Chapter 1
Introducing Managed Software Evolution

Ralf Reussner, Michael Goedicke, Wilhelm Hasselbring, Birgit Vogel-Heuser, Jan Keim, and Lukas Märtin

"Software eats the world!" Although this famous statement by the entrepreneur Marc Andreesen targets the disruptive change of business models enabled through software, it also describes a process ongoing over decades. Software already invaded basically all parts of our daily lives, at work as well as in private affairs. As a consequence, there is software in daily use to support critical processes in enterprises, machines, or production systems, which was initially developed decades ago. And still this software needs to be maintained and adopted to newly required functionality or modern information technology (IT) platforms. Estimations exist that assume that more than half of software budgets are spent in software maintenance [Gla01]. Sommerville states that the costs for running, maintaining, and evolution exceed

R. Reussner (✉) · J. Keim
Institute for Program Structures and Data Organization, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
e-mail: reussner@kit.edu; jan.keim@kit.edu

M. Goedicke
paluno – The Ruhr Institute for Software Technology, Specification of Software Systems, Universität Duisburg-Essen, Essen, Germany
e-mail: michael.goedicke@s3.uni-due.de

W. Hasselbring
Software Engineering Group, Department of Computer Science, Kiel University, Kiel, Germany
e-mail: hasselbring@email.uni-kiel.de

B. Vogel-Heuser
Technische Universität München, Institute of Automation and Information Systems, Garching, Germany
e-mail: vogel-heuser@tum.de

L. Märtin
Institute for Programming and Reactive Systems, Technische Universität at Braunschweig, Braunschweig, Germany
e-mail: l.maertin@tu-braunschweig.de

R. Reussner et al. (eds.), *Managed Software Evolution*,
https://doi.org/10.1007/978-3-030-13499-0_1

the development costs by a factor of at least two up to 100 [Som10]. Empirical studies from the industry support these numbers [Erl00, PA98, PM97]. However, it is actually even not clear how to interpret such figures. Are they bad signs, showing efficiency problems in maintenance, or are they good signs, showing that software is sufficiently good and valuable, that its maintenance is justified, as opposed to just throwing the software away and writing a new solution from scratch. Anyhow, the effects of software deterioration through long-running maintenance are well documented [Par94, VB02].

In the past, there were plenty of examples where software maintenance was challenging at least. The *Year 2000 problem*, also known as *Millenium bug*, struck many systems. Planning for long-living software did not factor in the turn of the millennium, causing a wide range of different problems. In 2009, customers of T-Mobile had no voice services or short message services (SMSs) available for several hours. Although the case was not a maintenance problem, at least the lack of knowledge about structural and architectural dependencies delayed the fix considerably. Flawed software on security chips of EC and credit cards caused a problem in 2010 because the card readers could not process the year properly. In 2016, the update of the operating system iOS for iPhones to version 10 caused alarm clocks to not go off any more because the new "bedtime alarm mode" interfered with the functionality of the existing alarm clock.

All these examples have in common the belief that problems could be related to lack of knowledge about the already existing system. From a bird's-eye view of software engineering, it is clear that knowledge is created during the process of developing software, but most of the time this type of knowledge is not documented. This leads to loss of knowledge about these systems, which can lead to problems in following development cycles and during maintenance. The results are much higher mean time to repair and much longer cycles until a new version of a system can be released. Additionally, lack of knowledge can also lead to more bugs, thus leading to a lower mean time to failure. In some cases, updates even introduce problems that were previously known and solved. For example, in 2017 an update for macOS accidentally reintroduced the critical "root bug".[1] Already in 1994 Parnas described the concept of hidden and lost knowledge [Par94]. Because of size and complexity, along with the interconnectedness of software systems, this problem gets worse.

Up to now, the focus in research and practice is mainly on developing new systems. New methods and tools are developed and existing ones refined to create optimal results for the initial operation of software systems. However, the long-term operation phase, along with the necessary adjustments and further development of software, is of paramount importance. This problem gains more weight when factoring in higher costs for maintenance and evolution in comparison with initial developments. Even in the research field of software evolution, the aspect of different evolutionary cycles for software and its execution and operating environment is yet not properly dealt with. The different life cycles of software and

[1] https://www.wired.com/story/macos-update-undoes-apple-root-bug-patch/.

