1

Introduction and Roadmap: History and Challenges of Software Evolution

Tom Mens

University of Mons-Hainaut, Belgium

Summary. The ability to evolve software rapidly and reliably is a major challenge for software engineering. In this introductory chapter we start with a historic overview of the research domain of software evolution. Next, we briefly introduce the important research themes in software evolution, and identify research challenges for the years to come. Finally, we provide a roadmap of the topics treated in this book, and explain how the various chapters are related.

1.1 The History of Software Evolution

In early 1967, there was an awareness of the rapidly increasing importance and impact of software systems in many activities of society. In addition, as a result of the many problems faced in software manufacturing, there was a general belief that available techniques should become less ad hoc, and instead based on theoretical foundations and practical disciplines that are established in traditional branches of engineering. These became the main driving factors for organising the first conference on Software Engineering in 1968 [391]. The goal of this conference, organised by the NATO Science Committee, was "the establishment and use of sound engineering principles in order to obtain reliable, efficient and economically viable software". Among the many activities of software engineering, *maintenance* was considered as a post-production activity, i.e., after the delivery and deployment of the software product.

This view was shared by Royce, who proposed in 1970 the well-known *waterfall life-cycle* process for software development [446]. In this process model, that was inspired by established engineering principles, the *maintenance* phase is the final phase of the life-cycle of a software system, after its deployment. Only bug fixes and minor adjustments to the software are supposed to take place during that phase. This classical view on software engineering has long governed the industrial practice in software development and is still in use today by several companies. It even became a part of the IEEE 1219 Standard for Software Maintenance [239], which defines software maintenance as "the modification of a software product *after delivery* to

T. Mens, S. Demeyer (eds.), *Software Evolution*.
DOI 10.1007/978-3-540-76440-3, © Springer 2008

correct faults, to improve performance or other attributes, or to adapt the product to a modified environment."

It took a while before software engineers became aware of the inherent limitations of this software process model, namely the fact that the separation in phases was too strict and inflexible, and that it is often unrealistic to assume that the requirements are known before starting the software design phase. In many cases, the requirements continue to change during the entire lifetime of the software project. In addition, knowledge gained during the later phases may need to be fed back to the earlier phases.

Therefore, in the late seventies, a first attempt towards a more evolutionary process model was proposed by Yau with the so-called *change mini-cycle* [559] (see Fig. 1.1). In this process, important new activities, such as change impact analysis and change propagation were identified to accommodate the fact that software changes are rarely isolated.

Also in the seventies, Manny Lehman started to formulate his, now famous, *laws of software evolution*. The postulated laws were based on earlier work carried out by Lehman to understand the change process being applied to IBM's OS 360 operating system [317, 318]. His original findings were confirmed in later studies involving other software systems [320].

This was probably the first time that the term *software evolution* (or program evolution) was deliberately used the stress the difference with the post-deployment activity of software maintenance. To stress this difference even more, Lehman coined the term *E-type software* to denote programs that must be evolved because they "operate in or address a problem or activity of the real world". As such, changes in the real world will affect the software and require adaptations to it.

Nevertheless, it took until the nineties until the term software evolution gained widespread acceptance, and the research on software evolution started to become popular [24, 403]. This also lead to the acceptance of so-called *evolutionary processes* such as Gilb's *evolutionary development* [200], Boehm's *spiral model* [71] and Bennett and Rajlich's *staged model* [57].

The staged process model, visualised in Fig. 1.2, is interesting in that it explicitly takes into account the inevitable problem of *software aging* [410]. After the initial stage of development of a first running version, the evolution stage allows for any kind of modification to the software, as long as the architectural integrity remains

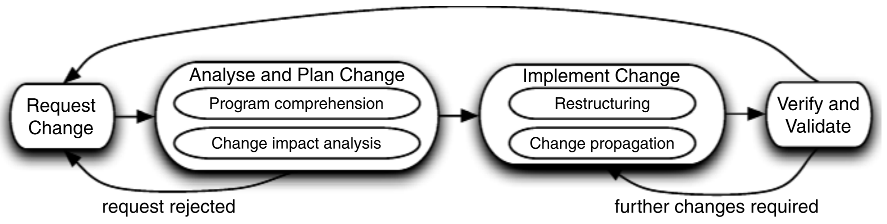

Fig. 1.1. The staged process model for evolution (adapted from [559] ©[1978] IEEE)

