Technische Universität Wien Vienna University of Technology
Fakultät für Informatik Faculty of Informatics

DIPLOMARBEIT – MASTER THESIS

Software Project Management in Unstable Environments: Handling Volatile Business Requirements Induced by Trade-Offs Among Client Stakeholders

zur Erlangung des akademischen Grades (for the obtainment of the academic degree)
Magister (Mag. rer. soc. oec.)

ausgeführt am (conducted at the)
Institut für Rechnergestützte Automation – (E183) – Institute for Computer Aided Design
Forschungsgruppe Industrielle Software – (INSO) – Research Group Industrial Software
der Technischen Universität Wien (at Vienna University of Technology)

unter der Anleitung von (under the supervision of)
Univ.-Prof. Dipl.-Ing. Dr. techn. Thomas Grechenig

durch (by)
Raoul Fortner
Rienößlgasse 17/18
A-1040 Wien
Österreich – Austria

Wien, August 2007 ______________________________

« Quand tu veux construire un bateau,
ne commence pas par rassembler du bois,
couper des planches et distribuer du travail,
mais réveille au sein des hommes le désir
de la mer grande et large. »

„Wenn du ein Schiff bauen willst,
dann trommle nicht Männer zusammen,
um Holz zu beschaffen, Aufgaben zu vergeben
und die Arbeit einzuteilen, sondern lehre sie
die Sehnsucht nach dem weiten, endlosen Meer.“

“If you want to build a ship,
don't drum up the men to gather wood,
divide the work and give orders.
Instead, teach them to yearn for
the vast and endless sea.”

Zugeschrieben – Attribuée á – Attributed to
Antoine de Saint-Exupéry
(1900 – 1944)

Dedication – Widmung

To Julia, Markus and my entire family – especially my parents
and my grandfather – for their remarkable support and patience.

Für Julia, Markus und meine gesamte Familie – insbesondere meine Eltern
und meinen Großvater – für ihre bemerkenswerte Unterstützung und Geduld.

Abstract

Trade-offs or conflicts among client stakeholders about the Business Requirements of a new software product are a practical problem in complex software projects which is not widely covered by academic literature. Most sources in Software Engineering literature cover the "human factor" only with regard to teamwork in the project team, but similar effects on the client side are often neglected. Therefore this thesis examines the underlying problems and risks in Requirements Engineering and highlights, how Software Project Manager can face problems with adversarial client stakeholders. The conclusion provides an overview for Software Project Managers who face such trade-offs or conflicts and supports them with "Contradictions that SWPM has to consider" as well as a Toolbox with possible measures. The major contributions of this thesis are the identification of an upcoming topic in Requirements Management and the related considerations about the nature of this problem and possible solutions.

Kurzfassung

Interessensgegensätze oder Konflikte der Kunden-Stakeholder untereinander über die Business Requirements eines neuen Softwareproduktes sind ein praktisches Problem in komplexen Softwareprojekten, das von der wissenschaftlichen Literatur derzeit kaum behandelt wird. Die meisten Quellen in Software Engineering besprechen den „menschlichen Faktor" nur beim Teamwork innerhalb des Entwicklerteams, während ähnliche Effekte auf Kundenseite ausgelassen werden. Daher untersucht diese Diplomarbeit die zugrunde liegenden Probleme und Risiken im Requirements Engineering und hebt hervor, wie Software Projekt Manager solche Probleme mit opponierenden Kunden-Stakeholdern bewältigen können. Die Schlussfolgerungen bieten einen Überblick für Software Projekt Manager die solche Zielkonflikte bewältigen müssen unterstützt diese „Denkanstössen in Form von Gegensatzpaaren" sowie einer Werkzeugkiste an möglichen Maßnahmen. Der wesentliche Beitrag dieser Arbeit sind die Identifizierung eines aufkommenden Themas im Requirements Management und die darauf aufbauenden Überlegungen über die Hintergründe und mögliche Lösungen dieses Problems.

<u>Keywords (Schlagworte)</u>:
Software Project Management, Requirements Engineering, Requirements Management, Business Requirements, Volatile Requirements, Anforderungsmanagement

Thesis Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institution. Information derived from the published or unpublished work of others has been acknowledged in the text and a list of references is given.

Vienna, August 2007 *Raoul Fortner*

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Wien, August 2007 *Raoul Fortner e.h.*

Acknowledgements

I have to express my gratitude to all those people who supported the creation of this master thesis

Dr. Thomas Grechenig, Mario Hiermann and the AMMA-group at the INSO

Stephanie Schultz and Disney Germany for figure 1.1

Hannes Eksler and the ITZ

Special thanks go to those friends and colleagues who supported me with their ideas and critics:
Sascha Frühwirth, Martin Spitzer, Alexander Brandstätter,
Martin Strasser, Alexander Fehr and Stephan Binder

Finally I express my deepest gratitude to my family – especially my parents and my grandfather –
for their ongoing support of my studies and their unbelievable patience during the creation of this thesis.

Table of Contents

List of Figures

List of Tables

1. INTRODUCTION

"Grau, teurer Freund, ist alle Theorie"
(Johann Wolfgang von Goethe, Faust I, Studierzimmer: 2038)

There is a real gap between (scientific) theory and daily practice in Software Engineering, which most of us know quite well, and which is usually camouflaged behind nice theories like the manifold Process Models already invented for Software Engineering and Software Project Management. It is the gap between well thought-out sophisticated plans on one hand (project plans, technical plans or any other kind of plans) and the uncertainties of everyday project reality on the other hand. This gap has various names, but the most appropriate synonym seems to be simply the "*human factor*".

There are many ways how the human factor can influence a Software Project – and there are also many theories, recipes or "best practices" to somehow master it, one of the newest "*silver bullets*" in Software Engineering is the so called "Risk Management (RM)". However, humans are not like computers or other machines, as it is quite human to err, change ones opinion or decide to go ones own direction. Some authors earn a lot of money by writing books on how to recapture the changes that the human factor brought into a software project – e.g. Tom DeMarco with his manifold books as well as Robert L. Glass with books and his "Loyal Opposition" in IEEE-Software or the authors of "*AntiPatterns*" [*Brown98*]. But unfortunately nobody ever really found the magical "*Philosopher's stone*" – neither in the quite young field of Software Engineering [*Brooks95:"No Silver Bullets*"], nor in any other engineering discipline. So we have to go on with regularly improving our theories and permanently learning with every new project we conduct – and sometimes one learns even more from a failure than from a success [*Glass98:19*]. The human factor can influence a software project in many different ways, as it can cause problems in the development team as well as in the project management itself or on the client side ("*Most software projects are chaotic, unpredictable and hazardous to careers*", [*Brown98:xxiv*]).

One selected example of these numerous effects will be under examination in this thesis: The various problems and influences that the "human factor", within client stakeholders, brings into the process of *Requirements Engineering* respectively *Requirements Management* – and how the Project Management can somehow master those effects. This is still a wide-area topic, but concise enough to write an interesting and up-to-date Master Thesis in Software Project Management about it. My personal motivation for choosing this topic was to combine my experiences in real software projects – especially from Project SIE during my last year abroad at the French Grande École SUPAERO – with my studies of "*Wirtschaftsinformatik*", a curriculum already combining both aspects of Software Project Management: Technical education as well as courses in economics and management.

The thesis starts with a survey about the state of the art in *Software Engineering* and highlights the current discussion about "home grounds" of plan-driven and agile methods, also in relation to Requirements Engineering (*Chapter 2*). Afterwards the human factor in *Software Project Management (SWPM)* is examined and focused on considerations about influences from the external environment. Also the manifold challenges that SWPM has to face, including many trade-off decisions, are underlined (*Chapter 3*). Subsequently an introduction to the concepts of *Stakeholders, Requirements and Requirements Engineering* underlines the strong relation between *Requirements Management* and Software Project Management. Also the main risks and problems in Requirements Engineering are addressed (*Chapter 4*).

The following survey about relations between requirements (process) risks and trade-offs among client stakeholders provides a brief introduction into the problem which is addressed by this thesis (*Chapter 5*). The conclusion provides an overview for Software Project Managers who face such trade-offs or conflicts and supports them with "Contradictions that SWPM has to consider" as well as a Toolbox with possible measures (*Chapter 6*). The climatic structure of this thesis is illustrated in figure 1.1.

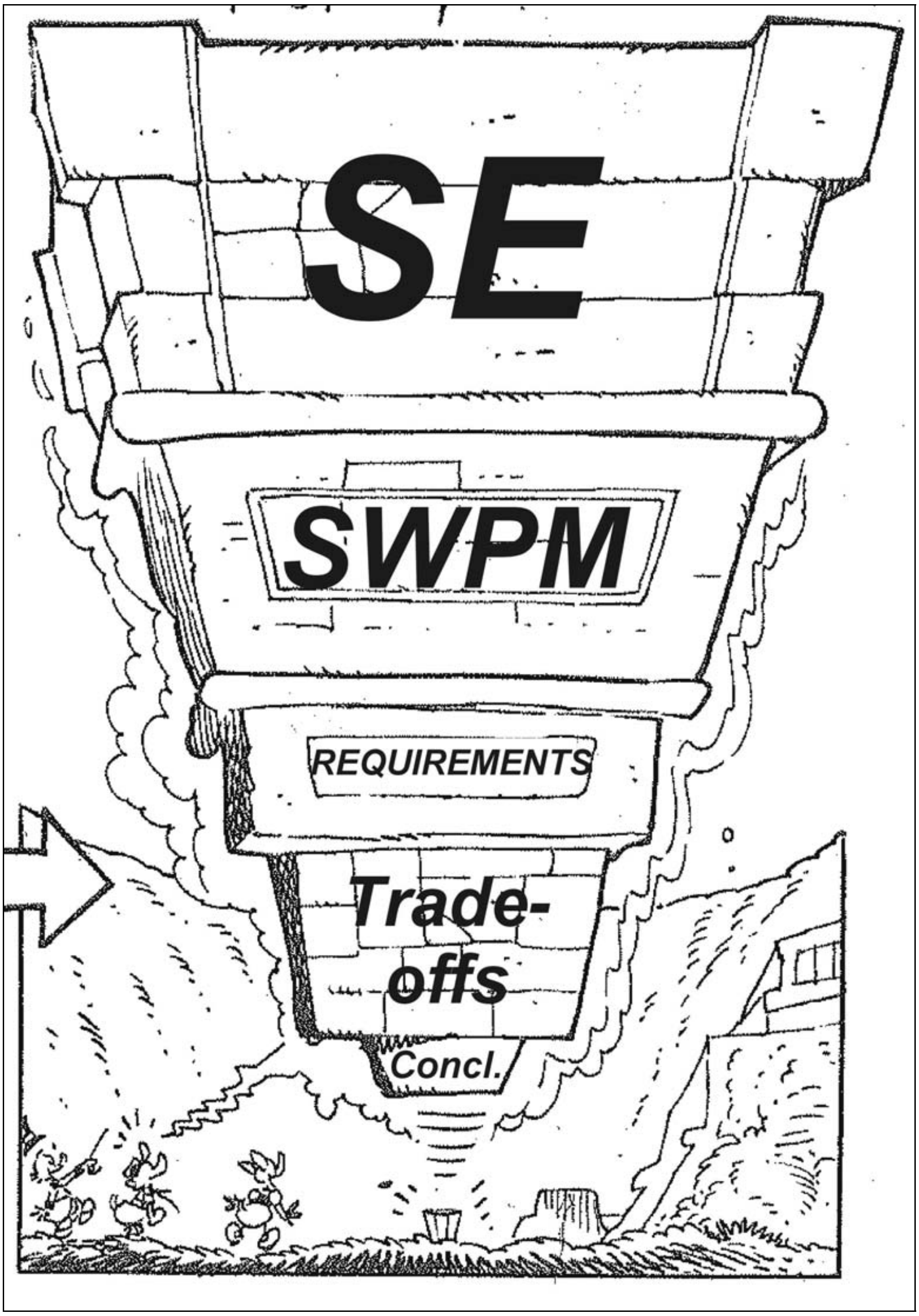


Figure 1.1: Inverted pyramid structure of this thesis (Derived from [*Disney87:123*])
(c) Disney - Reprinted with kind permission of Walt Disney Company (Germany) GmbH and Egmont Ehapa Verlag

2. EVOLUTION & PROCESS MODELS OF SOFTWARE ENGINEERING

"If builders built buildings the way programmers wrote programs,
then the first woodpecker that came along would destroy civilization."
(*Murphys Law – Weinberg's Second Law,* [*Bloch85:86*])

Compared to other Engineering Sciences, Software Engineering (SE) is a quite young discipline and still struggles somehow with its self-definition. While Architecture, Civil Engineering or Mechanical Engineering exist and evolve since centuries, Software Engineering itself only dates back to the 1960's, although the higher-ranking Computer Science has a more comprehensive history. Even within this short period since 1967/68, the meanings of "*Software Engineering*" and also the whole discipline have enormously evolved. Also more and more related domains have somehow been integrated into SE, which therefore now contains different technological aspects, manifold methodologies, various kinds of project- and people-management issues as well as the increasing significance of economic needs.

Considering this spectrum, a short chapter about the fundamentals of Software Engineering has to prioritise strongly, which is done with respect to the complex topic as well as to the needs of the later culmination of this thesis. Therefore this chapter starts with an overview about the diverse definitions and meanings of SE, followed by a taxonomy describing various established approaches and the key elements of SE. Afterwards two state-of-the-art Process Models of SE are displayed more detailed, as UP and XP are good examples for the current challenges in Requirements Engineering & Management. Finally a short excursus about Software Re-Engineering and Evolution is given.

2.1. Selected definitions and meanings of Software Engineering

The founding of the term and discipline of Software Engineering in the 1960's fulfilled an already noticeable need in computer science and started – like the Internet-predecessor Arpanet – with some well organised initial aid from the military, namely the North Atlantic Treaty Organization (NATO):

> "*Discussions were held in early 1967 by the NATO Science Committee ... on possible international actions in the field of computer science. In the Autumn of 1967 the Science Committee established a Study Group on Computer Science. ... In late 1967 the Study Group recommended the holding of a working conference on Software Engineering. The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering.*" [*NATO68:8*]

Certainly the term was not "invented" by the NATO Study Group on Computer Science, but it was popularised by the two famous "*NATO Software Engineering Conferences*" and is therefore often attributed to their godfather and first conference-chairman Friedrich Bauer from TU München [*Bauer93*]. The first conference took place between 7th and 11th October 1968 in *Hotel Sonnenbichl* in Garmisch-Partenkirchen (Germany) and was followed by a second – according to Brian Randell "*less harmonious and successful*" – conference in October 1969 in Rome. At least after these two conferences, the term Software Engineering was internationally established, gradually displacing "programming", and in the following decades a whole discipline in Computer Science evolved around this term.

Since the 1960's, the discipline itself progressed and evolved along with the term Software Engineering, which lead to various definitions of the term by scientists and practitioners, but till today there is no generally accepted agreement on what SE completely means. Quite the contrary, in some countries (like the USA and Canada) established engineering bodies even took legal actions to prevent the use of the term "*Software Engineer*" as a profession. So the definition, if SE is or is not an engineering discipline is quite more than an academic question and implies a lot of practical impacts, especially in terms of money and salary level. On the other hand, since the 1960's a lot of (excellent) programmers stressed the definition of programming as a skill, "craftsmanship" (Pete McBreen) or even "art" (e.g. Donald E. Knuth with his voluminous and still unfinished monographs "*The Art of Computer Programming*").

As this thesis strongly supports the engineering-approach underlined by SE-pioneers like Bauer, Boehm or Brooks and current standard SE-textbooks ([*Schach96*], [*GheJaz03*], [*ZuserGre04*], [*Sommer06*]), the following list gives a historical overview about common definitions of SE as an engineering discipline:

- **Friedrich Ludwig Bauer 1971** (Chairman of the 1968-NATO-Conference) [*Bauer71:530*]:
  "*[Software Engineering]… the establishment and use of sound engineering principles to obtain economically software that is reliable and works efficiently on real machines*"

- **Barry Boehm 1976** [*Boehm76:1226*] (A later recurrence in 1979 is quoted by [*ZuserGre04:23*]):
  "*Software Engineering: The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them.*"

- **David Lorge Parnas 1978** ["*Some Software Engineering Principles*", reprinted in *HofWeis01:257*]:
  "*[Software Engineering is the] … multi-person construction of multiversion software*"

- **Richard Fairley 1985** [*Fairley85*]:
  "*Software engineering is the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates.*"

- **IEEE Standards Coordinating Committee 1990** [*IEEE90:67*]:
  "*software engineering. (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).*" (REMARK: In1983 the IEEE initially stated a less precise definition of SE)

- **Ian Sommerville in the 1990s** [*Sommer06:6*] (at the latest in the 6th edition of his textbook):
  "*Software Engineering is an engineering discipline which is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.*"

- **Helmut Balzert 1996** [*Balzert96:36*] (who uses the half-German term "*Softwaretechnik*")
  "*… goal-oriented provision and systematic application of principles, methods, and tools for the cooperative, engineering-like development and deployment of large software systems*"

- **Ghezzi, Jazayeri & Mandrioli 2003** [*GheJaz03:1*]
  "*Software engineering is the field of computer science that deals with the building of software systems that are so large or so complex that they are built by a team or teams of engineers. Usually these systems exist in multiple versions and are in service for many years. During their lifetime, they undergo many changes: to fix defects, to enhance existing features, to add new features, to remove old features, or to be adapted to run in a new environment.*"

Many important SE-issues are spread in these definitions: "*engineering principles*", "*reliable*", "*efficient*", "*systematic, disciplined, quantifiable approach", "practical application of scientific knowledge", "technological and managerial discipline"," systematic application of principles, methods, and tools"*. These terms exemplify, that the definition of SE as an engineering discipline consequently implies the implementation of engineering principles (even when the agile movement currently tries to minimize some common SE-practices which they call too "bureaucratic"). So most of the SE-textbooks highlight more of these principles including reliability, responsibility (also for human lives, e.g. when constructing software for airplanes or medical machines), the establishment of a code of ethics, a well defined education and so on.

The difference between programming as an "art" and the engineering approach in SE can be exemplified as follows: A single programmer designing a simple homepage with a guestbook may do this without to much knowledge about "engineering" and can bring in "art" and skills in programming. But as soon as the homepage is supposed to be a professional E-Commerce-solution – e.g. a web application connected with a goods-database – a solid engineering approach will be necessary to guaranty values like reliability, security, maintainability, warranty and to cover even the related legal considerations for E-Commerce. All these issues are part of a professional engineering responsibility.

In 1996 Shaw compared the evolution of SE with the genesis of a professional engineering discipline and pointed out progressions that fields go through before they reach the level of professional engineering ([*Shaw96*], see figure 2.1). As Boehm already defined in 1976 ("*practical application of scientific knowledge"* [*Boehm76:1226*]), Shaw also states that only the professional unification of science and the commercial (practical) branch can finally lead to a professional engineering discipline. Since 1996, a lot of further progress has been made, especially by IEEE and ACM (e.g. with the definition of a code of ethics, a standard curriculum and the promising Software Engineering Body of Knowledge), but the current status of the entire SE within this diagram is still an open question (A more detailed and up-to-date overview can be found in [*Boehm06:16*] and will be presented in chapter 2.2.).

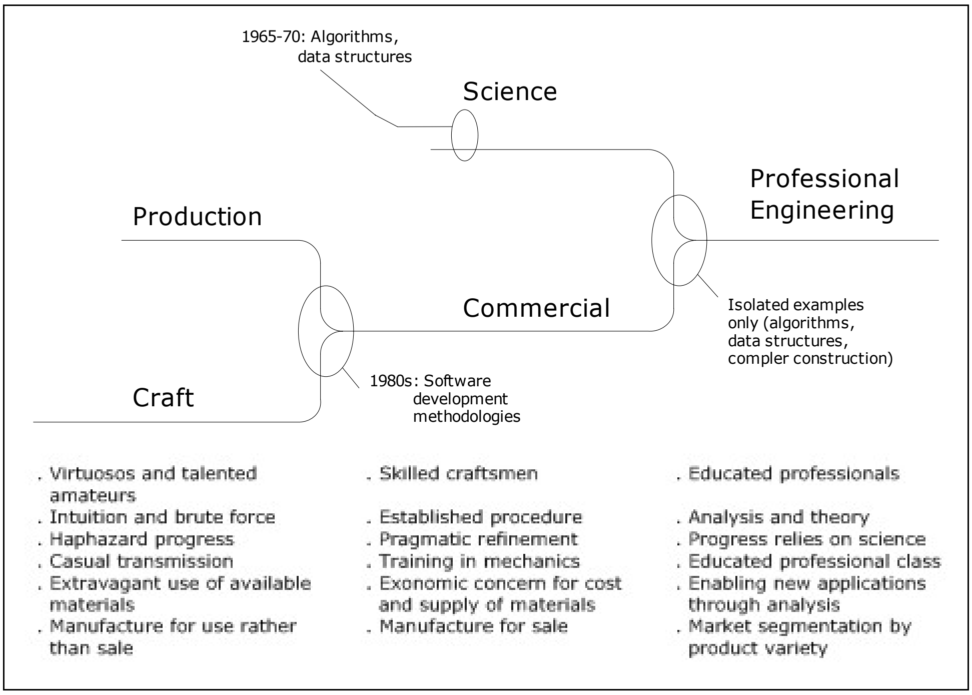

1965-70: Algorithms, data structures
Science
Professional Engineering
Production
Commercial
Isolated examples only (algorithms, data structures, compler construction)
Craft
1980s: Software development methodologies
. Virtuosos and talented amateurs
. Intuition and brute force
. Haphazard progress
. Casual transmission
. Extravagant use of available materials
. Manufacture for use rather than sale
. Skilled craftsmen
. Established procedure
. Pragmatic refinement
. Training in mechanics
. Exonomic concern for cost and supply of materials
. Manufacture for sale
. Educated professionals
. Analysis and theory
. Progress relies on science
. Educated professional class
. Enabling new applications through analysis
. Market segmentation by product variety

Figure 2.1: Evolution of SE as an engineering discipline [*ZuserGre04:22f* quoting *Shaw96*]

When comparing the already mentioned contrast between the "art"-approach and the engineering-approach in programming, one has to avoid misunderstandings: Even a house built by an architect or a bridge constructed by a civil engineer can also be beautiful art that really prettifies the landscape (e.g. the famous Golden Gate Bridge). But always more important will be, that their construction follows strict engineering rules (and even legal regulations) to guarantee a maximum of safety and durability, so engineering comes first, "art" second. Surely, every comparison between SE and other engineering disciplines has its limit (as Schach points out: "*A civil engineer, if asked to rotate a bridge through 90° or to move it hundreds of miles, would consider the requester to be bereft of his senses. However, we think nothing of asking a software engineer to convert …*" [*Schach96:6*]) – but also Schach underlines:

> "*Perhaps if software engineers treated an operating system crash as seriously as civil engineers treat a bridge collapse, the overall level of professionalism within software engineering would rise.*" [*Schach96:5*]

An important term for the problems with software development was coined between the 1960s and 1970s to subsume all the well known discontent with software that was "*delivered late, over budget and full of residual faults*" [*Schach96:4*]. This term is "software crisis" – and for a long time the argumentation was, that SE was found to solve this "crisis". Nowadays this concept seems out-of-date (and the term itself an empty buzzword), as the symptoms of the "crisis" are still there and not solved, only diminished (Schach proposes to call it the "*software depression, in view of its long duration and poor prognosis*" [*Schach96:5*]). Even Robert L. Glass who exposes "*16 colossal software disasters*" (e.g. Denver International Airport luggage system or the new air traffic control system for FAA) states in the same book clearly: "*I do not believe in the existence of a software crisis*" [*Glass98:6*]. The most realistic attitude toward this topic – showing that these are symptoms of overambitious expectations as well as childhood diseases of a quite young and still changing engineering discipline – is documented by the following appraisal:

> "*In reality, projects were late because the application was complex and poorly understood by both costumers and developers and neither had any idea how to estimate the difficulty of the task and how long it would take to solve it. Although the term 'software crisis' is still used sometimes, there is a general consensus that the inherent difficulties of software development are not short-term problems. New and complex application domains are inherently difficult to approach and are not subject to short-term, quick solutions*" [*GheJaz03:4f*]

So there seems to be no "crisis", but a lot of possibilities for advancement and increased (engineering) professionalism in SE, and even one of its pioneers, David Parnas, later called the word-combination Software Engineering "*an unconsummated marriage*" [*HofWeis01:Chapter32*]. Naturally there are different views on the solution of these problems: While (the mathematician) Dijkstra saw the problem (during a later transcript lecture from December 1993 in Austin, Texas) in the "*management community*" and "*insistence on teamwork*", sources which are more concerned about the "human factor" argue in the opposite direction: "*The major problems of our work are not so much technological as sociological nature*" [*DeMaLi87:4*]. The second approach is predominant in this thesis.

After this general overview about the current self-concept, definitions and boundaries of Software Engineering, the following subchapters present the current state-of-the-art in SE in condensed mainstream-form, as the focus of this thesis doesn't allow a more in-depth analysis of all ongoing discussions about each new paradigm, methodology or engineering-approach.

2.2. Taxonomy – Common approaches to Software Engineering

While the entire discipline of Software Engineering has been evolved since the 1960's, also various approaches for an optimal (often formalised and well organised) software development process have been developed and improved. The most characteristic indicators in SE for the increasing engineering-approach are – beneath general engineering principles and various (programming & design) paradigms – it's so called "Process Models". Since the ending 1960's, when the first well-defined Process Model – the Waterfall Model popularised by Winston Royce [*RoyceWi70*] – replaced the common "code and fix" approach, various Process Models and methods have been invented to organise the Software Development Process in a structured way with well defined stages (also called phases or activities) "*to produce high-quality software products reliably, predictably and efficiently*" [*GheJaz03:385*].

But as there is no generally accepted definition and meaning of Software Engineering, there is also no commonly accepted taxonomy of SE and its key methodologies – and some sources state that this is explicable, as SE has changed too strongly during the last decades and that especially technological progress in computer science played a leading role in this continuous re-invention of the whole discipline. On the other hand, even when programming languages and their paradigms change, at least some main ideas about the professional and engineering-like handling of a software development project should be "*timeless software engineering principles*" [*Boehm06:12*]. However, almost every established SE-textbook has its individual taxonomy of Software Engineering which is carefully delimited from other views, and which leads to various approaches how SE can be structured.

Especially there is a lot of disorder in the appropriate topology of Process Models and any other kind of models or methodologies related to the software development process, so terms like "*Process Model*", "*Process (assessment) framework*", "*Process*", "*Method(ology)*", "*Meta Process*" or even "*Software Life Cycle*" are mixed in a confusing way (e.g. [*KrollKru03:49ff*]). While some sources arrange them nearly on the same level (e.g. [*Schach96:52ff*], [*GheJaz03:403ff*], [*ZuserGre04:69ff*]), other sources develop sophisticated multi-layer-meta models (consisting of three or more levels) about their hierarchical dependency ([*Mayr05:79ff*], [*AmNa05:6f*]) and Sommerville simply classifies agile methods as part of the "rapid development"-section, far away from the software-process-section of his textbook [*Sommer06:93ff and 430ff*]. Various attempts for a SE-model-topology even try to establish different technical terms, sometimes leading to funny creations (e.g. [*KrollKru03:49ff*] introducing "*Low Ceremony*" for agile methods and "*High Ceremony*" for what less-diplomatic XP-believers normally call "*bureaucratic*").

Meanwhile practitioners from the software industry state with some pragmatism, that they normally mix between predefined process-based methods (e.g. codified industrial standards or Process Models) and some light-weighted methods in-between (e.g. prototyping and other agile approaches). So practical researchers already presume the "*foreseeable end of the agile hype*" [*Mayr05:99*] and expect that the pendulum that oscillates between "*process monsters*" (e.g. CMM/I) and agile methods will stabilise somewhere in-between [*Mayr05:78*]. Therefore SE-pioneer Barry Boehm argued during the last years to "*synthesize the best from agile and plan-driven methods to address our future challenges*" [*Boehm03:46*] (underlined by a quite enjoyable monkey-and-elephant-metaphor), so it appears that within some years a next generation of Process Models will bring more "*balance of agile and plan-driven methods*" [*Boehm02:69*].

As none of the mentioned SE-textbooks provides an appropriate diagram about the taxonomy of Software Engineering, a simple hierarchical overview about SE as a discipline of Computer Science has been developed for this thesis (see figure 2.2, supported by figure 2.3 about the SE-planning-spectrum by Boehm). The details about the mentioned SE-topics will be explained in the following subchapters, except topics like economics, metrics and tools for Computer Aided Software Engineering (CASE).

Activity level	Concerned with
COMPUTER SCIENCE (“Informatics”) ↓	Various sub-fields & disciplines (e.g. *technologies*, *algorithms & data structures*, *computation theory*, *architectures*, *programming languages*)
Discipline of SOFTWARE ENGINEERING ↓	Principles, profession (ethics, education, knowledge), evolution (history & future), paradigms, methods & process models, tools
BODIES (e.g. *ISO, IEEE*) **SOFTWARE INDUSTRY** **ENTERPRISE(S)** ↓	International or industrial **standards and QM-models for the assessment & improvement of Software Development Processes** (e.g. *CMM(I), ISO 90003, SPICE: ISO/IEC 15504*)
General planning of SOFTWARE (Development) LIFE CYCLES (SWDLC) and PROCESS MODELS ↓	Various **“Method(ologie)s”** (e.g. *Object-oriented*, *iterative*, *incremental*, *risk-driven*, *agile*) → **Predefined “Process Models”** (Frameworks with quite different scopes, e.g. *UP*)
INDIVIDUAL (Software Development) **PROJECT**	**Customising, tailoring and implementation of** one (or several) **Process Model(s) & Method(ologie)s** within the Project (e.g. *roles, products & artefacts, activities, iterations, tools*)

Figure 2.2: Attempt for a hierarchical taxonomy of Software Engineering [*own work*]

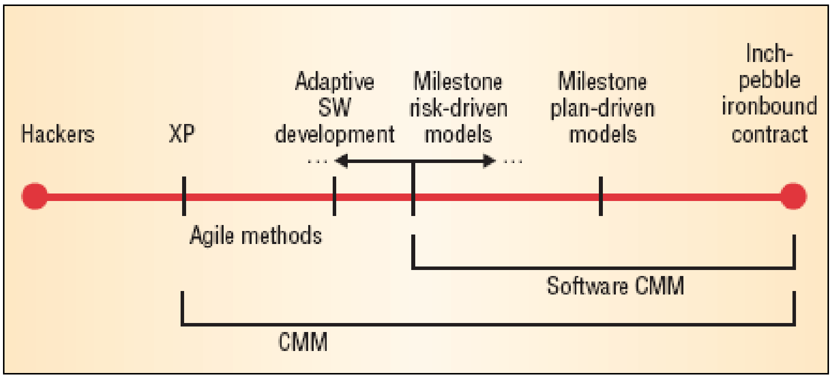

Hackers
XP
Adaptive SW development
Milestone risk-driven models
Milestone plan-driven models
Inch-pebble ironbound contract
Agile methods
Software CMM
CMM

Figure 2.3: The planning spectrum in Software Engineering [*Boehm02:65*]

2.2.1. Principles and qualities as an engineering discipline

> *„The whole trouble comes from the fact that there is so much tinkering with software.*
> *It is not made in a clean fabrication process which it should be.*
> *What we need is software engineering." (Friedrich L. Bauer in 1968* [*Bauer93*])

When developing Software in a professional way it is important to keep in mind that SE is an engineering discipline and the most important value for every engineer is always RESPONSIBILITY. As an engineer, one is responsible for the developed product and the production process, especially as clients trust the product and important institutions or maybe even human lives depend on it. Therefore one has to be aware about the future consequences of all engineering actions, as in the worst case even legal actions with subsequent penalties or imprisonment are possible (e.g. after a lethal accident).

Surely there are also significant differences between SE and other engineering disciplines, as Mayr points out (*Difficult cost estimation due to the common singularity of each SW-project; high number of different solutions; individuality of programmers and their performance varieties hinder a solid effort estimation; rapid technological change; absence of standardised components and modules; "invisibility" of software products hinders effective controls* [*Mayr05:55f*]). But there are some general engineering principles and qualities – transferred to Software Engineering – which (should) superpose the entire Software Development Process, regardless which task is performed. So one of the first engineering principles implemented in SE was the separation of the engineering stage from the production stage, which lead to the renunciation of the build-and-fix-approach followed by various Methods and Process Models [*RoyceWa98:8*]. From all examined SE-textbooks [*GheJaz03*] provide the most extensive and striking overview about such general engineering principles in Software Engineering and their consequences for the product quality:

<u>Software Engineering principles</u>: [*GheJaz03:42ff*]

(1) RIGOR AND FORMALITY: "*a necessary complement to creativity in every engineering activity … There is no need to be always formal during design, but the engineer must know how and when to be formal*"

(2) SEPARATION OF CONCERNS: "*The only way to master the complexity of a project*"

(3) MODULARITY & (DE-)COMPOSABILITY: "*A complex system may be divided into simpler pieces called modules … Modularity is an important property of most engineering processes and products*"

(4) ABSTRACTION: "*fundamental technique for understanding and analyzing complex problems*"

(5) ANTICIPATION OF CHANGE: "*ability of software to evolve does not happen by accident … requires a special effort … the one principle that distinguishes software most from other types of industrial production*"

(6) GENERALITY: "*asked to solve a problem, try to focus on the discovery of a more general problem that may be hidden behind the problem at hand … indeed, it may even be simpler*"

(7) INCREMENTALITY: "*the desired application is produced as a result of an evolutionary process*"

<u>Corresponding software qualities</u>: [*GheJaz03:17ff*]

Correctness, Reliability, Robustness, Performance, Usability, Verifiability, Maintainability, Repairability, Evolvability, Reusability, Portability, Understandability, Interoperability

More specific SE-principles will be presented in the following subchapters 2.2.2 (Code of Ethics), 2.2.4 (Paradigms) and 2.2.6 (Development methods and processes).

2.2.2. Profession: Codification of ethics, education and knowledge

Within the last decade, astonishing progress has been made in the international development of some key elements for a professional SE-discipline, mainly by the first-time-codification of *ethical principles*, *educational guidelines* and the existing *engineering knowledge* (at the best independent from specific trendy technologies). Since 1993 the Institute of Electrical and Electronics Engineers (IEEE) as well as the Association for Computing Machinery (ACM) had – in cooperation with the international Software Industry and related academic bodies – a leading role in this progress, which is likely just an intermediate step on the way to a fully developed engineering discipline ("*The contents of this Guide must therefore be viewed as … baseline for future evolution*" [*SWEBOK04:xx*]). But regarding the SE-evolution discussed in chapter 2.1 and the Shaw-model from 1996 (figure 2.1) these intermediate steps are important milestones which indicate an increasing level of maturity in Software Engineering. Therefore, three milestones within this codification-progress are presented:

● Software Engineering Code of Ethics and Professional Practice (IEEE-CS/ACM – 1999)

The "*Software Engineering Code of Ethics and Professional Practice (SECEPP)*" expands the fundamental engineering principles presented in chapter 2.2.1, especially by introducing the "ethical" approach, which is always a delicate question in science and engineering (e.g. when developing military applications and considering historical examples of obviously ethic-free-engineers like Wernher von Braun, who developed with the same enthusiasm and professionalism rockets for Adolf Hitler as he did later for US-President John F. Kennedy, moreover for the first he even abused concentration camp prisoners as forced labour slaves). The SE-code was developed by a multinational joint task force of the IEEE-Computer Society and ACM and the final version (5.2) was adopted by IEEE-CS and ACM in 1999. There is a "short" and a "full version" (with many detailed clauses), both determining eight principles "*intended as a standard for teaching and practicing software engineering … ethical and professional obligations of software engineers*" [*SECEPP99, background article:84*]. The main parts of the short version are:

> *"Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:*
> *1. PUBLIC: Software engineers shall act consistently with the public interest.*
> *2. CLIENT AND EMPLOYER: Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.*
> *3. PRODUCT: Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.*
> *4. JUDGMENT: Software engineers shall maintain integrity and independence in their professional judgment.*
> *5. MANAGEMENT: Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.*
> *6. PROFESSION: Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.*
> *7. COLLEAGUES: Software engineers shall be fair to and supportive of their colleagues.*
> *8. SELF: Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession."* [*SECEPP99*]

● *Education: Computing Curriculum (CC2005) & SE2004 (formerly CCSE, including SEEK)*

In 1998 the IEEE Computer Society and ACM started a Joint Task Force on Computing Curricula (CC), which should review the outdated Computing Curricula 1991 and develop a new curricula, originally named CC2001 (to "*define Educational Curricula for undergraduate, graduate, and continuing education*" [*SWEBOK04:vii*]). Due to the rapid technological progress and ongoing diversification of computer science, in 2001 the CC2001-task-force decided, that one single report was not useful, and there should be an own report (respectively volume) for each of the following disciplines: *Computer Science, Information Systems, Computer Engineering, Software Engineering* and *Information Technology* (last one added later). So the most recent report [*CC2005*] acts more as a survey and provides just a general overview about Software Engineering and its limitations (see the self-explanatory figure 2.4).

