2 Fundamentals

Back in the 70s and 80s of the 20th Century different approaches, concepts, standards and laws in software development were published – specifically in the areas of software maintenance and evolution. Their analysis is essential for a comprehensive definition and introduction.

The principles of this book involve the domains software life cycle, software maintenance and software modeling as well as the description of the appropriate methodology.

Software maintenance (Chapter 2.2) is a component of the *software life cycle* (Chapter 2.1) and is therefore classified in its context. This determination is necessary because the used advanced model-driven development approach differs from the traditional process models. Following, *software reengineering* (Chapter 2.4) and *software migration* (Chapter 2.5) are presented as two core areas of software maintenance. The focus of this book are aged and complex software systems, also called *legacy systems*. The associated problems and uncertainties contribute to their negative reputation[1] [III81, Seite 343][Ber05][Boe86, Seite 69] in the area of software maintenance. They are described in detail in Chapter 2.3.

The model-driven approach as well as different modeling concepts are explained in Chapter 2.6. This chapter examines different views from industrial practice [Kü06, Fav04a] and scientific research. This includes the distinction of various model definitions, model types and model relations.

The formal basis of the methodology developed in this book is a graphical notation that was devised by Brinkkemper et al. [Bri96, WBSV06, WBV07]. This has already been successfully used and extended by the author of this book[2]. A description of the methodology follows in Chapter 2.7.

In this book it is emphasized that development and operation/maintenance of software systems should be a continuous process. Techniques of model-driven development can be used to optimize the process of software

[1] "This perception of maintenance is Primarily a 'pest control ...'"

[2] compare to Chapter 2.7.2 [SHA12, She08, SHA08, CON12, ISJ+09, HSM10].

C. Wagner, *Model-Driven Software Migration: A Methodology,*
DOI 10.1007/978-3-658-05270-6_2, © Springer Fachmedien Wiesbaden 2014

reengineering and software migration. Furthermore existing software systems could gradually be extended and regenerated in a *continuous process*.

2.1 Software Life Cycle

This chapter considers different process models in software development and examines, whether and how operation/maintenance of software are represented there as well as what relevance is attributed to them. Subsequently the reason for the aging of software is explained based on principle laws in software evolution. Hence the need for maintenance will be justified. Finally, two scientific approaches are presented which emphasize the unity of development and operation/maintenance of software.

At the beginning of the software life cycle an application is developed for the first time, *creation ex nihilo*[3]. This first phase of the life cycle (*the development*) is completed on delivery of the software to the customer. All subsequent activities (operation, error correction, expansion and migration) are assigned to the second major section, *the operation*. In literature, the terms maintenance and evolution are used interchangeably [Som06, pp. 531]. In this chapter, the last section of the software life cycle is generally identified as operation/maintenance. A closer inspection and separation follows in Chapter 2.2.

So-called process models organize the development process and life cycle of software into structured phases. Certain tasks, methods and techniques are assigned to each phase. A transition criterion for phase change is defined and thus a logical order is produced.

The clear separation of development phase and operation/maintenance phase has certainly set the focus of research literature not on the entire life cycle [Ben95]. In recent decades, the first section, the development has been extensively studied and refined. The second part, operation/maintenance of software, often appears rather as an accessory or is listed only for completeness [Ben95] [Vli08, Seite 474]. This drawback is clearly recognizable in the existing process models.

2.1.1 Process Models: Classical

In scientific literature as well as in industrial practice different process models have emerged. These range in their original form from the waterfall model [Roy70] (Figure 2.1), developed in the 70s, to state-of-the-art meth-

[3] Latin for "creation out of nothing"