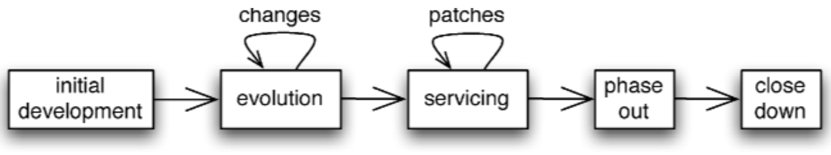

Fig. 1.2. The staged process model for evolution (adapted from [57] ©[2000] ACM)

preserved. If this is no longer the case, there is a loss of evolvability (also referred to as *decay*) and the servicing stage starts. During this stage, only small patches can be applied to keep the software up and running. If even such small patches become too costly to carry out, the phase-out stage starts, leading to ultimate close down of the system. If the system, despite of its degraded quality, is still valuable to its various stakeholders, it is called a *legacy system*. In that case, it may be wise to *migrate* to a new system that offers the similar or extended functionality, without exhibiting the poor quality of the legacy system. The planning to migrate to such a new system should be done as soon as possible, preferably during the servicing stage.

Software evolution is also a crucial ingredient of so-called *agile software development* [119, 351] processes, of which *extreme programming (XP)* [50] is probably the most famous proponent. In brief, agile software development is a lightweight iterative and incremental (evolutionary) approach to software development that is performed in a highly collaborative manner and explicitly accommodates the changing needs of its stakeholders, even late in the development cycle, because this offers a considerable competitive advantage for the customer. In many ways, agile methods constitute a return to iterative and incremental development as practiced early in the history of software development, before the widespread use of the waterfall model [312].

Nowadays, software evolution has become a very active and well-respected field of research in software engineering, and the terms *software evolution* and *software maintenance* are often used as synonyms. For example, the international ISO/IEC 14764 standard for software maintenance [242], acknowledges the importance of pre-delivery aspects of maintenance such as planning. Similarly, the Software Engineering Body of Knowledge (SWEBOK) [2] acknowledges the need for supporting maintenance in the pre-delivery as well as the post-delivery stages, and considers the following evolution-related research themes as being crucial activities in software maintenance: software comprehension, reverse engineering, testing, impact analysis, cost estimation, software quality, software measurement, process models, software configuration management, and re-engineering. These activities will be discussed in more detail in Section 1.2.

In this book, we will continue to use the term software evolution as opposed to maintenance, because of the negative connotation of the latter term. Maintenance seems to indicate that the software itself is deteriorating, which is not the case. It is changes in the environment or user needs that make it necessary to adapt the software.

1.2 Research Themes in Software Evolution

In this Section we provide an overview of some of the important research themes in software evolution. The various chapters of this book will explore some of these themes in more depth. Of course, it is not the aim of the book to provide complete and detailed coverage of all these themes. Instead, we have tried to offer a selection of important issues that are actively pursued by the research community. They have been identified, among others in the visionary articles by Bennett and Rajlich [57] and Mens et al. [371]. Therefore, in this section, we summarise some of the most important challenges and future research directions in software evolution, as reported in these articles.

1.2.1 Dimensions of Software Evolution

There are two prevalent views on software evolution, often referred to as the *what and why* versus the *how* perspectives [322].

The *what and why* view focuses on software evolution as a scientific discipline. It studies the *nature* of the software evolution phenomenon, and seeks to understand its driving factor, its impact, and so on. This is the view that is primarily taken in [338]. An important insight that has been gained in this line of research is that the evolution process is a multi-loop, multi-level, multi-agent feedback system that cannot be treated in isolation. It requires interdisciplinary research involving non-technical aspects such as human psychology, social interaction, complexity theory, organisational aspects, legislation and many more.

