattle, WA: Institute of Electrical and Electronics Engineers (IEEE).
- [HANS10] Hanschke, I. (2010). *Strategic IT Management*. Berlin, Heidelberg: Springer Berlin Heidelberg.
- [ICEC97] Bing Wu, D. L. (1997). The Butterfly Methodology : A Gateway-free Approach for Migrating Legacy Information Systems. *Proceedings of the 3rd IEEE Conference on Engineering of Complex Computer Systems (ICECCS97)* (pp. 200-205). Villa Olmo, Como, Italy: IEEE Computer Society.
- [JESI07] Simonsen, J. (2007). Involving Top Management In IT Project. *COMMUNICATIONS OF THE ACM* , 50 (8).
- [KEMI09] Stephen Kelly Micro Focus. (2009, 08 07). *COBOL – Still doing the business*. Retrieved 08 30, 2010 from [www.cio.com: http://advice.cio.com/stephen_kelly_micro_focus/cobol_still_doing_the_business](http://advice.cio.com/stephen_kelly_micro_focus/cobol_still_doing_the_business)
- [KSNW04] Urs Kuhlmann, H. S. (2004, 11). Workshop Reengineering Prozesse (RePro 2004) - Fallstudien, Methoden, Vorgehen, Werkzeuge. *Fachberichte Informatik* .
- [LAKE00], A. L. (2000). *Legacy System Anti-Patterns and a Pattern-Oriented Migration Response*. Canterbury: University of Kent at Canterbury.
- [LEBE85] M. M. Lehman, L. A. (1985). Program Evolution: Processes of Software Change. *A.P.I.C. Studies In Data Processing* , 27.
- [LFAN03] Luis Filipe Andrade and José Luiz Fiadeiro. (2003). *Architecture Based Evolution of Software Systems* (SFM 2003, LNCS 2804 ed.). Berlin Heidelberg: Springer-Verlag.
- [LILA99] Mikael Lind, A. L. (1999, April). Legacy Systems: Assets or Liabilities. *Borås Studies in Information Systems* , 20.
- [LMSS08] Grace A. Lewis, E. J. (2008). *SMART: Analyzing the Reuse Potential of Legacy Components in a Service-Oriented Architecture Environment*. Pittsburgh, USA: Carnegie Mellon University.
- [MALA09] Lankhorst Mark et al. (2009). *Enterprise Architecture at Work - Modelling, Communication and Analysis, 2nd Edition*. Berlin Heidelberg: Springer-Verlag Berlin Heidelberg.
- [MAWE91] Weiser, M. (1991, September). The Computer for the 21st Century. *Scientific American* .

- [MCAF06] McAfee, A. P. (2006). Enterprise 2.0: The Dawn of Emergent Collaboration. *MITSloan Management Review* , 47/3.
- [METO04] Tom Mens, T. T. (2004). A Survey of Software Refactoring. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING* (XX).
- [OPGR09] The Open Group. (2009). *TOGAF*. Retrieved February 13, 2011 from <http://www.opengroup.org/architecture/togaf9-doc/arch/>
- [ORM10] Media, O. (2010, 07 21). Retrieved 07 21, 2010 from oreilly.com: http://www.oreilly.com/news/graphics/prog_lang_poster.pdf
- [PARN94] Parnas, D. L. (1994, May). Software Aging. *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference* .
- [PECR10] Crosman, P. (2010, July 23). *Bank - System & Technology*. Retrieved September 01, 2010 from http://www.banktech.com/blog/archives/2010/07/citi_endorses_i.html
- [PMIP08] Project Management Institute. (2008). *A Guide to the Project Management Body of Knowledge: PMBOK Guide*. Project Management Institute.
- [REIF02] Reifer, D. J. (2002). *Making the Software Business Case: Improvement by the Numbers*. Boston, USA: Addison-Wesley.
- [REYN07] Reynolds, C. (2007, May 10). *Businesses offered a way to assess cost of downtime*. Retrieved September 6, 2010 from <http://news.techworld.com/applications/8793/businesses-offered-a-way-to-assess-cost-of-downtime/>
- [RMSI03] Dan Remenyi, A. M.-S. (2003). *The effective measurement and management of IT costs and benefits*. Oxford, Great Britain: Butterworth-Heinemann.
- [ROGE03] Rogers, E. M. (2003). *Diffusion of Innovations*. New York: The Free Press.
- [RUPA10] Ruparelia, N. (2010, February 25). *IT Architecture Report*. Retrieved February 11, 2011 from [wordpress.com: http://archreport.wordpress.com/2010/02/25/what-is-enterprise-architecture/](http://archreport.wordpress.com/2010/02/25/what-is-enterprise-architecture/)
- [SEPL03] Robert Seacord, D. P. (2003). *Modernizing Legacy Systems - Software Technologies, Engineering Processes and Business Practices*. Boston: Pearson Education, Inc.
- [SMIT10] Smith, M. (2010, 12 07). *www.hp.com*. Retrieved 01 31, 2011 from <http://h30507.www3.hp.com/t5/Talk-to-the-Modernization/Rules-of-Thumb-Custom-Software-Replacement-Cost/bc-p/86093>
- [SNWH10] Harr M. Sneed, E. W. (2010). *Softwaremigration in der Praxis*. Heidelberg, Germany: dpunkt.verlag GmbH.
- [SOA05] SAP News Desk. (2005, December 18). *SOA World Magazine*. Retrieved November 25, 2010 from [soa.sys-con.com: http://soa.sys-con.com/node/164624](http://soa.sys-con.com/node/164624)
- [STSA09] Everett M. Rogers, A. S. (2009). Diffusion of Innovations. In M. S. Don Stacks, *An integrated approach to communication theory and reasearch* (Vol. 2nd edition, pp. 418-435). New York: Routledge.