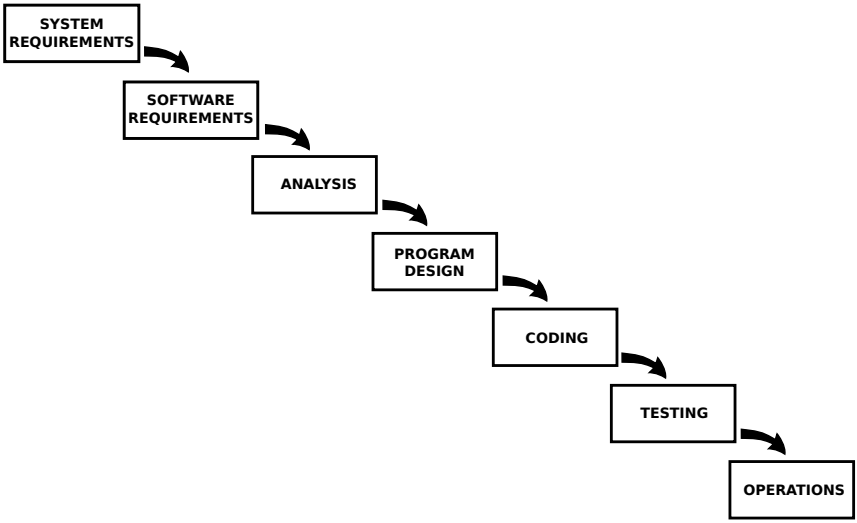

Figure 2.1: Waterfall model by Royce [Roy70]

ods like the Rational Unified Process (RUP) [Kru03]. All models have in common that they are focusing on the development of software exclusively.

The waterfall model by Royce, shown in Figure 2.1, is the first description of a relevant model of software development and the software life cycle. Despite its age of four decades it has been and still is used in a variety of projects[4]. The focus of the model is the refinement and organization of the development process in each phase. Inevitably, it is a sequential process. The transition criterion are documents that are created for each phase which leads to the term *document-driven*. Many subsequent process models are based on the phases defined in the strictly linear structured waterfall model[5].

The first phase, the development of software consists of the following phases: requirements analysis, design, implementation and testing of the software. The last section, called in most process models operation (Figure 2.1) or maintenance, is not further differentiated in contrast to software

[4] Until 2005, the company IBM used the waterfall model for the development of Websphere products [SH09]. In 2009, the market research and consulting company Forrester identified best practices in software development [Wes09]. Nevertheless still 33 % of the companies surveyed use the waterfall model for software projects.

[5] According to the process model, they are refined, summarized or iterated in several cycles.

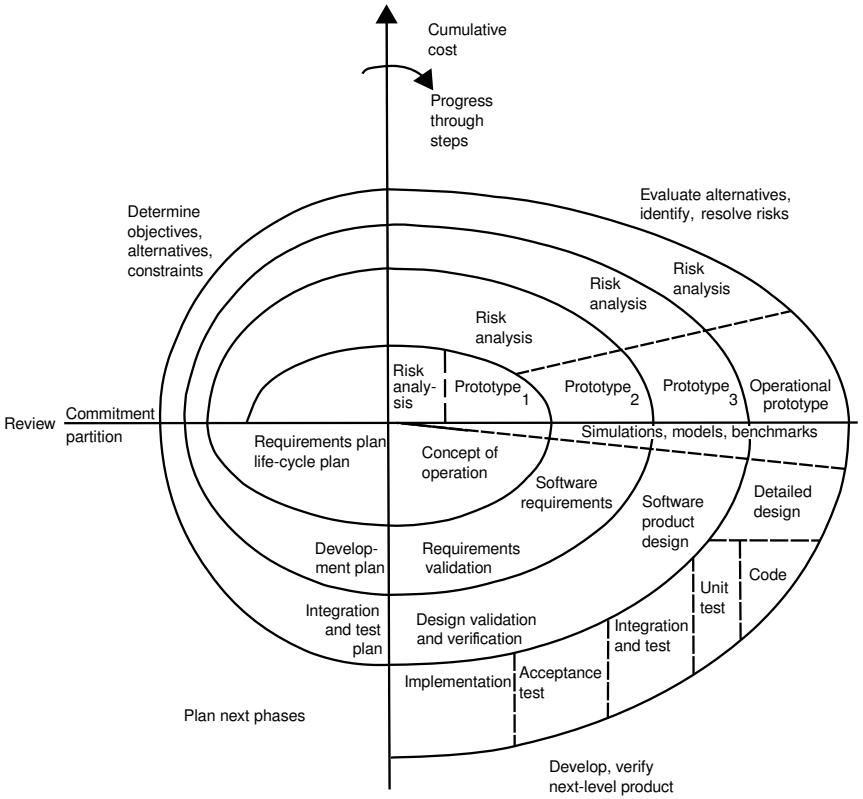

Figure 2.2: Spiral model by Boehm [Boe86]

development. Generally, there is a lack of specification in form of usable methods for adaptation, extension or migration of software.

A pragmatic solution to this problem is the *iterative* application of the process model. It is usually overlooked however, that in addition to new requirements an existing system has to be considered as well, *creatio ex aliquo*[6]. Inherently present problems are usually ignored (see Chapter 2.2).