Therefore the software-engineering-community within IEEE-CS and ACM got the mandate for an own volume within the CC-project, first called "*Computing Curriculum Software Engineering (CCSE)*" and later renamed to its current name: "*Software Engineering 2004 (SE2004)*". The SE2004 should "*provide guidance to academic institutions and accreditation agencies about what should constitute an undergraduate software engineering education*" [*SE2004:1*]. The SE2004 contains three main elements:

(1) Guiding principles and characteristics of software engineering graduates (Chapter 3):
"*Graduates of an undergraduate SE program must be able to*
1. Show mastery of the software engineering knowledge and skills, and professional issues necessary to begin practice as a software engineer.
2. Work as an individual and as part of a team to develop and deliver quality software artifacts.
3. Reconcile conflicting project objectives, finding acceptable compromises within limitations of cost, time, knowledge, existing systems, and organizations
4. Design appropriate solutions in one or more application domains using software engineering approaches that integrate ethical, social, legal, and economic concerns.
5. Demonstrate an understanding of and apply current theories, models, and techniques that provide a basis for problem identification and analysis, software design, development, implementation, verification, and documentation.
6. Demonstrate an understanding and appreciation for the importance of negotiation, effective work habits, leadership, and good communication with stakeholders in a typical software development environment.
7. Learn new models, techniques, and technologies as they emerge and appreciate the necessity of such continuing professional development." [*SE2004:15f*]

(2) The "*Software Engineering Education Knowledge (SEEK)*" (Chapter 4) – a body of knowledge for undergraduate programs in software engineering, so "*what every SE graduate must know*" [*SE2004:1*]. The SEEK is hierarchically organised with ten Knowledge Areas (KA), subdivided into Knowledge Units (KU) and Topics. The Knowledge Areas are: *Computing Essentials (CMP), Mathematical & Engineering Fundamentals (FND), Professional Practice (PRF), Software Modeling & Analysis (MAA), Software Design (DES), Software Verification & Validation (VAV), Software Evolution (EVL), Software Process (PRO), Software Quality (QUA), and Software Management (MGT).*

(3) Guidelines for curriculum designers (Chapters 5+6): "*ways that this knowledge and the skills fundamental to software engineering can be taught in various contexts*" [*SE2004:1*]

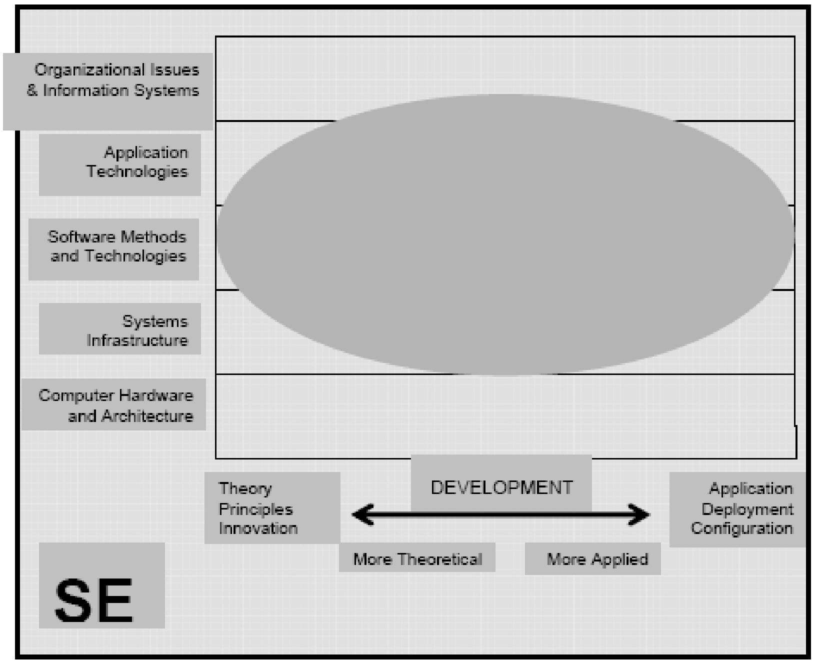

Organizational Issues & Information Systems
Application Technologies
Software Methods and Technologies
Systems Infrastructure
Computer Hardware and Architecture
Theory Principles Innovation
DEVELOPMENT
Application Deployment Configuration
More Theoretical
More Applied
SE

Figure 2.4: Computing Curriculum 2005: View on Software Engineering [*CC2005:21*]

● *(Guide to the) Software Engineering Body of Knowledge (SWEBOK) by IEEE-CS*

"Every profession is based on a body of knowledge and recommended practices" [*SWEBOK04:viii*]

The "*(Guide to the) Software Engineering Body of Knowledge (SWEBOK)"* is currently the most controversial outcome of the mentioned efforts for a professionalisation in Software Engineering. In a nutshell it can be described as a STRUCTURED, COMMENTED AND SELECTIVE BIBLIOGRAPHY about the discipline of Software Engineering as well as the knowledge which already exists in the published literature. SWEBOK claims to promote a consistent view of "*Generally Accepted Knowledge*" in software engineering worldwide, clarify its place and boundaries to other disciplines, provide a topical access and foundation for curriculum development as well as individual certification and licensing material. It was edited and published by the IEEE Computer Society – together with partners from the Software Industry as well as from academic bodies – and it was criticised by other parts of the SE-community as being too US-(licensing)-centred, textbook-oriented, Anglophone-centred and to strong biased toward "heavyweight"-methodologies.

SWEBOK structures the "*Generally accepted Knowledge*" in ten Knowledge Areas (KA) supplemented by related disciplines. The ten SWEBOK Knowledge Areas (which are further subdivided into detailed topics) are: *Software requirements, Software design, Software construction, Software testing, Software maintenance, Software configuration management, Software engineering management, Software engineering process, Software engineering tools and methods, Software quality*. The related disciplines are: *Computer engineering, Computer science, Mathematics, Management, Project management, Quality management, Software ergonomics, Systems engineering*. For each KA there are reference materials (book chapters, refereed papers or other recognized sources) and a matrix relating the reference material to the listed topics [*SWEBOK04*].

Project SEBOK was started in 1998 by the Software Engineering Coordinating Committee (SWECC), formed by the IEEE Computer Society (IEEE-CS) and the ACM. So initially the project was – like the SE-code of ethics and the SE-curriculum – an outcome of joint efforts between IEEE-CS and ACM. But on the 30th June 2000 the ACM Council decided to withdraw from the project, mainly due to critics about SWEBOK being the "*basis for licensing software engineers*" (which was intended by IEEE but not supported by ACM. Overall for an "international" project a quite US-centred debate about legal advantages after accidents – e.g. in trials – when using methodologies officially included in SWEBOK).

After 2000 the IEEE-CS continued project SWEBOK with interested partners from research and industry, so altogether the (Guide to the) SWEBOK evolved in three phases and versions: *Strawman* (1998), *Stoneman* (2000/2001) and the current version *Ironman* (2004). SWEBOK states, that the guide was "*the product of extensive review and comment*" [*SWEBOK04:xix*], including a review- and voting-process in which initially 500 people participated, later reduced to 120 in the Ironman-phase. Even so, criticism was coming also by people, who are still listed as part of the "Review Team" in SWEBOK 2004:

- Grady Booch stated: "*The SWEBOK I reviewed was well-intentioned but misguided, naive, incoherent, and just flat wrong in so many dimensions*" (XP-listserv on yahoogroups, 31st May 2003).
- Florida-Tech-SE-professor Cem Kaner published a negative evaluation of SWEBOK: "*excludes modern methods (such as agile development) … SWEBOK's criteria for inclusion and exclusion of topics is unsatisfactory … (Much in) SWEBOK is organized strangely, is dated, and many of the techniques (etc.) are marginal in terms of how often they are used*" (Kaners Blog, 27th June 2003)
- Wolfgang Zuser, an Austrian Reviewer, declares: „*A significant defect of the project is, that only Anglophone sources were used*"[*ZuserGre04:27*], which is at least conceded in the limitations of the guide [*SWEBOK04:xix*]. This thesis supports this critique, as SWEBOK currently excludes ideas and concepts which were published in other scientific world languages.

The concept of the project seems to be also a main source of all critics, as SWEBOK distinguished three knowledge-categories, and only the first one is regarded and included [*SWEBOK04:1-3*]:

(1) GENERALLY ACCEPTED KNOWLEDGE ("*Established traditional practices recommended by many organizations … should be included in the study material for the SE-licensing examination that graduates would take after gaining four years of work experience. … specific to the US style of education*")
(2) ADVANCED & RESEARCH KNOWLEDGE ("*tested and used only by some organizations*")
(3) SPECIALIZED KNOWLEDGE ("*practices used only for certain types of software*").

In the strongly diversified and rapidly changing Software Industry this approach seems to be quite problematic and subjective, as Software Engineering struggles since the 1960's to find an appropriate balance between the "art"- and the "engineering"-approach (see chapter 2.1 and [*Boehm02*, *Boehm03*]). In this context SWEBOK is criticised for pushing the "*traditional, heavyweight, rigid, documentation-heavy approaches*", even when the 2004-version already includes some books by Kent Beck.

Irrespective from SWEBOK other sources (supporting the engineering-approach in SE) also stated the need for a profound codification of the existing knowledge, e.g. [*GheJaz03:531*]: "*… reuse of previous knowledge and designs is standard practice in established engineering disciplines … A prerequisite of this reuse is the ability to codify existing knowledge … software engineering will be considered to have achieved the status of a true engineering discipline only after we have such handbooks that software engineers can use in their daily work*". Currently it seems, as SWEBOK has NOT already achieved this status, but it is an interesting intermediate step.

2.2.3. Evolution: History and current trends of Software Engineering

“*Panta rhei*”
(*Heraclitus of Ephesus*)

When presenting Software Engineering, the historical approach is quite a logical option – as SE is still a changeable, evolving and expanding field which continues to struggle with its self-definition as an engineering-discipline – but it is likely also the most voluminous way to illustrate the topic. Therefore one has to simplify and prioritise when doing so in a few pages, for which the best example in recent years is an article by Barry Boehm about the history and future of Software Engineering [*Boehm06*].

As it is quite hard to top Boehm with all his experience since the 1950’s and his overall quite unbiased overview, his striking article also builds the foundation of this subchapter (supported by [*Fleissner96*], [*Mahoney04*]). Boehm itself states that “*there are many types of software engineering … unlike the engineering of electrons, materials, or chemicals, the basic software elements we engineer tend to change significantly from one decade to the next*” [*Boehm06:12*]. Therefore his approach is likely the same as the metaphor of [*Mayr05:78*] about the “*pendulum*” which currently swings and foreseeable stabilises between “*process monsters*” and agile methods. But Boehm widely expands this pendulum-approach and presents the entire history of SE-evolution since the 1950’s as a decade-by-decade sequence of alternating contradictions – for which he applies the dialectic of the famous German philosopher Hegel (dissolving the contradiction between THESIS and ANTITHESIS by a SYNTHESIS – in German-speaking science and rhetoric nowadays often used by the “*Dialektischer Fünfsatz*”). The quintessence of his “*Hegelian View of Software Engineering’s Past*” is a graphical overview of six decades SE-history and current trends (see figure 2.5).

● *From the origins of computer science till the 1940’s*

The origins of modern computers trace back to the 17th century and the first ideas for mechanical calculating machines – by people like Wilhelm Schickard, Gottfried Wilhelm Leibnitz, Blaise Pascal – and the famous analytical engine by Charles Babbage with the worlds first computer programmer Lady Ada Lovelace [*Fleissner96:33ff*]. But even when there had been computers before, like mechanical computers (e.g. Z1 by Konrad Zuse or the Hollerith), simple electromechanical machines (e.g. Zuses Z2, Z3 and Z4 with relay’s) or the tube-technique used by ENIAC, modern computer science is strongly related to the “*explosive development*” [*HorHill89:xix*] of electronics and semiconductors.

While older “computers” used techniques like relay's or vacuum tubes, the first realization of a transistor in 1947 revolutionised electronics and later also computer science. The following evolutions of semiconductors lead to the invention of Integrated Circuits (about 1960 by Texas Instruments as well as by Fairchild Semiconductor) and microprocessors (in the early 1970s). This laid the electronic foundations for modern computer science and its hardware. Together with memory, bus and I/O-ports, microprocessors are still the key components which constitute the hardware, as even modern computers still trace back to the "*von-Neumann-architecture*" from 1945. Even when the von-Neumann-architecture has been improved in details and hardware has made quite big improvements (smaller size, higher speed, new storage technologies and groundbreaking input/output-devices like mouse’s, laser printers and LCD-screens), its basic principles remained the same, only challenged by quite few new paradigms as Harvard architecture or parallel computing [*HorHill89*]*;* [*Fleissner96*].

● *1950's – "Software Engineering is like Hardware Engineering"* [*Boehm06:13*]

"*Everyone in the GD [General Dynamics] software organization was either a hardware engineer or a mathematician, and the software being developed was supporting aircraft or rocket engineering. People kept engineering notebooks and practiced such hardware precepts as 'measure twice, cut once,' before running their code on the computer. ... On my first day on the job, my supervisor [...] said, 'Now listen. We are paying $600 an hour for this computer and $2 an hour for you, and I want you to act accordingly.'*"

In 1956, a software development process was invented (by hardware engineers from various disciplines) for the air-defence-project "*Semi-Automated Ground Environment (SAGE)*" – showing, that some kind of Waterfall-approaches were already common at this time (Boehm even points out, that the process-elements can also be arranged as an early form of a V-Model). Success was attributed to the fact that "*we were all engineers and had been trained to organize our efforts along engineering lines*". At this time also first versions of modern programming languages where invented (e.g. FORTRAN, COBOL and ALGOL).

● *1960's – "Antithesis: Software Crafting"* [*Boehm06:13f*]

According to Boehm "*people were finding out that software phenomenology differed from hardware phenomenology in significant ways ... easier to modify ... did not require expensive production lines to make product copies*". Also more and more "*non-engineering people flooded into software development*", which tempted to program with a "*code and fix*"-approach and lead to creative people producing "*heavily patched spaghetti code*" (and also the birth of the "*hacker culture*" and "*cowboy programmers*").

On the other hand, the infrastructure got better and mature high-order-languages simplified programming-practice (e.g. BASIC). The so-called "*software crisis*" was an upcoming topic, therefore the famous "*landmark*" Software Engineering Conferences sponsored by the NATO where held in 1968 and 1969 (see chapter 2.1): "*It was clear that better organized methods and more disciplined practices were needed to scale up to the increasingly large projects and products that were being commissioned.*"

● *1970's – "Synthesis and Antithesis: Formality and Waterfall Process"* [*Boehm06:14f*]

"*The main reaction to the 1960's code-and-fix approach involved processes in which coding was more carefully organized and was preceded by design, and design was preceded by requirements engineering*". The GOTO-Statement was questioned (e.g. by Dijkstra, Boehm-Jacopini): "*Showing that sequential programs could always be constructed without goto's led to the Structured Programming movement*" – which had two branches: "*formal methods … focused on program correctness, either by mathematical proof or … via a 'programming calculus'*" and a "*mix of technical and management methods, top-down structured programming with chief programmer teams*". The second branch created books like "*The Mythical Man Month*" [*Brooks95*] or "*Psychology of Computer Programming*" [*Wein04*].

"*The success of structured programming led to many other 'structured' approaches applied to software design.*" (e.g. modularity, cohesion, information hiding, abstract data types and structured design). In 1970 Winston Royce published his version of the waterfall-model, including ideas like "*Requirements-driven process*", "*build it twice*" and even the idea of iterations [*RoyceWi70*]. But it was most frequently "*interpreted as a purely sequential process … These misinterpretations were reinforced by government process standards emphasizing a pure sequential interpretation of the waterfall model … The sequential waterfall model was heavily document-intensive, slowpaced, and expensive to use*". Also languages like PASCAL, C, PROLOG and SQL were developed.

● ***1980's – "Synthesis: Productivity and Scalability"*** [*Boehm06:15ff*]

"*1980's led to a number of initiatives to address the 1970's problems, and to improve software engineering productivity and scalability … organizations spending 60% of their effort in the test phase found that 70% of the "test" activity was actually rework that could be done much less expensively if avoided or done earlier … could reduce costs through investments in better staffing training, processes, methods, tools, and asset reuse*". Contractual Standards were developed to make processes more compliant (e.g. DoD-STD-2167, MIL-STD-1521B) – strongly reinforcing the waterfall model. The Software Capability Maturity Model (SW-CMM) was invented and stepwise improved, while at the same time also ISO-9001 developed practices for software: "*The threat of being disqualified from bids caused most software contractors to invest in SW-CMM and ISO-9001 compliance. Most reported good returns on investment due to reduced software rework … Improved software processes contributed to significant increases in productivity by reducing rework, but prospects of even greater productivity improvement were envisioned via work avoidance*".

There were two approaches for work-avoidance: revolutionary ("*emphasized formal specifications and automated transformational approaches to generating code from specifications*") and evolutionary ("*mixed strategy of staffing, reuse, process, tools, and management, supported by integrated environments*"). The famous paper "*No Silver Bullet*" [*Brooks95*] from 1986 distinguished between "*accidental repetitive tasks*" and "*'essential' tasks unavoidably requiring syntheses of human expertise, judgment, and collaboration*". Brooks promoted solutions like "*great designers, rapid prototyping, evolutionary development … and work avoidance via reuse*". Especially reuse became an important topic in work avoidance and lead to powerful OS, DBMS, GUI-builder (with WYSIWYG-editing) and middleware. These effects where supported by the appearance of "*very high level*" programming languages, object-oriented programming (OOP) and visual programming, documented by the invention of C++, Smalltalk, Perl or Turbo Pascal.

● ***1990's – "Antithesis: Concurrent vs. Sequential Processes"*** [*Boehm06:18f*]

"*The strong momentum of object-oriented methods continued into the 1990's. Object-oriented methods were strengthened through such advances as design patterns; software architectures and architecture description languages; and the development of UML. The continued expansion of the Internet and emergence of the World Wide Web strengthened both OO methods and the criticality of software in the competitive marketplace*". The importance of Software strongly increased and the need for short time-to-market "*caused a major shift away from the sequential waterfall model to models emphasizing concurrent engineering of requirements, design, and code … shift to user-interactive products with emergent rather than prespecifiable requirements*". Boehm qualifies his 1988-risk-driven Spiral Model [*Boehm88*] as "*process to support concurrent engineering*". Stakeholders got more attention by "*software risk management activities and the use of the stakeholder win-win Theory W*" as well as by a "*set of common industrycoordinated stakeholder commitment milestones … Life Cycle Objectives (LCO), Life Cycle Architecture (LCA), and Initial Operational Capability (IOC)*".

Overall, iterative and evolutionary development methods where an upcoming topic, and simultaneously the Free Software Movement and Open Source Software Development became more important, illustrated by the success of Linux. The "*increased usability of software products by non-programmers*" challenged the development of "*programmer-friendly user interfaces*" and lead to more effort in the field of "*human-computer-interaction (HCI)*". New programming languages supported these trends, e.g. Java, PHP, Visual Basic and C# (which was already on its way for the new millennium).

- ***Since 2000 – "Antithesis and Partial Synthesis: Agility and Value"*** [*Boehm06:19ff*]

"*So far, the 2000's have seen a continuation of the trend toward rapid application development, and an acceleration of the pace of change in information technology (Google, Web-based collaboration support), in organizations (mergers, acquisitions, startups), in competitive countermeasures (corporate judo, national security), and in the environment (globalization, consumer demand patterns)*". Furthermore, Boehm diagnoses an "*increasing frustration with the heavyweight plans, specifications, and other documentation imposed by contractual inertia and maturity model compliance criteria*", leading to the Agile Manifesto in 2001 and an emergence of agile methods, "*such as Adaptive Software Development, Crystal, Dynamic Systems Development, eXtreme Programming (XP), Feature Driven Development and Scrum*". But Boehm also underlines the need for analysis of the "*home grounds*" of agile and plan-driven-methods: "*agile methods were most workable on small projects with relatively low at-risk outcomes, highly capable personnel, rapidly changing requirements, and a culture of thriving on chaos vs. order*" (see figure 2.14).

- ***Current trends*** [*Boehm06:19ff*]

Beside the obvious diagnosis of rapid change and the trend towards more agility, Boehm becomes less historical and more speculative in identifying important SE-trends, of which the most currents are:

- The emergence of "*Value Based Software Engineering (VBSE)*" is stressed by Boehm and linked to the agile movement ("*usability improvement via short increments and value-prioritized increment*"). A field, in which he is active and for which he sees increasing priorities – illustrated by a quote from W. Arthur: "*Computers are working about as fast as we need. The bottleneck is making it all usable*". He predicts an increasing influence of "*technology trends*" as a potential for "*user value*" and evolutionary "*I'll know it when I see it*"-design (IKIWISI).
- Therefore in the field of Requirements and Process Models Boehm predicts: "*requirements emergence is incompatible with past process practices such as requirements-driven sequential waterfall process models and formal programming calculi; and with process maturity models emphasizing repeatability and optimization. In their place, more adaptive and risk-driven models are needed. More fundamentally, the theory underlying software process models needs to evolve from purely reductionist 'modern' world views (universal, general, timeless, written) to a synthesis of these and situational 'postmodern' world views (particular, local, timely, oral) … The value-based approach also provides a framework for determining which low-risk, dynamic parts of a project are better addressed by more lightweight agile methods and which high-risk, more stabilized parts are better addressed by plan-driven methods*".
- According to Boehm, "*Criticality and Dependability*" will become a topic with high priority, as due to "*the high and increasing software vulnerabilities of the world's current financial, transportation, communications, energy distribution, medical, and emergency services infrastructures, it is highly likely that … a software-induced catastrophe will occur between now and 2025*" (which he compares to 9/11).
- "*Commercial-off-the-shelf (COTS) systems and components*" will increase productivity, and "*developers are spending more and more of their time assessing, tailoring, and integrating COTS products*".
- Further integration of Software Engineering and Systems Engineering (as both disciplines are more and more related and dependent on each other). Reuse and "*legacy software integration*" become more important, as well as "*Model-Driven Development (MDD) … developing domain models whose domain structure leads to architectures with high module cohesion and low intermodule coupling, enabling rapid and dependable application development and evolvability within the domain*".

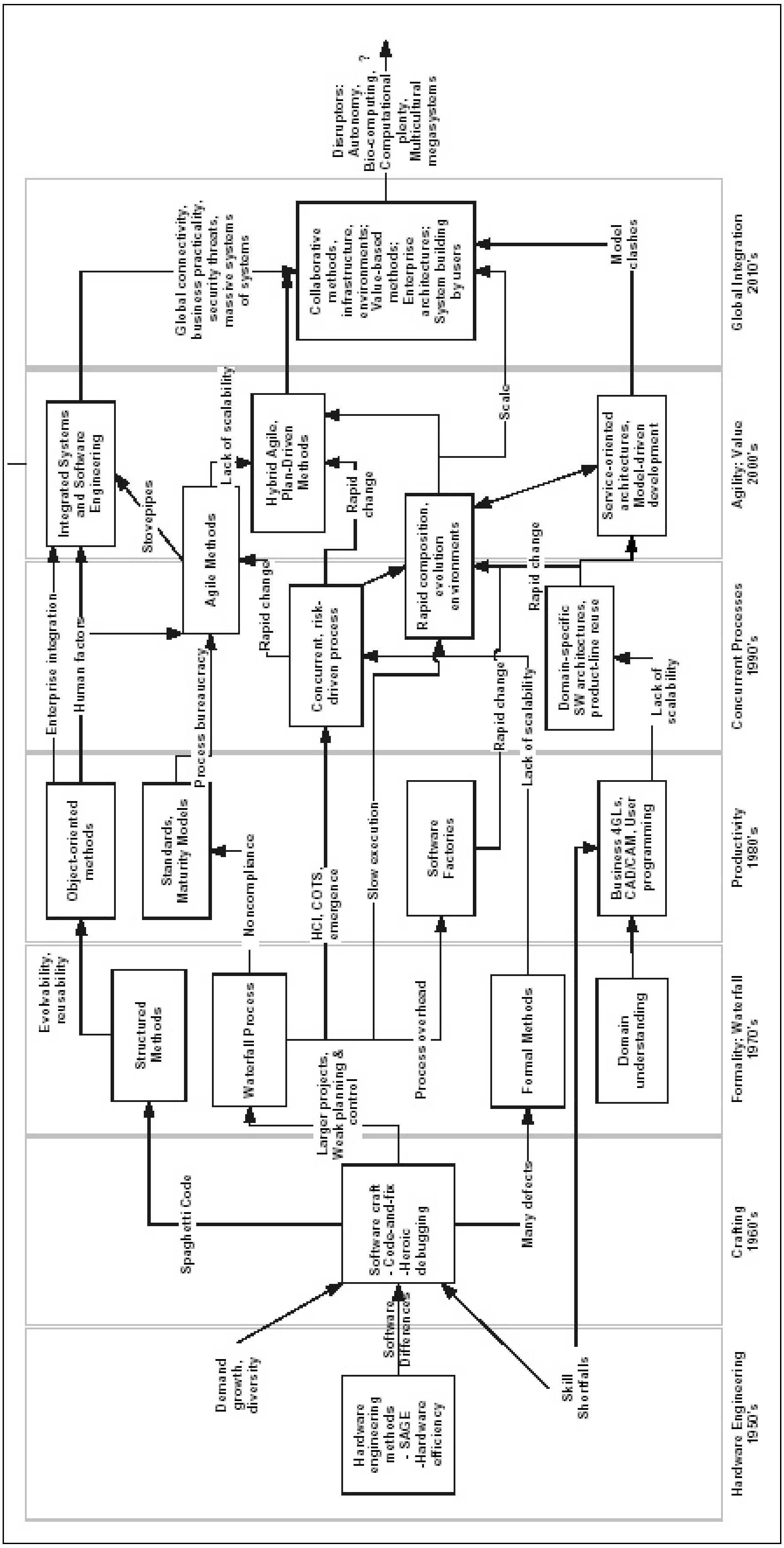

Hardware Engineering 1950's
Crafting 1960's
Formality: Waterfall 1970's
Productivity 1980's
Concurrent Processes 1990's
Agility; Value 2000's
Global Integration 2010's
Hardware engineering methods - SAGE -Hardware efficiency
Demand growth, diversity
Software Differences
Skill Shortfalls
Software craft - Code-and-fix -Heroic debugging
Spaghetti Code
Many defects
Larger projects, Weak planning & control
Structured Methods
Waterfall Process
Formal Methods
Domain understanding
Evolvability, reusability
Process overhead
Object-oriented methods
Standards, Maturity Models
Noncompliance
HCI, COTS, emergence
Slow execution
Software Factories
Business 4GLs, CAD/CAM, User programming
Enterprise integration
Human factors
Process bureaucracy
Rapid change
Concurrent, risk-driven process
Rapid change
Lack of scalability
Rapid composition, evolution environments
Domain-specific SW architectures, product-line reuse
Lack of scalability
Rapid change
Integrated Systems and Software Engineering
Stovepipes
Agile Methods
Lack of scalability
Hybrid Agile, Plan-Driven Methods
Rapid change
Scale
Service-oriented architectures, Model-driven development
Global connectivity, business practicality, security threats, massive systems of systems
Collaborative methods, infrastructure, environments; Value-based methods; Enterprise architectures; System building by users
Model clashes
Disruptors: Autonomy, Bio-computing, Computational plenty, Multicultural megasystems
?

Figure 2.5: A Full Range of Software Engineering History and Trends [*Boehm06:16*]

2.2.4. Quality Management in Software Development: Standards and Models

Standards, Models or Frameworks for the assessment and improvement of Software Development Processes are part of the quality-management-movement that became important in ANY kind of industry during the 1990's. Since that time, especially the widespread ISO 9000 (and all related norms like ISO 9001) became the best know standard in the global business world for establishing a certified *Quality Management System* (QMS) in a company. A QMS is a set of policies, monitoring processes and procedures (therefore quite a lot of documents) that guarantees and improves the quality of the way, how a company produces its outcome (goods or also services).

The main idea of ISO 9000-related QMS is that a product can only be as good as the way it was produced – therefore the standard is (like most quality-management-approaches) focussed on the assessment and improvement of the (production) PROCESS, NOT THE PRODUCT itself. Meanwhile, in the corporate world it has often become necessary for a company to be independently audited and "*ISO 9001 certified*" for being accepted as a supplier and not disqualified from bids.

In Software Development, there are also such kinds of QM-approaches, and they have to be carefully distinguished from the actual Process Models (which is often not done in an appropriate way, as many authors mix these topics, e.g. [*KrollKru03:49ff*] and in a milder form [*Mayr05:79ff*]):

- PROCESS MODELS (and related standards like ISO 12207) describe the sequence of tasks and related activities of the Software Development Process itself (see subchapter 2.2.6).
- QUALITY MANAGEMENT APPROACHES like CMM(I), ISO 9001 (& 90003) or SPICE (ISO/IEC 15504) describe methods to assess, improve or even certify the quality of the (existing) Software Development Process of a company. All with the aim to guarantee and standardise the (minimum) quality of external suppliers, as it is quite hard for a big company (or state-institution) to conduct an in-depth check for each supplier in a big project (see [*ZuserGre04:149ff*], [*Mayr05:128ff*], [*Boehm06:17*]).

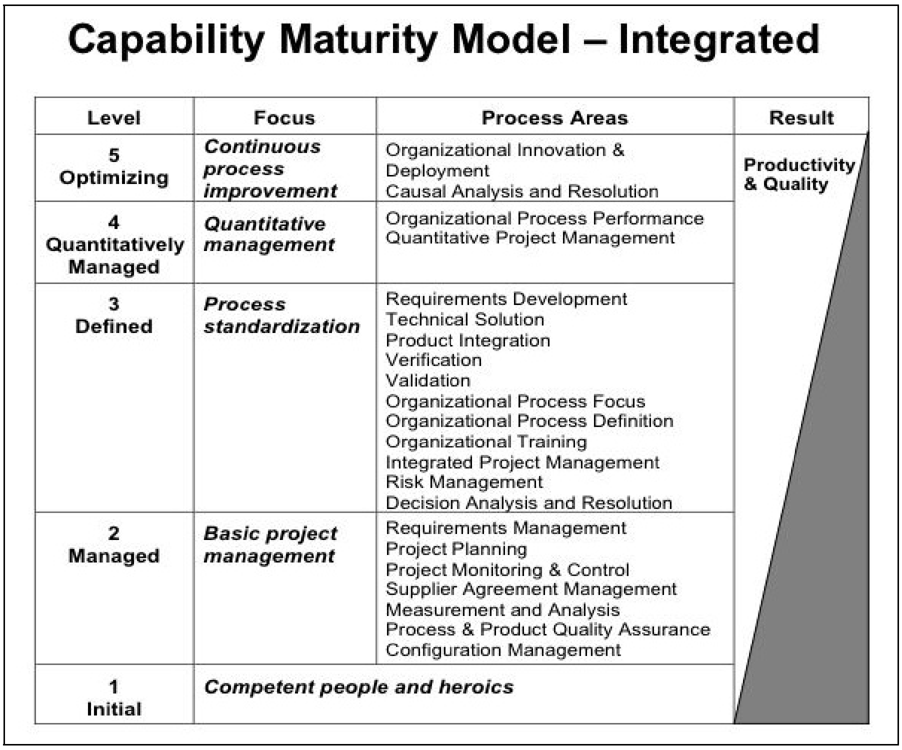
Certainly there are many interesting and mentionable methods for the assessment and improvement of Software Development Processes, an extensive European overview to *Software Process Improvement (SPI)* provides [*MessTull99*]. But the following survey focuses on the three most important approaches in international Software Development: *CMM(I)*, *ISO 9001* (combined with *ISO 90003*) and *SPICE (ISO/IEC 15504)*. Other standards exist, but in the current international Software Engineering practice they are at least superposed (or even superseded) by the three mentioned approaches, even when there are other mentionable standards like:

- Industry or military standards (e.g. "*MIL-STD-1521B"* of the US-Airforce for "*Technical Reviews and Audits for Systems, Equipments and Computer Software*");
- The European BOOTSTRAP-approach, other (trendy) methods like six-sigma or Total Quality Management (TQM) and the OPM3-approach by the *Project Management Institute*;
- Norms from national standardisation organisations (e.g. BSI in Britain, AFNOR in France, DIN in Germany, ÖNORM in Austria or SNV in Switzerland) – though in Software Engineering they often just publish a copy the international ISO-standard.

● ***Capability Maturity Model – Integration (CMM & CMMI) by SEI (since 1991)***

The first famous QM-model in SE reflects the need of big entities (like the military) for a strict classification system of their suppliers to "*discriminate between capable software developers and persuasive proposal developers*" [*Boehm06:17*]. So in 1986 the US-Air Force initiated the project for a "*Software Capability Maturity Model (SW-CMM)*", which was developed by the (quite new) Software Engineering Institute (SEI) at the Carnegie Mellon University (CMU) in Pittsburgh (Pennsylvania). Version 1.0 of the SEI-SW-CMM was released in 1991. The main idea was to "'*help organizations improve their software process' through 'the progression from an immature unrepeatable software process to a mature, well-managed software process*'" [*Benn95:9 quoting Paulk*]. So CMM is not a Process Model but it claims to describe how to get an effective model. CMM covers many Key Process Areas (KPA) of Software Development which have to be improved, defines five levels "*that measure the path from immaturity to maturity*" [*Benn95:9*] – the so called "*maturity levels*" (see figure 2.6) – and describes goals and key practices for the improvement of each KPA. This is normally a pure in-house activity – therefore CMM is not a certificate like ISO 9001.

During the 1990's the CMM-approach was also used to develop models for related issues like People Management (PCMM), Systems Engineering (SECM) or Integrated Product Development IPD-CMM). Therefore in 2000 the first "*Capability Maturity Model Integration (CMMI)*" was published by the SEI and superseded the original CMM (which definitely expires with December 2007). CMMI integrates the various CMM-approaches to overcome the problematic use of multiple models. In August 2006 the current version 1.2 was released (573-pages with 22 improvable process areas [*CMMI06*]), now renamed as "*CMMI for Development (CMMI-DEV)*" to distinguish it from future CMMI's for other issues. Maybe this complexity and ongoing change is one reason why CMM(I) is often associated with "*frustration*" about heavyweight plans: "*One organization recently presented a picture of its CMM Level 4 Memorial Library: 99 thick spiral binders of documentation used only to pass a CMM assessment*" [*Boehm06:19*].



Capability Maturity Model – Integrated

Level	Focus	Process Areas	Result
5 Optimizing	*Continuous process improvement*	Organizational Innovation & Deployment Causal Analysis and Resolution	Productivity & Quality
4 Quantitatively Managed	*Quantitative management*	Organizational Process Performance Quantitative Project Management	
3 Defined	*Process standardization*	Requirements Development Technical Solution Product Integration Verification Validation Organizational Process Focus Organizational Process Definition Organizational Training Integrated Project Management Risk Management Decision Analysis and Resolution	
2 Managed	*Basic project management*	Requirements Management Project Planning Project Monitoring & Control Supplier Agreement Management Measurement and Analysis Process & Product Quality Assurance Configuration Management	
1 Initial	*Competent people and heroics*		

Figure 2.6: Capability Maturity Model – Integrated: Five maturity levels [*mdob.larc.nasa.gov*]

• *ISO 9001:2000 and ISO 90003:2004 (since 1997)*

While the CMM(I) was invented for the field of Software Development, the ISO 9000ff-series is a "*generic*" set of standards for the establishment and assessment of a Quality Management System (QMS) as well as the final certification (by external and independent auditors) in ANY kind of company or organisation. ISO 9000ff was invented by the *International Organization for Standardization (ISO)* and the first version was released in 1987, including ISO 9000, 9001, 9002 and 9003 (of which the most comprehensive 9001 became well-known worldwide). In the years 2000 and 2005 the ISO 9000ff-series was strongly revised, especially the former standards 9002 and 9003 where included in ISO 9001:2000. Currently ISO 9000ff means: ISO 9000:2005 (*QMS – Fundamentals and vocabulary*), ISO 9001:2000 (*QMS – Requirements for certifications*) and ISO 9004:2000 (*QMS – Guidelines for performance improvements*).

Therefore the application of ISO 9000ff to Software Development is just a "*side-effect*" [*Mayr05:129*], but the increasing importance of software lead to an exceptional position: Standard *ISO 9000-3:1997* was published in 1997 to explain how ISO 9001 can be applied to software. Due to the later revision of the 9000ff-series, 9000-3 was replaced in 2004 by the *ISO/IEC 90003:2004* which "*provides guidance for organizations in the application of ISO 9001:2000 to the acquisition, supply, development, operation and maintenance of computer software and related support services … identifies the issues which should be addressed and is independent of the technology, life cycle models, development processes, sequence of activities and organizational structure*" [*www.iso.org – ISO/IEC 90003:2004*]. It contains "*recommendations*" and "*suggestions*" but IS NOT intended as "*assessment criteria*". So ISO 90003 is more comprehensive than CMM(I), as the superior ISO 9001 implies that "*the organization's quality management system should cover all aspects (software related and non-software related) of the business*" [*www.iso.org*]. But then CMM(I) is more detailed, voluminous and SE-oriented (ISO 9001 & 90003 together are less than 30 pages, while version 1.2. of CMM(I) contains 573 pages). Most sources state, that a firm which reached CMM-level three normally also passes the ISO 9001-certification.

• *ISO/IEC 15504: "Software Process Improvement and Capability Determination (SPICE)"*

In June 1992 [*Mayr05:132*] joint efforts of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) lead to project SPICE, with the goal to develop international standards for Software Development in two counts: For in-house "*Process Improvement*" and for the "*Capability Determination (or Evaluation)*" of potential suppliers. In 1998 the earlier trials lead to the official Technical Report (TR), which was split in nine parts, most of them published in 1998 (*ISO/IEC TR 15504-1 ff*). Since 2003 the *ISO/IEC 15504* became a fully developed standard, five parts haven already been published: "*Concepts and vocabulary*" (Part1); "*Performing an assessment*" (Part 2), "*Guidance on performing an assessment*" (Part 3); "*Guidance on use for process improvement (PI) and process capability determination (PCD)*" (Part 4), "*An exemplar Process Assessment Model (PAM)*" (Part 5) [*www.iso.org*].

ISO/IEC 15504 (including a reference model related to ISO 12207) is based on an assessment-matrix with two dimensions: (1) "*Process Dimension*" (exceeding the pure SW-Development and covering all related processes in a software business like management, support, organisation, etc.); (2) "*Capability Dimension*" (Maturity levels like CMM, but on a six-part-scale: 0: *Incomplete process*, 1: *Performed process*, 2: *Managed process*, 3: *Established process*, 4: *Predictable process*, 5: *Optimizing process*). According to [*Sommer06:728*] SPICE is more "*flexible*" than CMM(I), while [*Mayr05:132*] reflects the practitioners motto "*follow CMM(I), certify ISO*".

2.2.5. Software Life Cycle and Software Development Life Cycle

> "*The series of steps that software undergoes, from concept exploration through final retirement, is termed the life cycle*" [*Schach96:8*]

Life Cycles are a common concept in many sciences like Biology (where the term obviously comes from), Business Administration (e.g. Product or Technology Life Cycle) and of course also in Engineering. Especially in Systems Engineering the life-cycle-concept is an important part of the entire engineering-process from the first idea until the retirement of the product itself and also its production facility. Which is relevant, as most sources confirm that SE is quite close to Systems Engineering – and due to the increasing importance of software in any kind of complex systems (e.g. plants, factories or airplanes) there is an evident exchange between both disciplines [*GheJaz03:13*], [*Sommer06:34*].

Therefore it is quite astonishing that most SE-textbooks do not point out the apparent similarities between the Software Life Cycle and the Systems Life Cycle (although the trend toward "*integration of software and systems engineering*" [*Boehm06:22*] automatically integrates the Software Life Cycle in a superposed Systems Life Cycle). In addition, only few SE-sources present the FULL Life Cycle, also including the retirement-phase (e.g. [*AmNa05*] with "*The Enterprise Unified Process*" or [*Schach96:8f*]).

Instead, many sources don't distinguish properly between the "*Software Development Life Cycle*" and the more comprehensive "*Software Life Cycle*" (including evolution, migration or retirement) – so for those textbooks "maintenance" is often the last stage of the "life cycle" (e.g. [*RoyceWa98:73ff*], [*GheJaz03:6f*]). Furthermore, the life-cycle-idea is often mixed with various "*Software (Development) Life Cycle Processes*", so in some diagrams the "cycle" is presented as a waterfall, V or even spiral. This disorder reflects the fact, that even the underlying international standard ISO/IEC 12207 (1995) for "*Software life cycle processes*" which was derived from earlier US-military standards is somehow ambiguous (presenting a "*Life cycle process tree*"; distinguishing only five "primary" life cycle processes where "*development*" is an oversized conglomerate of 13 quite different activities; "*maintenance*" also comprises migration and retirement).

