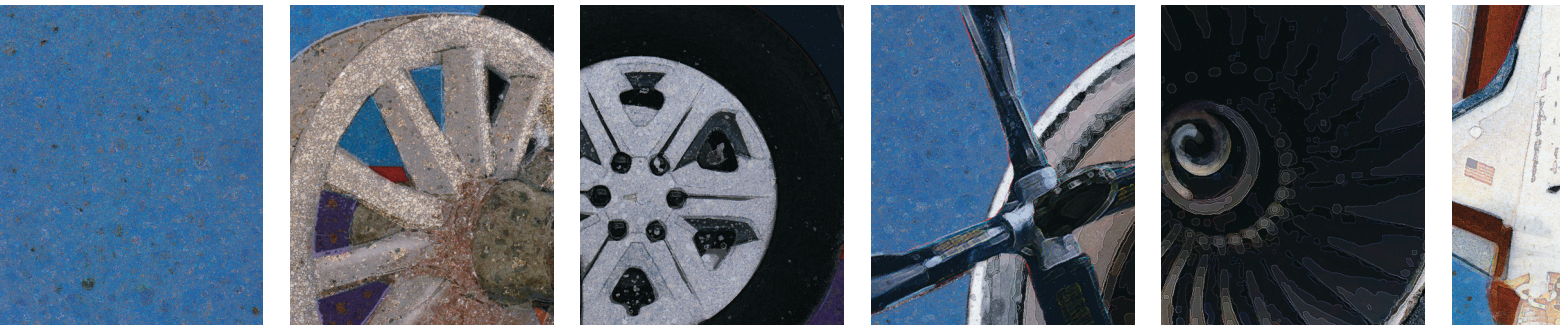

Software Evolution

Tom Mens, *Université de Mons*

Yann-Gaël Guéhéneuc, *École Polytechnique de Montréal*

Juan Fernández-Ramil, *The Open University, UK*

Maja D'Hondt, *Imec*

Modern society depends heavily on software systems. Software can enable or even accelerate human, social, economic, and technological changes. Software systems must often reflect these changes to adequately fulfill their roles and remain relevant to stakeholders, but the number of new requirements and maintenance requests often grows faster than software owners' abilities to implement them. Evolving and maintaining these systems is therefore critical and, consequently, most developers work on maintaining, incrementally enhancing, and adapting existing systems.

After two decades of growing interest in software evolution and related topics (see the sidebar "Software Evolution Concepts and Terminology"), many research groups are now working in this area and producing a growing body of knowledge (see the sidebar "Related References and Further Reading"). In May 2010, Google Scholar reported that for 2009, 70 publications had "software evolution" in the title, and more than 900 had "software evolution" somewhere in the text. Google Scholar data also shows, for these evolution-related publications, an increase of almost one order of magnitude between the yearly publication rates in the 1990s and those of the last decade.

This special issue brings this important topic further into the spotlight by providing a sample of recent research results with tools, lessons, and patterns that are applicable to real-world cases. We hope that this will stimulate further consideration, awareness, and research into software evolution.

Lehman's Laws

Practitioners involved in software evolution are likely to face, knowingly or not, some of the con-

0740-7459/10/$26.00 © 2010 IEEE

Related References and Further Reading on Software Evolution

Books

- *Software Evolution*, T. Mens and S. Demeyer, eds., Springer, 2008.
- *Software Evolution and Feedback—Theory and Practice*, N. Madhavji, J. Fernández-Ramil, and D. Perry, eds., Wiley, 2006.
- *Program Evolution: Processes of Software Change*, M.M. Lehman, L.A. Belady, eds., Academic Press, 1985; ftp://ftp.umh.ac.be/pub/ftp_infofs/1985/ProgramEvolution.pdf.

Website

- The European Research Consortium for Informatics and the Mathematics (ERCIM) Working Group on Software Evolution hosts a wiki with PhD theses, events, projects, tools, terminology, and more at http://wiki.ercim.eu/wg/SoftwareEvolution.