An iterative process is a central part of the spiral model of Boehm [Boe86], Figure 2.2. It was developed in the 80s in response to problems with the linearity in the waterfall model. At the beginning of software development, different prototypes are created (Figure 2.2, 1st quadrant) to test different aspects of the software as well as to evaluate risks. Therefore, Boehm speaks

[6] Latin for "creation from something"

of a risk-based methodology. Following the iterative risk assessment (1st phase) the actual software development according to the waterfall model is performed (see Figure 2.2, 2nd quadrant). To summarize, the development of software should change from an open process towards a closed loop with user feedback [Mil76, p. 266].

Boehm also discusses the applicability of his model in the operation/maintenance phase: The iterations in the model are solely used as preliminary risk assessment for the actual software development process. They do not necessarily have a direct correlation to the final product. Furthermore the model has no explicit operation/maintenance phase. Boehm said [Boe86, p. 69] that there is not a separate phase for operation/maintenance to avoid its second-class status. Software-in-use always creates new demands which are implemented in an additional maintenance spiral. The integration of existing software systems in this maintenance spiral (*creatio ex aliquo*) is not explicitly mentioned by Boehm.

2.1.2 Process Model: Continuous

In recent years[7] agile or evolutionary development methods[8] emerged in addition to the plan- or document-driven models[9] (classic). Common to all continuous approaches is the conviction that it is not possible to define all requirements at the beginning of a project. Furthermore, they focus on the integration of the user into the development process. The planning and implementation take place incrementally in small, timed steps. Adjustments or corrections are made immediately in consultation with the users.

Illustration 2.3 is a general sketch of the agile approach. At first the ideas and requirements of the customers are collected. The developers select a few requirements and implement a prototype. The prototype will be tested by the customer, which in turn creates new demands and ideas or discards old ones[10]. Thereafter the process starts again. Each iteration takes place in a short and clearly defined time frame. At the end of each iteration, a

[7] Initial ideas and approaches to evolutionary prototyping have already been documented in the early 80s by McCracken and Jackson [MJ82]. However, they gain importance in the last few years.

[8] The principles of these methods are defined in the agile manifesto: `http://www.agilemanifesto.org`.

[9] Phases, activities and goals are fixed at the beginning of the project. The progress is measured at the stage reached in the process model or at the generated documents. These models can be adapted to some degree, for example, individual phases can be iterated.

[10] This agrees with the assessment of Miller [Mil98], who states that computer users can not describe their requirements until they feel the results of their specification.

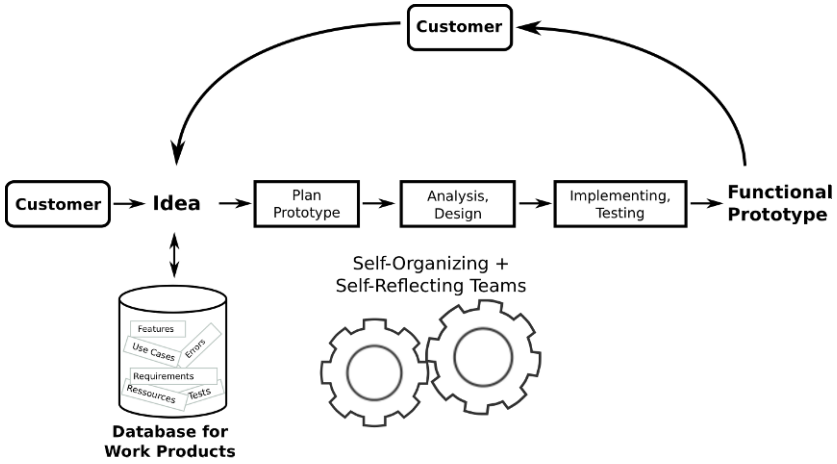

Figure 2.3: Sketch of an agile process model using the example of Scrum

running prototype (as opposed to the spiral model) must be available, which covers an increasing part of the wishes or requirements of the stakeholders.

With the help of agile process models a software life cycle model emerges, which removes the separation between initial development and operation/maintenance. Assuming that the described iteration never ends, the operation/maintenance phase is implicitly incorporated in the process model. The main difference to the plan- or document-driven models is due to the fact that the software development process does not end after the first cycle. If necessary it can be continued indefinitely. Development as well as operation/maintenance are interrelated. In the classical models, however, customer requirements can usually only be incorporated to the implementation phase and not beyond [RPTU84, p. 193].