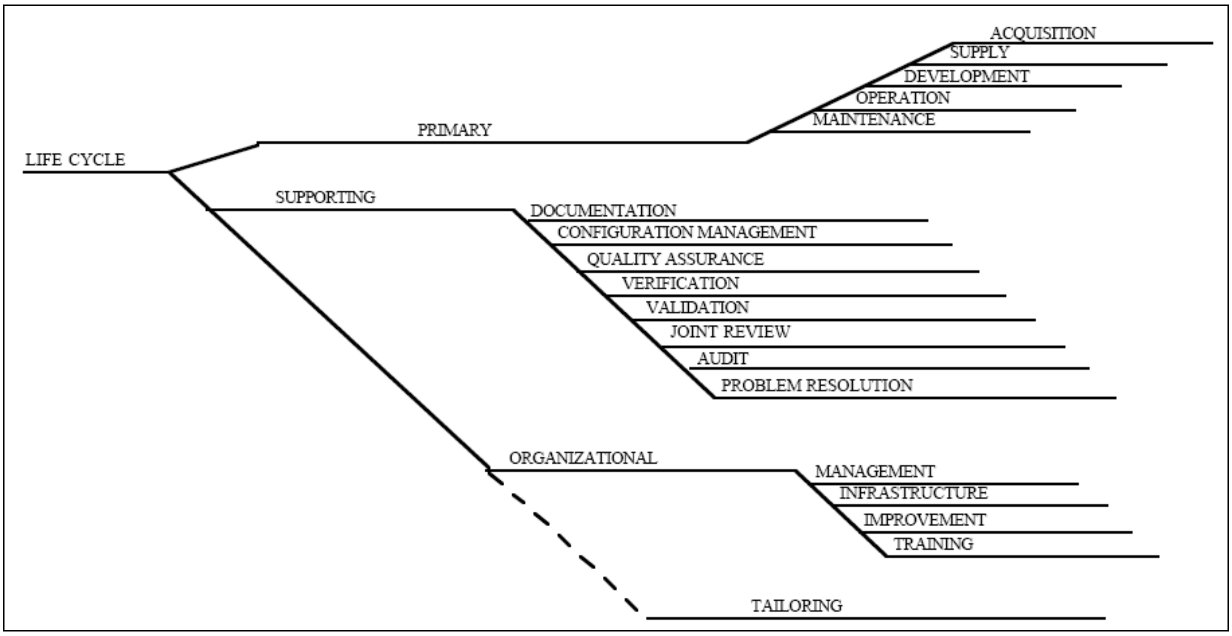


Figure 2.7: Questionable "*Life cycle process tree*" [*Standard ISO/IEC 12207*]

As "*generality*" is an important value of an engineering-discipline (see 2.2.1) this thesis supports the more comprehensive view of [*Schach96:8f*], [*AmNa05*] (or similar sources) and assumes:

● ***There is a significant distinction between the following three terms:***

- *Software Life Cycle* (SWLC: the full Software Life Cycle from first idea till retirement);
- *Software Development Life Cycle* (SWDLC: covers only the development-activities of SWLC; sometimes fragmented in several development cycles, e.g. "*iterations*" and/or "*increments*");
- Various "*Process Models*" (e.g. *Waterfall, V, Spiral, Unified Process, Enterprise Unified Process*), with quite different "*scopes*" [*Mayr05:79*], describing the required activities or phases of Software Development. Most of them consider only specific parts of the Software Development Life Cycle, only few try to cover the full Software Life Cycle (like the EUP described in [*AmNa05*]). These Models are basically what ISO/IEC 12207 defines as "*Life cycle model*".

● ***The entire Software Life Cycle (SWLC) is quite similar to the System Life Cycle (SLC):***

The SWLC comprises a number of stages (also called "phases") that every software is going through, independent of the applied methodology how these phases are performed (e.g. sequential, parallel, iterative, incremental, object-oriented, agile, model-driven) and independent of the corresponding Process Model (e.g. Waterfall, V-Model, Spiral, UP, EUP). Even a quite small software project – performed within some days – will somehow include these phases, even the design has happened with a pen(cil) on a simple sheet of paper. According to Schach "*These phases probably do not correspond exactly to the phases of any one particular organization … Similarly, the precise name of each phase varies from organization to organization*". Therefore he brings a concept "*chosen to be as general as possible*", which is correct in comparison to other examined textbooks. These "general" Life Cycle Phases are after [*Schach96:8f*]:

(1) REQUIREMENTS PHASE: "*concept is explored and refined … client's requirements are elicited*"
(2) SPECIFICATION PHASE: "*client's requirements are analyzed and presented in the form of the specification document … 'what the product is supposed to do' … sometimes called the analysis phase*"
(3) PLANNING PHASE (Software Project Management Planning): "*A plan is drawn up … describing the proposed software development in detail*"
(4) DESIGN PHASE: "*First comes architectural design in which the product as a whole is broken down into components, called modules. Then each module in turn is designed, this process is termed detailed design. … resulting design documents describe 'how the product does it'*"
(5) IMPLEMENTATION PHASE (Production): "*The various components are coded and tested*"
(6) INTEGRATION PHASE: "*components of the product are combined and tested as a whole … tested by the client (acceptance testing) …. product is accepted by the client and goes into operations mode*"
(7) MAINTENANCE PHASE: (7a) "*corrective maintenance (or software repair) … removal of residual faults*"; (7b) "*enhancement (or software update) … changes of the specifications and the implementation of those changes … two types of enhancement: … perfective maintenance, changes that (…) will improve the effectiveness of the product … adaptive maintenance, changes made in response to changes in the environment*"
(8) RETIREMENT: "*product is removed from service*" (may include migration-tasks for a new system)

There is no generally accepted SWLC-definition, therefore this concept and its phases may be questioned as well (e.g. missing of validation; no proper separation between maintenance, support and evolution). But it respects many engineering-principles like *generality* or *modularity* (see 2.2.1), is extensive and is in the development phases quite similar to other textbooks (e.g. [*Benn95:45ff*], [*GheJaz03:6f*]).

2.2.6. Software Development Processes: Methods, methodologies and Process Models

FROM 1970's "*self-fulfilling prophecy: 'We'd better hurry up and start coding, because we'll have a lot of debugging to do'*" [*Boehm06:15*]

TO "*... characteristic of a successful software development process is the well defined separation between 'research and development' activities and 'production'*" [*RoyceWa98:73*]

One of the first steps in SE toward professionalism and a fully-developed engineering discipline was "*the separation of the engineering stage from the production stage*" [*RoyceWa98:8*], which came up at the end of the 1960's, obviously influenced by hardware-related engineering techniques [*Boehm06:14*] – and is nowadays contested by agile approaches which incorporate the client in the development team. Today there is a wide repertory of Process Models for the Software Development Process, following various approaches, methods or methodologies and covering quite different scopes of the Software Development Life Cycle (SWDLC) or even the entire Software Life Cycle (SWLC) – often similar to the Systems Development Life Cycle (SDLC) or the System Life Cycle (SLC), see subchapter 2.2.5.

There is still no generally accepted taxonomy about these methods and every new approach brings new questions (e.g.: Is Extreme Programming already a Process Model or just a method for rapid software development, like [*Sommer06:Chapter17*] implies, see also figure 2.3). Therefore every textbook has its own classification of Process Models, also called "*Software development process (framework)*", "*Life cycle process*" or "*Life cycle model*" (supplemented by the term "*Vorgehensmodell*" in German-speaking sources and "*(modèle du) cycle de développement de logiciel*" in French). So this subchapter carefully delimits the practical PROCESS MODELS from quality-management-models like CMM (see 2.2.4) and the more comprehensive life-cycle-concept (see 2.2.5). Two terms have to be distinguished:

- The SOFTWARE DEVELOPMENT PROCESS is "*the way we produce software. It incorporates the software life-cycle model, the tools we use and the individuals, building the software*" [*Schach96:28*]. So it is the practical sequence of activities from the beginning of the project till its formal ending. In professional SE the development process normally follows one (or even several) Process Model(s), tailored and implemented for the purpose of an individual project.
- A PROCESS MODEL (also called "*life cycle model*" [*Schach96:52ff*]) is a structured model which describes how the entire Software Development Process can be organised, divided and carried out, mostly within a team and following one or several methods respectively methodologies. Basically it defines in an abstract way: A "*series of steps*" [*Schach96:52*] – mostly called PHASES – which the project passes through; Their order and how they relate to one another (e.g. overlapping, iterative); Perceivable transition criteria between the individual phase (e.g. milestones); Roles in the development-team and their responsibilities; Required activities and artefacts (e.g. documents, products) within each phase; Possible use of supporting tools for Computer Aided Software Engineering (CASE) or Computer Supported Cooperative Work (CSCW). The various existing Process Models aim to "*make the process predictable and controllable … achieve better control of the required qualities of the product*" [*GheJaz03:388*]. Even if they have quite different "*scopes*" [*Mayr05:79*], most of them reflect the Software Life Cycle (or at least its development part) as presented in subchapter 2.2.5.

When choosing and tailoring a Process Model for a project it has to be considered that "*different models may be suitable for different software projects or for different software development organizations*" [*Benn95:47*] and "*blanket prescriptions for the 'best methodology for software productivity' do not exist*" [*GheJaz03:385f*]. In spite of these remarks, ISO/IEC 12207 (see 2.2.5) claims to provide a common framework for so called "*Life cycle models*", but as newer approaches in SE (open source software, agile movement) obviously exceed the structure of this 1995-standard, it will not be covered further. Therefore the following condensed overview only presents the most significant methods and Process Models in SE, trying to cover the full planning spectrum displayed in figure 2.3 (So minor approaches like Microsoft Solutions Framework (MSF) [*ZuserGre04:96ff*], Transformation Model [*GheJaz03:413f*], *Cleanroom* or the "*S(h)ashimi*"-Model are not covered). Independent of the discussion about the most suitable Process Models "*we should at least introduce each software developer to any defined model. … improvement can take place later*" [*Mayr05:100*].

● *Paradigms, methods and methodologies*

Like Boehm demonstrates (see 2.2.3, [*Boehm06*]), the field of paradigms, methods and methodologies in SE is an ongoing history of contradictions (e.g. sequential vs. iterative, structured vs. object-oriented), that often influenced the creation of new Process Models. In SE, mainly three kinds of paradigms can be distinguished and lead to various kinds of "*Patterns*" and "*AntiPatterns*" [*Gamma94*], [*Brown98*]:

(1) PROGRAMMING PARADIGMS (related to programming languages and coding)

(2) DESIGN & ARCHITECTURAL PARADIGMS (related to the design activities):

The *structured paradigm*, the first influential SE-approach in the 1970's and 1980's, lead to Structured Analysis and Design (SADT) as well as Structured Programming and Testing. It was superseded by the *object-oriented paradigm* which traces back to the 1960's, advanced in the 1980's and had its final breakthrough in the 1990's. It originated Object Oriented Programming (OOP) with related languages, Object Oriented Design (OOD) and the Unified Modeling Language (UML). Significant newer approaches are also Model Driven Development (MDD), Value Based Software Engineering (VBSE) and Component Based Software Engineering (CBSE) [*Schach96:15f*], [*ZuserGre04:58ff*], [*Boehm06:18ff*]. Architectural paradigms concern the system- and software architecture (e.g. in distributed systems: client/server vs. n-tier).

(3) PLANNING & MANAGEMENT PARADIGMS (for projects, processes and phases):

As figure 2.3 shows, there is a wide planning spectrum, ranging from plan-oriented ("*'plan' includes documented process procedures that involve tasks and milestone plans*" [*Boehm02:64*]) to agile approaches (exaggerations are "*death by planning*" [*Brown98:221ff*]) and undisciplined "cowboy" hacking). Relevant "driven"-approaches are: *requirements-driven*, *plan-driven*, *risk-driven* (referencing to risk management), "*document(ation)-driven*" [*GheJaz03:409*] and agile design approaches like *feature driven* (FDD) or *test-driven* (TDD). Significant process-phase-orders are: SEQUENTIAL ("*a phase should be completed before the next phases can be started*" [*GheJaz03:404*]), ITERATIVE ("*sequence of incremental steps or iterations. Each iteration includes some, or most, of the development disciplines*" [*KrollKru03:6*]; similar to the Japanese "*Kaizen*"-concept of continuous improvement [*Mayr05:86*]) and INCREMENTAL "*builds*" [*Schach96:60ff*], [*GheJaz03:410ff*] ("*whose stages consist of expanding increments of an operational software product*" [*Boehm88:63*]). Thus Requirements Engineering and the "*inevitable tendency of software requirements to change during the process*" [*GheJaz03:390*] are increasingly addressed.

(0) Hacker approach: Build-and-fix

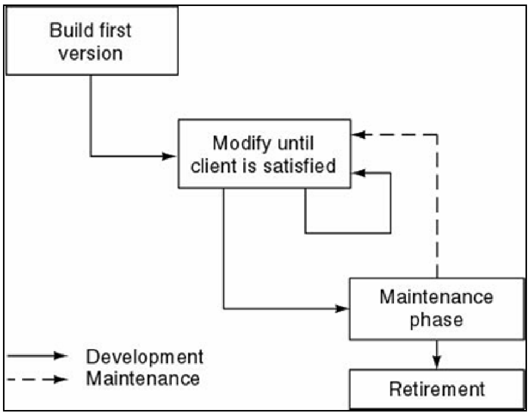


Figure 2.8: Simple "*Build-and-fix" model* [*Schach96:53*]

Many SE-textbooks introduce the *build-and-fix approach* (also called "*code-and-fix*" or "*cowboy coding*") as negative example, how software development should NOT take place (e.g. [*Schach96:52f*], [*GheJaz03:387*], [*ZuserGre04:70*]). It was common "*during the first few decades of software development*" [*Benn95:45*] and mainly consisted of two steps: (1) Write some code; (2) Fix the problem in the code (errors, better functionality, new features). So all tasks were performed at the same time and reworked till the client was satisfied. There was no (formal) separation between requirements, design, implementation and testing, which "*may work well on short programming exercises … is totally unsatisfactory for products of any reasonable size*" [*Schach96:53*]. There are three primary difficulties with this model:

(1) After some fixes the code is poorly structured, so further maintenance is quite expensive and less reliable without design documents (due to higher chances for a "*regression fault*").

(2) There is often "*such a poor match to users' needs*" [*Boehm88:61f*] that only expensive redevelopment saves the software from being rejected.

(3) Maintenance is mainly reduced to the initial programmer and its memories.

(1) Linear Process Models: From "stagewise" to the (sequential) Waterfall Model (WF)

"*Sequential, or not sequential: that is the question*"
(Derived from *William Shakespeare, Hamlet, 3rd Act, 1st Scene*)

The Waterfall (WF) was the first popular form, how the individual phases of the Software (Development) Life Cycle (see 2.2.5) were ordered to enforce a disciplined development and push back the build-and-fix approach. It was named "*from the way each phase cascades into the next*" [*Benn95:46*]. Especially in the 1970's many approaches in SE where somehow called a Waterfall, so there are many variations of the Waterfall Model, reaching from very rigid sequential approaches to more dynamic versions with "feedback loops". But "*it has become fashionable to blame many problems and failures in software development on the sequential, or waterfall, process*" [*Kruchten04:53*]. So many sources focus on the most rigid "sequential" WF-version (as negative example) and furthermore they often wrongly attribute it to Royce (e.g. [*Schach96:53*], [*Sommer06:96*]). Only few thorough sources point out, that Royce's famous 1970-paper [*RoyceWi70*] already pushed back the pure sequential approach, examined its problems and presented basic solutions for an improvement of the Waterfall model, e.g. [*LarBas03:48*]: "*Many—incorrectly—view Royce's paper as the paragon of single-pass waterfall*" (see also [*RoyceWa98:6ff*], [*Boehm06:14f*]).

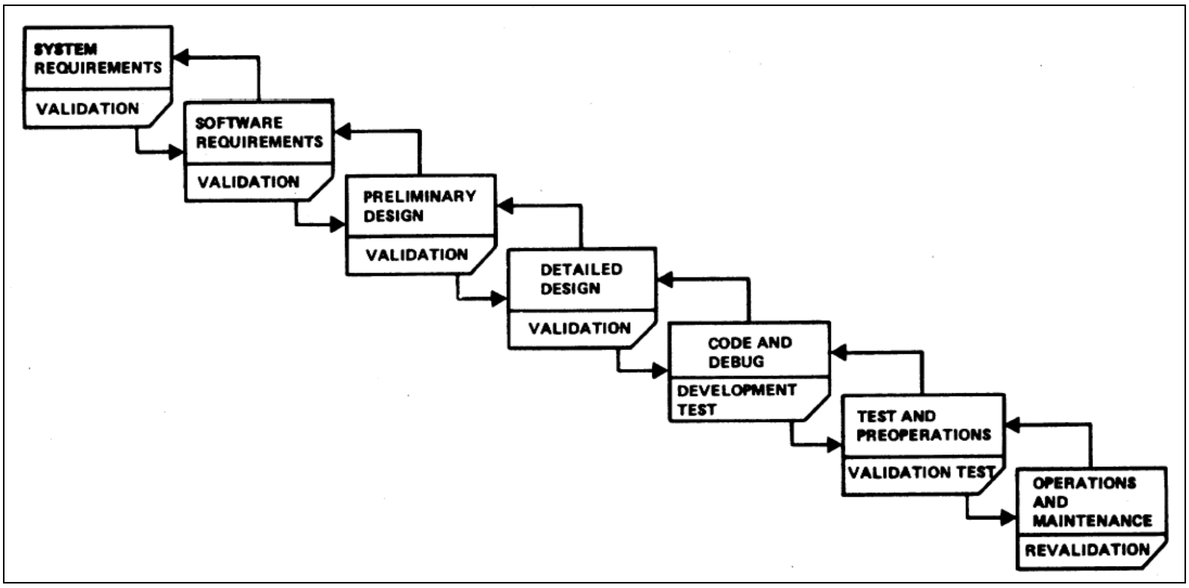


Figure 2.9: One common occurrence of the Waterfall Model in the 1970's [*Boehm76:1227*]

Exemplified by 1950's-project SAGE, Boehm points out that "*sequential waterfall-type models have been used in software development for a long time*" [*Boehm06:13*], often named the "*stagewise*"-approach with "*successive stages*" [*Boehm88:63*] (A "*requirements-driven*" process obviously influenced by hardware engineers). The most rigid – pure sequential – form implies that "*a phase should be completed before the next phases can be started, and each phase results in the preparation of one or more documents that form the input to the next phase*" [*GheJaz03:404*]. Therefore elaborated documents for each phase are the transition criteria to the next phase, therefore this model is often identified as "*document driven*". The "*organizations software development methodology*" [*GheJaz03:405*] often defined a detailed framework how the outputs of each stage should be produced, leading to various sequential-oriented industrial and military (contractual) standards.

In 1970 Royce examined this popular approach and already stated necessary improvements [*RoyceWi70*], leading to an influential "*refinement of the stagewise model*" [*Boehm88:63*] (which many textbooks today present as the source of the Waterfall Model as "conventional" software process). Main ideas where: "*build it twice*"-prototyping, interactive iterations between the phases ("*as each step progresses and the design is further detailed, there is an iteration with the preceding and succeeding steps but rarely with the more remote steps in the sequence*") and "*involve the costumer*". This lead to (quite formal) "*feedback loops*" between successive phases (to avoid excessive feedback among many stages), as presented in figure 2.9. Later there where more improvements leading to many (still linearly) variants, one even with "*overlapping*" phases [*Benn95:46f*].

The main advantage of this model was that it enforced "*much-needed*" [*GheJaz03:407*] discipline and clear documentation in SE ("*to manage and control all the intellectual freedom associated with software development*" [*RoyceWa98:6*]). Also the validations at the end of each phase improved quality and the model fit to related systems engineering approaches. But there are also known disadvantages: Too strong document-driven (can cause misunderstandings as clients are not used to the "language" of specification-documents); Early commitments and phase rigidity (too early "freeze" of results to progress to the next phase, e.g. fix requirements before clients really know what they want); "*Monolithic*" ("*Product is delivered as a whole, months or even years after the requirements were elicited, analyzed and specified … the application may be delivered when the user's needs have changed, and this will require immediate rework*" [*GheJaz03:408*]); Ignores the need for anticipating changes which leads to high maintenance costs.

(2) Evolutionary Process Models: Iterative and Incremental Software Development

„*How do you eat an elephant? One bite at a time!*" [*Kruchten04:60*]

Evolutionary Process Models address the major risks and disadvantages of the Waterfall approach, as they split the rigid, linearly and voluminous workflow of the Waterfall into smaller and therefore more manageable parts to "*avoid a single-pass sequential, document-driven, gated-step approach*" [*LarBas03:47*].

SE-literature is ambiguous about the meaning of "*evolutionary*" development: While some sources connect it predominantly with the incremental approach [*GheJaz03:410ff*], other sources link it to iterative methods [*Sommer06:98*], and even Boehm is somehow ambiguous [*Boehm88:63, Boehm06:19*] (Some also classify the implementation-oriented Prototyping as evolutionary approach). As newer sources and significant Process Models (like the Unified Process) increasingly merge iterative and incremental approaches ([*LarBas03:48*] identify "*compelling evidence of iterative and incremental development's (IID's)*"), this thesis supports this broader view of evolutionary development. So "*evolutionary development*" can have at least two dimensions, both leading to a series of "*mini-waterfalls*":

- ITERATIVE "*refinement & rework*": A pure iterative approach (see figure 2.10) means to rework and refine the ENTIRE software in several iterations till it fits the client-needs and can be released: "*Each iteration includes some, or most, of the development disciplines (requirements, analysis, design, implementation, and so on)*" [*KrollKru03:6*]. This traces back to "*do it twice*"-approaches already recommended in the 1970's by [*RoyceWi70*] or [*Brooks95*] – and according to [*Mayr05:86*] also to Japanese "*Kaizen*"-concepts of continuous improvement.
- INCREMENTAL "*growing*": In the (pure) "*Incremental Model*" [*Schach96:60ff*] the project is divided into several useful "*builds*" (also called "*releases*"), leading to a sequence of basically independent developments (Waterfalls) for each build (similar to figure 2.10 when replacing the word "*iteration*" by "*build*" and expanding the catenation between builds to all levels). This models "*stages consist of expanding increments of an operational software product*" [*Boehm88:63*].

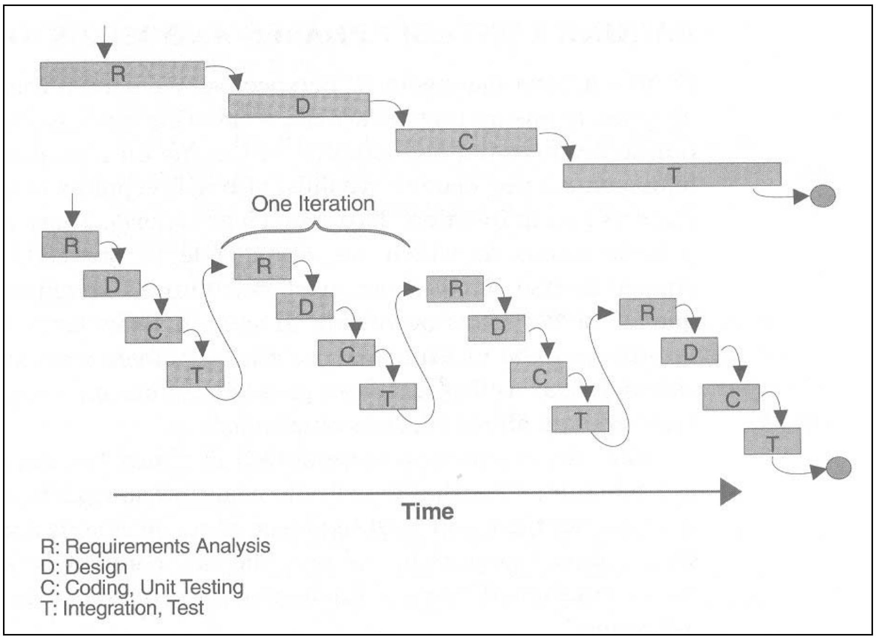


Figure 2.10: "*From a sequential to an iterative lifecycle*" [*Kruchten04:61*]

The use of iterative and incremental concepts has been eased by object-oriented approaches (and UML) which allow the division of complex products in manageable parts. Today significant Process Models (like the Unified Process) mix both concepts and profit from COMBINED ADVANTAGES: Avoid the "*Late Design Breakage*" [*RoyceWa98:12*] of the Waterfall via the possibility to prioritise and build the most essential, critical or risky parts first. So "*risks are usually discovered or addressed during early integrations … easier and less costly*" [*KrollKru03:8*]. Flexibility for "*evolution of requirements"* (refine and adapt changing requirements in a stepwise and controlled way) and detection of "*requirements errors"* while design and coding for indisputable parts already started. Initially unclear requirements can be specified by the client while seeing first builds of the product. Regular feedback to the client and a "*continuous validation of what is being developed*" [*GheJaz03:413*] instead of a "*black box*" [*GheJaz03:390*] or "*big bang*" at the end of the project [*Schach96:62*]. "*Concurrent engineering*" can be applied to parallelise the development activities (within a team or even with several specialised teams). The development team and the client "*learn along the way*" [*KrollKru03:8*], so their improved knowledge can be incorporated in later iterations.

There are also POSSIBLE RISKS of the iterative and incremental approach: "*Evolutionary process may resemble the old code-and-fix unstructured process. … be careful to retain the discipline introduced by the waterfall model*" [*GheJaz03:411*]; "*can too easily degenerate into the build-and-fix-approach. Control of the process as a whole can be lost*" [*Schach96:63*]. Expensive rework and rebuilt of already implemented parts can be necessary in case of big changes in late iterations (e.g. changing the entire architecture). Also the integration of parallel developed builds which do not fit together can cause problems and a rebuilt. Therefore the real engineering- and project-management skill is to find the appropriate number, size and duration of increments and iterations for a specific project (beneath an "architecture first"-approach).

(3) (Rapid) Prototyping and Rapid Application Development (RAD)

A special occurrence of evolutionary development is *prototyping*, which intends to test possible technical solutions or minimise problems with understanding and fixing of requirements. A prototype can be a draft for parts of the User Interface as well as a working demonstration of subsets of the desired product, so there is "*horizontal prototyping*" (one level of the system is realised, e.g. the GUI or the database) and "*vertical prototyping*" (a limited functional part of the system is realised on ALL levels). "(*Rapid) prototyping*" (already popular in the 1970's [*Brooks95*]) is a "*rapid*" initial phase of cyclic prototype-improvements (e.g. "*throwaway prototype*" or "*proof of concept*", see figure 2.11). It is followed by the "classical" development process, based on a better requirements-understanding [*Schach96:58*]. *Rapid Application Development (RAD)* is a special form of prototyping (invented in the 1980's) for data-intensive applications and is strongly related to CASE-tools [*Sommer06:439ff*]. "*Evolutionary prototyping*" can replace the development cycle, so the prototype is "*progressively transformed into the final application*" [*GheJaz03:412*].

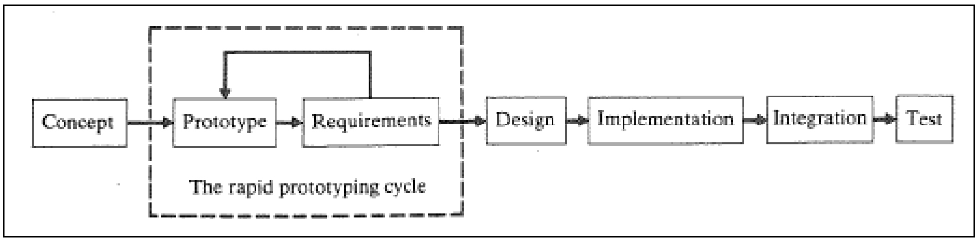


Figure 2.11: Rapid Prototyping Cycle in a sequential development process [*Benn95:47*]

(4) Risk-driven Process Model: Spiral Model

The cyclic Spiral Model was mainly invented and refined by Barry Boehm in the 1980's (first popular publication in 1988: [*Boehm88*]). It combines and refines the iterative and incremental techniques as well as prototyping. Even when other models also reduce risks, the Spiral Model was the first model explicitly addressing and minimising risks as intended activity (and is therefore called "*risk-driven*"). It iterates in a cyclic form until the product is complete (the number of iterations depends on the risks, so it is not fully predictable). Each cycle consists of four stages ending with a review (see figure 2.12):

(1) CYCLE'S GOAL DETERMINATION: Determine objectives, alternatives and constraints;
(2) RISK ANALYSIS: Evaluate alternatives; identify and resolve risks (with prototypes etc.);
(3) DEVELOPMENT: Develop the planned products (e.g. design-documents, artefacts, code) and verify the next-level product(s) – e.g. following Waterfall or evolutionary models;
(4) REVIEW & PLANNING: Review the achieved results and plan the next cycle (iteration).

The radial dimension represents the cumulative costs and the angular shows the progress made in completing a specific cycle. Each cycle can build on the results of the preceding cycle (enabling incremental development). The Spiral Model is a generic model ("*Metamodel*" [*GheJaz03:416*]) that can be used with other approaches ("*spiral model can accommodate most previous models*" [*Boehm88:64*]). For a successful application it needs a skilled risk-manager, and as it is quite extensive, the model is mostly recommended for large and complex projects ("*is applicable to only large-scale software … no sense to perform risk analysis if the cost of performing the risk analysis is comparable to the cost of the project*" [*Schach96:69*]).

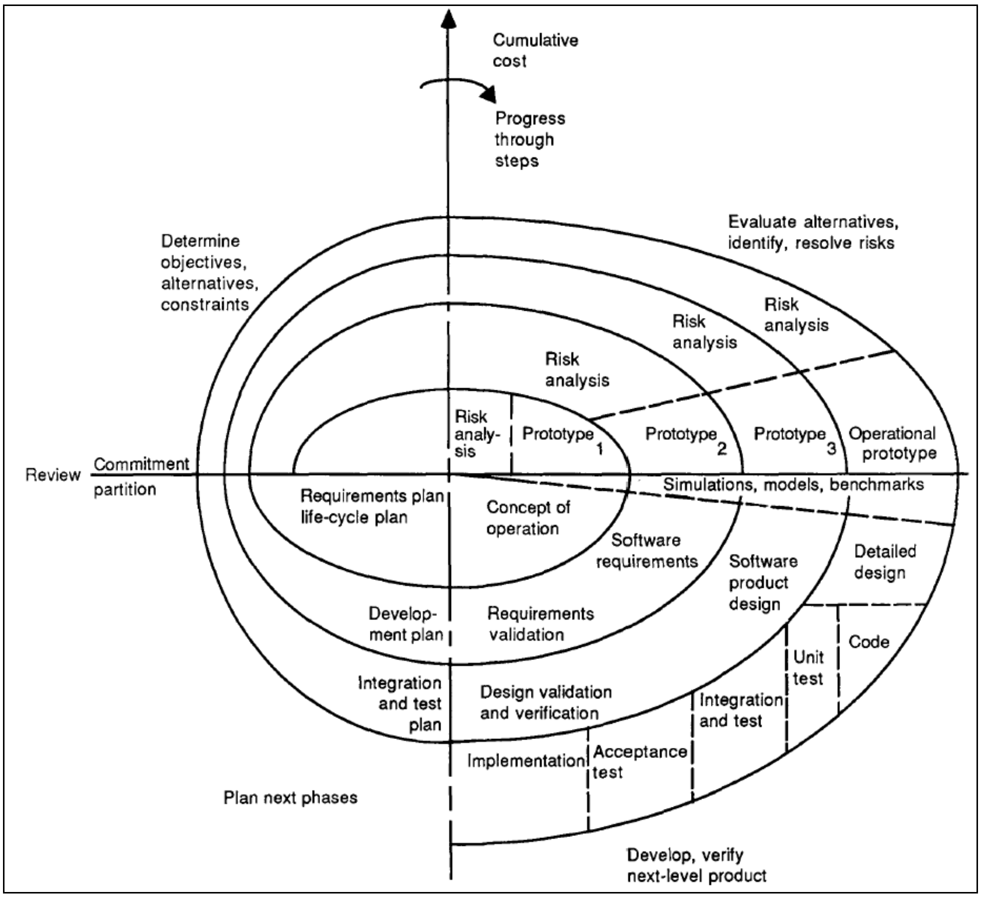


Figure 2.12: Spiral Model of software development [*Boehm88:64*]

(5) V-Model: V-Modell 92, V-Modell 97 and V-Modell XT

The basic V-Model is not a completely new Process model but a further development of the Waterfall and its phases, tracing back to ideas already raised by Barry Boehm in the late 1970's [*ZuserGre04:72*]. Its main idea is to contrast every development phase (respectively product) of the Waterfall Model with a corresponding validation (or test) phase on the same abstraction-level. So the Requirements Analysis is confronted with Acceptance Testing, Design and Architecture is contrasted with Integration Testing and each implemented module is opposed by Unit Testing. As these opposing phases can be arranged in V-shape (with the classical Waterfall-phases on the left descending side and the new test-phases on the right ascending side) the model got its one-character name, also often related to "Validation".

One prominent realisation of this idea was initiated by the German military in 1986 and has meanwhile become the official standard of the German government for IT-projects, today also applied by many companies. Even in Austrian the "*Bundesrechenzentrum (BRZ)*" (Austrian Federal Computing Centre) propagates the use of the German "*V-Modell*". The German *V-Modell* has evolved since its first civil version was published in 1992 (*V-Modell 92*) and rapid changes in SE stimulated a new version in 1997 (*V-Modell 97*). In 2005 the latest version *V-Modell XT* was released – now far beyond the basic Boehm-concept – and is still evolving [*VModellXT*]. XT means "*Extreme Tailoring*" and claims to demonstrate that the new version has overcome too bureaucratic approaches in the earlier versions of the *V-Modell* (criticised by practitioners) and is now more customisable (what needs to be proven in future practice).

(6) Unified Process (UP), Rational UP (RUP), Enterprise UP (EUP) and OpenUP

The *Unified Process (UP)* – mainly developed in the 1990's by the "*Three Amigos*" (Jacobson, Booch, Rumbaugh [*JacoBo99*]) – is a successful melting pot of already proven SE-concepts like: *iterative and incremental development* (balanced with an *architecture-centric approach*), *object-oriented design* with *UML* and *use-case-driven concepts*, *risk-addressing*, and *regular reviews*. Therefore it combines the advantages of all these concepts and easily supports tailoring (e.g. with a *light*, *average and large configuration* [*KrollKru03:61ff*]). The UP itself is a generic model, but it is intrinsically tied to the US-company Rational (now IBM), as the "*Three Amigos*" worked there and were also mainly responsible for the exceeding *Rational Unified Process (RUP)* ® – a commercial improvement of the UP-concept that is still evolving. As UP and RUP ® are "*a widely adopted industrial standard*" [*GheJaz03:444*] they will be presented in detail in chapter 2.3.1. The *Enterprise UP (EUP)* is an extension that covers the full Software Life Cycle [*AmNa05*] and will be discussed in chapter 2.4. Other current extensions are the *OpenUP* (for open source) and the *Agile UP*.

(7) Agile approaches: eXtreme Programming (XP), Scrum, Crystal, FDD, ASD & DSDM

The *Agile Movement* – unified in the "*Agile Manifesto*" (2001) – is the most recent branch of SE-methodologies (see 2.2.3) and forms a countermovement against heavyweight approaches (called bureaucratic "*Process Monsters*" [*Mayr05:78*]). Agile models are often a re-invention of already existing concepts (e.g. prototyping or RAD) so a future "*balance of agile and plan-driven methods*" [*Boehm02:69*] is foreseeable (and necessary to lessen the problems of existing models like "*death by planning*" [*Brown98:221ff*]). Currently "*eXtreme Programming (XP)*" is the most popular agile concept (see chapter 2.3.2). Other significant agile models are: *Scrum*, *Crystal*, *Feature Driven Development (FDD)*, *Adaptive Software Development (ASD)* and the *Dynamic Systems Development Method (DSDM)* [*Sommer06:430ff*].

● *Selection and tailoring of a Process Model for the real Software Development Process*

> "*If you examine the wide array of processes in the industry, you will most likely determine that you can tailor one (or more) to meet your assessed needs*" [*AmNa05:289*]

When introducing the various Process Models, SE-textbooks often omit an own section for "*Tailoring*", so readers may remain with the impression that Software Project Management (SWPM) only implies to pick up any Process Model and follow the model in every detail without reflecting the given project environment. In contrast, customising, "tailoring", configuring and implementing one (or even several) Process Model(s) for an individual software development project is a classical and outstanding skill of an experienced project manager (see chapter 3), and therefore mostly covered in SWPM-textbooks or practitioners guides (e.g. [*Benn95:63f*], [*RoyceWa98:209ff*], [*Mayr05:81f*]). Modern Process Models (e.g. *RUP*, *V-Modell XT*) already encourage the tailoring of its elements (*roles, products & artefacts, activities, size and number of iterations, tools*) for a specific project (e.g. [*KrollKru03:61ff*] introduce the four different projects "*Deimos*", "*Ganymede*", "*Mars*" and "*Jupiter*" to demonstrate various RUP-configurations throughout their book). So "*there is no unique, perfect, and ready-to-use process model that can be adopted once and for all, in all organizations, for all kinds of products or product families*" [*GheJaz03:418*] and "*Process tailoring is best done in an iterative manner: tailor some, implement some, and then repeat.*" [*AmNa05:290*].

[*RoyceWa98:Chapter14*] brings a striking survey about the necessary consideration when tailoring the process (even when explained with RUP-vocabulary quite general applicable): He states "*two dimensions of discriminating factors: technical complexity and management complexity*" (see figure 2.13). Afterwards he distinguishes six relevant "*process parameters*" which affect the process tailoring: SCALE (*Size of the project, scale of the software application, size of the team): 5 people (small), 25 people (moderate size) 125 people (large) and +625 people (huge)*; STAKEHOLDER COHESION OR CONTENTION (*can range from cohesive to adversarial*); PROCESS & CONTRACT FLEXIBILITY OR RIGOR (*required coordination*); PROCESS MATURITY (*of the development organisation*); ARCHITECTURAL RISK (*degree of technical feasibility*) and DOMAIN EXPERIENCE (strongly influences the number of *prototype release iterations*).