The *how* view focuses on software evolution as an engineering discipline. It studies the more pragmatic aspects that aid the software developer or project manager in his day-to-day tasks. Hence, this view primarily focuses on technology, methods, tools and activities that provide the *means* to direct, implement and control software evolution.

It is the latter view that is followed throughout most of the chapters in this book. Nevertheless, it remains necessary to develop new theories and mathematical models, and to carry out empirical research to increase understanding of software evolution, and to invest in research that tries to bridge the gap between the what and the how of software evolution.

As another "dimension" of software evolution, we can consider the types of changes that are being performed. Based on earlier studies by Lientz and Swanson [329], the ISO/IEC standard for software maintenance [242] proposes four categories of maintenance:

- *Perfective maintenance* is any modification of a software product after delivery to improve performance or maintainability.
- *Corrective maintenance* is the reactive modification of a software product performed after delivery to correct discovered faults.
- *Adaptive maintenance* is the modification of a software product performed after delivery to keep a computer program usable in a changed or changing environment.

- *Preventive maintenance* refers to software modifications performed for the purpose of preventing problems before they occur.

For completeness, we also mention the work of Chapin et al. [109], who further extended this classification, based on objective evidence of maintainers' activities ascertainable from observation, and including non-technical issues such as documentation, consulting, training and so on. A related article that is worthwhile mentioning is the work by Buckley et al. [94], in which a taxonomy of software change is presented based on various dimensions that characterise or influence the mechanisms of change.

1.2.2 Reverse and Re-Engineering

An important theme within the research domain of software evolution is *reverse engineering* [112]. This activity is needed when trying to understand the architecture or behaviour of a large software system, while the only reliable information is the source code. This may be the case because documentation and design documents are unavailable, or have become inconsistent with respect to the code because they have not been updated. Reverse engineering aims at building higher-level, more abstract, software models from the source code. *Program comprehension* or *program understanding* are activities that try to make sense of the wealth of information that reverse engineering produces, by building mental models of the overall software architecture, structure and behaviour. Program comprehension also includes activities such as task modelling, user interface issues, and many others.

Reverse engineering can also be regarded as the initial phase in the process of *software reengineering* [23]. Reengineering is necessary when we are confronted with *legacy systems*. These are systems that are still valuable, but are notoriously difficult to maintain [149]. Following the terminology used in the staged life cycle model of Fig. 1.2, we consider these systems to be in the servicing stage.

The goal of reengineering is thus to come to a new software system that is more evolvable, and possibly has more functionality, than the original software system. The reengineering process is typically composed of three activities, as captured by the so-called *horseshoe model* visualised in Fig. 1.3 [271]. First, reverse engineering may be necessary when the technological platform of the software system (language, tools, machines, operating system) is outdated, or when the original developers are no longer available. This activity is typically followed by a phase of *software restructuring* [22] in which we try to improve crucial aspects of the system. Finally, in a *forward engineering* phase we build a new running system based on the new, restructured, model.

The topic of reengineering is very important and relevant to industry, and therefore the second part of this book will be entirely devoted to it. Chapter 5 will focus on the reengineering of object-oriented software systems. Chapter 6 will address the need for, and means to, migrate data when reengineering large information systems. Chapter 7 discusses how to reengineer legacy systems into service-oriented systems.

Another very important research topic in reengineering research is the quest for new and better visualisation techniques that aid in a better program comprehension,

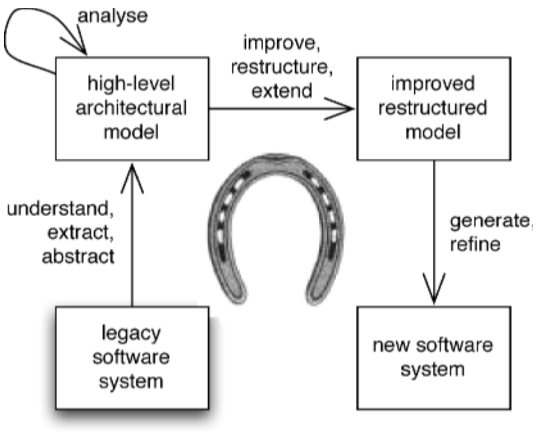
analyse
improve, restructure, extend
high-level architectural model
improved restructured model
understand, extract, abstract
generate, refine
legacy software system
new software system

Fig. 1.3. The horseshoe process model for reengineering

as well as a better understanding of the evolution of software. Such visualisation techniques are explored in Chapter 3.