[UMAR97] Umar, A. (1997). 8: Dealing with Legacy Applications: An Overview. In A. Umar, *Application Reengineering: Building Web-Based Applications and Dealing with Legacies*. Prentice Hall.

[VEMO03] Viswanath Venkatesh, M. G. (2003). User Acceptance of Information Technology: Toward a Unified View. *MIS Quarterly* , 27 (3), 425-478.

[WAI99] Warren, I. (1999). *The Renaissance of Legacy Systems: Method Support for Software-System Evolution*. New York, USA: Springer-Verlag.

[WERE08] Weidl-Rektenwald, J. (2008, March 6). *Software Wartung und Evolution*. Retrieved December 5, 2010 from <http://www.infosys.tuwien.ac.at/linksites/teaching/courses/SWE/swe.html>

[WISM03] Lloyd G. Williams, C. U. (2003). *Making the Business Case for Software Performance Engineering*. Retrieved November 17, 2010 from Performance Engineering Services: www.perfeng.com/papers/buscase.pdf

[WIUL02] William Ulrich. (2002). *Legacy Systems: Transformation Strategies*. Upper Saddle River: Prentice Hall PTR.

Umar, A. (1997). *Application Reengineering: Building Web-Based Applications and Dealing with Legacies*. Prentice Hall .

CURRICULUM VITAE

Personal Details

Name	Hannes Böhm
Date of Birth	25 th December 1983
E-mail	hannes.boehm@gmx.at

Work Experience

Fa. Gerhard Böhm 7434 Bernstein	Office Clerk IT administration, billing and reporting
Chamber of Commerce Burgenland Branch Office Oberwart 7400 Oberwart	Trainee Different tasks in the field of training course administrations, Trade fair preparations
Siemens Financial Services Austria & CEE 1140 Vienna	Student Trainee Inter- and Intranet Content Management, Migration Responsibility for Finance Applications
University of Vienna 1010 Vienna	Tutor in a variety of Lectures Database systems, Interoperability, Evolution, Semantic Information systems, Integration

Education

University of Vienna 1010 Vienna	Master Program in Business Informatics With Focus on Large Scale Systems
University of Vienna & Vienna University of Technology 1010 Vienna	Bachelor Program in Business Informatics With Focus on Semantic Web
Business School Oberwart 7400 Oberwart	Higher vocational college With Focus on Business Informatics and Organisation
BRG Oberschützen 7432 Oberschützen	Academic secondary school With Focus on French

Additional Skills

Modelling Skills	Meta-modelling and Modelling techniques and methodologies Semantic Web (OWL, RDF), Business Process (BPMN, BOC Adonis), Software Modelling (UML), UN/CEFACT's Modeling Methodology (UMM), Profitability Modelling (e3value), Goal-oriented Modelling (i*)
Framework Experiences	Enterprise Architecture (TOGAF) Software Engineering (RUP, Spiral Model) Project Management (PMI Method)
Programming Skills	very good skills in JAVA and J2EE, SQL, JavaScript, PHP, BASH Script, HTML,

Personal Interests	CSS
Technology Sports	Related to communication and entertainment and music Volleyball Running Board sports (Snowboarding, Surfing, Wakeboarding) Skiing Climbing
Music	

References

Available on request.