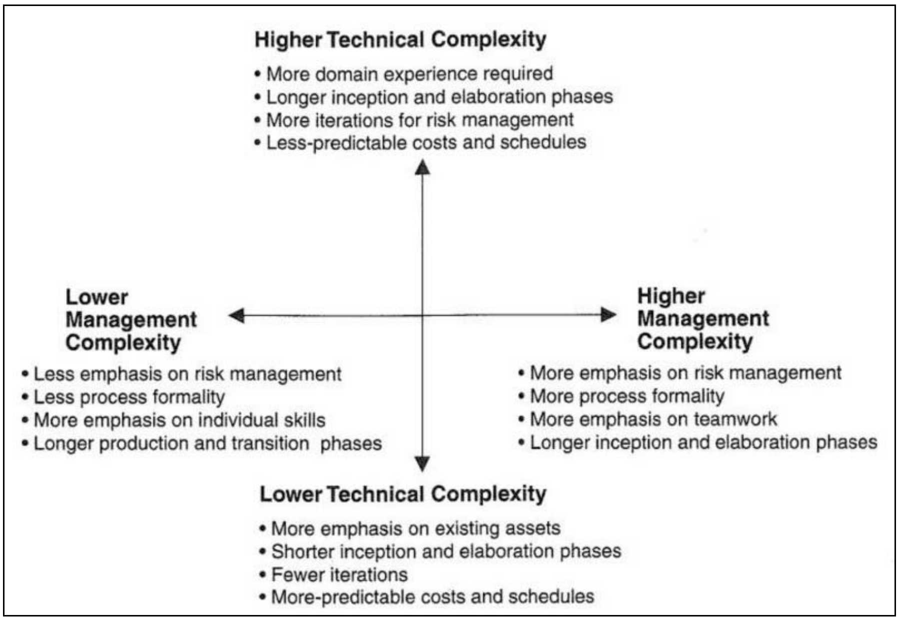

Higher Technical Complexity
• More domain experience required
• Longer inception and elaboration phases
• More iterations for risk management
• Less-predictable costs and schedules
Lower Management Complexity
• Less emphasis on risk management
• Less process formality
• More emphasis on individual skills
• Longer production and transition phases
Higher Management Complexity
• More emphasis on risk management
• More process formality
• More emphasis on teamwork
• Longer inception and elaboration phases
Lower Technical Complexity
• More emphasis on existing assets
• Shorter inception and elaboration phases
• Fewer iterations
• More-predictable costs and schedules

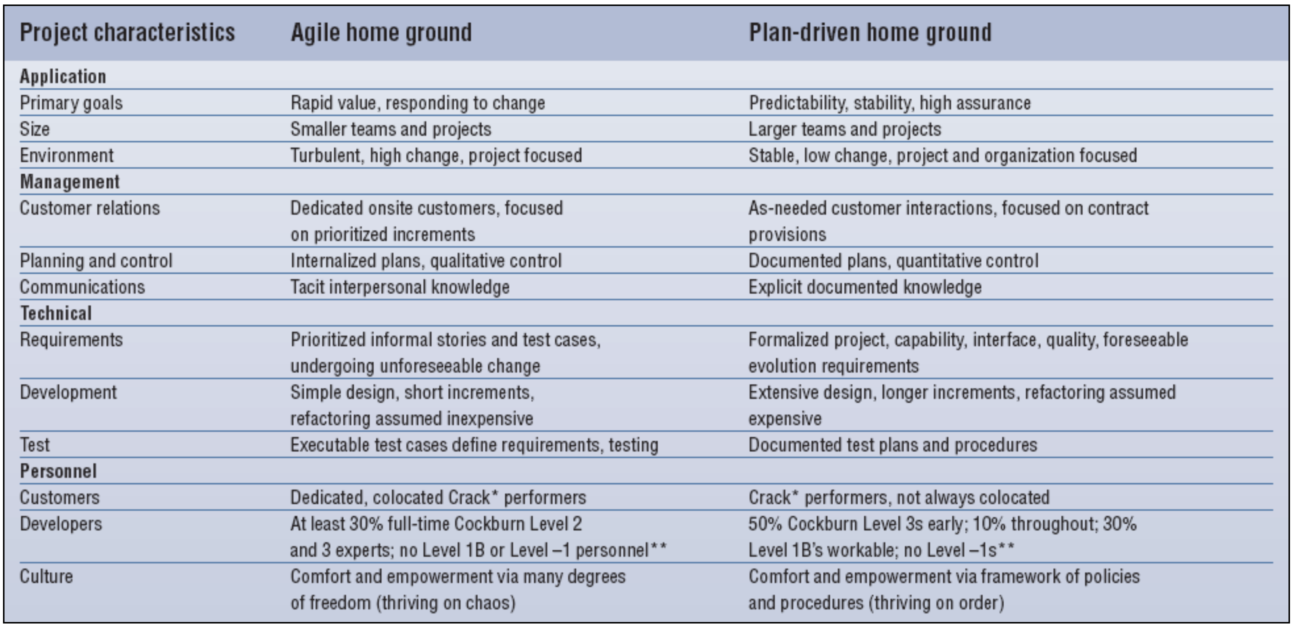
Figure 2.13: Two dimensions for tailoring the software development process [*RoyceWa98:211*]

2.3. Software Development: Two significant Process Models

> "*Greece vs. Rome – Two Very Different Software Cultures: Greeks would fit pretty well into the Agile camp, Romans would be working mightily to improve their CMM level, and Barbarians would say "huh?" if you mentioned either one!*" [*Glass06:111f*]

There is an extensive repertoire of various Process Models in Software Engineering and only a representative selection of significant models was shown in subchapter 2.2.6 (A more practical comparisons of "*Modern Software Processes*" is given in [*AmNa05:286f*]). So why is this chapter focusing on a detailed presentation of *Unified Process (UP)* and *eXtreme Programming (XP)*? Not only because they are "state-of-the-art" in the software industry [*EssMey03:2*] and were both used in project SIE (see chapter 1). More important is the fact, that both models reflect very well the contradictions between plan-driven and agile approaches (a current debate in SE and somehow a continuation of the well-known "art" versus "engineering" discussion described in chapter 2.1). Furthermore UP and XP are both practical and matured examples of their respective methodology, even if XP is currently quite "hype" and UP is already a "compromise" (as it combines the best ideas of many previous models in a customisable way and is often wrongly related to bureaucratic QM-approaches like CMM, e.g. [*Mayr05:78*] calls the RUP absurdly a "*Process monster*"). Therefore both models are appropriate to show the current range of underlying SE-concepts in which Requirements Management is embedded

In 2002 Boehm advocated the "*best balance of agile and plan-driven methods*" but also stated that "*each have strengths and weaknesses*" [*Boehm02:65ff*]. Therefore he compared them for certain criteria and constituted the idea of "*home ground(s)*" for both (similar to the Royce-concept in figure 2.13). In 2003 this concept was refined and now provides a comprehensive set of characteristics for both methods, where also requirements and the environment play an important role (see figure 2.14): "*The home grounds for the agile and plan-driven methods encompass the sets of conditions under which they are most likely to succeed.*" [*BoehTur03:58*].



Project characteristics	Agile home ground	Plan-driven home ground
Application		
Primary goals	Rapid value, responding to change	Predictability, stability, high assurance
Size	Smaller teams and projects	Larger teams and projects
Environment	Turbulent, high change, project focused	Stable, low change, project and organization focused
Management		
Customer relations	Dedicated onsite customers, focused on prioritized increments	As-needed customer interactions, focused on contract provisions
Planning and control	Internalized plans, qualitative control	Documented plans, quantitative control
Communications	Tacit interpersonal knowledge	Explicit documented knowledge
Technical		
Requirements	Prioritized informal stories and test cases, undergoing unforeseeable change	Formalized project, capability, interface, quality, foreseeable evolution requirements
Development	Simple design, short increments, refactoring assumed inexpensive	Extensive design, longer increments, refactoring assumed expensive
Test	Executable test cases define requirements, testing	Documented test plans and procedures
Personnel		
Customers	Dedicated, colocated Crack* performers	Crack* performers, not always colocated
Developers	At least 30% full-time Cockburn Level 2 and 3 experts; no Level 1B or Level –1 personnel**	50% Cockburn Level 3s early; 10% throughout; 30% Level 1B's workable; no Level –1s**
Culture	Comfort and empowerment via many degrees of freedom (thriving on chaos)	Comfort and empowerment via framework of policies and procedures (thriving on order)

Figure 2.14: Agile and plan-driven home grounds [*BoehTur03:58*]
(** Collaborative, Representative, Authorized, Committed, Knowledgeable; ** Cockburn's Levels of Software Understanding*)

2.3.1. Plan-driven: Iterative, use-case-object-oriented Unified Process (UP, RUP)

> "*Consider the RUP as a smorgasbord of best practices. Rather than eat everything, eat your favourite dishes, the ones that make sense for your specific project*" [*KrollKru03:35*]

The *Unified Software Development Process (UP)* is the result of a continuous evolution in the OO-community in the 1990's [*ZuserGre04:79f*]. In 1987 Ivar Jacobson founded the firm *Objectory* and invented a development-approach that introduced use-cases ("*The Objectory process defines the core from which the RUP … later evolve*" [*AmNa05:8*]). In the mid-1990's three leading pioneers of object-oriented design (Jacobson, Booch, Rumbaugh – also called the "*Three Amigos*", later joined by Kruchten) partnered up under the roof of *Rational Software* company and unified their previous work to constitute the *Unified Process (UP)* and also the *Unified Modeling Language (UML)*. In 1998 the (commercial) RUP 5.0 came out (a continuation of the *Rational Objectory Process ROP 4.0*) and the (free) Unified Process as generic framework was published in 1999 [*JacoBo99*]. Since then the *Rational Unified Process (RUP)* was enhanced as a commercial product by Rational (since 2002 part of IBM), and now includes the (improved) process itself and supporting web-based tools for an easy customisation of (e.g. own views for the different roles). RUP ® is now part of IBM's *Rational Method Composer* (currently RMC V 7.2).

(R)UP is *iterative and incremental* – hence has IID-advantages stated in section 2.2.6 (2) – balances rework-risks with an *architecture-centric approach* and combines *UML-(component)-based object-oriented design*, *use-case-driven concepts*, *risk-addressing*, and *regular reviews*. Basically the RUP ® has two dimensions (figure 2.15):

- 9 DISCIPLINES ("*Content-Axis*", UP has only five): 6 *technical* and 3 *supporting* disciplines are passed iteratively (with varying intensity) during the whole SWDLC (unlike the linear WF).
- 4 PHASES ("*Time-Axis*"): "*four major stages … (the) project goes through over time*" [*AmNa05:15*]. Each phase is split in several "controlled" iterations and ends with a review and a milestone.

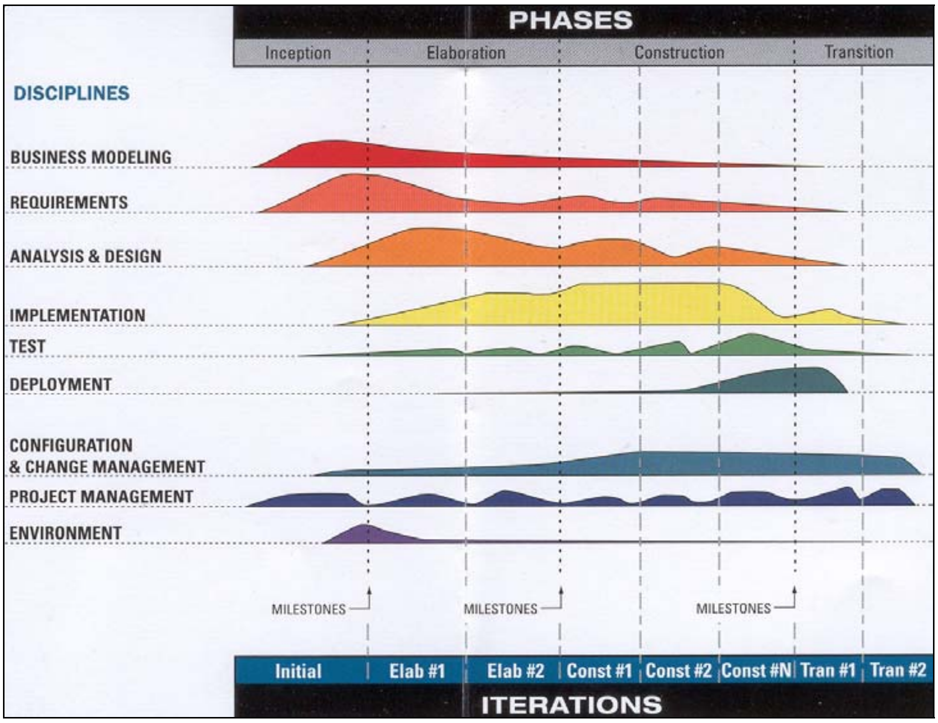


Figure 2.15: RUP ® development cycle: Disciplines, Phases and Iterations [*Kruchten04:Poster*]

Readable introductions for all the details of the RUP ® are [*Versteeg00*], [*KrollKru03*], [*EssMey03*], [*Kruchten04*], [*AmNa05*]. Only the most important elements of the RUP ® can be introduced here:

● PRINCIPLES: "*Best practices*" of the RUP are introduced by [*Kruchten04:5ff*] (e.g. *Develop iteratively, Manage requirements, Use component-based architecture, Visually Modeling, Continuously verify software quality, Control Change*) and enlarged by [*AmNa05:22ff*]. Other sources present more general UP-principles:

- "*any large software project should be broken into controlled iterations (miniprojects) that provide increments of the product… Increments can be additive or perfective*" [*GheJaz03:444*]. Iterations mostly go through all technical disciplines, establishing a serial cycle of *requirements, analysis, design, implementation, testing, and evaluation.* This combines the iterative and incremental approach as described in 2.2.6 (2), avoids the WF-*late-design-breakage*, allows early risk detection and "*lets you take into account changing requirements*" [*Kruchten04:23*].
- UML is an important factor in the RUP: "*the rich collection of languages that constitute UML provides specific notations to specify, analyze, visualize, construct, and document the artefacts that are developed in the life cycle of a software system.*"; "*use cases are employed as the primary means of communication with stakeholders in the requirements workflow … the main input to the analysis, design, implementation, and testing workflows"; "software architecture is the primary artefact used for conceptualizing, construction, managing, and evolving the system being developed*" [*GheJaz03:444ff*]
- Milestones and Reviews at the end of each iteration (or at least each phase) encourage control, feedback and decision-making: "*A milestone consists of delivering an intermediate set of artefacts that can be subject to quality control via reviews and inspections*" [*ibid.*]. So [*GheJaz03:446*] reason: "*UP achieves the goals of flexibility and incrementality without retreating to unstructured development practices … fine-grained iteration steps help in continuous validation and ensure that early changes in the process may be performed to redirect the development if required*".

● (R)UP-PHASES are quite different to WF-phases. Each UP-phase performs with varying intensity all disciplines and ends with a major milestone: INCEPTION (*Define project scope, estimate cost and schedule, define risks, develop business case, prepare project environment*), ELABORATION (*Specify requirements in detail, identify and validate architecture, evolve project environment, staff project team*), CONSTRUCTION (*Model, build and test system; develop documentation*) and TRANSITION (*System testing, user testing, rework, system deployment*). The appropriate iteration-number for each phase depends on the size and complexity of the project.

● 4 ELEMENTS combine disciplines and phases to form a unique process [*Kruchten04:35ff*]. Tailoring to project-needs is advisable ([*KrollKru03:61ff*], "*process must be made as lean as possible*" [*Kruchten04:30*]):

- Up to 30 ROLES (grouped in 5 categories: *Analyst, Developer, Manager, Tester, Production and Support, Additional*) define WHO is responsible for certain activities and artefacts. The mapping is done by the SWPM, multiple mappings are possible [*Kruchten04:273ff*].
- ACTIVITIES define HOW roles perform their work ("*a unit of work that an individual in that role may be asked to perform and that produces a meaningful result*" [*Kruchten04:38*]).
- ARTEFACTS define WHAT is produced ("*tangible products*", e.g. *models & model elements, documents, source code, executables*). They are organised in nine *artefact sets*, corresponding to the nine disciplines [*Kruchten04:277ff*] and should be traced with a version-control.
- WORKFLOWS (mostly related to one of the nine disciplines) describe "*meaningful sequences of activities … that produces a result of observable value*" [*Kruchten04:45*]

2.3.2. The rise of Agile Software Development: eXtreme Programming (XP)

> "*Individuals and interactions over processes and tools*
> *Working software over comprehensive documentation*
> *Customer collaboration over contract negotiation*
> *Responding to change over following a plan*
>
> *... while there is value in the items on the right,*
> *we value the items on the left more.*"
>
> Agile Manifesto 2001 [*www.agilemanifesto.org*]

> "*If you want to start*
> *a religious or software war,*
> *issue an edict or manifest*"
>
> Ken Orr 2002 [*Mayr05:96*]

• The Agile Movement and its foreseeable balance with the "traditional" models

Considering the Hegelian SE-history introduced by Boehm (see 2.2.3), agile methodologies are definitely an "*Antithesis*": The agile movement is a COUNTERMOVEMENT officially oriented against "*conventional*", "*document-driven*", "*heavyweight*", "*rigid*" or "*bureaucratic*" software development, as which many programmers sensed the rise of very formal processes in the 1990's (but forgot that QM-Models like *CMM(I)* and Process Models like the *V-Modell* where initiated by the military – so they reflected very rigid clients needs as described in 2.2.4 and needed *tailoring* in practice). The agile movement is also loosely linked to two other "hot" SE-topics: *FOSS – Free and Open-Source Software* (*code sharing* and other cooperative "agile" practices are close to well-known FOSS-approaches, see [*Raymond99*], [*Cox00*], [*GheJaz03:431*], [*Ebert07*]) and "*outsourcing-fears"* about process-improvement-models as SPICE or CMM(I) making (US-)programmers as "*suppliers*" more easily exchangeable (e.g. Cem Kaner argues with this against SWEBOK and outsourcing was already addressed in 1992 by Yourdon [*Your92*]).

Since mid-1990 (while RUP was still in its infancy) some programmers advocated to overcome the disadvantages of traditional development methods by pushing them away (instead of improving) and proposed "lighter" alternatives. So 17 of them signed in February 2001 the "*Agile Manifesto*" (replacing the previous term "*lightweight*" by "*agile*") and defined agile principles (see quotation above and [*Beck00*], [*Cockb01*], [*MarcSuc03*]). The *agile software development movement* started quite dogmatic and provocative ("*The future of our Information Age economy belongs to the agile*" [*MarcSuc03:9*]). So some sources diagnose a "*religious eagerness*" [*Sommer06:431*] or even a "*religious war*" [*Mayr05:78*] and there is already a countermovement against the "agile dogma" (e.g. [*StepRose03*]), examining the thin line between "agility" and "fragility". It is often criticised, that some agile proponents ignore the right side of the Agile Manifesto, using it as a genial excuse for "*cowboy coding*" and documentation-avoidance [*Mayr05:96*]. But Mayr also concedes that overcoming process-"dinosaurs" would have been impossible without some provocation. Meanwhile "*agile vs. plan-driven*" increasingly becomes a false contradiction, as "traditional" models started to react and integrate useful "agile" practices (e.g. *Agile Unified Process* and agile plug-ins for *V-Modell XT*). So there are "*home grounds*" for both (see figure 2.14), and some already forecast the future "balance" between agile and traditional models ([*Boehm02*], [*Mayr05:78*], [*BoehTur05*]). Also, strictly speaking, agile models are not as new as they claim to be, as evolutionary development models (*iterative, incremental or RAD*) are well-known for decades (see 2.2.6). But the agile movement definitely refined them and pushed them to their limits (with very short iterations and small increments, strictly bottom-up development and as little planning as possible).

● *eXtreme Programming (XP)*

"*How little can we do and still build great software?*" (Kent Beck [*MarcSuc03:Inner Cover*])

Currently *eXtreme Programming (XP)* is the best-known "agile" software development method. It was created by Kent Beck (and colleagues) during the Chrysler-C3-project (payroll systems unification) that he rebooted in 1996. While Beck and XP became famous (see "*XP Series*", [*Beck00*], [*MarcSuc03*]), the C3-project using XP got into troubles (e.g. the customer representative quit due to a burn-out) and was shut down in February 2000 (ironical, but the huge C3-project was surely the wrong domain as "*XP is not appropriate for every software project … bigger projects have problems when applying it*" [*ZuserGre04:102*]). XP fits for small teams (and projects) with self-disciplined and self-organised programmers and a competent client who is willing to be strongly involved in the team (see figure 2.14). Under these terms one can quickly react to changing requirements (in this case Waterfall & Co. can be called the "*Artillery*" and XP a "*guided missile*" that perfectly hits the "*moving target*" [*MarcSuc03:59*]). XP is more a set of practices than a process, still evolving and – due to its agility – not as well defined as traditional models ("*For some people Extreme Programming (XP) is a new set of rules, for others it is a humanistic set of values, and to some it is a very dangerous oversimplification*" [*Don Wells, MarcSuc03:5*]). So its most important elements are:

● 5 VALUES: Communication (*collaboration and frequent verbal exchange*), Simplicity (*extras can be added later*), Feedback (*from the system via unit tests, from the client via acceptance tests, from the team to the client via user story estimation*), Courage (*e.g. to refactor or throw code away*), Respect (*in the team*).

● 4 BASIC ACTIVITIES and RELATED PRACTICES [*www.extremeprogramming.org*] (see figure 2.16):

- Planning & Listening: *User stories, Planning game, Release planning for schedule, frequent small releases, iteration planning starts each iteration, move people around, stand-up meeting starts each day*;
- Coding: *customer always available, code Unit Tests first, pair-programme all code, integrate often, collective code ownership and coding standard, optimisation at last, 40 hours per week, no overtime*;
- Designing: *Simplicity, Class-Responsibilities-Collaboration (CRC) cards for sessions, reduce risk with spike (prototype) solutions, no unnecessary functionality, refactor whenever and wherever possible*;
- Testing: *all code must have and pass unit tests before it can be released, frequent acceptance tests.*

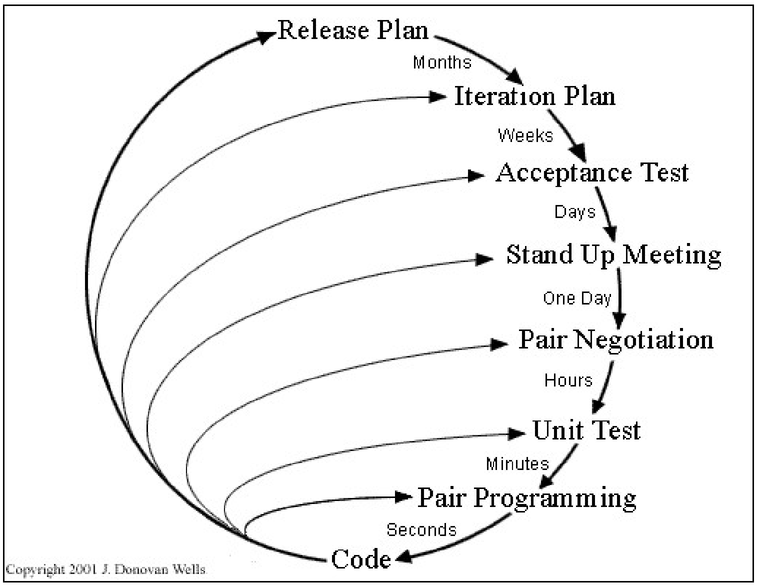


Figure 2.16: XP – Planning & feedback loops with time dimensions [*www.extremeprogramming.org*]

● *Agile vs. rigid in Barry Boehm's "land of metaphor": The monkey and the elephant*

As chapter 2.3 already began with a metaphor ("*Greece vs. Rome*" [*Glass06:111ff*]), it is just fair to end also with a notable metaphor that anticipates the future "*balance of agile and plan-driven methods*" [*Boehm02:69*]:

> *Once upon a time, in the land of Metaphor, there lived a monkey and an elephant. They both lived on one side of a wide, swiftly flowing river. On both sides of the river there were many fruit trees. The monkey was very agile. He could climb to the top of the fruit trees and eat as much fruit as he needed. The elephant was very tall. He could reach up with his trunk and eat as much fruit as he needed.*
>
> *But the trees grew taller. Soon the elephant could not reach enough fruit to eat. But he was strong and self-sufficient. He found that when he was hungry, he could just pull down a tree and have fruit to eat.*
>
> *The monkey watched the elephant pull down most of the fruit trees. He was not happy. He said to the elephant, "Don't do that! I will climb up the trees and get fruit for you."*
>
> *The elephant said, "I am hungry. I am strong. I can do things for myself."*
>
> *The monkey said, "You dumb elephant. Soon there will be no trees or fruit for you either."*
>
> *The elephant said, "I only work on one problem at a time. Things may change. Maybe more fruit trees will come. Or maybe the trees will get shorter. If they don't, I'll work out a solution then."*
>
> *The monkey had to agree. He only worked on one problem at a time too.*
>
> *Soon there were no more fruit trees left on their side of the river.*
>
> *The elephant said, "I have a solution. I will go across the river and pull down trees over there."*
>
> *The monkey said, "You dumb elephant. That didn't work on this side of the river, and it won't work on the other side of the river. You will starve us both. Let's duke it out. I am agile and will run rings around you."*
>
> *The elephant said, "That is fine with me. I am big and strong and I will pulverize you."*
>
> *So they began to duke it out. The monkey ran rings around the elephant. But he was not able to stop the elephant from trying to pulverize him. The elephant thrashed around with his strong trunk and legs, trying to pulverize the monkey. But the monkey was too agile, and the elephant missed every time.*
>
> *It was a hot day. Soon the monkey and the elephant got tired. They sat down and tried to figure out what to do next.*
>
> *The monkey said, "I am agile. I will just scamper across the river and bring back some fruit."*
>
> *He got a running start toward the river and went scamper, scamper, scamper … splott! The river started to carry him away. "That dumb monkey!" said the elephant. He waded into the river, picked up the monkey, put him on his back, and waded back to shore. They sat and thought for a long time while the monkey dried out.*
>
> *Finally the monkey said, "Maybe this is a solution. You can carry me across the river on your back. When we get across, I can climb the trees and get enough food for us both." The elephant thought for a moment and answered, "That sounds like it is worth a try." So they tried it, and it worked. And they lived happily ever after, until the end of their days.*
>
> *Some morals to the story:*
>
> - *Monkeys can do things elephants can't do.*
> - *Elephants can do things monkeys can't do.*
> - *Working on one problem at a time may not be a good idea.*
> - *Duking it out may not be a good idea.*
> - *Finding a collaborative win-win solution may be a good idea.*
>
> —*Barry Boehm* [*Boehm03:45*]

2.4. Beyond Development: Maintenance, Reengineering, Evolution and Retirement

> "*As we learned during the Year 2000 (Y2K) crisis, many systems remain in production decades after they were originally implemented*" [*AmNa05:33*]

It was already stated in subchapter 2.2.5, that there is an important difference between the *Software Life Cycle (SWLC)* and the *Software Development Life Cycle (SWDLC*), as there is still much to do after a software product has been successfully developed, delivered, installed and has passed the final acceptance test. The development project may be finished, but the software itself will be in use for a long time and has to be MAINTAINED, REENGINEERED or EVOLVED until it can be replaced and enters the end of its life cycle – the RETIREMENT. All this is strongly influenced by the original development and the achieved *Maintainability, Repairability, Portability* or *Evolvability* (see 2.2.1 and [*GheJaz03:23ff*]), but many SE-textbooks focus on the development and neglect the time after. Therefore the wording "*Once the system passes all the tests, it is delivered to the costumer and enters the maintenance phase*" [*GheJaz03:418*] sounds a bit like "*... and they lived happily ever after*" but doesn't fully reflect reality.

There are quite different definitions and classifications about the naming of these activities: Schach divides this "*maintenance phase*" in three fields (*corrective, perfective and adaptive maintenance*, see 2.2.5 and [*Schach96:462*]). Even if [*GheJaz03:23*] already questions the term "maintenance" (and slightly proposes "*software evolution*"), it adopts the Schach-concept and ads "*preventive maintenance*" (planned improvements to prevent future maintenance) [*GheJaz03:421*]. All four maintenance-terms were already defined in an IEEE-standard [*IEEE90:46*]. The amalgamation of maintenance and evolution ("*maintenance is evolutionary developments ... maintenance is continued development*") can also be found in [*SWEBOK04:6-3*].

In contrast the EUP defines for these activities the "*Production Phase*" to which it assigns the tasks: *operate systems, support systems, manage change requests, monitor systems, prepare for and recover from disasters* [*AmNa05:32*]. Furthermore the "*Enterprise Unified Process (EUP)*" [*AmNa05*] is the most comprehensive concept, covering the FULL SWLC and including also business aspects that drive software evolution (see 2.4.3: Unification of business strategy and IT-strategy). The EUP extends the RUP ® with:

- Two new phases (*Production, Retirement*) and a 4th support discipline (*Operations & Support*)
- Seven additional "*enterprise disciplines*" which also concern the four earlier RUP-phases: *Enterprise Business Modeling, Portfolio Management, Strategic Reuse, Enterprise Architecture, People Management, Enterprise Administration, Software Process Improvement (SPI).*

All these "*beyond-development*"-concepts are often associated with well known "*RE-activities*" in Software Engineering like: *Reuse, Refactoring, Reengineering (incl. Reverse Engineering and Redocumentation), Redesign, Restructuring, etc.* In this context a source of MISUNDERSTANDING between engineers and managers is the term "*Reengineering*": While managers associate this term with *Business Process Reengineering (BPR)* and therefore with a "*radical redesign of business processes*" [*HamCha94:32*], software engineers will think about *Software Reengineering* which is defined as "*the examination and alteration of a subject system to reconstitute it in a new form*" [*ChiCo90:15*] (a pure technical "renovation" without any functional change). So what the manager calls "*Reengineering*" in the business context, the engineer in his SE-context calls "*Software evolution*". As Information Technology (IT) plays an important role in BPR [*Thorp98:55*], also some software-related books mix both concepts and cause misunderstandings, e.g. [*YangWa03:2f*].

So to avoid misunderstandings, this thesis emphasises the careful distinction between four different "*beyond-development*"-activities, using concepts from several sources ([*ChiCo90:15*], [*Tomic94:29ff*], [*ZuserGre04:Chapter13*], [*AmNa05*], [*Sommer06:Chapter21*]):

- Maintenance of the original software (service, support, "*repair*" of faults, see 2.4.1)
- Software Reengineering (for technological change or to restore *maintainability*, see 2.4.2)
- Software Evolution (due to – mostly business driven – changed requirements, see 2.4.3)
- Complete Retirement (including migration to a new system, see 2.4.4)

It is self-evident, that in practice, these activities cannot be distinguished as sharp as in theory, and there is a smooth transitions or even overlapping between some of them. The question is: How long does the software continue to be the "original" system and when does it become a reengineered or evolved system? (e.g.: A new GPS navigation system doesn't change an entire car, but a new motor may do).

Also the decisions about the future of legacy software (maintain, evolve or replace) are not always fully clear for the responsible engineers and managers: As stated above, they first depend on the original system and HOW foresighted it was developed (*agile* products could face future critics about this). Another criterion are the expected (future) maintenance costs [*Sneed: PoloPiat03:201ff*], [*Sommer06:534ff*]. Considering the famous "*Laws of Software Evolution*" (Lehman et. al.), Sommerville proposes a two-dimensional *decision matrix* (*system quality; business value of the system*) that classifies the various systems and helps to decide between (partial) replacement and evolution [*Sommer06:545ff*]. Certainly these activities imply effort and many textbooks declare (with percentages from 55% to 90% [*PoloPiat03:vii*]): "*Software maintenance has become the most costly stage of the software life cycle*" [*YangWa03:1*]. But at the same time Reengineering and evolution are a key factor for OPTIMISATION: Of the software itself in various dimensions (maintenance costs, technical aspects like the architecture or the data model) and (often more important) for the optimisation and improvement of related business processes and workflows.

2.4.1. Maintenance of the original software system

Software doesn't "*ware out*", so its maintenance (in terms of *repairs, services and support*) is not comparable to the maintenance of other technical systems. It is defined as the "*process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment*" [*IEEE90:46*]. This standard already defined the four "classical" categories, adopted by many textbooks, e.g. [*Schach96:462ff*], [*ZuserGre04:396f*], [*GheJaz03:23f;420f*] and outspoken challenged by [*Sommer06:534f*]:

(1) Corrective maintenance: "*to correct (residual) faults*" (original faults or "*regression fault and undesirable behaviours*" due to modification [*YangWa03:19*]);

(2) Adaptive maintenance: make software "*usable in a changed (external) environment*" (e.g. new hardware, operation system or databases – also after "disasters");

(3) Perfective maintenance: "*improve qualities like performance, maintainability, or other attributes*"(e.g. Code-*Refactoring* & *Restructuring*, maybe also additional functions);

(4) Preventive maintenance: *Anticipate future (maintenance) problems, ease maintainability.*

In fact, many of these tasks – when not limited to the original system and functions – may already constitute a new (development) project and lead over to SW-Reengineering and evolution, often also a contractual question: Which services and "*fault reports*" are still comprised by the maintenance contract?

2.4.2. Legacy Software, technological change and Software Reengineering

Legacy software is "*software that already exists in an organization and usually embodies much of the organization's processes and knowledge*" [*GheJaz03:24*]. This normally means "old" software ("old" *languages, architectures, technologies*), related to essential business processes and therefore with a high "value", also because a lot of money was invested. It is often "*mission critical*" and linked to other systems, so it cannot be replaced easily or is considered "irreplaceable" by some users: "*Economic and technical constraints make it impossible in most cases to discard the existing and legacy systems and develop replacement systems from scratch … legacy system migration strategies are often preferred to replacement*" [*PoloPiat03:151*]. And "old" may be ambiguous, as "*new software becomes legacy software quickly*" [*YangWa03:1*]. A readable survey offers [*Sommer06:65ff*].

Therefore the main idea of *Software Reengineering* (also known as "*renovation*") is "*the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form*" [*ChiCo90:15*] (see also [*Byrne92*], [*Hall92*]). So just the technical realisation of the system is altered, optimised or modernised (often supported by CASE-tools), but its functions remain the same (Within this thesis, new functions already belong to *Software Evolution*). To avoid a "big-bang"-effect, SW-Reengineering can happen partially or incrementally, implying prioritisation [*Your92:253f*]. Today there is a broad spectrum of Software Reengineering possibilities (*code-restructuring, modularisation, program- and architecture restructuring*) [*Sommer06:542ff*], but the basic concept (see figure 2.17) is still a sequence of:

- REVERSE ENGINEERING: Analyse the (unknown) structure of the legacy system (*requirements, functions, main components & relationships, algorithms, data structure*) and create better representations or documentation (*Design Recovery*, *Redocumentation*). At worst this could be illegal [*Lee in Hall92:559*], but a "*lack of documentation and … poor development … are the main factors affecting the need for – and the cost of – reverse engineering*" [*GheJaz03:420*].
- Followed by FORWARD ENGINEERING: Classical development cycle, may include improvement or optimisation as well as Restructuring on the same hierarchical level.

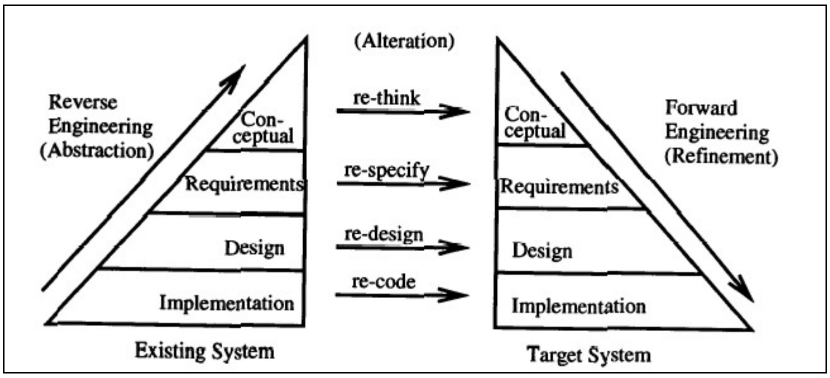


Figure 2.17: General model for Software Re-engineering [*Byrne92:230*]

Main Reasons: (1) MAINTENANCE PROBLEMS: Often "*the source code is the only documentation available*" or the available documentation is "*useless*" [*Schach96:40*], which can also cause more *regression faults*. "*Maintenance is [also] hindered by previous poorly performed maintenance interventions*" [*GheJaz03:420*] which gets worse with every maintenance. (2) CHANGED (technical) ENVIRONMENT (hard- & software). (3) OPTIMISATION POSSIBILITIES in the initial system but also with new technologies due to the pace of technological change (e.g. new hardware- and software possibilities) [*ZuserGre04:410*]. (2) and (3) depend, if the initial developer(s) already anticipated upcoming technological trends [*Mayr05:41*].

2.4.3. (Business-driven) Software Evolution

> "*To succeed, you need to look beyond IT and consider the larger picture – that of the entire business process … a business system isn't successful unless the business is*" [*AmNa05:116*]

As soon as the original requirements of the legacy software are questioned in general – due to changed needs of users, departments or the entire company – maintenance and Software Reengineering (technical "*renovation*") reach their limits. In this case, when related workflows and business processes are the primary driver(s), the company could develop a completely new system ("*from scratch*") and retire the legacy system (see 2.4.4). Or – due to the reasons mentioned in 2.4.2 – one may decide to renew and evolve the valuable legacy system, combined with Software Reengineering (or at least Reverse Engineering) to transfer the precious elements of the "old" system into a new one ("*System evolution is so common that a development from scratch is the exception*" [*YangWa03:8*]). For extensive and modular legacy systems a partial evolution and partial retirement is possible [*Your92:253f*]. Often "*a balance must be struck between the constraints imposed by the existing legacy systems and the opportunities offered by [BPR]*" [*PoloPiat03:151*].

Whether the original authors often used the term Reengineering for both, *Software Reengineering* and *Software Evolution*, the basic idea of (business driven) *Software Evolution* is not as new as some "consultant-driven" sources try to suggest (see figure 2.18, [*Your92:235ff*], [*Brown92*]). But those ideas increasingly attracted interest, as they were linked to the upcoming and far-reaching concepts of *Business Process Management (BPM)* and *Business Process Reengineering (BPR)* ("*fundamental rethinking and radical redesign of business processes to achieve dramatic improvements in critical contemporary measures of performance, such as cost, quality, service, and speed*" [*HamCha94:32*]). Even when Hammer later softened his "radical" approaches: After Harvard-related consultants declared "*Information Technology as an Enabler of Process Innovation*" [*Daven97:37ff*], the top-management spotlighted these suddenly popular issues and often created an own *Chief Information Officer (CIO).* This gave software (departments) much more positive attention, but also spotlighted problems in this domain ("*frequent reality that we cannot demonstrate a connection between money spent on IT and business results*" [*Thorp98:xix*]) – leading to the fact, that *IT-outsourcing* has become a key issue in practice as well as in the related literature (e.g. [*ApplAus03:561ff*], [*Boehm06:22*]). Also *Component Based Software Engineering (CBSE)* and *Commercial Off-The-Shelf-products (COTS)* are up and coming.

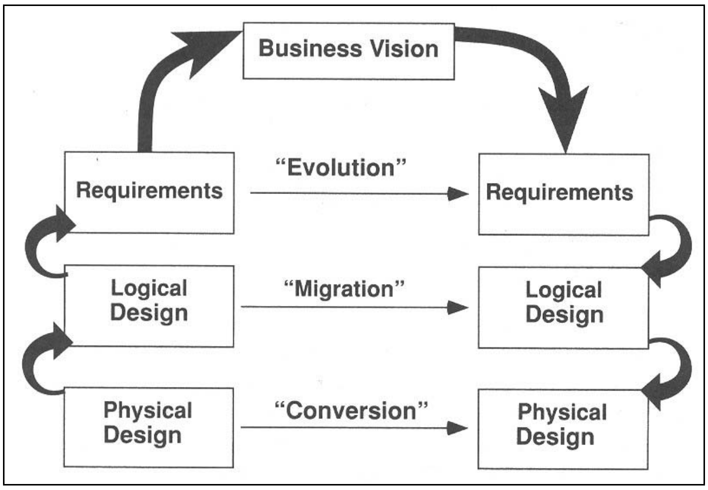


Figure 2.18: Business-driven Software Evolution [*Your92:254*]

Since the 1990's, the potential for optimisation and modernisation with software evolution caused an ongoing wave of concepts, often related to *Information Systems (IS)* and *Business Intelligence (BI)*, like: *Information Management (IM), Knowledge Management (KM), Data Warehousing and Data Mining, Enterprise Application Integration (EAI), Enterprise Resource Planning systems (ERPs)* like *SAP, Workflow Management Systems, Costumer Relationship Management (CRM)* and with some time delay also *E-Government*. Meanwhile even *IT-Governance* has been covered by Corporate Governance, but currently there is no clear taxonomy about all these somehow related concepts. A promising approach for a rough classification is presented by Thorp who distinguishes "*Three stages of IT evolution*" (*Automation of work, Information Management, Business Transformation*) [*Thorp98:14*]. And one has to consider that also "*the misuse of technology can block reengineering altogether by reinforcing old ways of thinking and old behaviour patterns*" [*HamCha94:83*]

Therefore there are currently movements toward an "*integrated strategy*" (combination of enterprise strategy and IT-strategy) like the *BTOPP-system* (*business, technology, organisation, process, people* [*Thorp98:72*]) and many sources examine "*the Impact of IT on Strategic Decision Making*" [*ApplAus03:34ff*]. Even leading management consulting firms have already established specialised branches for IT-related issues. But as IT-optimisation is therefore increasingly associated with outsourcing and cost-cutting, a project for Software Evolution implies the increasing risk of frictions and resistance among some stakeholders – an effect which is not totally new and was already described by Yourdon [*Your92:252f*].