Articles

- M.M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution," *Proc. IEEE*, IEEE Press, vol. 68, no. 9, 1980, pp. 1060–1076.
- E.J. Chikofsky and J.H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, vol. 7, no. 1, 1990, pp. 13–17; doi: 10.1109/52.43044.
- K.H. Bennett and V.T. Rajlich, "Software Maintenance and Evolution: A Roadmap," *Proc. Conf. Future of Software Eng.*, ACM Press, 2000, pp. 73–87; doi: 10.1145/336512.336534.
- N. Chapin et al., "Types of Software Evolution and Software Maintenance," *J. Software Maintenance and Evolution: Research and Practice*, vol. 13, no. 1, 2001, pp. 3–30; doi: 10.1002/smr.220.
- T. Mens et al., "Challenges in Software Evolution," *Proc. 8th Int'l Workshop on Principles of Software Evolution*, IEEE CS Press, 2005, pp. 13–22; doi: 10.1109/IWPSE.2005.7.
- M. Petrenko et al., "Teaching Software Evolution in Open Source," *Computer*, vol. 40, no. 11, 2007, pp. 25–31; doi: 10.1109/MC.2007.402.
- A. van Deursen et al., "Model-Driven Software Evolution: A Research Agenda," *Proc. Int'l Workshop on Model-Driven Software Evolution* (MoDSE07), Vrije Universiteit Amsterdam, 2007, pp. 334–350; www.cs.vu.nl/csmr2007/workshops/p19.pdf.
- H.M. Sneed, "The Drawbacks of Model-Driven Software Evolution," *Proc. Int'l Workshop Model-Driven Software Evolution* (MoDSE07), Vrije Universiteit Amsterdam, 2007; www.cs.vu.nl/csmr2007/workshops/p9.pdf.
- M. Godfrey and D. German, "The Past, Present and Future of Software Evolution," *Proc. Int'l Conf. Software Maintenance*, IEEE CS Press, 2008; doi: 10.1109/FOSM.2008.4659256.

Standards and Similar Sources

- ISO/IEC 14764 IEEE Std. 14764-2006, *Software Engineering—Software Life Cycle Processes—Maintenance*, IEEE, 2006; doi: 10.1109/IEEESTD.2006.235774.
- *The Guide to the Software Engineering Body of Knowledge* (SWEBOK; www.swebok.org) includes a chapter on software maintenance, including measurement, planning, and support.
- The IT Infrastructure Library (ITIL; www.itil-officialsite.com/home/home.asp) consists of concepts and good IT management practices. Software project managers can use ITIL in planning and organizing software maintenance processes (for example, service support and delivery, software asset management, and other practical aspects of IT management).

straints implied by the laws of software evolution Meir M. Lehman introduced in the 1970s. These laws or, rather, empirical hypotheses, suggest that any actively used software system must continually change to satisfy its stakeholders. On the basis of such changes, the system gradually degrades over time because such changes are likely to increase its complexity. If the complexity isn't controlled (for example, via refactoring in object-oriented systems or restructuring), then evolution costs can escalate, leading to even more complex changes—a vicious circle—until the software owners must make a radical decision, such as reengineering, migration, redevelopment, or replacement. But even replacement by an off-the-shelf system doesn't avoid the need for evolution: the software's owner, whether it's a vendor or the open source community, must still implement the changes.

Articles in This Issue

Software evolution spans a wide range of topics within software engineering. This special issue opens with two invited and complementary perspectives on software evolution by two well-known authors in software engineering, Kent Beck and Barry Boehm. Boehm argues that the time of "one software evolution process fits all" is over and provides guidelines to select the most appropriate evolution-friendly process under various circumstances. Beck acknowledges the inevitability and difficulty of evolving a software design. He reminds us of the various factors that software

Software Evolution Concepts and Terminology

We present a short glossary of important terms in software evolution to accompany the contributions to this special issue.

Software maintenance is different from the maintenance of physical goods because it normally leads to a changed or improved software system. Physical goods maintenance seeks to restore the item or entity (for example, an airplane or a car) as close as possible to factory condition. Software doesn't wear out but it "degrades" through suboptimal "quick fixes" and through an increasing number of unfulfilled requirements. Nontrivial software isn't free from defects: some maintenance is always required as long as the software is in use.

Software evolution is also different from the evolution of physical goods. It's mainly concerned with changes in a software system over versions or releases of the same system, while physical goods tend to evolve over "generations of products" (that is, there's little or no evolution of a specific physical product).