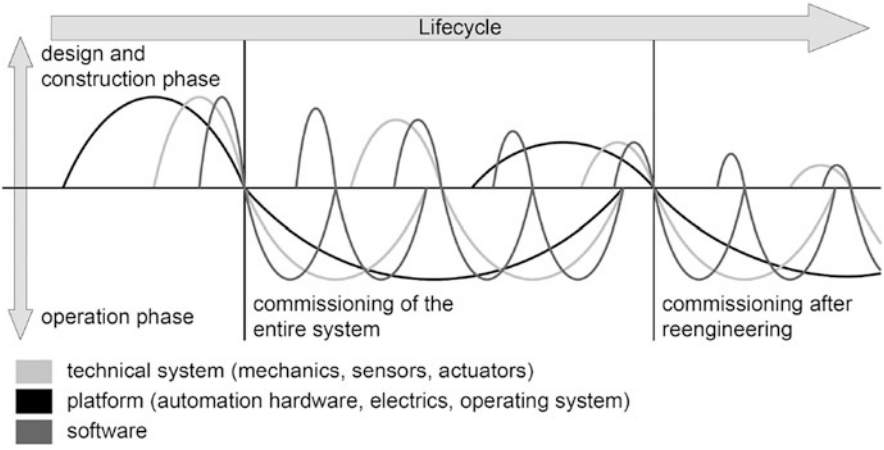

Fig. 1.1 Integration of the development and operation of hardware/software systems [Li+12]

its platform, as well as the technical systems, are shown in Fig. 1.1. There you can see the life cycles of software in grey, platforms in black, and technical systems in light grey. Vertical amplitudes show in which phase the software, platform, or technical system is. Life cycles of platforms are much longer, starting with longer design phases, and have much longer operation phases. In contrast, software is pretty short-lived and new versions replace older software rather fast. These differences need to be addressed. Some techniques used to simplify the creation of new software systems like Software-as-a-Service can lead to problems in combination with these varying life cycles. Although the potential exchange of services is seen as a benefit on evolution, published service interfaces are even harder to change than internal interfaces; hence, services lead to frozen interfaces hindering evolution. In addition, the required complex technology stack creates many—often undocumented—dependencies, which makes evolution to new platforms even harder, as the knowledge to decouple business functionality and platforms is lost rather soon after the initial development.

This leads us to problem areas for long-living software systems, which are explained in the following. Firstly, lacking understanding and knowledge about functionality, structure, dependencies, and other properties of software systems impedes a proper evolution of these systems, which are in agreement with the originally stated requirements. This leads to a deficit in those systems. Secondly, functional correctness and conformity with the architecture can often not be guaranteed because of misunderstood methods and techniques for software evolution. Finally, the complexity of development from the functional point of view on one side and the development of platforms and technologies on the other side obstruct each other regularly and are hindering the evolution of applications and application systems.

This shows that there is the necessity to make systems adaptable to changing requirements and environments and to make knowledge about systems accessible. Additionally, instead of separating the development, adaptation, and evolution of software and their platforms, as well as operation, monitoring, and maintenance, all these should be integrated into the process. This new paradigm should be developed and elaborated. For this we created three major guiding themes that are explained below.

Following this line of motivation, the German Research Council ("Deutsche Forschungsgemeinschaft (DFG)") initiated in 2012 the Priority Programme "Design for Future – Managed Software Evolution", to develop fundamentally new approaches in software engineering with a determined focus on long-living software systems. Over its funding periods, 59 proposals were evaluated by a board of scientifically outstanding international reviewers from the fields of software engineering and automation technology (see Board of reviewers section). The accepted 14 projects for each funding period included in total over 50 researchers and 31 principal investigators. As an anchor for these projects, three guiding themes were put into foreground, namely:

"Knowledge carrying software"
: This is the overarching theme of the whole Priority Programme. The principle of this guiding theme is that knowledge contained in software or its underlying design needs to be integrated and made accessible, both for functional and for quality properties. To realise this, sophisticated meta-models need to be developed for defining and managing suitable models.

"Methods and processes"
: They have to ensure that knowledge is preserved and integrated into the design and evolution of software. Therefore, a new model for the life cycle of software or software/hardware systems needs to be developed. This model needs to allow and consider different evolution cycles on different levels of the software, platform, and hardware stacks.

"Platforms and environments for evolution"
: One goal is to develop suitable middleware and robust runtime environments for monitoring and updating during operation to provide infrastructure for the evolution of software and software/hardware systems. It is an important principle for this guiding theme that design and runtime information need to be made accessible wherever needed during the operation of systems.