The agile approach is to be regarded as the ideal case. The extensive effects, for example on the continuity of a development team or the evolution of the software architecture, are not discussed in this book.

At this point it must be mentioned that similarities, differences (cf. [CLC04, pp. 4]) and influences [ASRW02] between spiral model and agile methods are the subject of current controversy. Common points are the definition and status of a prototype, time periods between iterations, incorporating customer requirements and risk management into the project. From the perspective of this book, the spiral model is an intermediate step from the waterfall model to an agile software development model. Both with the aim of continuous development of software. The agile models are consid-

ered more rigorously with regards to incremental development, prototyping and customer engagement.

According to the methodology of this book, it is necessary to avoid the gap between initial development and operation/maintenance for sophisticated and useful software products. Especially it is often ignored that software products come to maturity after putting them into operation [Ber05].

Therefore the operation/maintenance phase of software should be inherently anchored in a process model[11]. *Ex aliquo* is the rule and *ex nihilo* the special case. Simultaneously the evolution of software[12] is improved by introduction of models leading to a different level of abstraction.

2.1.3 Laws in the Software Life Cycle

The need of software evolution is due to the increasing complexity in the course of the life cycle of software. Lehman and Belady [LB85] have formulated this principle in the mid-80s in several laws[13]. The first law is called *the law of continuous change*. It states that software will only be used if it is continuously adapted to the changing requirements.

> *"A large program that is used undergoes continuing change or becomes progressively less useful. The change process continues until it is judged more cost - effective to replace the system with a recreated version." [LB85, Seite 250]*

The second principle refers to the inner complexity of software. Usually, it increases by any change in the software.

> *"As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it." [LB85, Seite 253]*

Today, these laws are still valid. They define that any software in use, is subject to evolution. It necessarily follows an increasing inner complexity. Evolution refers to the time from first operation to retirement/replacement of software (see Figure 2.5). From the perspective of the software life cycle, evolution is a part of the operation/maintenance phase.

[11] cf. Mills 1976 [Mil76, p. 272] and Berg 2005 [Ber05, p. 58].
[12] The concept of evolution is described in detail in Chapter 2.1.3.
[13] Lehman and Belady have formulated five Principles to the character of the software life cycle. In this book, the first two Principles are considered in detail.

Table 2.1: Empirical surveys on the percentage of maintenance in the software life cycle, partly from Müller [Mü97, p. 8]

Survey	Year	Percentage of maintenance
Mills [Mil76]	1976	75 %
de Rose and Nymann [DRN78]	1978	60 %-70 %
Zelkowitz [ZSG79, Seite 9]	1979	67 %
Cashman and Holt [CH80]	1980	50 % - 80 %
Lientz and Swanson [LS80]	1980	≥50 %
Reutter [III81]	1981	70 %
McKee [McK84]	1984	66 %-75 %
Ramamoorthy [RPTU84]	1984	60 %
Grady [GC87]	1987	50 %
Moad [Moa90]	1990	60 %-90 %
Nosek [NP90]	1990	60 %
Zilahi [ZS95]	1995	≥66 %
Erlikh [Erl00]	2000	85 % - 90 %
van Vliet [Vli08]	2008	75 %

Percentage of Maintenance in the Software Life Cycle

Several empirical surveys determined the amount of software maintenance in the entire life cycle (operation of software is not listed). A tabular overview of two decades can be found in Müller [Mü97, p. 8]. The percentage of software maintenance in this period was between 50 % and 80 %. Table 2.1 illustrates the summary by Müller chronologically and is extended with current surveys.

The studies in Table 2.1 point out that the lower bound of maintenance continues to rise. One survey estimates an effort of 90 % [SPL03, pp. 5].

However, these studies can only be compared with reservation. They determine the effort based on various criteria such as percentage of total budget as well as number of staff hours. Furthermore they use different definitions of maintenance, summarize maintenance and operation, are informal or are based on experience; they might also estimate future expenses. But all highlight that the percentage of maintenance is extremely high and has increased steadily in the past. Based on the principles of Lehman and Belady outlined above, it can be assumed that the proportion of maintenance steadily increases during the lifetime of software.