An extensive concept for the practical combination of software development and evolution with these business issues provides the "*Enterprise Unified Process*", that looks "*beyond the needs of a single system*" and addresses "*cross-system issues that your IT organization must deal with on a daily basis*" [*AmNa05:34*] (see also introduction of 2.4 and [*Macias01:14f*]). A practical example for BPR with Software Evolution – a new architecture makes a system web-fit and enlarges the business processes – shows [*PoloPiat03:151ff*].

2.4.4. Retirement and migration

"*Software systems do not last forever*" [*AmNa05:32*]

As stated above, partly of fully retirement of software systems is the last phase of the *Software Life Cycle*. This is done, when the system is not needed any more or when "*further maintenance is not cost-effective*" [*Schach96:41*]. It is normally initiated by a strategic decision, that Software Reengineering or Software Evolution are not useful, or lead to a completely new system which replaces the existing legacy system and incorporates worthy elements of it (maybe supported by some reverse-engineering) [*Sommer06:545ff*]. Schach distinguishes therefore a "*true retirement*" (without replacement) which is very rare and done when the system is obsolete so "*the client organization no longer requires the functionality provided by the product*" [*Schach96:41*]. The system can also be fully replaced by a COTS-product (e.g. SAP). In case of a (partly or fully) replacement some projects "*treat the retirement of an existing system as a subproject of the initial development of the system replacing it … but this is just a management convenience; the fact remains that retirement is still part of the lifecycle of the original system*" [*AmNa05:33*].

Retirement is always a complex task and needs careful preparation. Important tasks are: Analyse legacy system interactions ("*legacy analysis*": current input- and output-data as well as other interactions with other systems); Determine retirement strategy (*"big-bang"-retirement, staged retirement or parallel systems*); Test new system, Migrate users and data (e.g. 95% automatically, rest manually) [*AmNa05:71ff*].

3. SOFTWARE PROJECT MANAGEMENT AND THE HUMAN FACTOR

> "*There is no cookbook for software management. There are no recipes for obvious good practices. I have tried to approach the issue with as much science, realism, and experience as possible, but management is largely a matter of judgement, (un)common sense, and situation-dependent decision making. That's why managers are paid big bucks.*" [*RoyceWa98:xxiv*]

As professional Software Development is normally performed by a team of developers within a defined project, *Software Project Management (SWPM)* is a key factor for the success of a Software Project. Even if this thesis strongly supports the engineering-approach in SE (see chapter 2.1), there is no contradiction to emphasise at the same time the importance of the "human factor" in SE (as underlined by authors like Tom DeMarco). Engineering means to conduct software projects on engineering principles and with careful planning (instead of *build-and-fix*), but it is mainly the responsibility of SWPM to take into account the human factor when creating and following this plan. Therefore the main ideas of SWPM – especially those concerning the influence of the "human" factor – will be presented in this chapter. After a survey of the basic elements of SWPM one significant influence of the human factor in software projects is picked out: *Environmental influences and risks* (including also client-side effects). All this is presented in consideration of this thesis focus: Human risks and influences in Requirements Management, which also affect the entire SWPM when they are not resolved early.

3.1. Software Project Management in a nutshell

> "*Successful Project Management needs more than the elaborate utilisation of PowerPoint and MS Project*"
> (Raoul Fortner, own saying)

Sommerville noticed, that *Software Project Management (SWPM)* is a too broad topic to be covered only by a single chapter and it is impossible to give a universal description of all necessary tasks [*Sommer06:125*]. This is true, as there are manifold kinds of software projects and various ways how to organise, plan and manage them. Size is an important factor, as it would be quite excessive (and maybe also harmful) to apply the full SWPM-repertoire within a small and simple project. But normally the problem in SE is quite contrary, and many projects have a pure technical focus, so systematic SWPM is often absent.

SWPM implies *administrative paperwork* (time plans, cost estimation and reporting) as well as *human-centred leadership* (teamwork, motivation, monitoring, problem- and conflict-solving). Also SWPM has *internal aspects* (team) and *external aspects* (Client, external stakeholders), and mostly the project manager (PM) has to be the "interface". The different SE-development methods (see 2.2.6) also influence SWPM: Some SWPM-books highlight plan-driven SE-models (e.g. RUP-oriented sources like [*RoyceWa98*], [*Versteeg00*]) and other ones support the upcoming *Agile Project Management (APM)* [*August05*]. So SWPM is complex and normally there are three different VIEWS on the project [*Mayr05:30ff*]: (1) *business-view* (Time, Costs and Quality – the "*Iron Triangle*"); (2) *technical view* (focused on techniques and delivered functionalities); (3) *sociological view* (team-spirit and other human factors). Often the PM's vita influences, which view is "privileged", but experienced PM's will bring them all together in a reasonable manner. Therefore only the most significant aspects of SWPM can be highlighted below.

3.1.1. Projects and Project Management (PM) in general

Significant and readable sources about projects and *Project Management (PM)* in general are [*Karnov02*], [*PatzRatt04*], [*Gareis04*], [*PMBOK04*]. Alternative views are presented in [*Fröhlich02*], [*Gärtner04*].

● *Projects*

A general and commonly used DEFINITION of a project can be found in the „*Guide to the Project Management Body of Knowledge (PMBOK)*" which is published by the *Project Management Institute (PMI)*:

> "*A project is a temporary endeavor undertaken to create a unique product or service. Temporary means that every project has a definite beginning and a definite end. Unique means that the product or service is different in some distinguishing way from all other products or services.*" [*PMBOK04:4*]

So projects have CHARACTERISTICS which distinguish them from routine tasks [*PatzRatt04:18f*]:

- The project is *unique* and *temporary* (a one-time job with a definite end);
- Projects are *goal- and problem-directed* (which may include also "hidden" problems), the goal(s) should be observable, measurable and achievable (challenging but realistic). If the goal(s) change(s) during the project, it can become "moving target"-problem;
- *Time, budget* and *human resources* are *limited* (the combined restrictions of time, budget and goal(s) is often called the "*iron triangle*", sometimes goal(s) are substituted by quality);
- The problem can be *new*, *complex* and *dynamically linked* to many *different disciplines*. The goal(s) can only be achieved by the *multidisciplinary cooperation diverse of skills*;
- Projects are embedded into a *specific environment* in which they have a high relevance (especially *stakeholders* who have expectations and/or fears toward the project).
- Because of its "uniqueness" projects have *particular risks*, especially concerning *time planning, cost estimation and applied technologies*. Minimising those risks is important for a successful project management [*ZuserGre04:40*].

Projects can be CLASSIFIED according to one or several of the following attributes [*Mayr05:28f*]:

Size (people, budget, tasks, etc.), *Duration, Client-status* (internal or external project), *Complexity*, (Problem) *Domain*, *Difficulties, Specifics, Relevance and impact, Risk, Costs, Intensity* (full-time or concurrent), *Number of subprojects* (in case of multi-project management).

● *Project Management (PM)*

Project Management (PM) is a particular form of *Management* (directing and controlling a group to achieve a goal) and is therefore commonly defined as follows (see also figure 3.1) [*PMBOK04:6*]:

> "*Project management is the application of knowledge, skills, tools, and techniques to project activities to meet project requirements. Project management is accomplished through the use of the processes such as: initiating, planning, executing, controlling, and closing. The project team manages the work of the projects, and the work typically involves:*
>
> - *Competing demands for: scope, time, cost, risk, and quality.*
> - *Stakeholders with differing needs and expectations.*
> - *Identified requirements*"

Typically projects and the related Project Management are divided "*into several project phases to improve management control*" (also known as the "*project life cycle*") [*PMBOK04:11*]. While Karnovsky defines only three of them [*Karnov02:18*], Patzak and Rattay define at least four (with any number of "*execution-phases*"). The PMBOK avoids a clear limitation, but presents in its PM-definition five phases (*initiating, planning, executing, controlling, closing*) and shows "*representative project life cycles*" [*PMBOK04:13ff*].

The activities during the planning-phase and the execution-control-phase (for software projects) are examined in 3.1.3, therefore only the importance of the first phase is described here: Karnovsky states "*Tell me, how you start a project and I tell you how it ends*" [*Karnov02:40*] which is – according to the experiences of this thesis author – fully true. A good "*kick-off*" is the foundation for the essential team spirit and a harmonious ambience during the entire project, and a bad start is hard to counterbalance later. Karnovsky defines the *project start phase* as period between the birth of the project (first idea) and the formal kick-off which starts the project. It implies the clarification and limitation of the intended goals as well as a rough estimation about the necessary resources (time, budget, and staff). Also the project framework must be defined (Responsibilities, Project Management, team structure, etc.). Maybe a feasibility study or pilot studies are carried out to evaluate the initial idea and clarify open questions. All this leads to a formal definition and/or contract for the project. Based on this formal agreement, the kick-of-meeting can take place and the team can start to work (see 3.2). Karnovsky strongly advocates to find and fix the "*godfather*" of the project (German: "*Pate*"), thus a high-rank manager on the client side who promotes the project and to whom the project management has direct access.

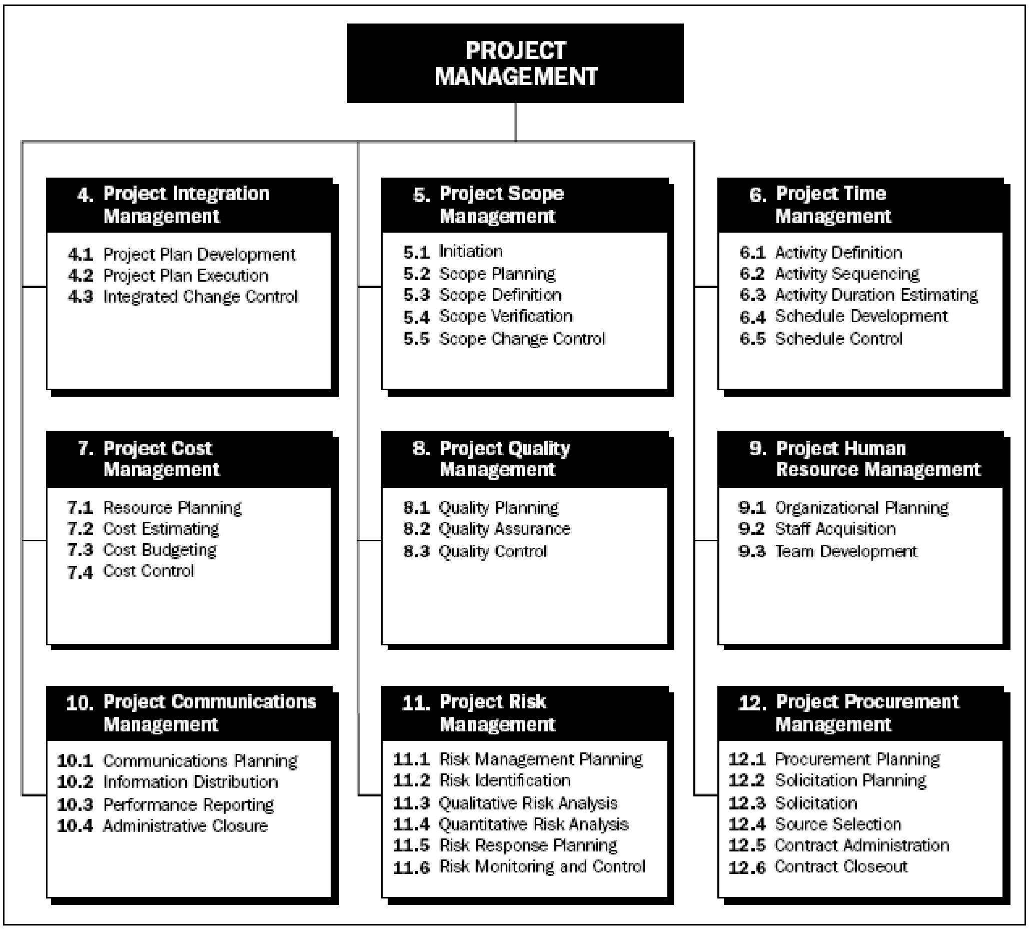

PROJECT MANAGEMENT
4. Project Integration Management
4.1 Project Plan Development
4.2 Project Plan Execution
4.3 Integrated Change Control
5. Project Scope Management
5.1 Initiation
5.2 Scope Planning
5.3 Scope Definition
5.4 Scope Verification
5.5 Scope Change Control
6. Project Time Management
6.1 Activity Definition
6.2 Activity Sequencing
6.3 Activity Duration Estimating
6.4 Schedule Development
6.5 Schedule Control
7. Project Cost Management
7.1 Resource Planning
7.2 Cost Estimating
7.3 Cost Budgeting
7.4 Cost Control
8. Project Quality Management
8.1 Quality Planning
8.2 Quality Assurance
8.3 Quality Control
9. Project Human Resource Management
9.1 Organizational Planning
9.2 Staff Acquisition
9.3 Team Development
10. Project Communications Management
10.1 Communications Planning
10.2 Information Distribution
10.3 Performance Reporting
10.4 Administrative Closure
11. Project Risk Management
11.1 Risk Management Planning
11.2 Risk Identification
11.3 Qualitative Risk Analysis
11.4 Quantitative Risk Analysis
11.5 Risk Response Planning
11.6 Risk Monitoring and Control
12. Project Procurement Management
12.1 Procurement Planning
12.2 Solicitation Planning
12.3 Solicitation
12.4 Source Selection
12.5 Contract Administration
12.6 Contract Closeout

Figure 3.1: PMBOK – Nine Knowledge Areas and related Processes [*PMBOK04:8*]

3.1.2. Characteristics of Software Project Management (SWPM)

> "*Enable the knowledge workers to do what they are being paid for*"
> (Peter Ferdinand Drucker [*Drucker77:273*])

Peter F. Drucker – an Austrian-born mastermind in modern US-management sciences – already coined in his early works the term "*knowledge worker*" ("*An employee whose major contribution depends on his employing his knowledge rather than his muscle power*" [*Drucker77:348*]). This term seems to be well applicable for software developers, as Software Projects are quite different from many other kinds of technical projects (see 2.2.1). Therefore also *Software Project Management (SWPM)*, which is defined as "*the process of planning, organizing, staffing, monitoring, controlling, and leading a software project*" [*Benn95:3*], has characteristics:

- Software Project Management is normally strongly tied to the methods and Process Models for software development (see 2.2.6). They have a complex relation like a chicken and an egg (Which was first?): SWPM chooses the Process Model(s) and tailors the process for the specific needs of an individual project (see figure 2.13 in combination with figure 2.14). At the same time the Process Model (and its underlying methodology) strongly influences the way, how SWPM is conducted. Plan-driven Process Models (e.g. RUP, V-Modell XT) imply guidelines for SWPM [*Versteeg00*]. Also for agile Process Models there is something similar like the Agile Project Management (APM) [*August05*] (Even if sometimes the combination of agile Process Models and SWPM seems like a contradiction, as agile approaches where often invented to "free" programmers from SWPM). So there is no universal form of SWPM, there are many.
- Software Development teams have no "blue collar workers", as programming implies a lot of creativity and brain-work, so it is "*less deterministic*" [*Benn95:3*] than a classical production process (software is produced by "projects", not by a production line). SE made progress in minimising the negative implications of this fact, with Process Models (2.2.6) and Software Process Improvement (2.2.4), but the fact cannot be neglected.
- The fact remains, that "*software projects are less measurable, more difficult to estimate, and more dependent on subjective human factors*" and even if modern Process Models created a better framework "*this has led to a new problem: developers complained that they were spending too much time on documentation and too little on the actual development of code*" [*Benn95:3*]. Which leads back to Peter Drucker who stated in 1977: "*highly paid and competent scientists are not allowed to do their work, but are instead forced to attend endless meetings to which they cannot contribute and from which they get nothing*" [*Drucker77:273*]. So "*a middle ground should be sought between … the freelance-style project … and the over-standardized, over-documented project*" [Benn95:3]. Royce therefore pleas for "*balance*" and "*pragmatic*" in SWPM [*RoyceWa98:xxiii-f*] and Boehm highlights the same questions in his "planning spectrum" (see figure 2.3) and the possible "home-grounds" of divers Process Models (see figure 2.14).
- A critical summary about the consequences of these problems in real practice is given by Royce: "*Software development is still highly unpredictable. Only about 10% of software projects are delivered successfully within initial budget and schedule estimates*" [*RoyceWa98:5*]

Other SWPM-characteristics are similar to general PM as stated in the introduction of 3.1. and in 3.1.1.

3.1.3. Planning and Controlling: Time Schedules and Cost Estimation

The planning and control tasks in Software Project Management imply a lot of *administrative paperwork* (which should be tailored for the specific needs of the individual project) and in the most extensive form a *Software Project Management Plan (SPMP)* can be as comprehensive as described in [*IEEE98b:4*]. The significant elements of these activities are presented in the following.

● ***Contracting and effort-estimation***

Before the initial idea and request from the client can be specified in a formally signed contract (maybe with a formal Project Management Plan), software development projects normally start with estimating the expected effort to fulfil this request (time, money and staff) [*Schach96:290ff*], [*Sommer06:660ff*]. The form of the customer-developer relationship strongly influences the detailed form of the contract [*Benn95:25ff*]. There are five basic parameters for cost-estimation for which Royce even provides a simple formula: "*size, process, [capabilities of the] personnel, environment, and required quality*" [*RoyceWa98:5*]. Royce names several popular cost estimation models (e.g. *COCOMO, CHECKPOINT, ESTIMACS,* …) but he also warns that a common approach is: "*This project must cost $X to win this business. … Here's how to justify that cost*" [*RoyceWa98:28*]. A practical list of common approaches provides [*ZuserGre04:112*]:

(1) "*Aus-dem-Bauch-Methode*" (Instinctively and formless estimation based on experience)
(2) *Analogy method* (Similarities to former projects are discovered and adapted)
(3) *Multiplication method* (The project is split in modules which are estimated exactly and summarised)
(4) *Weighting method* (cost-drivers are identified – normally code-lines or functions points – and the effort is estimated with the help of a defined formula, e.g. in the COCOMO-model)
(5) *Percentage method* (One single part of the project is estimated in detail and afterwards extrapolated for the entire project).

● ***Planning***

In Software Projects the planning of tasks, responsibilities, workflows and resources often follows some specific rules of the applied Process Model (e.g. RUP), but the following planning methods are quite common in Software Engineering (see [*GheJaz03:476ff*], [*ZuserGre04:120ff*], [*Sommer06:130ff*]):

- A *Work Breakdown Structure (WBS)* splits the project into components (tasks or objects [*Karnov02:56*]) and is a hierarchical tree structure where the root is the project goal. "*Once these pieces have been identified, they can be used as units of work to assign to people*" [*GheJaz03:477*].
- Dependencies and sequences between tasks of a project can be planned with graphs, e.g. with a PERT-diagram (*Program Evaluation and Review Technique*) or a "*Netzplan*".
- Time Schedules for the individual activities, team members and resources can be planned with a *Gantt-Chart*, where each activity, person or resource forms a row and each time unit (day, week) forms a column. Gantt-Charts are popular, as they provide a concise overview and can be easily extracted from the Work Breakdown structure. Figure 3.2 shows a practical Gantt-example from a students project at TU Wien.

Today a lot of software-tools exist to support the planning of projects and the mentioned diagrams, which also support the detection of the "*critical path*" (most critical activities that can delay the project).

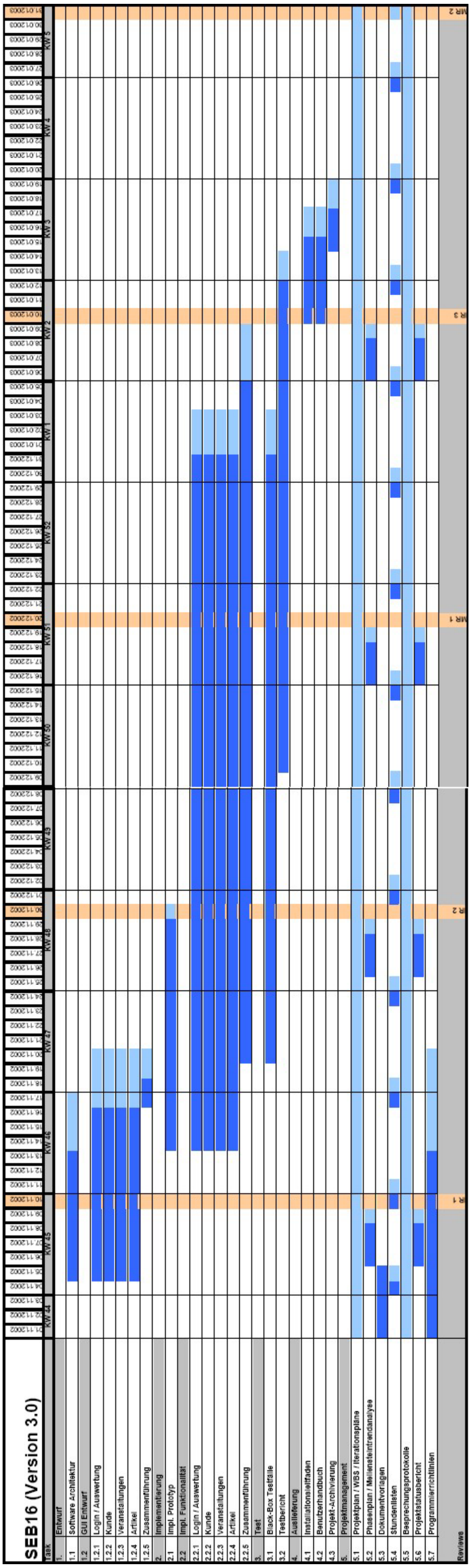


Figure 3.2: Gantt-chart of a simple Students Project at TUW [*A. Fehr, R. Fortner*]

Anticipating chapter 3.1.4, it has to be emphasised that administrative planning tasks like time schedules and resource planning cannot be successful and realistic without respecting the human factor. The planning in software projects is "*intimately tied to the problem of how to estimate the productivity of software engineers*" (question of metrics in software engineering) as well as to the problem that "*it is often difficult to specify the software requirements completely a priori*" [*GheJaz03:461*]. So schedules should be prepared together with the client and the team to make them realistic, "human-oriented" and generally accepted. "Buffer"-times are recommendable. Further interesting ideas provide [*DeMarco97*], [*DeMarco01*], [*Gärtner04*].Advices for realistic schedules are given in [*ZuserGre04:118ff*]: The schedule must not scamper or demoralise the team. It should be accomplishable but also ambitious enough to guarantee continuing progress. The product has priority over the schedule. Long tasks should be split in subprojects to make them more clearly. Subproject should produce a satisfying and motivating result.

● ***Controlling***

Controlling of the ongoing progress is one of the most important tasks for the project management after the project has been started. It "*includes the efficient management of the development team members and requires constant awareness of the real status of their work on the project*" [*Benn95:3*]. Especially for iterative or evolutionary development methods it also implies a regularly replanning of the requirements and time schedules for the next iteration. Therefore "*controlling requires the measurement of performance against plans and taking corrective action when deviations occur*" [*GheJaz03:460*]. How to handle such deviations is a key task for the project management and requires many of the skills presented in the next subchapter 3.1.4. Common methods to control the progress in a project are regular status-reports to the project management in a written form or during periodical meetings (Informal: "*Jour-fixe*"; Formal: Reviews).

As described in 2.3.1, there can be major or minor milestones that define products (documents, code, etc.) which should be achieved at a particular point in time. During every formal meeting the project management checks those milestones and their current estimation. A *Milestone Trend Analysis* illustrates those estimations in a demonstrative way: The x-axis is the time-axis of the project, and the y-axis represents new milestone estimations during a meeting. Commonly the estimations start to shift rightwards during a project (as illustrated in figure 3.3), identifying future problems at an early stage.

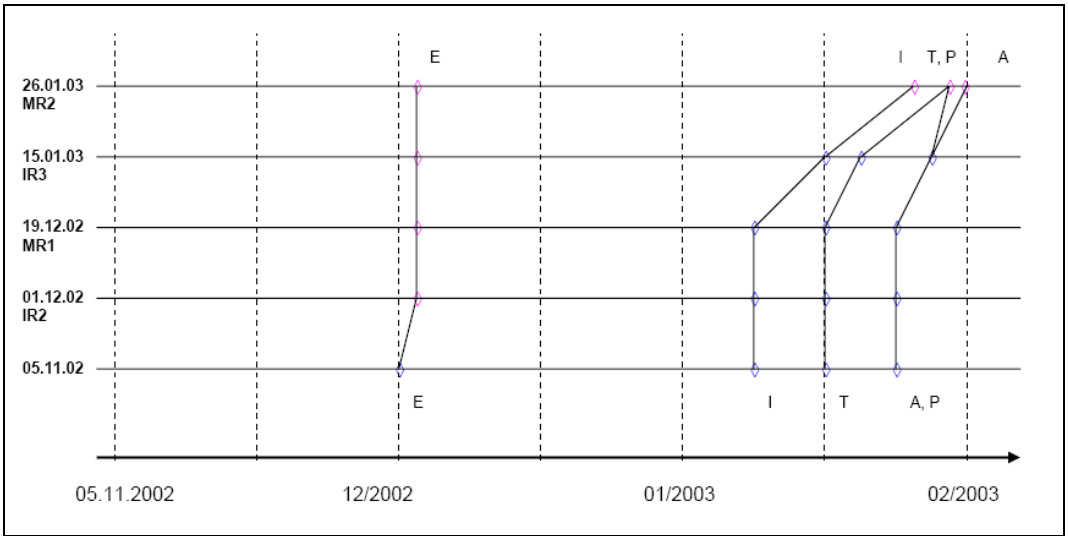


Figure 3.3: Milestone Trend Analysis of a simple Students Project at TUW [*A. Fehr, R. Fortner*] (*IR: Internal Review; MR: Management Review; E: Design Phase; I: Implementation; T: Test; A: Delivery; P: Project*)

3.1.4. Software Project Managers: Leadership, soft skills and client-team-link(er)

"I am not sure there is such a thing as management training other than on-the-job experience"
(Walker Royce, [*RoyceWa98:xxiv*])

As stated at the beginning of 3.1, successful SWPM needs more than the elaborate utilisation of PowerPoint and MS Project. It requires leadership and a lot of so called "*soft skills*". Project Managers have to reconcile the needs of all project-participants and act as a *link(er)* between them (see figure 3.4).

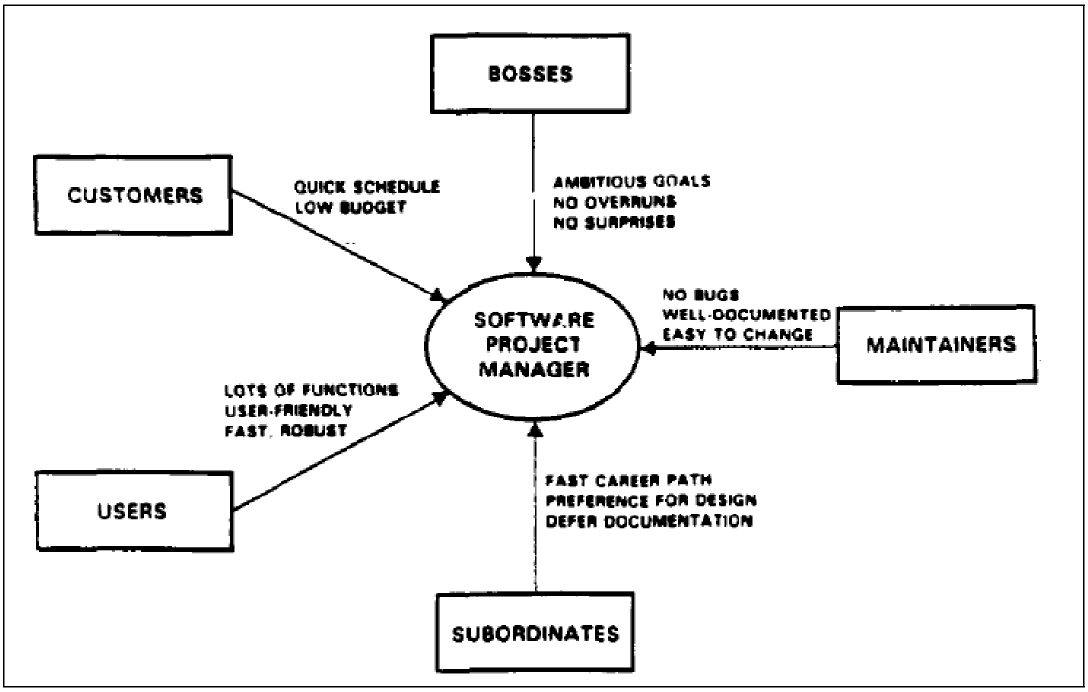


Figure 3.4: The software project manager's problem (timeless diagram) [*BoehRos89:903*]

Depending on the rules of a project, the competencies of the Software Project Management "*can reach from simple moderation to disciplinal authority*" [*Karnov02:109*]. Also the required professional and managerial competencies strongly depend on the needs of the conducted project. Therefore the PMBOK defines "*key general management skills that are highly likely to affect most projects*" [*PMBOK04:21ff*]:

- LEADING (*Establishing direction; Aligning people; Motivating and inspiring*);
- COMMUNICATING (*implies knowledge about Sender-receiver models, Choice of media, Presentation techniques, body language, Management of meetings*);
- NEGOTIATING (*conferring with others to come to terms with them or reach an agreement, e.g. about: Scope, cost, and schedule objectives; Changes to scope, cost, or schedule; Contract terms and conditions; Assignments; Resources*);
- PROBLEM SOLVING (*Problem definition & Decision-making: Distinguishing between causes and symptoms. Problems may internal, external, technical, managerial, or interpersonal; Identify viable solutions, and then making a choice from among them. Once made, decisions must be implemented*);
- INFLUENCING THE ORGANISATION (*Ability to "get things done"; Requires an understanding of both the formal and informal structures of all the organizations involved. Influencing the organization also requires an understanding of the mechanics of power and politics*).

All these skills require "*on the job-experience*" and they are hardly to teach and learn via books ("*That's why managers are paid big bucks*" [*RoyceWa98:xxiv*]). But some readable sources which provide "*food for thought*" have to be mentioned:

- An all-around tour about InterPersonal Skills for managers (*Self-awarness, Communication, Motivating, Leading, Teaming, Problem Solving*) provides [*RobHun03*].
- An academic introduction about all related aspects of social psychology (*Perception and Interaction*) gives [*Herkner91*].
- More practical advices can be found in the voluminous opus of Friedemann Schulz v. Thun: *Anatomy and four aspects of a message, Problems in interpersonal communication* [*Thun81*], *Communication Styles* [*Thun89*], The "*interior team*" of a person and related conflicts [*Thun98*]. A survey of all these topics for managers is presented in [*ThuRup00*].
- A timeless Austrian source for all aspects of Body Language is Samy Molcho who identifies that "*The body is the glove of our sole*" [*Molcho94:20*].
- Another Austrian source is communication scientist Paul Watzlawick who became famous for his entire (voluminous) work, but mostly for his "*hammer*" and his five axioms: "*One Cannot Not Communicate*"; "*Every communication has a content and relationship aspect such that the latter classifies the former and is therefore a metacommunication*"; "*The nature of a relationship is dependent on the punctuation of the partners communication procedures*"; "*Human communication involves both digital and analog modalities (words are "digital"; non-verbal and analog-verbal communication are not)*"; "*Inter-human communication procedures are either symmetric or complementary, depending on whether the relationship of the partners is based on differences or parity.*"

Most SE-textbooks reduce Software Project Management on its technical and administrative aspects. If the human factor in SWPM is covered at all, it is reduced on its internal dimension, therefore only on team management and people management (see subchapter 3.1.5), while the considerations about the same effects on the client side are often missing or hidden in Risk Management (Even if software project managers could also need a book about "*The Psychology of dealing with clients*" in many situations, like Weinberg's famous book [*Wein04*]).

A survey about the "*customer-developer relationship*" gives [*Benn95:25ff*] (even if human factors are a minor topic) who states: "*Customer relations: In some projects, contact with the costumer is a major activity. This includes documenting the customer's requirements, controlling changes required by the customer, handling the customer's involvement in the development process, providing reports and organizing reviews and product demonstrations*" [*Benn 95:3*]. External risks and aspects in SWPM (concerning the client) can also be found in [*GerlGerl05*].

Royce widely covers the uncertainty and influence of human factors in SWPM. He pleas for two values in SWPM: "*balance … achieve balance among the objectives of the various stakeholders*"; "*pragmatic … having no illusions and facing reality squarely*" [*RoyceWa98:xxiii-f*]. He also covers "*Adversarial Stakeholder Relationships*" [*RoyceWa98:15ff*] which will be examined later in this thesis.

Agile Project Management (APM) also covers client stakeholders and states: "*Form a Guiding Coalition: The coalition should have a core of senior managers who have the power, credibility, and experience to lead the change represented by your project*" [*August05:71*]. This seems like an "agile" version of the Karnovsky-rule, that the PM should always find a "*godfather*" for the project among the top-management [*Karnov02:42f*].

3.1.5. Teamwork in Software Development Projects

An important sub area of SWPM which is strongly related to the human factor is teamwork (one of the first sources covering this topic was Weinberg in 1971 [*Wein04:143ff*]). Therefore modern SE-textbooks already cover issues like Team Management or People Management by default (see [*Benn95:68ff*], [*ZuserGre04:169ff*], [*Sommer06:635ff*]). Also standard PM-sources address this issue ([*PatzRatt04:53ff*], [*PMBOK04:107*]). Alternative or practical views are given by [*LanBrau85*], [*DeMaLi87*] and [*Spreng00*].

● *Individual, group, team*

As stated in 3.1.2, SWPM differs from other forms of PM because programmers are "knowledge workers". Therefore Bennatan states: "*According to many studies, managing software engineers is more difficult than managing engineers in most other areas of technology. The typical software engineer (if such a person exists) is often characterized as being both artistic and logical, as well as possessive and temperamental. These traits can be found in any group of people, but they appear to be more prevalent among software engineers*" [*Benn95:68*]. According to Bennatan this explains productivity ratios in SE-teams of 1:5 or even more (what modern development methods and Process Models as presented in 2.2.6 try to improve). Most software development projects are organised as teams, even bigger projects are divided to enable smaller teams. According to Bennatan "*the ideal size of a development team is between four and six developers*" [*Benn95:73*]. So to reach an appropriate proficiency level, a conglomerate of INDIVIDUALS (with no common goal) has to be organised as a GROUP (to follow a common goal) and should become a real TEAM (where a real synergy-effect leads to an extraordinary productivity level and a common "team-spirit") [*ZuserGre04:181*].

● *Team Development*

[*ZuserGre04:183f*] presents the four phases of team development as proposed by Tucker, to which he later added a fifth phase (see figure 3.5):

(1) FORMING: *Team members learn and agree about the common goal(s) in their best behaviour.*

(2) STORMING: *Competition of ideas and interests leads to first confrontations and power struggles. Coalitions and cliques are formed and may last. This phase is important for the growth of the team (and can also end destructive), so conflicts should not be burked but moderated to overcome them.*

(3) NORMING: *Team members start to coordinate themselves and cooperate in a positive and efficient way. Common rules and a pleasant behaviour enable real team-work and a factual conflict resolution.*

(4) PERFORMING: *The group has become a self-organised team and is able to perform on a high level without much external intervention. Later it can fall back to earlier phases, e.g. after structural changes.*

(5) ADJORNING: *The project is finished and the team is dissolved, but personal contacts may remain.*

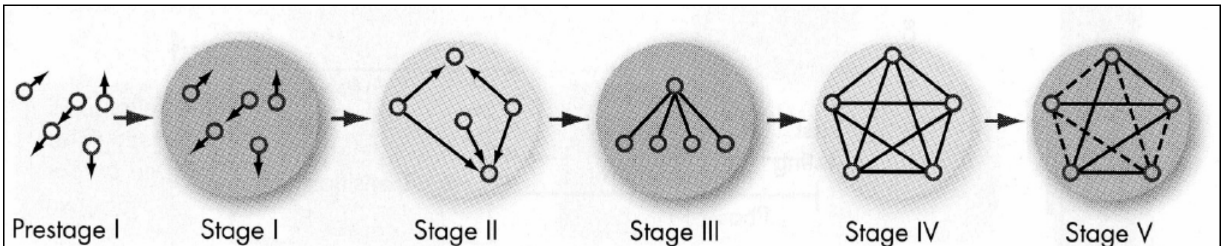


Figure 3.5: Stages of Team Development [*Robbins03*]

● *Organisation and structures*

A team has normally an organised structure that strongly influences its rules and workflows. Structures and organisation deal with "*devising roles for individuals and assigning responsibilities for accomplishing projects goals … motivated by the need for cooperation when the goals are not achievable by a single individual in a reasonable amount of time … The aim of an organizational structure is to facilitate cooperation towards a common goal*" [*GheJaz03:483*].

The choice of the appropriate organisational structure depends – like the entire SWPM – on the used SE-methodologies and Process Models, and is also related to the characteristics of the project (size, duration, etc.). There are structures with *centralised control, decentralised control* and *mixed-control* – and all of them occur in modern SE, which ranges from military software development to (similarly successful) open source projects. In big companies projects can range across many departments, so the project can be formed as *staff unit, line organisation, matrix organisation* or as *informal organisation* [*Karnov02:125ff*].

For Bennatan the "*hierarchical structure is based on the four cornerstones of management: delegation, authority, responsibility and supervision. … it is the project manager's responsibility to select the structure best suited for the project … the larger the project the more critical the organizational structure becomes*" [*Benn95:68*]. He also distinguishes three kinds of teams: *Democratic teams, chief engineer teams* (also called *chief programmer teams*) and *expert teams* (see figure 3.6). Schach adds a fourth (more practical approach), the so called "*modern programming team*", consisting of a team-manager for non-technical management and a team leader for technical management [*Schach96:390ff*]. For large projects, this structure can be scaled up, leading to several independent teams (each with an own team leader) who all report to a common project leader.