1.2.3 Incremental Change Techniques

In the change mini-cycle proposed by Yau et al. [559], and visualised in Fig. 1.1, a number of important activities related to the change process become apparent.

During the planning phase, program comprehension is of course essential to understand what parts of the software will be affected by a requested change. In addition, the extent or impact of the change needs to be assessed by resorting to *change impact analysis* techniques [74]. By predicting all parts of the system that are likely to be affected by a change, they give an estimation of how costly the change will be, as well as the potential risk involved in making the change. This analysis is then used to decide whether or not it is worthwhile to carry out the change.

Because of the fact that a change may have a non-local impact, support is needed for what is referred to as *change propagation* [424, 425]. It is necessary when a change to one part of a software system requires other system parts that depend on it to be changed as well. These dependent system parts can on their turn require changes in other system parts. In this way, a single change to one system part may lead to a propagation of changes to be made throughout the entire software system.

During the implementation phase, it may turn out that the change cannot be implemented directly, and that a *restructuring* or *refactoring* of the software is required first in order to accommodate the requested change. The goal is thus to improve the software structure or architecture without changing the behaviour [21, 183].

During the validation and verification phase, techniques to revalidate the software after having performed changes are crucial in order to ensure that the system integrity has not been compromised. *Regression testing* is one of those techniques [66]. Rather than repeating all tests for each new software release (which would be too costly, take too much time, and consume too many resources), a carefully selected subset of the tests is executed to verify that the changes did not have inadvertent effects. Chapter 8 of this book provides an excellent overview of software testing, and its interplay with software evolution.

1.2.4 Managerial Issues

Managerial issues are equally crucial to software evolution. Despite this fact, it remains a challenge to increase awareness among executives and project managers about the importance and inevitability of software evolution. Indeed, various studies and surveys indicate that over 80% of the total maintenance effort is used for non-corrective actions [1, 416]. In addition, other studies indicate that software maintenance accounts for at least 50% of the total software production cost, and sometimes even exceeds 90% [329, 457, 296].

According to Lehman, software evolution problems start to appear when there are at least two management levels involved in the software production process. This is confirmed by Brooks [85], who calls this the *large program problem*. A very important managerial issue has to do with the *economics of software evolution* [72]. It turns out that, in many cases, the reason for evolving software is non-technical. More specifically, it is an economic decision, driven by marketing or other reasons.

The main challenge is therefore to develop better predictive models, based on empirical studies, for measuring and estimating the cost and effort of software maintenance and evolution activities with a higher accuracy [261, 466, 427, 177]. Similar techniques may also be useful to measure the cost-effectiveness of regression testing [444].

Another point of attention for managers is the need for *software quality assurance*. If proper support for measuring quality is available, this can provide crucial information to determine whether the software quality is degrading, and to take corrective actions if this turns out to be the case. Numerous software metrics have been proposed, studied and validated as measures of software quality characteristics such as complexity, cohesion, coupling, size and many others [83, 39, 171, 231].

Besides metrics, other more heuristic approaches may be used to detect "bad smells" or other indicators of poor-quality software. For example, Chapter 2 of this book studies techniques to detect and remove software redundancies and code clones, which are generally considered to be an indication of poor quality. Chapter 4 analyses software failures stored in a bug repository to predict and improve the software quality over time.

1.2.5 The Software Process

An important area of research is to find the software process model that is most appropriate to facilitate software evolution. In Section 1.1 we already introduced a number of such process models. The IEEE standard for software maintenance [239] and the ISO/IEC standard for software maintenance [242] also propose such a maintenance process model.