From Table 2.1 it can be deduced that software projects developed *ex nihilo*, only account for a small portion of all software projects. In most cases, besides new requirements of the customer, an existing software product needs to be expanded or improved. Conversely, this means that most projects do not start on a greenfield[14]. They have to deal with *legacy systems* and to incorporate them. This emphasizes the need to focus on the operation/maintenance phase.

The Maintenance Crisis

Rising costs especially have led some authors to speak of an impending maintenance or legacy crisis [SPL03, p. 6] – based on the concept of software crisis[15], which was introduced in 1968 for the first time. The authors assume that a large amount of resources are necessary for maintenance and lesser resources are available for new developments. Parnas thinks the term crisis inappropriate, since it refers to something rather short-term and unexpected. Therefore, he prefers the term *chronic disease* [Par94, p. 286].

From the perspective of this book it is not the question of whether there should be maintenance, but how much costs may be associated with it. Improvements of methods and techniques mostly have focused on software development. However, they are amortized by the growing size and the increasing complexity of software. They play a subordinate role regarding overall duration of the software life cycle (see Table 2.1). Accordingly, the interrelation of development and operation/maintenance is needed for a continuous development process for the design and maturity of long-lived software [Ber05].

2.1.4 Process Models: Research Approaches

First research approaches in this field address the above mentioned problems. In addition to the agile process models, new scientific methods holistically integrate the operation/maintenance phase into the software life cycle. Two representatives are the Continuous Software Engineering (CSE) [Web99], a former research project of the Federal Ministry of Education and Research (BMBF) at the Technical University of Berlin and the

[14] Also called greenfield approach: Software projects that are developed from scratch must take no account to organically grown environment.

[15] It was first spoken of a software crisis on a NATO conference in Garmisch-Partenkirchen (Germany) in 1968 [Mü97]. This is also considered as the birth of software engineering.

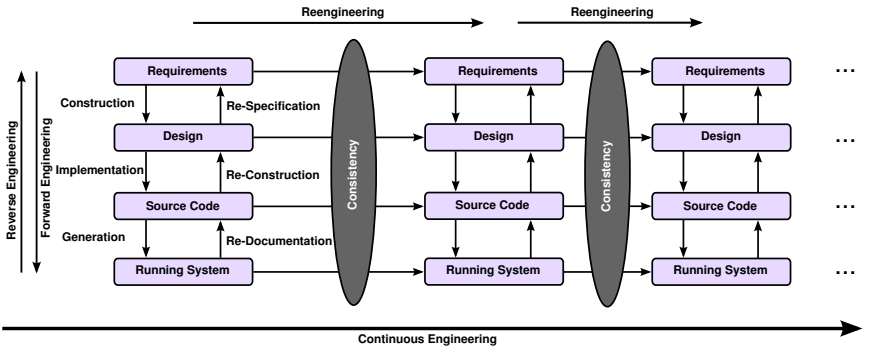

Figure 2.4: Continuous Software Engineering by Mann et al. [MBE+00]

Continuous Model-Driven Engineering (CMDE) [MS09a], developed at the Technical University of Dortmund and the University of Potsdam.

CSE addresses the continuous maintenance of software, which includes associated development artifacts[16]. It involves all phases of software development in order to avoid inconsistencies in documents, models and source code over time. Each step in the development must therefore be largely formalized. Any adjustment on one level may require changes at other levels. Therefore, effects and relationships need to be analyzed. The basic approach is shown in Figure 2.4. After each evolutionary step the consistency of all development artifacts is established, before starting with the next iteration [Web99].

The second approach (CMDE) is a continuous development approach based on the model-layer. It was originally developed for modeling and execution of process graphs. In this book it is applied to integrate existing software systems in a continuous development process (see Chapter 3.3.1). By modeling, further evolution of software is facilitated and the boundary between development and operation/maintenance is removed.

Previous approaches for continuous development and evolution of software (agile development methods, CSE, CMDE) assume that they are applied from the beginning. A subsequent switch and the insertion of a continuous process is not described yet. In literature numerous statements that emphasize the integration of existing systems into a continuous software de-

[16] Requirements, models, documentation, source code, etc., are various components that are part of a software development. They are commonly referred to as artifacts.

http://www.springer.com/978-3-658-05269-0

Model-Driven Software Migration: A Methodology
Reengineering, Recovery and Modernization of Legacy Systems
Wagner, C.
2014, XXV, 304 p. 95 illus., Softcover
ISBN: 978-3-658-05269-0