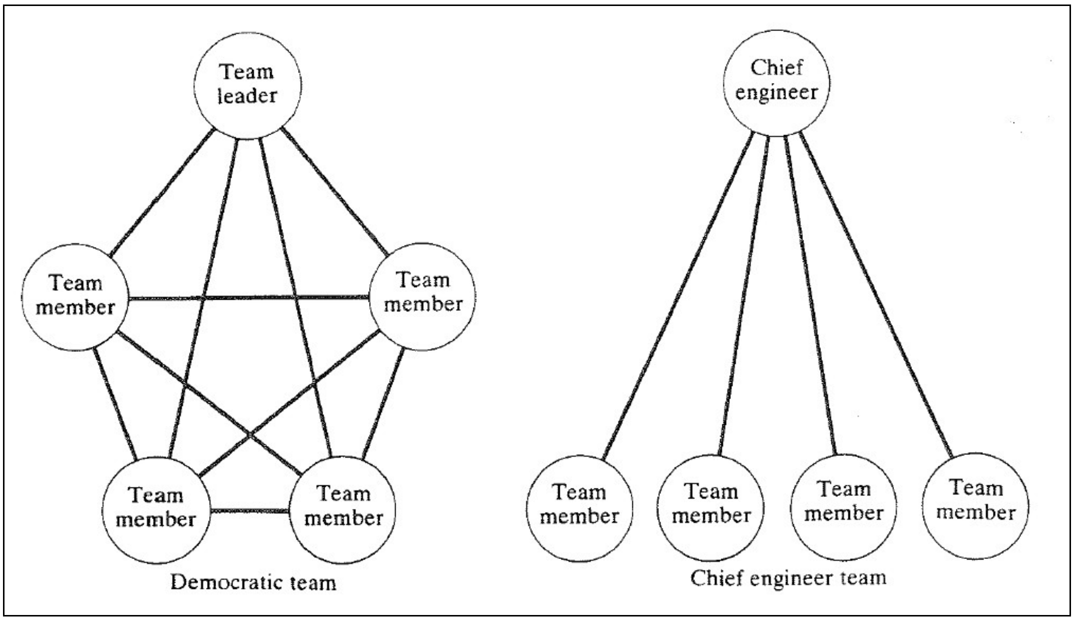


Figure 3.6: Possible team organisations [*Benn95:74*]

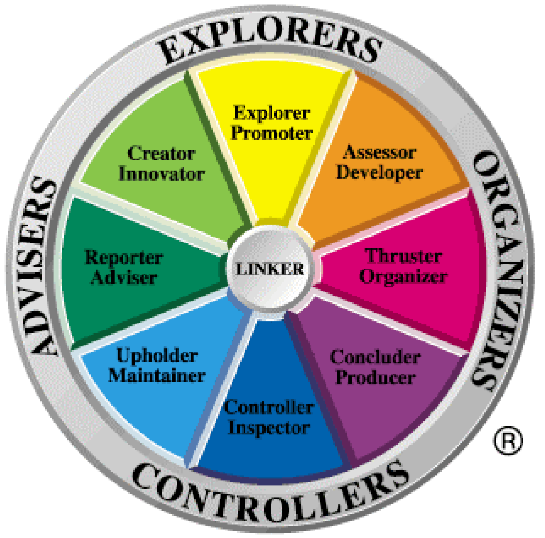
Other SWPM-tasks concerning team management are *recruiting* ("staffing"), appropriate conditions of work (including breaks and a coffee machine) as well as the entire field of *people management* (which also includes giving team members a perspective for their personal career and development beyond the current project). DeMarco and Lister plea for a human-oriented view in software team management and state: "*The major problems of our work are not so much technological as sociological nature*" [*DeMaLi87:4*].

• *Roles*

> *"All the world's a stage, and all the men and women merely players:*
> *They have their exits and their entrances; and one man in his time plays many parts"*
> (*William Shakespeare, As You Like It, 2nd Act, 7th Scene*)

Software development teams can have official (formal) roles as they are defined in the RUP (see 2.3.1), like *Client, User, or Management* as well as *Project leader, team leader*, developer-roles (*analyst, architect, programmer*), *tester* [*ZuserGre04:172*]. The assignment of roles depends on the project, the used SE-method and on the enterprise policy ("*specialists*" vs. "*generalists*"). Each team member can also have one or several informal role(s) which are examined by social sciences. One example is the *Team Management Wheel* (by Margerison and McCann, based on theories of C.G. Jung) [*ZuserGre04:174ff*], which describes possible *working styles and work preferences* of team members. It can be the foundation for team assessment and work assignment decisions (see figure 3.7). A similar model is given in [*Robbins03*] (see figure 3.8).

Related issues concern *leadership* (authoritarian vs. democratic) [*ZuserGre04:184ff*], *factors and characteristics* for successful teamwork [*ZuserGre04:182*], [*PatzRatt04:56ff*], *communication* in the team [*ZuserGre04:179f*], [*Sommer06:649*], *motivation* [*Sommer06:641*] and *conflict management* [*PatzRatt04:368ff*]. Most of these issues imply soft-skills-knowledge as described in 3.1.4, therefore those qualities are not only restricted to the project managers and are instead recommended (but not formally required) for any team member.



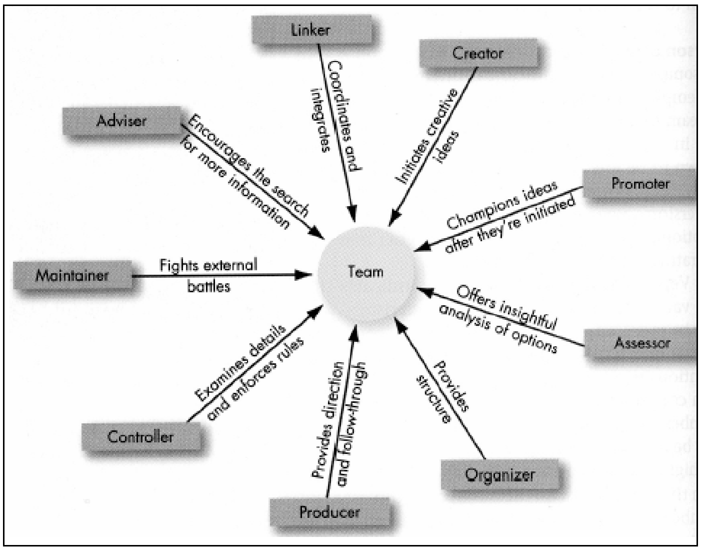

Linker
Creator
Adviser
Promoter
Maintainer
Team
Assessor
Controller
Organizer
Producer
Coordinates and integrates
Initiates creative ideas
Encourages the search for more information
Champions ideas after they're initiated
Fights external battles
Offers insightful analysis of options
Examines details and enforces rules
Provides direction and follow-through
Provides structure

Figure 3.7: Team Management Wheel [*www.tms.com*]; **Figure 3.8: Key Roles of Teams** [*Robbins03*]

• *Current challenges in our flat world of globalisation*

Currently new challenges arise: Comfortable tools for *Computer Supported Cooperative Work (CSCW), distributed work and virtual teams* (with team members located in different places, countries or even globally distributed on different continents) become increasingly important in our "flat world" of globalisation and outsourcing. But the use of CSCW-methods and Emails often masks the *need for intercultural competencies* in multi-national teams. Even if the underlying technical views are quite similar (e.g. see [*Bénard92*] how a Francophone source covers SWPM-problems likewise), but language barriers and cultural differences often influence the teamwork subtly. Rothlauf describes influences on the cultural environment of international business [*Rothlauf06:48f*] and provides an extensive list with required competencies for international teams [*Rothlauf06:115*]. A practical survey is given in [*Baumer02*].

3.2. Environmental influences and risks acting on the project

"Here be dragons (lat.: Hic sunt dracones)"
(Sign for dangerous or unexplored territories,
attributed to mediaeval cartographers)

Standard-textbooks for Project Management already cover the full spectrum of problematic or risky "human" influences on a project – including the (also external) "*environment*". In contrast, most SE-textbooks focus on the human factor in the development team, but neglect the same effects on the client side or cache them as sub-item in topics like Risk Management or Quality Management (where they are often hardly recognisable). It is likely that this will change in the future, but currently it seems like most SE-textbooks are too noble to call a spade a spade – maybe because SE still struggles a lot with so many internal problems (see 2.1) or because in the past the client has been too often blamed wrongly (So Fröhlich strongly criticises software engineers for commonly shifting the blame to the client – for which he uses the grave German phrase "*Dreiste Schuldabwälzung*" [*Fröhlich02:65*]).

But when one examines well-known "*Software Disasters*" like the new baggage-handling system at Denver International Airport (DIA) [*Glass98:23ff*] or the new air-traffic control system for the US-Federal Aviation Administration (FAA) [*Glass98:56ff*] it is not convincing to search the responsibility solely in Software Engineering, like Scientific America did in 1994 for the DIA-project (while Glass and DeMarco deliver insights to the complex environment of those projects, see [*DeMaLi03a:Chapter 3*]). Both examples show, how a problematic environment can bring a project totally out of control.

But there is also some important responsibility for SE and SWPM: To detect such external problems early and react accordingly, which may include some uncomfortable communication and negotiations with the client or other parts of the external environment. Here the fault is also an underdeveloped set of practices in SE and SWPM that doesn't comprise considerations about such issues in an appropriate dimension. Instead, the ideas of pioneers who push frontiers and focus on sociological SE-problems – like DeMarco, Lister, Glass – are neglected or dismissed by some parts of the SE-community who search the philosophers stone somewhere else ("art"- or "craftsmanship"-approaches, solely focus on technical engineering-principles and CMM-levels). Surely not every software project manager is able to "*speak truth to power*" [*Booch07:12*] as direct as Grady Booch did, but caching problems doesn't help to solve them (a principle which is already accepted in the storming-phase of SE-team development, see 2.1.5). So obviously there is still a need for more theoretical and practical research in *soci(ologi)cal SWPM*.

PM-literature covers the risks and influences from the external environment by default, so German-speaking PM-textbooks present concepts like the "*Projektumweltanalyse*" [*Gareis04:Chapter3*] or "*Projektumfeldanalyse (PUA)*" [*PatzRatt04:68ff*], both terms roughly translatable into English as "*Project Environment Analysis*" (even if the English term "environment" is also related to ecological issues). A similar – but not identical – concept in the Anglophone PM-literature is the "*Stakeholder Analysis*" which is mostly known and used in the business-context. A related topic, also coming from the corporate world (corporate finance as well as insurance industry) is the so called "*Risk Management*" which was later introduced in Project Management and has – assisted by authors like Boehm [*Boehm89*], Lister [*Lister97*] or DeMarco [*DeMaLi03a*] – meanwhile also found its place in SE-textbooks [*Sommer06:135ff*].

3.2.1. Significant SE-risks and reasons for Software Project Runaways

What is a risk? Project Management knows the overall project risk ("*Gesamtrisiko*") which is composed of individual risks ("*Einzelrisiken*") [*PatzRatt04:42*]. "Risk" needs to be defined particularly for projects as the concept of Risk Management was originally defined in a business- and military context:

> "*Project risk is an uncertain event or condition that, if it occurs, has a positive or a negative effect on a project objective. A risk has a cause and, if it occurs, a consequence. … Project risk includes both threats to the project's objectives and opportunities to improve on those objectives.*" [*PMBOK04:127*]

SE-textbooks often provide listings about "typical" risks and problems in software projects, based on manifold studies (e.g. *CHAOS-report* from the *Standish Group*). Some sources associate those problems with the so called "*Software crisis*" (e.g. [*Versteeg00:2ff*]) while other sources deny this concept ("*I do not believe in the existence of a software crisis … Most of those who cry crisis and are trying to sell or promote something are offering a better technology for building software. But most of the case studies of software failure find that poor management technique, not poor technology, is the cause of the problems*" [*Glass98:6*]). Irrespective of the reasons "*only about 10% of software projects are delivered successfully within initial budget and schedule estimates*" [*RoyceWa98:5*]. Some significant lists about typical risks, failures and problems in software projects follow below.

● *Bennatan* discusses "*basic problems which a project manager is likely to find in any software project*" [*Benn95:13f*]:

> (1) *Inadequate initial requirements and frequent changes*; (2) *Dependence on external sources (vendors, subcontractors etc.)*; (3) *Difficulties in concluding the project*; (4) *Frequent replacement of development personnel (staff turnover)*; (5) *Poor estimates*; (6) *Inadequate tracking and supervision*; (7) *Uncontrolled changes*

● In 1988 *Boehm* (also a pioneer in Software Risk Management [*Boehm89*]) presented a Top-10-list of risks in software projects, last updated in 1998 (see [*Boehm88:70*], [*GheJaz03:493*], [*Mayr05:193ff*]):

> (1) *Personnel shortfalls;* (2) *Unrealistic schedules and budgets; inappropriate application of the process;* (3) *Shortfalls in COTS and externally furnished components;* (4) *Requirements mismatch (mistaken requirements);* (5) *User interface mismatch (Developing the wrong UI);* (6) *Bad architecture, performance and overall quality;* (7) *Requirements changes and development of mistaken or wrong functionalities;* (8) *Dealing with Legacy software (Inclusion into project or evolution);* (9) *Shortfalls of externally-performed tasks;* (10) *Straining technologies and computer science capabilities over their limits.*

● In 1998 *Glass* wrote an entire book with 16 practical examples (like DIA and FAA) covering seven reasons for so called "*Software Runaways*" (mostly based on a 1995-KPMG-study) as there are [*Glass98*]:

> (1) *Project Objectives Not Fully Specified*; (2) *Bad Planning and Estimating*; (3) *Technology New to Organization*; (4) *Inadequate/No Project Management Methodology*; (5) *Insufficient Senior Staff on the Team*; (6) *Poor Performance by Hard/Software-Suppliers*; (7) *Performance (Efficiency) Problems.*

● *Versteegen* notices ten reasons for project failures and links them to the "*software crisis*" [*Versteeg00:2ff*]:

> (1) *Unclear requirements*; (2) *Changing technologies*; (3) *Inadequate communication within the project*; (4) *Too late integration (leading to the "Late Design Breakage")*; (5) *Too strongly document-oriented*; (6) *Missing or inappropriate Process Model*; (7) *Inadequate staff education*; (8) *Missing (human) resources*; (9) *Missing quality management*; (10) *Neglecting the 80:20-rule(s) for software metrics by Boehm.*

● DeMarco and Lister have found five "core risks" for software projects [*DeMaLi03b:99*]: (1) *Intrinsic schedule flaw*; (2) *Specification breakdown*; (3) *Scope creep*; (4) *Personnel loss*; (5) *Productivity variation.*

3.2.2. Risk Management for software projects

"*If there's no risk on your next project, don't do it*" [*DeMaLi03a:3*] means, that only risky projects are interesting and profitable. This idea was already stated by Peter F. Drucker in 1975, pursuing ideas from the Austrian economist *Eugen Böhm-Bawerk*: "*existing means of production will yield greater economic performance only through greater uncertainty, that is, through greater risk*" (later quoted in one of the earliest SE-related Risk Management books, see [*Boehm89:53*]). So the quotation "*Risk Management is Project Management for adults*" [*Lister97*] means, that the "*manager adopts an adult attitude toward things that might go wrong during the project, a marked difference from the prevailing can-do attitude … to look problems directly in the eye*" [*DeMaLi03b:99*]. So Risk Management is *pro-active*, while Crisis Management is *reactive* (when the risk already occurred).

● *Risk Management as integral part of Project Management*

Risk Management is known in many disciplines (corporate finance, insurance industry, etc) but every domain has its particular needs and therefore Risk Management – which has meanwhile become an integral part of modern Project Management [*PatzRatt04:42*] – has been defined for projects as follows:

> "*Risk management is the systematic process of identifying, analyzing, and responding to project risk. It includes maximizing the probability and consequences of positive events and minimizing the probability and consequences of adverse events to project objectives. ... To be successful, the organization must be committed to addressing risk management throughout the project*" [*PMBOK04:127f*]

Risks are defined in the preceding chapter (3.2.1). Three main categories are distinguished, that can be applied and continuously refined for a specific domain, company or project [*PatzRatt04:47*]: Factual Environment risks (*Natural risks and disasters, technical, economical, laws*); Social Environment risks (*Client, partner, subcontractor*); Internal project risks (*Technical, contract, calculation, personnel, organisation, information*). Humans are different and so is their risk tolerance: Important stakeholders may have different attitudes toward project risks (*risk-seeking, indifferent, risk-averse*). Therefore a company as well as a specific project may have a particular "*general risk policy*", so the policy of client and developer should fit [*PatzRatt04:51*].

Risk management isn't a singular task, the assessment and control of risks has to be continued during the entire project. Risk Management can be based on qualitative as well as quantitative facts, as there are significant attributes of risks: Risk PROBABILITY (of occurrence: 0..1 or 0% ..100%); Risk IMPACT (qualitative or quantitative, e.g. damage or financial loss); LEAD TIME (possible time to react, see figure 3.9); Risk MANAGEABILITY (is it possible to avoid, transfer, minimise or mitigate the risk, e.g. with countermeasures); Risk COUPLING (is the risk linked with other risks) [*Mayr05:192*].

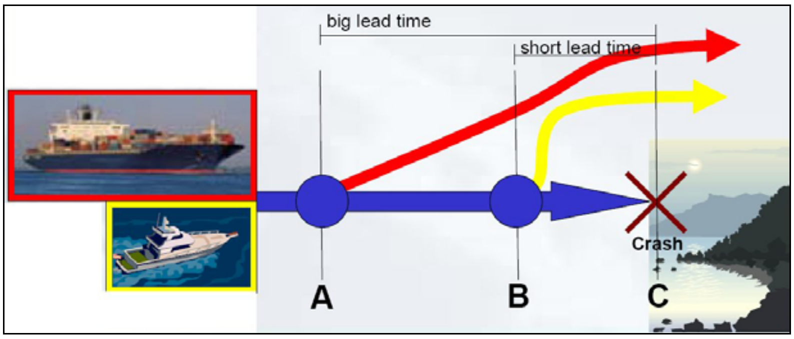


Figure 3.9: Different lead times for different risks and projects [*BiffHein05:Slide8*]

● *Risk Management in Software Project Management*

Risk Management in software projects was already mentioned by Boehm in the 1980's, when he proposed the risk-driven Spiral Model (see 2.2.6, section 4) and a related "*Software Risk Management Plan*" that consisted of the following elements: "*1. Identify the project's top 10 risk items; 2. Present a plan for resolving each risk item; 3. Update list of top risk items, plan, and results monthly; 4. Highlight risk-item status in monthly project reviews, Compare with previous month's rankings, status.; 5. Initiate appropriate corrective actions*" [*Boehm88:70*]. In 1989 Boehm proposed a taxonomy for Software Risk Management that is basically still valid and he already covered quantitative approaches (*Risk Exposure, RRL, etc.*) [*Boehm89*]. Later other authors like DeMarco and Lister covered the topic from a more sociological perspective [*DeMaLi03a*].

Meanwhile Risk Management is covered in most SE-textbooks by default (see [*Benn95:18ff*], [*GheJaz03:490ff*], [*AmNa05:164ff*], [*Sommer06:135ff*]) and it is also an integral part of Process Models like the RUP [*Kruchten04:118ff*] and "mature" processes as defined by CMM(I) and ISO 15504 (see 2.2.4). By contrast, it is still not very well established in practice, as Mayr quotes a 2004-survey which states "*Two thirds of the polled European SW-companies don't quantify their risks*" [*Mayr05:194*]. Boehm already stated in 1989 the four main reasons for Risk Management in software projects: (1) Avoiding Disaster (*like runaway budgets and schedules*); (2) Avoiding Rework (*of erroneous requirements, design and code that typically consumes 40-50% of the entire costs*); (3) Avoiding Overkill (*by focusing on critical areas instead of low-level risks*); (4) Stimulating Win-Win situations (*for participants*).

The main tasks of Risk Management in software projects are therefore [*Mayr05:196ff*], [*BiffHein05*]:

- RISK ASSESSMENT: Risk identification (*by the team and the client, e.g. brainstorming, interviews, etc.*); Risk analysis (*probability, impact and related costs, lead time*); Risk prioritisation and "clustering" (*e.g. with Risk Matrix, Risk Exposure calculation – see figure 2.10; often it is too expensive to cover all risks, so it is recommended to focus on the most significant N risks, e.g. N=10*).
- RISK CONTROL: Risk planning (*"plan B" for the real occurrence of a risk*); Risk Resolution (*Strategies: avoidance, transfer or acceptance, e.g. mitigate the risk with countermeasures*); Risk Monitoring (*regularly update of the risk planning with new risks, priorities or measures*).

A quantitative approach to Software Risk Management was first given by Boehm in 1989 [*Boehm89:6ff*] (*UO: Unsatisfactory outcome*): **RE (Risk Exposure) = (Probability$_{UO}$) x (Loss of Utility$_{UO}$).**

RE helps to detect important risks which should be addressed first (see figure 2.10: *risks A, C and H*). Boehm also defined the *Risk Reduction Leverage (RRL)* to choose among countermeasures. RRL is the cost-benefit ratio for a measure (Difference between RE before and after a measure, weighted with the costs of this measure). There are many tools supporting quantitative Risk Management in projects.

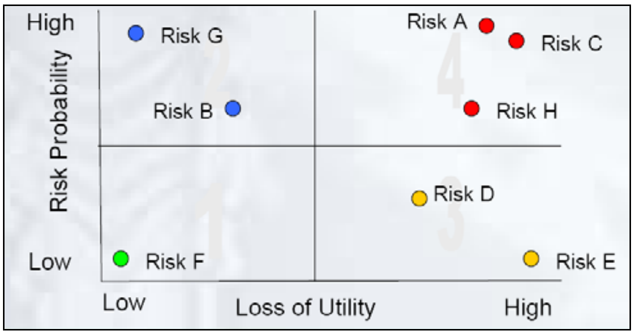


Figure 3.10: Risk Exposure Diagram [*BiffHein05:Part12*]

3.2.3. Project Environment Analysis (Projektumfeldanalyse – PUA)

Another useful technique to detect, assess and control project risks – which is especially famous in the German-speaking PM-literature – is the "*Projektumfeldanalyse (PUA)*" (also called "*Projektumweltanalyse*"), roughly translatable into English as "*Project Environment Analysis*" (but without any ecological meanings). It is similar to the Anglophone "*Stakeholder Analysis*" or "*Force Field Analysis*", but more PM-oriented.

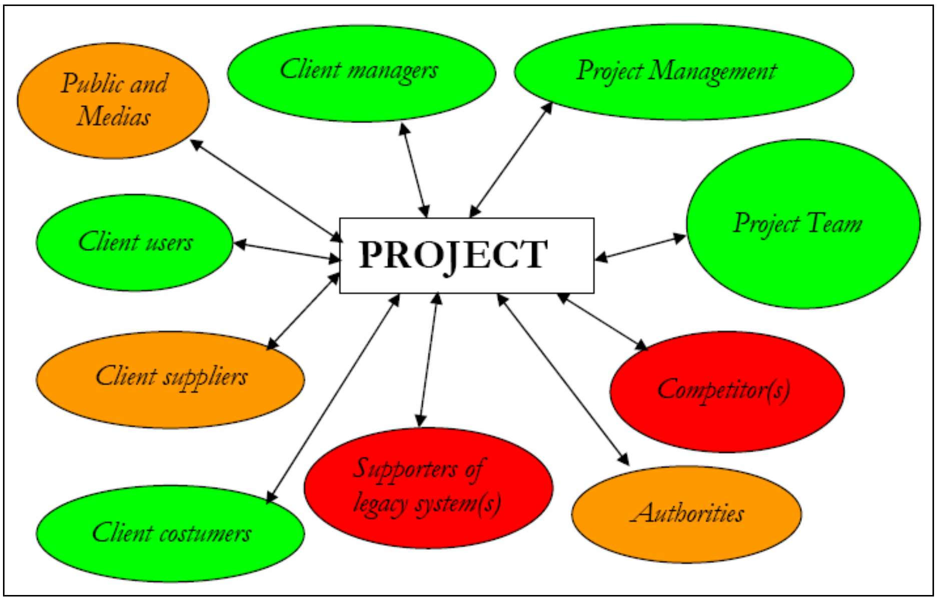


Figure 3.11: Typical "Project Environment" (own work, derived from [*PatzRatt04:69*])

The PUA is a helpful tool, as the PM is strongly forced to deliberate inconvenient issues in a structured way (like in Risk Management). So "*professional acting replaces spontaneous reacting*" [*PatzRatt04:68*]. The basic idea of the PUA is, that every project has an environment: People, groups, organisations or institutions that are relevant and (can) influence (or even jeopardise) the project [*Gareis04:Chapter3*], [*PatzRatt04:68ff*]. There is an *internal environment* (Project team, project management) and an *external environment* (client, competitors, suppliers). These groups can have relations with each other, that can be strong (e.g. team and project management) or weak (e.g. suppliers). The main PUA-tasks are:

(1) Holistic IDENTIFICATION of the Project Environment (internal, external)
(2) CLUSTERING: *Organisational-social groups* (int.; ext.) and *factual influences* (e.g. laws)
(3) ASSESSMENT of the environment and detailed ANALYSIS (see figure 3.11, table 3.1)
(4) Derivation of (counter-)MEASURES AND STRATEGIES

Environment group (Stakeholder)	***Attitude towards the project (☺ 😐 ☹)***	***Relevance and Influence (1..5)***	***(+) Expectations (-) Fears***	***Measures Strategies***
Client users	☺	4	*(+) Workflow easier*	*Involve*
Client managers	☺	5	*(+) Performance, gains*	…
Client costumers	☺	3	*(+) Web access …*	*E-Commerce*
Legacy-system supp.	☹	3	*(-) Loss of influence*	*Find new role*
Project team	☺	2	*(+) Interesting project*	…
Competitors	☹	2	*(-) Market shares*	…

Table 3.1: Typical project environment spreadsheet (own work, derived from [*PatzRatt04:71*])

4. STAKEHOLDERS AND REQUIREMENTS IN SOFTWARE PROJECTS

> "*There is a consensus, among both software developers and customers, that the activities of eliciting, understanding, and specifying requirements are the most critical aspects of the software engineering process*" [*GheJaz03:392*]

Requirements are the starting point for every software project, and their specification is maybe one of the most important and critical (risky) tasks, as all subsequent activities in Software Engineering build up on requirements (see Software Life Cycle in 2.2.5). A professional software project cannot start without any requirements, irrespective of the chosen (and tailored) Process Model which may support the later refinement and evolution of the requirements (like iterative models as RUP or agile approaches as XP, see 2.2.6 and 2.3). The cost- and time-estimations as well as the formal contract (see 3.1.3) need a factual base, some – even rough – description about "what" has to be developed.

At the same time, ambiguous, unstable and volatile requirements as well as insufficient Requirements Management are one of the biggest SE-problems: "*A recent European survey showed that the principal problem areas in software development and production are the requirements specification and the management of customer requirements*" [*SomSaw97:vii*] (For Austria see [*Kappel04:30*]). Wiegers states: "*Between 40 and 60 percent of all defects found in a software project can be traced back to errors made during the requirements stage*" [*Wiegers99:5*].

There is a strong influence of the "human factor" on requirements, the *stakeholders*. The last (14th) *IEEE Requirements Engineering Conference (RE06)* in September 2006 focused on this inseparable relation between stakeholders and requirements [*GlinzWie07:18*]. Its title "*Understanding the stakeholders' desires and needs*" addressed the fact, that Requirements Engineering should primarily satisfy client stakeholders. Wiegers states: "*Nowhere more than in the requirements process do the interests of all the stakeholders in a software project intersect*" [*Wiegers99:5*]. In view of the focus of this thesis, basic concepts of *requirements, stakeholders, Requirements Engineering* and *related risks* will be described in this chapter, so that they can be applied later.

4.1. Stakeholder(s) in Software Projects: Definition and classification

The term *stakeholder* was first coined in business sciences in the 1980's (like the term shareholder), meaning people who affect or can be affected (by) the goals and activities of a firm. Later this concept was adopted for projects (meanwhile also in software projects) and is defined as follows:

"*Project stakeholders are individuals and organizations that are actively involved in the project, or whose interests may be positively or negatively affected as a result of project execution or project completion; they may also exert influence over the project and its results.*" [*PMBOK04:16*]. The *IEEE Glossary for SE-Terminology* doesn't define the term until now [*IEEE90*]. Macaulay defined stakeholders as "*all those who have a stake in the change being considered, those who stand to gain from it, and those who stand to lose*". Glinz and Wieringa defined for SE: "*A stakeholder is a person or organization who influences a system's requirements or who is impacted by that system.*" [*GlinzWie07:19*]

Reflecting the internal-external-environment concept presented in 3.2.3, mainly two kinds of stakeholders are distinguished in software projects: *customers* and *developers* [*Macias01:4*]. Typical customer roles are: *managers, investors, system users, maintenance and service staff, etc.* [*Hull02:97f*], typical developer roles are described in section 2.3.1. (RUP) and 3.1.5 (teamwork). Identifying, analysing and prioritising stakeholders is one of the first activities when eliciting requirements [*SWEBOK04:2-3f*] (see 4.3).

4.2. Requirement(s) in Software Projects: Definition and classification

Wiegers states that "*no clear, unambiguous understanding of the term 'requirement' exists*" [*Wiegers99:7*]. Also Sommerville explicates, that the term *requirement* is not consistently used in the software industry, its meaning ranges from an *abstract* description of a service to a *detailed (formal)* definition of a function or system service [*Sommer06:150*]. There are also different views on requirements (*user view*, *developers view*), different *hierarchical levels* (Sommerville defines two; Wiegers three) and a technical distinction between *process- and product-requirements* as well as between *functional* and *non-functional* requirements. And the PMBOK-definition of project management also included the term *project requirements* (see 3.1.1).

● ***Definition***

The SWEBOK states "*A software requirement is a property which must be exhibited in order to solve some problem in the real world.*" [*SWEBOK04:2-1*] while the *IEEE Glossary for SE-Terminology* defines requirements as:

> "*(1) A condition or capability needed by a user to solve a problem or achieve an objective.*
> *(2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.*
> *(3) A documented representation of a condition or capability as in (1) or (2)*" [*IEEE90:62*]

As Wiegers points out, the IEEE-definition includes the users view ("*external behaviour of the system*") as well as the developers view [*Wiegers99:6*]. Which is one characteristic of (software) requirements: They are the primary connection between the *problem domain* ("*home of real users and other stakeholders, people whose needs must be addressed*" [*LeffWid04:19*], full of "needs" and "visions") and the *solution domain* (leading to a design). So Ebert compares the ambiguity of requirements with the list of wishes of a little boy to his parents: "*speedy car, bicycle like at Jos, round ball, toy that Paula brought to the kindergarten*" [*Ebert05:10*].

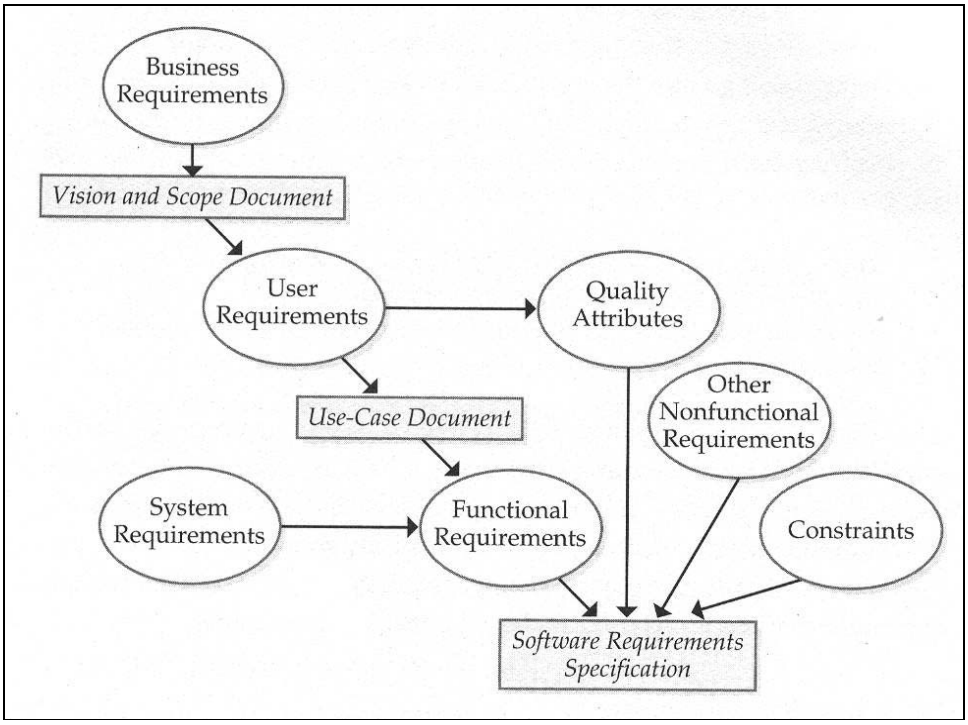


Figure 4.1: Classification of several kinds of requirements [*Wiegers99:8*]

● *Classification*

Basically there are two main kinds of requirements in software projects [*Ebert05:11*]:

- PROCESS requirements: *Process Model, CMM(I)-requirements, roles, organisation, rules or standards for the specific problem domain (e.g. standard DO-178B for aviation software)*
- PRODUCT requirements (see further classification and figure 4.1).

Process requirements will not be covered furthermore, even though Ebert cites the Conway-law, that structure and architecture of a product reflect the organisational structure of the involved people. Product requirements can be classified in tow dimensions (see figure 4.1):

- HIERARCHICALLY (three levels, as applied in this thesis and defined in [*Wiegers99:7f*])

  (1) *Business requirements* are the high-level "business" objectives (or goals) of the customer (person or organisation) for the software project. They should be captured in a "*Project Vision and Scope*"-document and are important to limit the project. "*To manage scope creep, start with a clear statement of the project's vision, scope, objectives, limitations, and success criteria*" [*Wiegers99:12*]. All other requirements must align with the business requirements. Other sources call this a "*concept document*" [*Benn95:48f*], and Sommerville recommends to "*make a business case for the system*" [*SomSaw97:49f*]. As described in chapter 2.4.3, IT- and business-strategies are increasingly growing together, so a common business vision helps all involved stakeholders to develop and secure a common understanding of the project in a written form. A software system will only be successful if it serves the business goals of the costumer(s), so this document also helps the project team [*ZuserGre04:224*].

  (2) *User requirements* describe "*tasks the user must be able to accomplish with the product*" [*Wiegers99:7*], so they are normally related to workflows and activities of the users. There are various forms to capture them, depending on the used methodology or Process Model. The RUP collects "*Use Cases*" while XP works with "*User stories*". Typically they are written in a natural language and describe in a more or less formal (structured) way – supported by diagrams or models – the desired functions of the system.

  (3) *System requirements and functional requirements* describe in a very detailed and often also very formal way the individual functionalities of the system.

- TECHNICALLY
  Functional requirements: Functions that the system shall execute to satisfy the business- and user-requirements. Maciaszek calls this a *service statement* ("*describes how the system should behave with regard to an individual user or … to the whole user community*" [*Macias01:22*]). A set of logically related functional requirements is a called a "feature".

  Non-functional requirements and constraints: Describe further characteristics of the system, like *performance, portability, reliability or usability* (see 2.2.1). They are also called "*supplementary requirements*" or "*constraint statements*" ("*restriction on the system's behaviour or on the system's development*" [*Macias01:22f*]). They can also be described in a quantitative form.

The detailed requirements are normally written down in a document like the "*requirements specification*" that describes what "*stakeholders expect to be satisfied in the implemented and deployed system*" [*Macias01:47*].

4.3. Requirements Engineering and Requirements Management in Software Projects

> "*Before developing any system, you must understand what the system is supposed to do and how its use can support the goals of the individuals or business that will pay for that system. This involves understanding the application domain; the system's operational constraints; the specific functionality required by the stakeholders; and essential system characteristics such as performance, security, and dependability.*" [*Sommer05:16*]

There is no generally accepted definition of *Requirements Engineering* and *Requirements Management*, each source has its own view. Ebert equates both terms and uses exclusively Requirements Management [*Ebert05:18*]. Wiegers takes the opposite direction and finely distinguishes Requirements Development from Requirements Management, both sub-disciplines of the entire Requirements Engineering (see figure 4.2) [*Wiegers99:19ff*]. Sommerville strikes a balance between both views, he sites Requirements Management in the middle of all requirements-related activities (see figure 4.3) [*Sommer05*]. And each sources diagnoses the same problem: "*Confusion about requirements terminology extends even to what to call the whole discipline. Some authors call the entire domain 'requirements engineering' while other refer to it as 'requirements management*'" [*Wiegers99:19ff*]. SWEBOK avoids both terms "*For reasons of consistency*" [*SWEBOK04:2-1*].

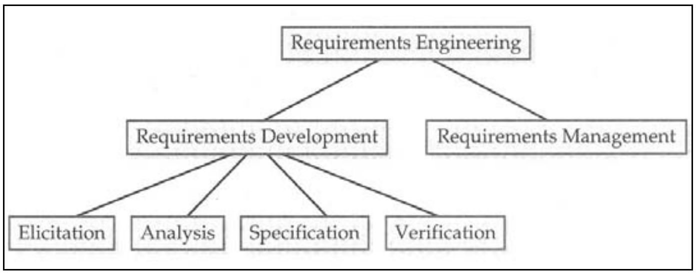


Figure 4.2: Hierarchical decomposition of Requirements Engineering by Wiegers [*Wiegers99:19*]

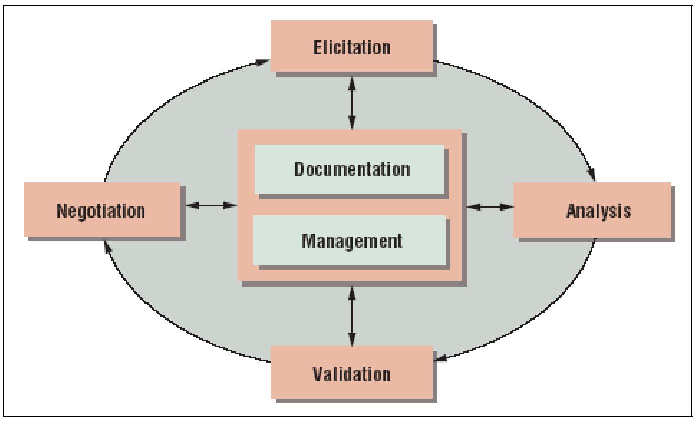


Figure 4.3: Requirements engineering activity cycle by Sommerville [*Sommer05:17*]

THIS THESIS builds up on both ideas and uses the following approach (also inspired by Maciaszek [*Macias01:59*]): Requirements Engineering is the entire discipline that comprises ALL requirements-related activities. Requirements Management is in the middle of this discipline, and like Software Project Management in Software Engineering, Requirements Management is responsible to coordinate and manage all the other requirements-related activities, and to align them with the superior SWPM.

Ebert points out, that Requirements Engineering is related to many other disciplines like *Systems Engineering, Computer Science and Project Management* [*Ebert05:18*] – one more reasons to equate the relation between Requirements Engineering and Requirements Management with the relation between SE and SWPM. Basically Requirements Engineering is not a software-specific task (and happens also in other engineering disciplines), but this thesis focuses on Requirements Engineering in software projects.

Requirements also had a leading role in the evolution of SE (see 2.2.6), as it was the *Waterfall Model* that displaced the "build-and-fix"-approach by establishing the "*requirements first*"-principle (with the well-known "Requirements Phase" at the beginning of a project). Later *iterative, incremental and evolutionary approaches* tried to minimise the negative side-effects of the sequential Waterfall and created possibilities to react to changing requirements even when the project was already on its way. *Risk-driven* (and still iterative) approaches brought some prioritisation in those iterations and the underlying decisions. Today *agile approaches* claim to be the silver bullet for changing requirements. But as agile approaches try to push back documentation as much as possible, they also seem like a return to those days in SE when a common joke was: "*Don't worry about that specification paperwork. We'd better hurry up and start coding, because we're going to have a whole lot of debugging to do*" [*Boehm84*:75].