It is important to observe that, due to the fact that the activity of software evolution is a continuous feedback process [338], the chosen software process model itself is likely to be subject to evolution. The research area of *software process improvement* aims to reduce cost, effort and time-to-market, to increase productivity and reliability, or to affect any other relevant properties. Software process improvement can be based on theory or empirical industrial case studies [208].

As software systems become larger and more complex, and are being developed in a collaborative and distributed way, it becomes inevitable to resort to dedicated *software configuration management* tools. Among others, they provide automated support for the change process, they allow for software *versioning* and *merging*, and they offer procedures (verification, validation, certification) for ensuring the quality of each software release. Even today, research in this area is continuing in order to advance the state-of-the-art.

Another aspect of software process improvement is the exploration and introduction of novel development paradigms such as agile software development [119, 351], aspect-oriented software development [247], model-driven software development [474], service-oriented architectures [393], and many more. All of these development paradigms claim to improve software development and to lead to higher productivity, higher quality, and more adaptable and maintainable software. Some of these claims are investigated in Chapter 9 for aspect-oriented development.

Of particular interest is the open source movement, which has provided a novel, strongly collaborative way of developing and evolving software. The question arises whether this style of software development is subject to the same laws that govern the evolution of traditional software development approaches [318]. This topic is under active study [206, 481, 461] and will be addressed in Chapter 11 of this book.

1.2.6 Model Evolution

One of the main difficulties of software evolution is that all artefacts produced and used during the entire software life-cycle are subject to changes, ranging from early requirements over analysis and design documents, to source code and executable code. This fact automatically spawns many subdisciplines in the research domain of software evolution, some of which are listed below:

Requirements evolution. The main objectives of *requirements engineering* are defining the purpose of a software system that needs to be implemented. Requirements evolve because requirements engineers and users cannot predict all possible uses of a system, because not all needs and (often mutually conflicting) goals of the various stakeholders can be taken into account, and because the environment in which the software is deployed frequently changes as well. Because the topic of requirements evolution is not covered in this book, we direct the reader to [571, 570, 191] for more information.

Architecture evolution. Based on an (initial) description of the software requirements, the overall software architecture (or high-level design) and the corresponding (low-level) technical design of the system can be specified. These are inevitably subject to evolution as well. The topic of architectural evolution is explored in detail in Chapter 10. The related problem of evolving software product families is not covered in this book, but we refer to [253, 252] for an in-depth treatment of this topic.

Data evolution. In information systems and other data-intensive software systems it is essential to have a clear and precise description of the database schema. Chapter 6 explores in detail how to evolve and migrate such schemas.

Runtime evolution. Many commercial software systems that are deployed by large companies need to be constantly available. Halting the software system to make changes cannot be afforded. Therefore, techniques are needed to change the software while it keeps on running. This very challenging problem is known under a variety of terms, including *runtime evolution*, *runtime reconfiguration*, *dynamic adaptation* and *dynamic upgrading* [297, 284].

Service-oriented architectures (SOA) provide a new paradigm in which a user-oriented approach to software is taken [162]. The software is developed in terms of which services are needed by particular users, and these users should be able to easily add, remove or adapt services to their needs. While this approach has many similarities with the component-oriented approach [486], services are only bound together at runtime, whereas components are statically (i.e., at design time) composed together. A service-oriented approach thus promises to be inherently more flexible than what is available today. This is crucial, especially in e-commerce applications, where rapid and frequent change is a necessity in order to respond to, and survive in, a highly competitive market. Chapter 7 of this book will be devoted to the migration towards service-oriented architectures.

Language evolution. When looking at languages (whether it be programming, modelling of formal specification languages), a number of research directions come to mind. The first one is the issue of co-evolution between software and the language that is used to represent it. Both are subject to evolution, albeit at different speed [167]. The second challenge is to provide more and better support for evolution in the context of multi-language software systems. A third challenge is to improve the design of languages to make them more robust to evolution (e.g., traits [451]). This challenge has always been the main driver of research in design of new computer languages. Unfortunately, every new programming paradigm promises to improve the software development process but introduces its own maintenance problems. This was the case for object-oriented programming (where the inheritance hierarchy needs to be mastered and kept under control when evolving software), aspect-oriented programming (where aspects need to be evolved next to the base code, see Chapter 9 for more details), component-oriented programming, and so on. In general, every new language or technology should always be evaluated in the light of its potential impact on the software's ability to evolve.