In Fig. 1.2 the three guiding themes are set in relation to relevant fields of research for today's software engineering [Gol+15]. The guiding themes are embedded into various areas of software engineering like requirements management, software architecture design, artefact management, and operation and infrastructure. All these areas play an influential role for the Priority Programme.

As a second means for project integration, the Priority Programme established two community case studies: the Common Component Modelling Example (CoCoME) for business-oriented software systems and the Pick-and-Place Unit

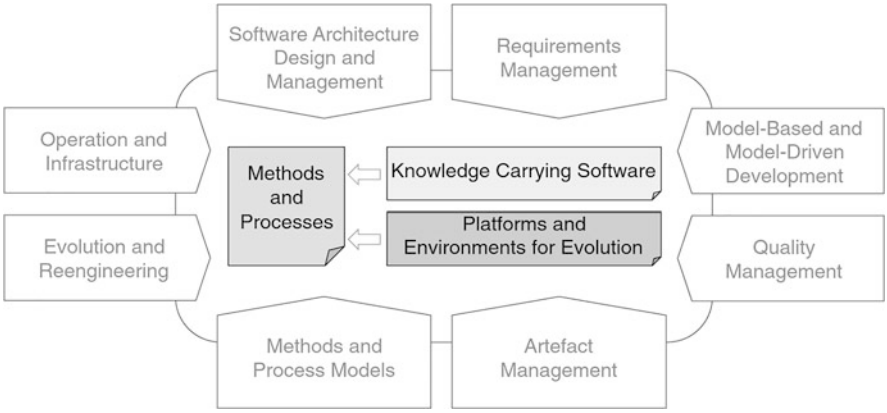

Fig. 1.2 Guiding themes related to current research in software engineering [Gol+15]

(PPU) as an exemplary automated production system. Each of the projects contributed to at least one of these studies.

Although or even because several hundred internationally high-ranked scientific publications were created during the course of the Priority Programme, the principal investigators see the need for a more integrated way to present the results to the scientific community and to academically trained practitioners in the field. Therefore, we wrote this book, with all the projects contributing in an integrated way. Hence, the following chapter overview describes the overarching results of the Priority Programme.

Overall the book is split into three major parts. The first part of the book deals with introductions into the topics. In Chap. 2, an introduction to the nature of software evolution is given, followed by the challenges that occur in Chap. 3. Lastly in this part, an introduction to the case studies we used is given in Chap. 4. In the second part of the book, there are the main chapters about knowledge-carrying software, starting with Chap. 5 on tacit knowledge in software evolution. Next, continuous design decision support will be covered in Chap. 6. Chapter 7 covers SPL round-trip engineering, followed by performance analysis strategies in Chap. 8. Maintaining security in software evolution is tackled in Chap. 9, before the topic about learning from evolution for evolution, which is tackled in Chap. 10. This second part in the book is completed with Chap. 11 on formal verification of evolutionary changes. Finally, the last part of the book presents results and spin-offs. There, Chap. 12 describes the case studies for the community, along with their benefits and deliverables. The lessons learned are collected in Chap. 13. We close the book in Chap. 14 with an overview of future research topics.

Chapters without author names are written by the editors of the book, while other chapters refer to the scientists who contributed as authors. A complete author list can be found at the end of the book. The editors would like to thank all the authors for their considerable effort in writing a cohesive book on the results of

the Priority Programme. Additionally, we would like to thank all the authors who peer reviewed the chapters, which helped improve the quality of this book. We also would like to thank the office of the DFG, in particular Dr. Gerrit Sonntag and Dr. Andreas Raabe and their teams, for all their administrative support and for organising the review process. We also like to thank very cordially our international reviewers, who not only evaluated projects but also provided us with very valuable feedback during the whole funding period of the Priority Programme. Special thanks go to the managers of the Priority Programme, Dr. Lukas Märtin und Jan Keim, who served and organised the whole programme in an excellent manner and also managed the writing process of this book extremely well. We also want to thank Prof. Dr. Wilhelm Schäfer, who supported us invaluably as a programme director before his health condition unfortunately disallowed further contributions. We are also deeply indebted to Prof. Dr. Ursula Goltz, the first speaker of the coordination board. She successfully brought the programme through the review and application process and set it up in 2012, leading the programme during its first phase. Her unfortunate and sudden health problems made it impossible for her to carry on with this responsibility. We wish her good luck and furthermore a good recovery.