So there are also various forms of Requirements Engineering in SE, depending on the chosen and tailored SE-method or Process Model for a specific project (see figures 2.13 and 2.14). While the RUP constitutes "*a precise distinction between the requirements and analysis workflows*" [*GheJaz03:444*], eXtreme Programming (XP) collects user stories (instead of a well defined requirements specification) and manages them within the "planning game" (see 2.3). According to Sommerville, XP doesn't try to anticipate changes (like plan-driven methods) and therefore future changes can weaken the entire software in a way, that further changes become more and more difficult [*Sommer06:435*]. XP softens this side-effect with regularly "refactoring" to simplify the software. But currently the relation between agile methods and "traditional" Requirements Engineering is still a quite new topics with various interesting approaches (see [*Macias01:34ff*], [*ZuserGre04:247ff*], *LeffWid04:383ff*], [*Ebert05:245ff*]), and as stated in 2.3, the future may bring a reasonable balance between plan-driven and agile methods. Therefore this thesis focuses on the traditional approaches in Requirements Engineering as defined in the SWEBOK [*SWEBOK04:Chapter2*]. Even the "criticised" SWEBOK (see 2.2.2) is balanced enough to forewarn its readers that "*a risk inherent in the proposed breakdown is that a waterfall-like process may be inferred*". So SWEBOK defines a generic Requirements Process and states:

> "*The process-based breakdown reflects the fact that the requirements process, if it is to be successful, must be considered as a process involving complex, tightly coupled activities (both sequential and concurrent), rather than as a discrete, one-off activity performed at the outset of a software development project.*" [*SWEBOK04:2-1*]

But SEWBOK also states about requirements-related activities: "*It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly.*" [SWEBOK04:2-1]. So the problems of appropriate Requirements Engineering in software projects are crucial for the entire Software Engineering discipline. Wiegers therefore criticises that "*educational curricula favour technical topics over the softer requirements issues*" [*Wiegers99:vii*] and that "*many organizations still apply ad hoc methods for these essential project functions*" [*Wiegers99:5*].

4.3.1. Requirements Management: Managing all requirements-related activities

As stated above, this thesis assumes that Requirements Management is in the center of Requirements Engineering (as illustrated in figure 4.3), so it coordinates and manages all requirements-related activities. Maciaszek supports this view (while other sources limit it to managing requirement changes):

> "*Requirements have to be managed. Requirements management is really a part of an overall project management. It is concerned with three main issues:*
>
> 1. *Identifying, classifying, organizing, and documenting the requirements.*
> 2. *Requirements changes (i.e. with processes that set out how inevitable changes to requirements are proposed, negotiated, validated, and documented).*
> 3. *Requirements traceability (i.e. with processes that maintain dependency relationships between requirements and other system artefacts as well as between the requirements themselves)*" [*Macias01:59f*]

Leffingwell points out, that Requirements Management is an ongoing process during the entire project and defines it as "*a systematic approach to eliciting, organizing, and documenting the requirements of the system, and a process that establishes and maintains agreement between the customer and the project team on the changing requirements of the system*" [*LeffWid04:383ff*] (see also figure 4.4). As stated above, Requirements Management is responsible to align all requirements activities with the entire SWPM, as "*specific management issues for requirements development arise in connection with planning; monitoring progress; controlling changes*" [*Hull02:164*].

This *relation between requirements, Requirements Management and SWPM* is quite complex: Important decisions (and problems) in the requirements-domain automatically involve the requirements management and maybe also the superior SWPM – at least when facing decisions or problems with the *Business Requirements* or *Project Scope*, which are highly relevant for the entire project planning (time schedules, cost estimation, iteration planning, etc.) [*LeffWid04:207ff*], [*Wiegers99:95ff*]. Wiegers therefore states for bigger changes: "*Renegotiate project commitments when requirements change*" [*Wiegers99:53*]. In contrast, the SWPM (and also the Requirements Management) should not interfere with every detailed requirements- and design-decision, e.g. size and colour of a button (SWPM-relevant only if related discussions harm the entire project). Typical "engineering" tasks are "*elicitation, analysis, specification and verification*" of requirements – Wiegers defines this as "*Requirements Development*" [*Wiegers99:20*].

Requirements Management is also responsible for tailoring the Requirements Engineering process which "*varies immensely depending on the type of application being developed, the size and culture of the companies involved, and the software acquisition processes used.*" [*Sommer05:16*]. Sommerville explains this more practical: "*For large military and aerospace systems, there is normally a formal RE stage in the systems engineering processes and an extensively documented set of system and software requirements. For small companies developing innovative software products, the RE process might consist of brainstorming sessions, and the product "requirements" might simply be a short vision statement of what the software is expected to do.*" [*ibid.*]. Also the requirements-related roles depend on the project size and team organisation, which determine, if there is a specialised "*business analyst*" or if other team members fulfil this role. This relation acts also in the opposite direction: If the requirements (especially business requirements) are uncertain or "volatile" at the beginning of a project, one may chose a more iterative-incremental approach as if they are fully clear (see figure 2.14). So Requirements Management implies also team activities [*LeffWid04:33ff*], high-level negotiations and organisational questions like choosing an appropriate Process Model and a software tool for the administration of all requirements activities. So a lot of SWPM-relevant skills are also relevant in this domain (see chapter 3).

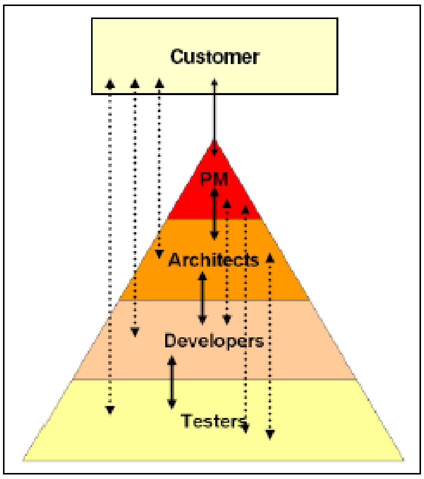


Figure 4.4: Requirement communication in a diversified development team [*MikHeis06:922*]

4.3.2. Requirements Elicitation

“*I just wanted to ask a few questions*
... Just one more thing”
(Peter Falk as “*Columbo*”)

As described in 4.2, requirements are the primary link between the “*problem domain*” and the “*solutions domain*”. Requirements transform (often) diffuse visions and needs for a new (software) product into (more or less) precise descriptions of the required product. Ideally they define every detail of the desired product in an unambiguous way, so that those details can be analysed and transformed into a design, which is then implemented, tested and delivered. In practice, the achievement of such well-defined requirements is hard work. As Wiegers states: “*Don't expect your customers to present the requirements analyst with a succinct, and well-organized list of needs*” [*Wiegers99:139*] (see also the Ebert-example in 4.2).

SE-textbooks use the verb “elicit” for this complex and strongly human-oriented task, so “*Requirements Elicitation*” has been coined as a technical term to describe all activities which are necessary to gain utilisable requirement definitions from the costumer. Every *domain, project, product and client* is different – and so is Requirements Elicitation. Project size and project domain are influential factors for the necessary workflow. In a very technical domain – when software engineering is part of systems engineering – the developers may get very good requirements from their users (which are also engineers). In a pure business context this can be completely different, as managers often don't speak the “language” of engineers – and vice versa. But they have to find a common language, as otherwise the engineer will not understand the visions and needs of the client – and therefore fail to deliver the right product. Ebert emphasises the importance of a goal or vision for the product before collecting requirements, because clearly defined goals always encourage target-oriented work in a project [*Ebert05:87f*]. Otherwise – if vision and scope of the project are not precisely defined and not limited by the underlying contract – requirements will be developed “on the fly”, and may grow uncontrollable. As described in 2.4.3, the project vision will often be dominated by a Business Vision, which is useful for writing a vision-document at the beginning, but may also limit the project [*ZuserGre04:224*]. The business vision may also be the result of a compromise, and the underdog may become a “negative stakeholder”. One has to be prepared for everything, as every new project is a different adventure.

According to Wiegers there are three levels of requirements (see 4.2): "*business, user, and functional. These come from different sources at different times during the project, have different audiences and purposes, and need to be documented in different ways*" [*Wiegers99:43*] (other sources call the business requirements a "*concept*" and distinguish more carefully between the activities before and after signing the contract [*Benn95:48*]).

So in a new project, the elicitation of business requirements is normally the first step in Requirements Engineering. Often a preliminary form of elicitation in a preceding feasibility (or pilot) study can take place before the official contract is signed. Once again: Every project is different, size and domain will also strongly influence, when and how the three levels of requirements can be elicited: "*Elicitation, analysis, specification, and verification don't take place in a tidy linear sequence: these activities are interleaved, incremental, and iterative*" [*Wiegers99124*]. Also the responsibilities have to be defined in advance, because figure 4.4 demonstrates how many communication channels between customer and project team exist in a complex software project. So SWPM should define firm responsibilities.

Once the project vision is defined and the contract is signed, normally the detailed requirements elicitation begins. An important starting point (maybe already done before) is the identification, analysis and prioritisation of client stakeholders who will be "sources" of requirements (once again: firm responsibilities – who is a source and who takes decisions – help to avoid later problems).

The elicitation is often conducted by a business analyst who "*discovers the system requirements through consultation, which involves customers and experts in the problem domain*" [*Macias01:49*]. In smaller projects a team member may perform this role. The business analyst has to delve into the DOMAIN, its STAKEHOLDERS and their NEEDS to elicit the necessary REQUIREMENTS and capture "*the unique character of the organization – the way the business is done here and now or how it should be done*" [*ibid.*]. There are various methods of requirements elicitation with specific advantages and disadvantages [*Schach96:198f*], [*Macias01:50ff*], [*Hull02:197ff*], [*LeffWid04:87ff*], [*SWEBOK04:2-5f*], [*Mayr05:137ff*]:

- *Structured (formal)* or *unstructured (informal)* interviews with customers and domain experts: "*Interviews with domain experts are frequently a simple knowledge transfer process. … Interviews with customers are more complex. Customers may have only a vague picture of their requirements. They may be unwilling to cooperate or be unable to express their requirements in understandable terms. They may also demand requirements that exceed the project budget or that are not implementable. Finally, it is likely that the requirements of different customers may be in conflict*" [*Macias01:50f*];
- Questionnaires: *Efficient way of gathering information from many customers. (Normally used in addition to interviews. "Passive", therefore no clarification possible)*;
- Passive, active or explanatory observation: "*Observing the process*";
- Study of documents and existing software systems: *forms, policy plans, minutes of meetings, official correspondence, computer screens, manuals, business domain journals, etc.*;
- Prototyping: "*most frequently used method of modern requirements elicitation*" [*Macias01:54*];
- Brainstorming and idea reduction: *Idea generation and idea reduction; combines various ideas*;
- Requirements Workshops: *Gathers all stakeholders together. Short but intensely focused period*;
- Scenario exploration techniques like *passive, active or interactive* storyboarding;
- Schach states: "*The most accurate and powerful requirements analysis technique is rapid prototyping*" [*Schach96:199*] (which is maybe true, but it is also more expensive). Maciaszek proposes *Joint application development (JAD)* and *Rapid application development (RAD)* [*Macias01:56*].

It is obvious, that those people who perform the requirements elicitation should have communicative competencies and also some of the other “soft skills” as described in chapter 3.1.4. The elicitation methods presented above cover a wide spectrum, and one should know when to apply which technique. Figure 4.5. exemplifies those considerations about the appropriate communication channel.

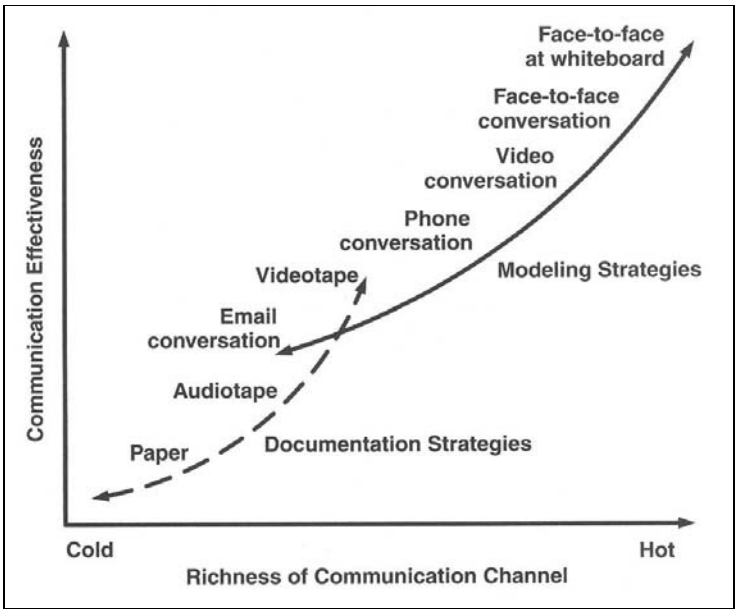


Figure 4.5: Comparing the effectiveness of different communication channels [*AmNa05:42*]

4.3.3. Requirements Analysis and Modeling

After the requirements have been elicited in an informal (narrative) or formal way, they are analysed, classified and models are derived from them (ER-diagram, Data Flow Diagram, etc. [*Wiegers99:176ff*]). Sommerville gives a practical definition “*Analysis: Understand the requirements, their overlaps, and their conflicts.*” [*Sommer05:16*], while the *IEEE Glossary for SE-Terminology* defines *requirements analysis* as.

> “*(1) The process of studying user needs to arrive at a definition of system, hardware, or software requirements.*
> *(2) The process of studying and refining system, hardware, or software requirements*” [*IEEE90:62f*]

As stated in 4.3.2, requirements are often *ambiguous, hardly understandable or even conflicting* – all this can be accidental (different stakeholders have different “*viewpoints*”) or in the worst case the involved stakeholders can even be contradictory [*GheJaz03:393*]. Therefore a requirements analysis takes place and transforms the various inputs into a homogenous model of the desired system (if possible) – or detects and highlights existing contradictions, “*errors, omissions or other deficiencies*” [*Wiegers99:46*]. In the end – combined with the other requirements engineering activities and after several iterations – there should be a coherent view to form a basis for the subsequent design activities.

The traditional view is that the requirements analysis is still part of the “problems domain” and should not forestall the “solution” (design activities): “*Their purpose is to aid in understanding the problem, rather than to initiate design of the solution*” [*SWEBOK04:2-6*]. This strict separation has become increasingly difficult in times of object-oriented analysis & design techniques (often based on UML for both domains) and agile approaches without much formal specification and strong developer-user-interaction.

4.3.4. Requirements Specification and Documentation

When the detailed requirements of the desired software system (*user requirements, system requirements*) have been elicited and analysed, they are documented. Normally they are consolidated in the Requirements Specification Document "*which describes what the analysis has produced*" [*GheJaz03:393*]. Wiegers calls this the Software Requirements Specification (SRS) [*Wiegers99:148*]. Afterwards this document has to be verified and discussed with the relevant stakeholders (validation, negotiation) as it is later (in a stable version) used to develop the design for the new system. Therefore one "*must document them [the requirements, ed.] in some consistent, accessible, and reviewable way*" [*Wiegers99:48*].

A recommendation is made by Sommerville: "*Write down the requirements in a way that stakeholders and software developers can understand*" [*Sommer05:17*]. This helps to avoid later problems, because users are not used to specification documents and their "language". So maybe errors are detected very late (e.g. when the first prototype is presented) and have to be corrected with much more effort.

Style and extensiveness of this document depend on the specific domain, size, type and criticality of the desired software system (see also 4.3.1). A checklist recommends covering (at least) information about the DOMAIN, FUNCTIONAL REQUIREMENTS, NONFUNCTIONAL REQUIREMENTS, and PROCESS REQUIREMENTS [*GheJaz03:394*]. A glossary for the domain-specific vocabulary and a list with "open questions" is also highly recommended [*ZuserGre04:226*].

Each Process Models provides its own templates for the specification document(ation), e.g. the RUP is mainly based on Use Cases and UML-diagrams. An overview about the requirements-artefacts in the RUP provide [*Kruchten04:157ff*], [*ZuserGre04:225ff*]. The style of this document is formal and SWEBOK states: "*Software requirements are often written in natural language, but, in software requirements specification, this may be supplemented by formal or semi-formal descriptions … notations should be used which allow the requirements to be described as precisely as possible … the choice of notation is often constrained by the training, skills and preferences of the document's authors and readers.*" [*SWEBOK04:2-8*]. A readable survey can be found in [*Ebert05:107ff*].

To avoid ambiguities, diagrams and graphical models should be used for overviews and explanations. Currently the comprehensive Unified Modeling Language (UML) is an internationally established notation for this. In 1998 the IEEE has released a "*Recommended Practice for Software Requirements Specification*" [*IEEE98a*], including an extensive proposal for the requirements specification document.

Wiegers defines "*Characteristics of excellent requirements*" for the:

- requirement statements (*complete, correct, feasible, necessary, prioritised, unambiguous, verifiable*);
- requirement specification (*complete, consistent, modifiable, traceable*) [*Wiegers99:16*].

A similar list with alternative priorities is given in [*Hull02:89f*]. Maciaszek also recommends, that all requirements should be clearly identified, numbered and hierarchically structured [*Macias01:60f*]. He proposes the use of a requirements dependency matrix (or interaction matrix) to detect and control conflicting or overlapping requirements [*Macias01:58f*]. This already anticipates the ideas of traceability and change management.

4.3.5. Requirements Verification, Validation and Negotiation

VERIFICATION is performed within the project team and means that the specification document is reviewed, to check if it was created or updated correctly and meets all formal criteria ("*Check that the requirements document meets your standards*" [*SomSaw97:192ff*]). Wiegers defines: "*Verification activities ensure that requirement statements are accurate, complete, and demonstrate the desired quality characteristics*" [*Wiegers99:49*].

VALIDATION takes place with the client (stakeholders) and guarantees that the needs of the client are fully understood and the right product will be delivered. It is normally performed with reviews or formal inspection and misunderstandings should be clarified before design starts: "*Validation: Go back to the system stakeholders and check if the requirements are what they really need.*" [*Sommer05:16f*].

NEGOTIATION is necessary when conflicting requirements have been detected ("*There will always be conflicts, overlaps and omissions in any set of requirements*" [*SomSaw97:125*]). Requirements can conflict with each other, or they can also be in conflict with the project scope and vision. These conflicts are often swept under the carpet [*SomSaw97:125*], but they have to be resolved, because "*You can specify a product vaguely, but you can't implement it vaguely*" [*DeMaLi03b:101*]. Conflicts can happen on different requirement levels, and therefore their solution may involve only the analyst – or in case of serious trouble also the project management. Normally trade-off decisions are made to solve such conflicts (see 4.3.1 and 6.).

"*How do you know when you are done?*" is always a tricky question in requirements development. Wiegers states "*No simple, clear signal will indicate when you're done gathering requirements … you'll never be completely done*" [*Wiegers99:143f*], but identifies some indicators for an end: *Users can't think of any more use cases, new use cases are redundant, new use cases are out scope (therefore a scope is important), new use cases become more and more "low priority" functions, users propose functions that can be implemented "sometime in the lifetime of the product".*

4.3.6. Change Management and Requirements Traceability

"*Change is not a kick in the teeth, but unmanaged change is*" [*Macias01:61*]

Change Management and *Requirements Traceability* are an integral part of *Requirements Management* and have been already defined in subchapter 4.3.1. Every domain has its own characteristics, so "*the formality with which change is managed will depend upon the nature and state of the project*" [*Hull02:31*].

To some extent, changing requirements are normal in a software project, but to keep them under control the essential questions are: WHICH requirements LEVEL (*Business, User, System*) is affected?; WHEN does a change request occur? (Before or after the related part is already designed or even implemented); What is the IMPACT (costs, effort, etc.) of the change?: "*During the early stages, changes can and must be made with ease so that progress can be made. However, there comes a time at which a commitment must be made and formal agreement struck. From this time, it is usual to have a more formal arrangement in which changes are not just inserted at the whim of anyone on the project. Instead a process is used in which changes are first requested or proposed and then they are decided upon in the context of their impact on the project*" [*Hull02:31*]. It depends on the nature of the project, who takes this decision, but in serious cases (changing business requirements) the project management is normally involved. Like SWPM, Change Management is strongly influenced by the "*human factor*", as it implies many tricky trade-off decisions and is affected by (requirements) risks and stakeholders. These aspects will be pursued in the chapters 4.4. and 5.

But Change Management involves also a lot of data and the administration of manifold dependencies [*Wiegers99:297ff*]. Therefore *Requirements Traceability* is a supporting activity which deals with the storage and administration of all relevant information about requirements relationships and their changes [*Macias01:62*]. Ebert defines *three relevant traceability relationships* which have to be tracked [*Ebert05:179ff*]:

- Requirement–Requirement *("Horizontal", even if different requirement levels are involved*);
- Requirement–Other Product *("Vertical": Related design modules, code, classes, test cases*);
- Requirement–Source (*Responsible stakeholder/s*).

These many-to-many relationships have to be examined or even updated in case of a change (request) [*Hull02:141ff*]. In big or difficult projects their administration can be quite complex, often a *traceability matrix* is used for this purpose. Therefore a premise for Requirement Traceability are clearly identified, numbered and hierarchically structured requirements and products (see 4.3.4, [*Macias01:60f*]). Requirements Traceability is therefore normally supported by an own database and/or CASE-tool.

4.3.7. Requirements Engineering Good Practices

Wiegers defines in his textbook a "*Requirements Bill of Rights*" and a "*Requirements Bill of Responsibilities*" for Software Customers [*Wiegers99:27*] – a kind of Code of Ethics (see 2.2.2) for Requirements Engineering. He also presents the most comprehensive list about "*Requirements Engineering Good Practices*" of all examined textbooks (more than 40 practices in seven categories), which is worth to be summarised (as it will be addressed later) [*Wiegers99:38f*]:

(1) KNOWLEDGE: *Train requirements analysts, Educate user representatives and managers about requirements, Train developers about the application domain, Create a glossary.*

(2) REQUIREMENTS MANAGEMENT: *Define a change control process, Establish a change control board, Perform change impact analysis, Trace each change to all affected work products, Baseline and control versions of requirements documents, Maintain change history, Track requirements status, Measure requirements stability, Use a requirements management tool.*

(3) PROJECT MANAGEMENT: *Select appropriate life cycle (Process Model), Base plans on requirements, Renegotiate commitments, Manage requirements risks, Track requirements effort.*

(4) ELICITATION: *Vision and scope (document), Define requirements development procedure, Identify user classes, Select "product champions", Establish focus groups, Identify use cases, Hold JAD sessions, Analyse user workflow, Define quality attributes, Examine problem reports, Reuse requirements.*

(5) ANALYSIS: *Draw context diagram, Create prototypes, Analyse feasibility, Prioritise requirements, Model the requirements, Create a data dictionary, Apply Quality Function Deployment.*

(6) SPECIFICATION: *Adopt a Software Requirements Specification template, Identify sources of requirements, Label each requirement, Record business rules, Create requirements traceability matrix.*

(7) VERIFICATION: *Inspect requirements documents, Write test case from requirements, Write a user manual, Define acceptance criteria.*

Other significant sources for "best practices" in Requirements Engineering are: [*SomSaw97*], [*Hull02*], [*LeffWid04*]. A more academic perspective can be found in [*Ebert05*].

4.4. Significant risks related to Stakeholders and Requirements

Significant risks in software projects have been discussed in chapter 3.2.1 and it was already stated in the introduction of chapter 4, that requirements-related tasks are one of the most critical activities in software development. Many studies are quoted by various sources, all with one message: "*Many of the problems encountered in software development are attributed to shortcomings in the processes and practices used to gather, document, agree on, and alter the products requirements … the problem areas might include information gathering, unstated or implicit functionality, unfounded or uncommunicated assumptions, inadequately documented requirements, and a casual requirements change process*" [*Wiegers99:4*] (see also [*Hull02:3f*], [*Ebert05:23ff*]).

A specific problem with requirements errors is, that requirements are the foundation of the entire development process, and the later an error is detected, the more it costs: "*If requirements errors can be fixed quickly, easily, and economically, we still may not have a huge problem*" [*LeffWid04:10*]. In 1981 Boehm "*found that correcting a requirement error discovered after the product was in operation cost 68 times as much as correcting an erroneous requirement during the requirements phase. More recent studies suggest this defect-cost amplification factor can be as high as 200*" [*Wiegers99:15*]. Therefore the listings about significant SE-risks (see 3.2.1) identify requirements-related risks as one of the most important factors for software project failures (the Top-5-list of DeMarco and Lister includes two requirement-issues: *Specification breakdown* and *Scope creep*).

(Requirements) Risk Management has to detect such risks, prioritise them (by probability and impact) and address the most important risks early (see 3.2.2). So Requirements Management and Software Project Management (SWPM) are intimately connected in fulfilling this "risky" task (see 4.3.1). Wiegers states: "*Because requirements play such a central role in software projects, the prudent project manager will identify requirement-related risks early and control them aggressively*" [*Wiegers99:79*]. Therefore a survey about significant requirements- and stakeholder-related risks (from relevant textbooks) is given in the following sections.

Why is RE so risky? Cheng and Atlee state that the "*challenges faced by the requirements-engineering community are distinct from those faced by the general software-engineering community, because requirements reside primarily in the problem space whereas other software artifacts reside primarily in the solution space*" (see 4.3.2). Therefore several consequences from this distinction "*cause requirements engineering to be inherently difficult*" [*ChenAtl07*]:

- "*Requirements analysts start with ill-defined, and often conflicting, ideas … and must progress towards a single, detailed, technical specification of the system … The requirements problem space is less constrained than the software solution space … there are many more options to consider and decisions to make about requirements*"; "*Taking into consideration environmental conditions significantly increases the complexity of the problem at hand, since a system's environment may be a combination of hardware devices, physical-world phenomena, human (operator or user) behavior, and other software components.*";
- "*Reasoning about the environment includes identifying not only assumptions about the normal behavior of the environment, but also about possible threats or hazards*";
- "*The resulting requirements artifacts have to be understood and usable by domain experts and other stakeholders, who may not be knowledgeable about computing. Thus, requirements notations and processes must maintain a delicate balance between producing descriptions that are suitable for a non-computing audience and producing technical documents that are precise enough for …developers*"
- "*RE activities, in contrast to other SE-activities, may be more iterative, involve many more players who have more varied backgrounds and expertise, require more extensive analyses of options, and call for more complicated verifications of more diverse (e.g., software, hardware, human) components*"

● *Risks and problems primary related to stakeholders*

Stakeholders – in particular Client Stakeholders – are one of the main reasons why the "human factor" influences software development. Maciaszek states for the domain of Information Systems that they are "*social systems. They are developed by people (developers) for people (customers). The success of a software project is determined by social factors – technology is secondary*" [*Macias01:4*] (which corresponds with the already quoted statement "*The major problems of our work are not so much technological as sociological nature*" [*DeMaLi87:4*]).

Maciaszek provides a list with reasons why projects fail at the customer end [*Macias01:4*]:

- Customer needs are misunderstood or not fully captured;
- Customer requirements change too frequently;
- Customers are not prepared to commit sufficient resources to the project;
- Customers do not want to cooperate with developers;
- Customers have unrealistic expectations;
- The system is no longer of benefit to customers.

Other sources mention the problem of "*adversarial stakeholder relationships*" among client stakeholders [*RoyceWa98:15f, 214*], [*DeMaLi03b:101*]. Reasons for project-failures at the developer end are also well known, but not in the specific focus of this thesis, even if Fröhlich strongly criticises software engineers for commonly shifting the blame to the client [*Fröhlich02:65*]. Considering the ideas of chapter 3.2, the following statement of Wiegers about Stakeholders in software projects is quite balanced:

> "*Nowhere more than in the requirements process do the interests of all the stakeholders in a software project intersect. These stakeholders include customers, users, business or requirements analysts (people who gather and document customer requirements and communicate them to the development community), developers, testers, authors of user documentation, project managers, and customer managers.*
> *Handled well, this intersection can lead to great products, happy customers, and fulfilled developers. Handled poorly, this intersection is the source of misunderstanding, frustration, and friction that can undermine the quality and business value of the final product*" [*Wiegers99:5*]

● *Risks and problems primary related to requirements*

Maciaszek provides a list with "requirements risks" [*Macias01:59*], an adapted version of a similar list in [*SomSaw97:137ff*]. A combination of both lists is given below:

- Technical risks (*requirement is difficult to implement*);
- Performance risks (*requirement can adversely affect the response time of the system*);
- Safety and security risks (*requirement can expose the system to security beaches*);
- Database integrity risks (*requirement can cause data inconsistency*);
- Development process risks (*requirement calls for unconventional development methods*);
- Implementation technology risks (*requirement may require the use of unfamiliar technology*);
- Schedule risks (*requirement may be technically difficult and may threaten the development schedule*);
- Political risks (*requirement may prove difficult to fulfil for internal policy reasons*);
- External risks (*requirement implementation may involve external contractors, etc.*)
- Legal risks (*requirement is in conflict with current or upcoming laws, regulations, etc.*);
- Volatility / Stability risks (*requirement is likely to keep changing or evolving during development*).

● *Risks and problems related to the entire requirements process*

> "*If your organization is serious about software success, it must accept that the days of sliding some vague requirements and a series of pizzas under the door to the programming department are over*" [*Wiegers99:26*]

Some of the risks in the last section point in the direction of the requirements process. So Wiegers calls a section in his book "*When bad requirements happen to nice people*" and provides a list with risks from "*inadequate*" requirements processes that threaten project success ("*success can be defined as delivery of a product that satisfies user expectations of functionality and quality at agreed-on cost and timeliness*") [*Wiegers99:11*]:

- Insufficient user involvement leads to unacceptable products;
- Creeping user requirements contribute to overruns and degrade product quality;
- Ambiguous requirements lead to ill-spent time and rework;
- Gold-plating by developers and users adds unnecessary features;
- Minimal specifications lead to missing key requirements;
- Overlooking the needs of certain user classes leads to dissatisfied customers;
- Incompletely defined requirements make accurate project planning and tracking impossible.

Ebert compiled a list with typical requirements process risks [*Ebert05:26ff*], an extension of an earlier article by him, Lawrence and Wiegers ("*The Top Risks of Requirements Engineering*" [*LawWie01*]):

- Overlooking crucial requirements;
- Modeling only functional requirements;
- Inadequate customer representation;
- Uncontrolled requirement changes;
- Representing requirements in the form of designs;
- Not inspecting or validating requirements;
- Attempting to perfect requirements ("*Gold Plating*"; before beginning construction.

Ebert also presents a list with requirement-related failures which are relevant for SWPM [*Ebert05:68ff*]:

- Ambiguous requirements;
- Changing requirements;
- Unstable product- or design-basis (in case of software evolution);
- Ad-hoc Requirements Management with unclear responsibilities;
- Gap between customer expectations and project scope;
- Insufficient customer management;
- Aggressive project definition with unachievable milestones;
- Superficial or inaccurate effort- and impact-estimations;
- Project plans not observed;
- Uncontrollable subcontracts with external suppliers.

Many of these risks can lead to the known "*moving target syndrome*" [*Benn95:15*], [*Schach96:469f*]. Therefore requirements management and SWPM have to recognise early symptoms and address those problems.

● ***Requirements uncertainty and volatile requirements***

Changing or volatile requirements are normal in Requirements Management (see 4.3.6), and Sommerville states: "*Requirements change is inevitable, because the business environment in which the software is used continually changes—new competitors with new products emerge, and businesses reorganize, restructure, and react to new opportunities. Furthermore, for large systems, the problem being tackled is usually so complex that no one can understand it completely before starting system development. During system development and operational use, your stakeholders continue to gain new insights into the problem, leading to changes in the requirements*" [*Sommer05:18*].

Different types of volatile requirements are defined in [*SomSaw97:249ff*]: MUTABLE requirements (*due to changes in the environment*), EMERGENT requirements (*emerge as the system is designed and implemented*), CONSEQUENTIAL requirements (*assumptions on how the system will be used turn out to be wrong*), COMPATIBILITY requirements (*depend on other equipment or processes*).

Requirements uncertainty can also become a big problem: "*A key reason for project failures is insufficient management of changing requirements during all stages of the project life cycle.*" [*EberMan05:553*]. Important questions – Which requirement level is affected? When does a change request occur? – have already been addressed in chapter 4.3.6. The crucial question for Requirements Management, Risk Management and SWPM is also: *WHY do requirements change? Are there early warning signs?* Ebert and Man have tried to find root causes and early symptoms for requirements uncertainty and subsequent delays in software projects (see figure 4.6) – and an absent vision about the project is a leading factor [*EberMan05*].

This field his highly unpredictable and full of "*undiscovered ruins*" [*LeffWid04:91*], so assume: "*You will not get change prediction 100% right and you will not be able to identify all volatile requirements*" [*SomSaw97:250*]. Change Management has a "bureaucratic" side: Administration and tracing of change requests and related requirements. But it is also a human-centred activity, full of tricky trade-off decisions which will be discussed in chapter 5. Every project has its own nature and the Project Manager will have to find an appropriate balance between "freezing" requirements and accepting reasonable changes even late in the project to satisfy the customer. "*During the development of stakeholder requirements there will be a period of rapid and intense change. At this stage it is not sensible to have a formal change control process in place, because the situation is too dynamic … However, at some point stability will begin to emerge and the requirements manager can determine when the requirements are sufficiently stable to subject further changes to a more formal process*" [*Hull02:168*].

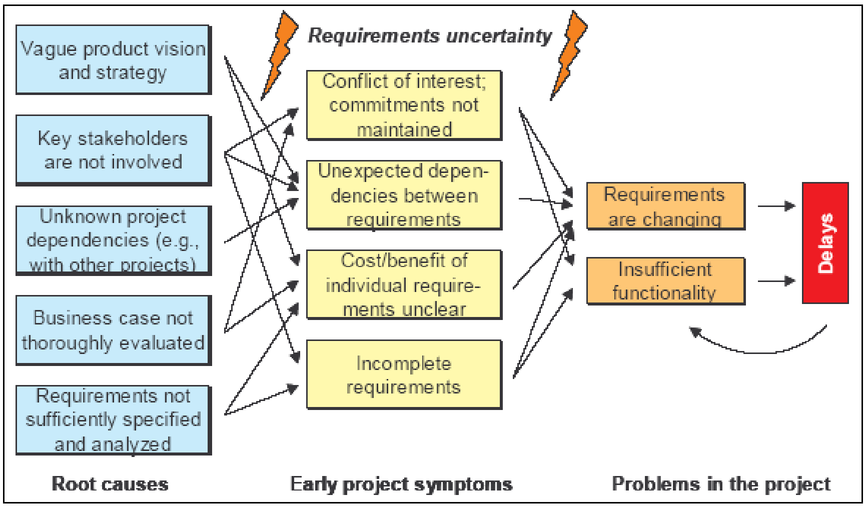

Requirements uncertainty
Vague product vision and strategy
Key stakeholders are not involved
Unknown project dependencies (e.g., with other projects)
Business case not thoroughly evaluated
Requirements not sufficiently specified and analyzed
Conflict of interest; commitments not maintained
Unexpected dependencies between requirements
Cost/benefit of individual requirements unclear
Incomplete requirements
Requirements are changing
Insufficient functionality
Delays
Root causes
Early project symptoms
Problems in the project

Figure 4.6: Root causes of delays from requirement uncertainty [*EberMan05:556*]

5. REQUIREMENT TRADE-OFFS AMONG CLIENT STAKEHOLDERS

> *"You can specify a product vaguely,*
> *but you can't implement it vaguely"*
> [*DeMaLi03b:101*]

Software Project Management (SWPM) is full of trade-off decisions (see chapter 3), and many of them are directly related to requirements. Already in their influential paper about "*Theory-W*", Boehm and Ross addressed these decisions (see figure 3.4) and proposed the (meanwhile well-known) "*WinWin-approach*" to satisfy all stakeholders [*BoehRos89*]. Their negotiation-strategy has (basically) four steps:

(1) Separate the people from the problem.
(2) Focus on interests, not positions.
(3) Invent options for mutual gain.
(4) Insist on using objective criteria.

In 2000, Boehm addressed these problems again, and presented a more extensive "*Model-Clash Spiderweb diagram*" which identifies main trade-offs and conflicting interests between four groups of stakeholders: *Users, Acquirers, Maintainers and Developers* (see figure 5.1). Most of these decisions are relevant for Requirements Management and can influence the requirement risks discussed in chapter 4.4. Therefore SWPM and Requirements Management have to cooperate to avoid requirement risks induced by such trade-offs, in particular for decisions which have to be made among adversarial client stakeholders.

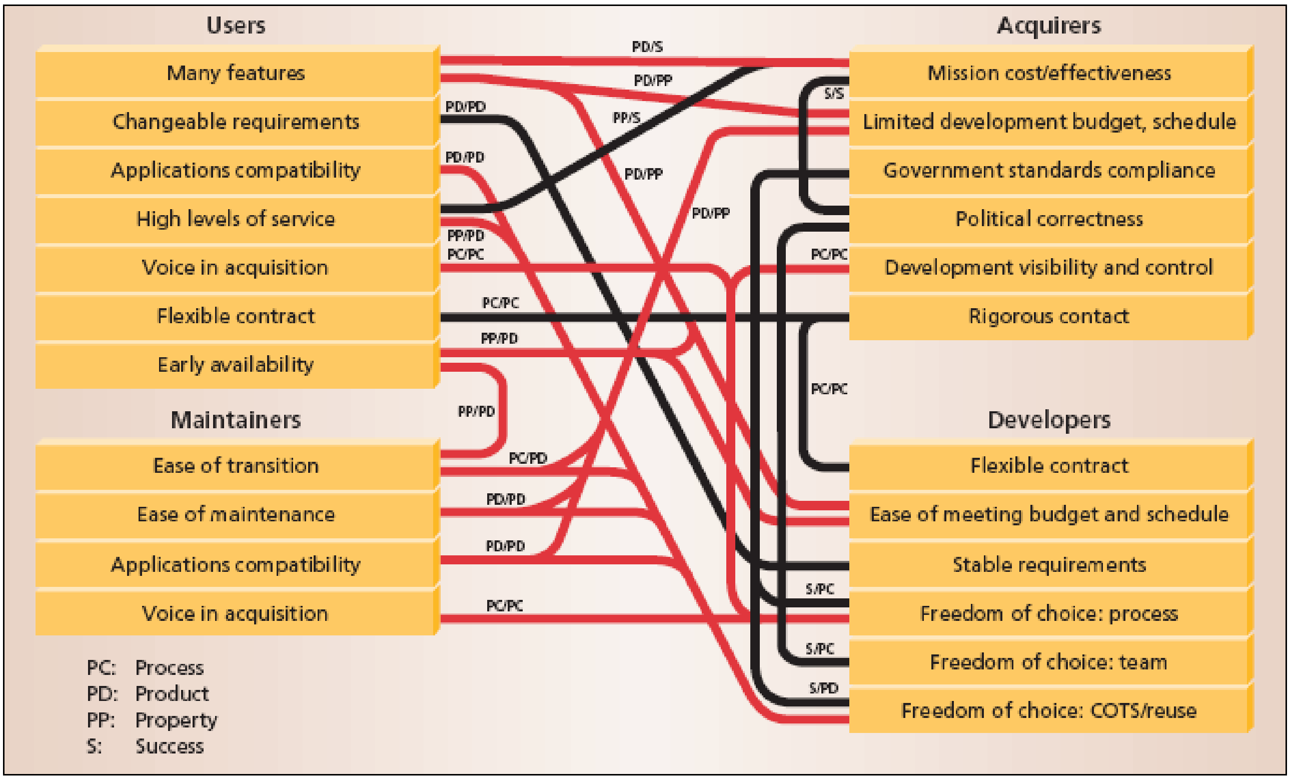


Figure 5.1: Model-Clash Spiderweb diagram [*BoehPort00:121*]

The importance of stakeholders and their interests for the discipline of Requirements Engineering was recognised by the last (14th) *IEEE Requirements Engineering Conference (RE06)* in September 2006, which was focused on this inseparable relation between stakeholders and requirements [*GlinzWie07:18*]. Its title "*Understanding the stakeholders' desires and needs*" addressed the fact, that Requirements Engineering should primarily satisfy client stakeholders. Wiegers states: "*Nowhere more than in the requirements process do the interests of all the stakeholders in a software project intersect*" [*Wiegers99:5*]. So more and more sources state that "*Stakeholders are a recognized source of significant software project risk*" [*Woolr07:36*].