Interestingly, when starting to study evolution of software artefacts different from source code, new challenges arise that need to be dealt with, regardless of the type of software artefact under consideration. For example, we need techniques that ensure a *traceability* link between software artefacts at all different levels of abstraction, ranging from very high-level requirements documents to low-level source code [16].

In presence of many different types of software artefacts that co exist, we also need *inconsistency management* and *consistency maintenance* techniques to control the overall consistency of the software system [471], as well as techniques for *co-evolution* and *incremental synchronisation* of all related software artefacts [363].

1.3 Roadmap

The remainder of the book is structured into three parts, each containing at least three chapters. All chapters provide a detailed overview of relevant research literature.

Part I of the book, called *Understanding and Improving Software Evolution* is about understanding software evolution by analysing version repositories and release histories, and improving software evolution by removing software redundancies and fixing bugs:

- In Chapter 2, Koschke discusses and compares various state-of-the-art techniques that can be used to detect and remove software clones. In addition, he describes techniques to remove clones through refactoring and summarises studies on the evolution of clones.
- In Chapter 3, D'Ambros et al. report on how information stored in version repositories and bug archives can be exploited to derive useful information about the evolution of software systems.
- In Chapter 4, Zimmermann et al. explore how information about software failures contained in a bug database can be mined to predict software properties and to improve the software quality. Their results are validated on a number of industrial case studies.

Part II of the book, called *Reengineering of Legacy Systems* contains three chapters devoted to the topic of legacy software systems, and how one may migrate to, or reengineer these systems into a system that is no longer outdated and more easy to maintain and adapt:

- In Chapter 5, Demeyer discusses the state-of-the-art in object-oriented software reengineering. In particular, he focuses on the techniques of *refactoring* and *reengineering patterns*, and shows how these techniques can be used to capture and document expert knowledge about reengineering.
- In Chapter 6, Hainaut et al. address the problem of platform migration of large business applications and information systems. More specifically, they study the substitution of a modern data management technology for a legacy one. They develop a reference framework for migration strategies, and they focus on some migration strategies that minimize program understanding effort.
- In Chapter 7, Heckel et al. discuss an important research trend, namely the migration of legacy software systems to web services and service-oriented architectures by introducing architectural styles. In particular, they report on experience with an industrial case study in the context of a European research project, relying on the technique of graph transformation.

Part III of the book, called *Novel Trends in Software Evolution* addresses the relation between software evolution and other essential areas of software engineering such as software testing, software architectures, aspect-oriented software development, and open source software.

- In Chapter 8, van Deursen et al. discuss the current state of research and practice on the interplay between software evolution and software testing. In particular, they discuss and compare approaches for regression testing, unit testing (and the impact of refactoring on unit tests), test smells, and many more. They also consider tool support for test comprehension.
- In Chapter 9, Mens and Tourwé highlight some evolution-related issues and challenges that adopters of aspect-oriented software development approaches encounter. They discuss state-of-the-art techniques addressing the issues of aspect mining, extraction and evolution, and point out some issues for which no adequate solutions exist yet. This chapter can serve as a guideline for adopters of aspect technology to get a better idea of the evolution issues they may confront sooner or later, of the risks involved, and of the state-of-the-art in the techniques currently available to help them in addressing these issues.
- In Chapter 10, Barais et al. provide a detailed treatment of state-of-the-art approaches to evolving software architectures. In addition, they discuss in more detail *TranSAT*, one particular framework for software architecture evolution. The proposed solution combines ideas from aspect-oriented software development with architectural description languages.
- In Chapter 11, Fernandez-Ramil et al. discuss state-of-the-art techniques to study characteristics of evolving open source systems and their processes based on empirical studies. Results of the application of these techniques are given, including growth patterns, productivity, complexity patterns, social networks, cloning, processes and quality in open source systems, and so on.