Therefore textbooks about Requirements Engineering cover by default trade-off-issues. There are technical and quality trade-offs, concerning decisions about conflicting quality attributes (e.g. *Flexibility vs. Efficiency, etc.*) [*Wiegers99:204ff*]. There are also trade-offs with regard to conflicting, overlapping or "questionable" requirements – user- or system-requirements. All this is not in the focus of this thesis.

But there is a third kind of requirements trade-offs which is increasingly addressed by scientific sources and which is the final focus of this thesis: Trade-offs among client stakeholders concerning the BUSINESS REQUIREMENTS, less diplomatic called "CONFLICTING GOALS". These issues are more complicate than the others and affect the entire project much more than the "details" mentioned above. In the worst case they can challenge the entire project and SWPM can get a very rough task in such an environment. One paper that addresses these questions – which are strongly related to the "human factor" in SE – was written by DeMarco and Lister. They have found five "core risks" for software projects (see 3.2.1), and one risk which was covered in-depth is the so called "*specification breakdown: failure to achieve stakeholder consensus on what to build*":

> "*In the past, IT projects were most often tasked to satisfy a single user's requirements. They were relatively easy, but sadly we finished all such projects years ago. Today a new IT project is likely to affect several different stakeholders from different parts of the organization, in different locations, with different interests, and little or no common stake. Perhaps the biggest core risk is that these stakeholders will fail to concur on project goals. Our data leads us to expect this to happen to a disruptive extent on approximately one project out of seven.*
>
> *Failure to achieve total concurrence would be no more than an annoyance if all could agree to disagree. We would end up delivering products that satisfied different stakeholders to differing degrees, with no one left completely out. Unfortunately, nonconcurring projects seldom play out this way. The problem is that organizational culture might require all the stakeholders to cooperate, or at least seem to cooperate. This does not make dissent go away, but forces it underground. And dissent always exists—count on it. New IT products introduce change into organizations, and change is never uniform in its impact on different constituencies. Our basic rule is Every time an IT product is delivered, somebody gains power and somebody else loses power. Both the power gainers and the power losers are, by definition, stakeholders. You can expect some stakeholders on any complex project to be adversaries. They might not be allowed to act adversarially, but many other possibilities are open to them.*" [*DeMaLi03b:99ff*]

No other source addressed this problem as direct as this paper, even if [*Woolr07:43*] states (with a reference to [*DeMaLi03b*]): "*Although several stakeholders might react similarly, some might be more able to derail the project – by refusing to make themselves available, withholding approval, using their power to force decisions and priorities, or adding gratuitous requirements to drag the project down*". Software project managers will find themselves in an uncomfortable situation when they do not identify this problem and related risks right in time. And not all of them are as brave as Booch who advocates "*Speaking truth to power*" [*Booch07*].

These problems with conflicting business requirements among client stakeholders are covered by textbooks about Requirements Engineering more diplomatic: Wiegers ties this issue to the excellent idea of a Vision and Scope Document for the project (see 4.2) and states: "*The requirements will never stabilize if the project stakeholders do not share a common understanding of the business needs the product must satisfy and the benefits it will provide*" [*Wiegers99:96*]. Therefore he recommends a clear project vision, which points in the direction of DeMarco & Lister who state: "*Failure to concur is clearly a political matter, not a technical matter. ... Projects that can't achieve a signoff on the boundary census by the approximate 15 percent point probably need to be cancelled*" [*DeMaLi03b:101*].

Sommerville and Sawyer also mention the topic and recommend "*Plan for conflicts and conflict resolution*" [*SomSaw97:125ff*] and "*Be sensitive to organisational and political considerations*" [*SomSaw97:69ff*]:

> "*In spite of years of experience, many organisations still do not allow enough time to resolve requirements conflicts. The reason for this is, perhaps, that conflicts are considered as some kind of 'failure' and it is not accepted to plan for failure. This view is completely wrong. Conflicts are natural and inevitable. They reflect the fact that different stakeholders in the system have different needs and priorities.*" [*SomSaw97:125*]
>
> "*If you understand organisational politics, you are more likely to be able to understand the real rationale for some requirements … When eliciting requirements, there are a number of things you should watch for as these suggest organisational and political influences on the requirements: Conflicting goals … Loss or transfer of responsibility … The organisational culture … Management attitudes and the morale of the organisation … Departmental differences*" [*SomSaw97:69f*]

One particular stakeholder problem in the context of Software Reengineering and Evolution is addresses by Yourdon when stating possible rejections of such a new system by the maintenance programmers of the current legacy system:

> "*I don't need any help: Many maintenance programmers are perfectly satisfied with the level or work the are doing and are sincerely convinced that they are as productive as could be reasonable expected. … Don't dare touch my program: From a negative perspective, we might argue that the program represents job security … The old ways are better*" [*Your92:252*]

Yourdon assumes, that such tricky stakeholder relations are often only resolvable when "*the sole living expert*" retires or quits. Or when the senior management has finally decided to stop being "*held hostage to the whims of a single indispensable person*" [*Your92:258f*]. The same cause for conflicting goals in context of Software Reengineering and Evolution arises, when parts of the senior management feel strongly connected with a legacy system, because they were (or still are) responsible for its development.

So in software engineering practice it is well known, how conflicting interests and trade-off conflicts among client stakeholders can affect the entire project or how they make Business Requirements "volatile". As shown above, some academic sources already cover this issue, but there is still no common classification about such conflicts. Some sources address these problems with "prioritisation"-approaches [*Ebert05:183ff*] and project scope management. At least it is recommended to make a business case or vision document on which everybody has to agree [*Wiegers99:95ff*], [*SomSaw97:49ff*]. Ebert and Gärtner also address the management-related aspects of this topic and recommend best practices how to deal and negotiate with such stakeholders [*Gärtner04:78ff*], [*Ebert05:60ff*]. Chapter 6 provides an overview about useful considerations and tools for software project managers who have to handle "volatile" Business Requirements due to trade-offs among client stakeholders.

6. CONCLUSIONS

"Wir stehen selbst enttäuscht und sehn betroffen, den Vorhang zu und alle Fragen offen."
(Bertolt Brecht, Der gute Mensch von Sezuan, Epilog [*Brecht53*]*)*

The focus of this thesis covers a topical and complex question in Software Project Management (SWPM) which is currently not widely covered by academic sources:

How to handle volatile business requirements induced by trade-offs (or even conflicts) among client stakeholders?

Therefore chapter 3 underlined the manifold challenges that SWPM has to face, including many trade-off decisions – often influenced by the external environment (see 3.2). Chapter 4 introduced the concepts of Stakeholders, Requirements and Requirements Engineering as well as the strong relation between Requirements Management and SWPM. The main risks and problems in Requirements Engineering where addressed in chapter 4.4. Chapter 5 finally combined those risks with trade-offs among client stakeholders in Requirements Engineering on the level of business requirements. Therefore the following conclusions provide a helpful overview for software project managers who have to face similar problems. They are split in two sections:

- Contradictions that SWPM has to consider when facing trade-offs or conflicts
- A Toolbox for software project managers when facing trade-offs or conflicts

6.1. Contradictions that Software Project Management has to consider when facing trade-offs or conflicts

● Decision making: System godfather vs. democratic

Karnovsky strongly advocates to find and fix the "*godfather*" of the project (German: "*Pate*"), thus a high-rank manager on the client side who promotes the project and to whom the project management has direct access – and who can take all critical decisions if necessary. Many other sources recommend a collaborative working style when deciding about requirements (e.g. Gärtner, Wiegers, Ebert).

Both styles can degenerate (leading to another contradiction: Autocratic versus anarchy). Both styles have advantages and disadvantages: System godfather is the easy way and enable fast decisions. But it covers tensions and conflicts, so neglected or adversarial (negative) stakeholders may find other ways to express their opinion and work against the project. This contradiction can be already decided by the contract and the internal structures at the client side, but the respective side-effect will be the same.

- ***Taking sides: Absolute truth vs. tactics***

A delicate question: How much honesty is appropriate when facing internal conflicts among client stakeholders? Can the PM and his team start to collaborate with parts of the client stakeholders to protect the project? Which influences has this "taking sides" for the relationships to other stakeholders? Managers often refer to Machiavelli [*Machia1532*] who recommended partial honesty. Other sources plea for a neutral and collaborative role. SWPM can get strongly involved in internal conflicts when starting to collaborate with one side, but project success may require doing so.

- ***Sophisticated (detailed) organisation and documentation vs. agile methods***

This issue was already discussed in chapter 2.3. Figure 2.14 gives an overview about "home grounds" of each approach. Anyway, agile methods always clame to be more appropiate for volatile requirements, but they have another problem: Who is the client stakeholder that works with the development team? This person has much more influence on the project than all other "relevant" stakeholders and may come in trouble when "his" (or "her") decissions are not supported by the others.

- ***Death March projects: Surrender vs. fight it out***

DeMarco and Lister recommended stopping a project when a certain percentage of agreement about the project goals is not achievable. Wiegers recommend a Vision and Scope document but does not say what to do when this document is not achievable. SWPM has to consider carefully when to stop a project and which measures are tried before to restart or protect the project (also a finance-issue).

- ***Ebert: Paralyse vs. Uncertainty***

Some sources recommend starting a project before requirements problems prevent everybody from doing something [*EberMan05*]. In the same paper, uncertainties are addressed which could be prevented when starting later. So the question is: When is the project determined enough to start?

- ***Software Reengineering and Evolution – a particular problem***

As exemplified in chapter 5 (and anticipated in chapter 2.4.3), projects which build up on the reengineering or evolution of existing legacy systems face particular problems. There may be supporters of the legacy system (technical staff and management staff) which act adversarial to the "new" project. Also related "Business Process Reengineering" may affect client stakeholders who therefore act adversarial on the new software project.

- ***What can Software Project Managers learn from other Engineers?***

Other engineering disciplines face similar problems with complex projects, abstract requirements and trade-offs among their client stakeholders (e.g. architects). What can we learn from them?

6.2. Toolbox: Tools that Software Project Management can use when facing trade-offs or conflicts

- ***Apply the WinWin-approach for negotiations*** [*BoehRos89*], [*Gärtner04:78ff*], [*Ebert05:60ff*]

- ***Apply the Project Environment Analysis ("Projektumfeldanalyse – PUA") at an early stage, in particular for requirements and related stakeholders*** *(see chapter 3.2.3)*

- ***Apply Risk Management to Requirements Management for 5 to 10 important risks***

- ***Use prototyping to make the new product "tangible" and to convince or overcome adversarial stakeholders***

- ***Search for an appropriate Software Development Process Model for the specific project environment and tailor it accordingly*** *(see 2.2.6 and 2.3)*

- ***In complex, big and risky environments: Install a steering-committee that takes binding decisions (even if it is bureaucratic it will help to make conflicts more transparent)***

- ***If desired, use collaborative techniques like stakeholder-meetings, workshops and conferences***

- ***Insist on a Vision and Scope document (or some other form of general agreement about the main goals and limits of the project)***

- ***Use Viewpoints to understand the views of different stakeholders and find compromises***

- ***Include also relations between requirements and negative stakeholders (who oppose this requirement) in Requirements Tracebility***

- ***Make tactical games from negative client stakeholders transparent***

7. REFERENCES

7.1. Reference books

[**AmNa05**] Ambler, Scott W.; Nalbone John; Vizdos Michael J.: *The Enterprise Unified Process: Extending the Rational Unified Process*, 2005, Prentice Hall PTR

[**ApplAus03**] Applegate, Lynda M.; Austin Robert D.; McFarlan F. Warren: *Corporate Information Strategy and Management*, 6th (International) ed.: 2003 (1st ed.: 1983), McGraw-Hill

[**August05**] Augustine, Sanjiv: *Managing Agile Projects*, 2005, Prentice Hall PTR

[**Balzert96**] Balzert, Helmut: "*Lehrbuch der Software-Technik, Bd. 1.: Software-Entwicklung*" (Volume 1), 2nd ed.: 2000 (1st ed. 1996), Spektrum Akademischer Verlag, Heidelberg

[**Baumer02**] Baumer, Thomas: *Handbuch Interkulturelle Kompetenz*, 2002, Orell Füssli Verlag, Zürich

[**Beck00**] Beck, Kent (2nd ed. with: Andres Cynthia): *Extreme Programming Explained – Embrace Change* (Part of "*The XP Series*"), 2nd (revised) ed.: 2005 (1st ed.: 2000), Addison Wesley, (German title: *Extreme Programming: Die revolutionäre Methode für Softwareentwicklung in kleinen Teams*, 2003, Addison Wesley)

[**Bénard92**] Bénard, Christian: *Les 9 points clés de la conduite d'un projet informatique*, 1992, Les Éditions d'Organisation, Paris

[**Benn95**] Bennatan, E. M.: *Software Project Management – A Practitioner's Approach*, 2nd ed.: 1995 (1st ed.: 1992), McGraw-Hill (UK)

[**Boehm89**] Boehm, Barry W.: *Software Risk Management*, 1989, IEEE Computer Society Press

[**Brooks95**] Brooks, Frederick P.: *The Mythical Man-Month: Essays on Software Engineering*, 2nd strongly revised "*20th anniversary*"-edition (including the 1987-essay "*No Silver Bullet - essence and accidents of software engineering*"): 1995 (1st ed. 1975), Addison-Wesley

[**Brown92**] Brown, Ann (Ed.): *Creating a Business-based IT Strategy*, UNICOM-Series: Applied Information Technology, Vol. 14, 1992, Chapman & Hall, London et. al.

[**Brown98**] Brown, William J. et al.: *AntiPatterns – Refactoring Software, Architectures and Projects in Crisis*, 1998, John Wiley & Sons, New York (German title: *AntiPatterns – Entwurfsfehler erkennen und vermeiden*, 2004, mitp-Verlag, Bonn)

[**Cockb01**] Cockburn, Alistair: *Agile Software Development*, 2nd ed.: 2006 (1st ed.: 2001), Addison Wesley (German title: *Agile Software-Entwicklung,* 2003, mitp-Verlag, Bonn)

[**Daven97**] Davenport, Thomas H.: *Process Innovation – Reengineering Work through Information Technology,* Reprint 1997 (1st ed.: 1993), Harvard Business School Press, Boston

[**DeMaLi87**] DeMarco, Tom; Lister Timothy: *Peopleware – Productive Projects and Teams*, 1987, Dorset House Publishing, New York (German title: *Wien wartet auf Dich! – Der Faktor Mensch im DV-Management*, 1991, Carl Hanser Verlag, München-Wien)

[**DeMarco97**] DeMarco, Tom: *Warum ist Software so teuer? ... und andere Rätsel des Informationszeitalters*, 1997, Carl Hanser Verlag, München-Wien (English title: *Why does software cost so much? And other puzzles of the information age*, 1995, Dorset House Publishing, New York)

[**DeMarco01**] DeMarco, Tom: *Spielräume - Projektmanagement jenseits von Burn-out, Stress und Effizienzwahn*, 2001, Carl Hanser Verlag, München-Wien (English title: *Slack – Getting Past Burnout, Busywork, and the Myth of Total Efficiency*, 2001, Random House)

[**DeMaLi03a**] DeMarco, Tom; Lister Timothy: *Bärentango – Mit Risikomanagement Projekte zum Erfolg führen*, 2003, Carl Hanser Verlag, München-Wien (English title: *Waltzing with Bears – Managing Risk on Software Projetcs*, 2003, Dorset House Publishing, New York)

[**Drucker77**] Drucker, Peter Ferdinand: *People and Performance – The Best of Peter Drucker on Management*, 1977, Heinemann, London

[**Ebert05**] Ebert, Christof: *Systematisches Requirements Management – Anforderungen ermitteln, spezifizieren, analysieren und verfolgen*, 2005, dpunkt.verlag; Heidelberg

[**EssMey03**] Essigkrug, Andreas; Mey Thomas: *Rational Unified Process kompakt*, 2003, Spektrum Akademischer Verlag, Heidelberg-Berlin

[**Fairley85**] Fairley, Richard: *Software Engineering Concepts*, McGraw-Hill, 1985

[**Fleissner96**] Fleissner, Peter; Hofkirchner Wolfgang; Müller Harald; Pohl Margit; Stary Christian: *Der Mensch lebt nicht vom Bit allein …*, 2nd ed.: 1997 (1st: 1996), Peter Lang GmbH – Europäischer Verlag der Wissenschaften, Frankfurt am Main et. al.

[**Fröhlich02**] Fröhlich, Adrian W.: *Mythos Projekt – Projekte gehören abgeschafft. Ein Plädoyer*, 2002, Galileo Press, Bonn

[**Gamma94**] Gamma, Erich; Helm Richard; Johnson Ralph; Vlissides John: *Design Patterns – Elements of Reusable Object-Oriented Software*, 1st ed.: 1994, Addison-Wesley

[**Gärtner04**] Gärtner, Johannes: *Realistisches Projektdesign – Projektarbeit in einer wenig berechenbaren Welt*, 2004, vdf Hochschulverlag, ETH Zürich

[**Gareis04**] Gareis, Roland: *Happy Projects!*, 3rd (German) ed.: 2006 (1st ed.: 2003), Manz Verlag, Wien

[**GerlGerl05**] Gerlich, Rainer; Gerlich Ralf: *111 Thesen zur erfolgreichen Softwareentwicklung*, 2005, Springer Verlag, Berlin-Heidelberg

[**GheJaz03**] Ghezzi, Carlo; Jazayeri Mehdi; Mandrioli Dino: *Fundamentals of Software Engineering*, 2nd (International) ed.: 2003 (1st ed.: 1991), Pearson Education @ Prentice-Hall

[**Glass98**] Glass, Robert L.: *Software Runaways – Lessons Learned from Massive Software Project Failures*, 1998, Prentice Hall PTR

[**Hall92**] Hall, P. A. V. (Ed.): *Software Reuse and Reverse Engineering in Practice,* UNICOM-Series: Applied Information Technology, Vol. 12, 1992, Chapman & Hall, London et. al.

[**HamCha94**] Hammer, Michael; Champy James: *Reengineering the Corporation: A manifesto for business revolution*, 1994, Harper Business, New York

[**Herkner91**] Herkner, Werner: *Lehrbuch Sozialpsychologie*, 5th (revised) ed.: 1991 (1st: 1975: *Einführung in die Sozialpsychologie*), Verlag Hans Huber, Bern et. al.

[**HofWeis01**] Hoffman, Daniel M.; Weiss David M. (Eds.): *Software Fundamentals – Collected Papers by David L. Parnas*, 2001, Addison-Wesley

[**HorHill89**] Horowitz, Paul; Hill Winfield: *The Art of Electronics*, 2nd (revised) ed.: 1989 (1st: 1980), Cambridge University Press

[**Hull02**] Hull, Elizabeth; Jackson Ken; Dick Jeremy: *Requirements Engineering*, 2002, Springer Verlag, London

[**JacoBo99**] Jacobson, Ivar; Booch Grady; Rumbaugh James: *The Unified Software Development Process*, 1999, Addison Wesley

[**Kappel04**] Kappel, Gerti; Pröll Birgit; Reich Siegfried et. al. (Eds:): *Web Engineering – Systematische Entwicklung von Web-Anwendungen*, 2004, dpunkt Verlag, Heidelberg

[**Karnov02**] Karnovsky, Hans: *Grundlagen des Projektmanagements – Ein Leitfaden für die Projektpraxis*, 2002, Paul Bernecker Verlag, Wien

[**KrollKru03**] Kroll, Per; Kruchten Philippe: *The Rational Unified Process made easy: a practitioner's guide to the RUP*, 2003, Addison-Wesley @ Pearson Education

[**Kruchten04**] Kruchten, Philippe: *The Rational Unified Process: an introduction*, 3rd ed.: 2004 (1st ed.: 2003), Addison-Wesley @ Pearson Education

[**LanBrau85**] Langmaack, Barbara; Braune-Krickau Michael: *Wie die Gruppe laufen lernt –Anregungen zum Planen und Leiten von Gruppen*, 1985, Beltz Verlag, Weinheim-Basel

[**LeffWid04**] Leffingwell, Dean; Widrig Don: *Managing Software Requirements – A use case approach*, 2nd ed.: 2004 (1st ed.: 2003), Addison-Wesley @ Pearson Education

[**Macias01**] Maciaszek, Leszek A.: *Requirements Analysis and System Design*, 2nd ed.: 2006 (1st ed.: 2001), Addison-Wesley @ Pearson Education

[**MarcSuc03**] Marchesi, Michele; Succi Giancarlo; Wells Don; Williams Laurie: *Extreme Programming Perspectives* (Part of "*The XP Series*"), 2003, Addison-Wesley @ Pearson Education

[**Mayr05**] Mayr, Herwig: *Projekt Engineering – Ingenieurmäßige Softwareentwicklung in Projektgruppen*, 2nd (revised) ed.: 2005, Fachbuchverlag Leipzig @ Carl Hanser Verlag

[**MessTull99**] Messnarz, Richard; Tully Colin: *Better Software Practice for Business Benefit – Principles and Experience*, 1999, IEEE Computer Society Press

[**Molcho94**] Molcho, Samy: *Körpersprache*, Special edition 1994, Mosaik Verlag, München

[**PatzRatt04**] Patzak, Gerold; Rattay Günter: *Projektmanagement – Leitfaden zum Management von Projekten, Projektportfolios und projektorientierten Unternehmen*, 4th (revised) ed.: 2004, Linde, Wien

[**PoloPiat03**] Polo, Macario; Piattini Mario; Ruiz Francisco: *Advances in software maintenance management*, 2003, Idea Group Publishing

[Raymond99] Raymond, Eric S.: *The Cathedral and the Bazaar - Musings on Linux and Open Source by an Accidental Revolutionary*, 1999, O'Reilly Media
Online (regularly revised): www.catb.org/~esr/writings/cathedral-bazaar

[Robbins03] Robbins Stephen P.: *Organizational Behaviour*, 10th ed.: 2003, Prentice Hall

[RobHun03] Robbins, Stephen P.; Hunsaker Phillip L.: *Training in InterPersonal Skills: TIPS for Managing People at Work*, 3rd ed.: 2003 (1st ed.: 1989), Prentice Hall

[Rothlauf06] Rothlauf, Jürgen: *Interkulturelles Management*, 2nd (revised) ed.: 2006, Oldenburg Wissenschaftsverlag, München

[RoyceWa98] Royce, Walker (not to mistake for his father Winston W. Royce): *Software Project Management – A unified Framework*, 1998, Addison-Wesley

[Schach96] Schach, Stephen R.: *Classical and Object-Oriented Software Engineering*, 3rd ed.: 1996 (Earlier Editions: *Software Engineering*, 1990), IRWIN @ McGraw-Hill

[Shaw96] Shaw, Mary; Garlan David: *Software Architecture – Perspectives on an Emerging Discipline*, 1996, Prentice Hall

[SomSaw97] Sommerville, Ian; Sawyer Pete: *Requirements Engineering – A good practice guide*, 1997, John Wiley & Sons, Chichester et. al.

[Sommer06] Sommerville, Ian: *Software Engineering*, 8th ed. in German: 2007 (8th English ed.: 2006, 1st English ed. 1982), Addison-Wesley (Note: All citations refer to the German edition)

[Spreng00] Sprenger Reinhard K.: *Aufstand des Individuums – Warum wir Führung komplett neu denken müssen*, 2nd ed.: 2001 (1st ed.: 2000), Campus Verlag, Frankfurt New York

[StepRose03] Stephens, Matt; Rosenberg Doug: *Extreme Programming Refactored - The Case against XP*, 2003, APress

[Thorp98] Thorp, John: *The Information Paradox: Realizing the business benefits of Information Technology*, 1998, McGraw-Hill

[Thun81] Schulz von Thun, Friedemann: *Miteinander Reden 1: Störungen und Klärungen – Allgemeine Psychologie der Kommunikation*, 41st ed.: 2005 (1st ed.: 1981), Rowohlt Taschenbuchverlag, Reinbek bei Hamburg

[Thun89] Schulz von Thun, Friedemann: *Miteinander Reden 2: Stile, Werte und Persönlichkeitsentwicklung – Differentielle Psychologie der Kommunikation*, 25th ed.: 2005 (1st ed.: 1989), Rowohlt Taschenbuchverlag, Reinbek bei Hamburg

[Thun98] Schulz von Thun, Friedemann: *Miteinander Reden 3: Das "innere Team" und situationsgerechte Kommunikation – Kommunikation, Person, Situation*, 14th ed.: 2005 (1st ed.: 1998), Rowohlt Taschenbuchverlag, Reinbek bei Hamburg

[ThuRup00] Schulz von Thun, Friedemann; Ruppel Johannes; Stratmann Roswitha: *Miteinander Reden: Kommunikationspsychologie für Führungskräfte*, 2nd ed.: 2001 (1st ed.: 2000), Rowohlt Taschenbuchverlag, Reinbek bei Hamburg

[Versteeg00] Versteegen, Gerhard: *Projektmanagement mit dem Rational Unified Process*, 2000, Springer Verlag, Berlin Heidelberg

[Wein04] Weinberg, Gerald M.: *Die Psychologie des Programmierers – Seine Persönlichkeit, sein Team, sein Projekt*, 2004, mitp-Verlag, Bonn (English title: *The Psychology of Computer Programming*, 1998, Dorset House Publishing, New York – 25th Anniversary edition)

[Wiegers99] Wiegers, Karl E.: *Software Requirements*, 1999, Microsoft Press (German edition: 2005)

[YangWa03] Yang, Hongji; Ward Martin: *Successful Evolution of Software Systems*, 2003, Artech House Publishers, London Boston

[Your92] Yourdon, Edward: *Decline & Fall of the American Programmer,* 1992, Yourdon Press @ Prentice Hall PTR, Englewood Cliffs

[ZuserGre04] Zuser, Wolfgang; Grechenig Thomas; Köhle Monika: *Software-Engineering mit UML und dem Unified Process*, 2nd (revised) ed.: 2004 (1st ed.: 2001), Pearson Studium, München

7.2. Proceedings, papers and articles

[Bauer71] Bauer, Friedrich L.: "*Software Engineering*", In: *Information Processing 71 – Proceedings of IFIP Congress 71 (23rd – 28th August 1971, Ljubljana, Yugoslavia)*, 1972, North-Holland Publishing Co., Amsterdam, pp. 530-538
(Reprinted l972/75 in "*Advanced Course in Software Engineering*" by Springer Verlag, Berlin)

[Bauer93] Bauer, Friedrich L.: "*Software Engineering – Wie es begann"*, In: *Informatik Spektrum*, Springer Verlag, Vol. 16 (1993), Nr. 5, October 1993, pp. 259–260

[BiffHein05] Biffl, Stefan; Heindl Matthias et. all: „RMVU – *Skriptum zur Lehrveranstaltung Risikomanagement VU*", Lecture notes and slides: *Institute of Software Technology and Interactive Systems (IFS)* at TU Wien (http://qse.ifs.tuwien.ac.at), 2005

[Boehm76] Boehm, Barry W.: "*Software Engineering*", In: *IEEE Transactions on Computers*, Vol. C-25 (1976), Nr. 12, December 1976, pp. 1226-1241

[Boehm84] Boehm, Barry W.: "*Verifying and Validating Software Requirements and Design Specifications*", In: *IEEE Software*, Vol. 1 (1984), Nr. 1, January 1984, pp. 75-88

[Boehm88] Boehm, Barry W.: "*A Spiral Model of Software Development and Enhancement*", In: *IEEE Computer*, Vol. 21 (1988), Nr. 5, May 1988, pp. 61-72

[BoehRos89] Boehm, Barry W.; Ross Rony: "*Theory-W Software Project Management: Principles and Examples*", In: *IEEE Transactions on Software Engineering*, Vol. 15 (1989), Nr. 7, July 1989, pp. 902-916

[Boehm02] Boehm, Barry W.: "*Get Ready for Agile Methods, with Care*", In: *IEEE Computer*, Vol. 35 (2002), Nr.1, January 2002, pp. 64-69

[Boehm03] Boehm, Barry W.: "*Agility through Discipline: A Debate*" (*Kent Beck vs. Barry Boehm*), In: *IEEE Computer*, Vol. 36 (2003), Nr.6, June 2003, pp. 44-46

[BoehPort00] Boehm, Barry W.; Port Dan; Al-Said Mohammed: "*Avoiding the Software Model-Clash Spiderweb*", In: *IEEE Computer*, Vol. 32 (2000), Nr.11, November 2000, pp. 120-122

[BoehTur03] Boehm, Barry W.; Turner Richard: "*Using Risk to Balance Agile and Plan-Driven Methods*", In: *IEEE Computer*, Vol. 36 (2003), Nr.6, June 2003, pp. 57-66

[BoehTur05] Boehm, Barry W.; Turner Richard: "*Management Challenges to Implementing Agile Processes in Traditional Development Organizations*", In: *IEEE Software*, Vol. 22 (2005), Nr.5, September/October 2005, pp. 30-39

[Boehm06] Boehm, Barry W.: "A *View of 20th and 21st Century Software Engineering*", In: *Proceedings of the 28th ICSE*, May 2006, pp. 12-29

[Booch07] Booch, Grady: "*Speaking Truth to Power*", In: *IEEE Software*, Vol. 24 (2007), Nr. 2, March/April 2007, pp. 12-13

[Byrne92] Byrne, Eric J.: "*A Conceptual Foundation for Software Re-engineering*", In: *Proceedings for the Conference on Software Maintenance*, 1992, IEEE, pp. 226-235

[ChenAtl07] Cheng, Betty H.C.; Atlee Joanne M.: "*Research Directions in Requirements Engineering*", In: *Proceedings of the IEEE Future of Software Engineering Conference (FOSE'07)*, 2007

[ChiCo90] Chikofsky, Elliot J.; Cross James H. II: "*Reverse Engineering and Design Recovery - a Taxonomy*", In: *IEEE Software*, Vol. 7 (1990), Nr.1, January 1990, pp. 13-17

[Cox00] Cox, Alan: "*Dear Mr Brooks, or: Software engineering in the free software world*", Talk given at LinuxTag (LinuxDay) July 2000, Stuttgart (Repeated on the 13th Nov. 2000 at TU Wien) Online: http://ftp.linux.org.uk/pub/linux/alan/Talks/OGG/transcript.txt

[DeMaLi03b] DeMarco, Tom; Lister Timothy: "*Risk Management during Requirements*", In: *IEEE Software*, Vol. 20 (2003), Nr. 5, September/October 2003, pp. 99-101
Online: www.systemsguild.com/pdfs/s5req.lo%201.pdf

[EberMan05] Ebert, Christof; Man Jozef De: „*Requirements Uncertainty: Influencing Factors and Concrete Improvements*", In: *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pp. 553-560

[Ebert07] Ebert, Christof: "*Open Source Drives Innovation*", In: *IEEE Software*, Vol. 24 (2007), Nr. 3, May/June 2007, pp. 105-109

[Glass06] Glass, Robert L.: "*Greece vs. Rome: Two Very Different Software Cultures*", In: *IEEE Software*, Vol. 23 (2006), Nr. 6, November/December 2006, pp. 111 -112

[GlinzWie07] Glinz, Martin; Wieringa, Roel J.: "*Stakeholders in Requirements Engineering*“, In: *IEEE Software*, Vol. 24 (2007), Nr. 2, March/April 2007, pp. 18-20

[LarBas03] Larman, Craig; Basili Victor R.: "*Iterative and Incremental Development: A Brief History*", In: *IEEE Computer*, Vol. 36 (2003), Nr. 6, June 2003, pp. 47-56

[LawWie01] Lawrence, Brian; Wiegers Karl; Ebert Christof: "*The Top Risks of Requirements Engineering*", In: *IEEE Software*, Vol. 18 (2001), Nr. 6, Nov./Dec. 2001, pp. 62-63

[Lister97] Lister, Tim: "*Risk Management Is Project Management for Adults*", In: *IEEE Software*, Vol. 14 (1997), Nr. 3, May/June 1997, pp. 20, 22

[Mahoney04] Mahoney, Michael S.: "*Finding a History for Software Engineering*", In: *Annals of the History of Computing*, IEEE Computer Society, Vol. 26 (2004), Nr. 1, Jan/March 2004, pp. 8-19

[MikHeis06] Mikulovic Vesna, Heiss Michael: "*'How do I know what I have to do?'- The Role of the Inquiry Culture in Requirements Communication for Distributed Software Development Projects*", In: *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*, pp. 921-925

[NATO68] Naur, Peter; Randell Brian (Eds.): "*Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMITTEE (Garmisch, Germany, 7th to 11th October 1968)*", NATO Scientific Affairs Division, January 1969, Brussels. Online: http://homepages.cs.ncl.ac.uk/brian.randell/NATO (Including the two original NATO-reports from 1968 and 1969 as well as photographs of participants and sessions)

[RoyceWi70] Royce, Winston W. (not to mistake for his son Walker Royce): "*Managing the Development of Large Software Systems*", In: *Proceedings of the IEEE WESCON*, August 1970, pp. 1-9

[Sommer05] Sommerville, Ian: "*Integrated Requirements Engineering: A Tutorial*", In: *IEEE Software*, Vol. 22 (2005), Nr. 1, January/February 2005, pp. 16-23

[Tomic94] Tomic, Marijana: "*A possible Approach to Object-Oriented Reengineering of Cobol programs*", In: *ACM SIGSOFT Software Engineering Notes*, Vol. 19 (2004), Nr. 2, April 1994, pp. 29-34

[Woolr07] Woolridge, R.W.; McManus D.J.; Hale J.E.: "Stakeholder Risk Assessment: An Outcome-Based Approach", In: *IEEE Software*, Vol. 24 (2007), Nr. 2, March/April 2007, pp. 36-45

7.3. Standards and further online references

[**CC2005**] IEEE-CS/ACM Joint Task Force for Computing Curricula: *Computing Curricula 2005 – The Overview Report covering undergraduate degree programs in Computer Engineering, Computer Science, Information Systems, Information Technology, Software Engineering (A volume of the Computing Curricula Series)*, 30th September 2005
Online: http://computer.org/curriculum and www.acm.org/education/curricula.html

[**CMMI06**] Software Engineering Institute (SEI), Carnegie Mellon University (CMU): *Capability Maturity Model® Integration (CMMI) for Development (Version 1.2)*, August 2006, Pittsburgh
Online: http://www.sei.cmu.edu/cmmi/models/index.html

[**IEEE90**] Standards Coordinating Committee of the IEEE Computer Society: *IEEE Std. 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology* (Update and expansion of IEEE Std. 729-1983), Approved 28th September 1990, Published 10th December 1990 (IEEE-Status in August 2007: Active)

[**IEEE98a**] Standards Coordinating Committee of the IEEE Computer Society: *IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specification*, October 1998

[**IEEE98b**] Standards Coordinating Committee of the IEEE Computer Society: *IEEE Std 1058-1998, IEEE Standard for Software Project Management Plans*, December 1998

[**PMBOK04**] Project Management Institute (PMI): *A Guide to the Project Management Body of Knowledge (PMBOK® Guide)*, 3rd ed.: 2004 (Same title for the German translation)

[**SE2004**] IEEE-CS/ACM Joint Task Force for Computing Curricula: *Software Engineering 2004 – Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering (A Volume of the Computing Curricula Series)*, 23rd August 2004 (see also [*CC2005*])
Online: http://sites.computer.org/ccse/SE2004Volume.pdf

[**SECEPP99**] IEEE-CS/ACM Joint Task Force on Software Engineering Ethics and Professional Practices: *Software Engineering Code of Ethics and Professional Practice (SECEPP)*, 1999 (Final version 5.2), Online: www.computer.org/certification/ethics.htm or http://info.acm.org/serving/se/code.htm
(Background Material can be found in: "*Computer Society and ACM Approve Software Engineering Code of Ethics*", IEEE Computer, Vol. 32 (1999), Nr. 10, Oct. 1999, p. 84-88)

[**SWEBOK04**] Professional Practices Committee of the IEEE Computer Society: *Guide to the Software Engineering Body of Knowledge (SWEBOK)*, 2004 Version (February 2004), www.swebok.org

[**VModellXT**] Koordinierungs- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung (KBSt) Deutschlands (Federal Government Co-ordination and Advisory Agency for IT in the Federal Administration of Germany): *V-Modell XT (Extreme Tailoring)*, February 2005, Berlin, Online: www.v-modell-xt.de

7.4. World literature

[Bloch85] Bloch, Arthur: *Murphy's Law Complete – All the reasons why everything goes wrong!*, 1990 (1st ed.: 1985), Mandarin Paperbacks, London

[Brecht53] Brecht, Bertold: *Der gute Mensch von Sezuan* (English title: *The Good Person of Sezuan*), 1953

[Disney87] The Walt Disney Company (Ed.), Cavazzano, Giorgio (Drawings): “*Die umgekehrte Pyramide*”, In: *Walt Disneys Lustiges Taschenbuch Nr. 118 – Donald, der Weltenbummler,* 1987, Egmont Ehapa Verlag GmbH, Berlin (English title: “*The inverted Pyramid*”), p. 76 - 136

[Eco77] Eco, Umberto: *Wie man eine wissenschaftliche Abschlussarbeit schreibt,* 10th German ed.: 2003, C. F. Müller Verlag, Heidelberg (Italian original: *„Come si fa una tesi di laurea“,* 1st ed.: 1977, Bompiani, Milano)

[Goethe1808] Goethe, Johann Wolfgang von: *Faust – Der Tragödie erster Teil,* 1st ed. @ Reclam: 1986 (Original: 1808), Reclam, Stuttgart

[Machia1532] Machiavelli, Niccolò: *Il Principe* (English title: *The Prince*), 1532

Colophon

Word processing:	*Microsoft ® Office Word 2003*
Typeface:	*Garamond (Body Text: 12 pts.; Line spacing 1.2)*
Language:	*BE (British English)*
Printing:	*Infotec (Ricoh) IS 2235; HP Color Laserjet 3550*
Paper:	*XEROX Premier ECF A4 80 - 003R91720 (80 g/m^2, bright white 165)*
Bookbinding:	*Red full-linen with golden stamping*
Electronic version:	*www.fortner.at/WINF/2007DiplomarbeitRF.pdf*

© Raoul Fortner
A-1040 Wien (Vienna)
Österreich (Austria)
raoul@fortner.at
August 2007