There are various definitions of software evolution and software maintenance. The ISO/IEC/IEEE 14764:2006 standard states that software maintenance is "the totality of activities required to provide cost-effective support to a software system."[1] This standard includes pre-delivery (planning and logistics) and post-delivery activities (actual software modification) as part of maintenance work. One view considers both evolution and maintenance as synonyms that cover all the work done to update and fix a software system after its first release. Some practitioners might find it more natural to refer to evolution when functionality or some other aspect of the software is improved, with maintenance referring mainly to essential adaptations and fixes to keep the software running. Keith H. Bennett and Václav T. Rajlich consider evolution a stage of the software life cycle.[2] A complementary view is to consider software maintenance as wider and more encompassing than software evolution. The latter concerns updating and changing the code and other technical artifacts, while maintenance also includes regression testing, administrative, and support activities such as help desks and technical support. Ned Chapin and his colleagues provide a comprehensive classification of software maintenance and evolution work,[3] beyond the classical corrective, preventative, adaptive, and perfective in the ISO/IEC/IEEE 14764:2006 standard.

A legacy software system is any system that significantly resists modifications and changes while remaining of significant value for its owner. The system might have been developed using an outdated programming language or an obsolete development method. Most likely, it has changed hands several times and shows many signs of degradation.

Reengineering is "the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Reengineering generally includes some form of reverse engineering (to achieve a more abstract description) followed by some form of forward engineering or restructuring. This may include modifications to fulfill new requirements not met by the original system."[4]

Forward engineering is "the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system."[4]

Restructuring is "the transformation from one representation form [of a software system] to another, at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics)."[4]

Refactoring is the object-oriented specialization of restructuring. When applied to source code, the goal of both is to improve the internal structure (for example, reduce complexity) of software to make it easier to understand and modify.

Reverse engineering is "the process of analyzing a subject system to identify its components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction. Reverse engineering generally involves extracting design artifacts and building or synthesizing abstractions that are less implementation-dependent."[4]

References

1. *ISO/IEC 14764 IEEE Std. 14764-2006, Software Engineering—Software Life Cycle Processes—Maintenance*, IEEE, 2006; doi: 10.1109/IEEESTD.2006.235774.
2. K.H. Bennett and V.T. Rajlich, "Software Maintenance and Evolution: A Roadmap," *Proc. Conf. Future of Software Eng.*, ACM Press, 2000, pp. 73–87; doi: 10.1145/336512.336534.
3. N. Chapin et al., "Types of Software Evolution and Software Maintenance," *J. Software Maintenance and Evolution: Research and Practice*, vol. 13, no. 1, 2001, pp. 3–30; doi: 10.1002/smr.220.
4. E.J. Chikofsky and J.H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software, vol. 7, no. 1, 1990, pp. 13–17; doi: 10.1109/52.43044.

developers and managers must consider when evolving systems (such as cost, time, and risk), including the need to keep a system operational during its actual evolution, which is particularly relevant to some critical applications.

This special issue also includes five peer-reviewed articles that provide samples of industrially relevant research results aimed at facilitating various aspects of software evolution. They cover model-driven software evolution, architecture-driven modernization, reengineering legacy systems, program refactoring, reverse engineering, and software quality assessment and improvement during evolution.

Joris Van Geet and Serge Demeyer highlight the difficulties of applying mature techniques for software evolution in practice. They indirectly justify the need for increased technological transfer between research and practice. For example, although researchers have studied feature identifica-

tion and other techniques for years, industry's acceptance of such techniques is low, mostly because of the lack of out-of-the-box implementations.

By highlighting the need for a more practical approach to software evolution, Javier Luis Cánovas Izquierdo and Jesús García Molina illustrate how the architecture-driven modernization initiative by the OMG (Object Management Group) can concretely help in evolving legacy PL/SQL (Procedural Language/Structured Query Language) code to Java code.

Eric Bouwers and Arie van Deursen discuss when to perform evolution, while introducing a lightweight approach to monitor a system's architecture. How to perform evolution has been the subject of much research work, and Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni present a tool that evaluates whether an applied refactoring appears to be harmless by automatically generating relevant test cases, which can then be executed to check that the system's behavior is preserved and, consequently, added to the system's overall test suite. These approaches give developers confidence that they're performing the right kind of evolution at the right moment. They can also help convince managers that the time is right to perform evolution tasks.

However, managers could still consider that evolution tasks, technically sound, are a waste of time. Hongyu Zhang and Sunghun Kim address this dilemma by showing how to visualize defect trends and study patterns that might appear. They do so through the simple means of C-charts, which are commonly used in statistical quality control.

If you want to find out more, we provide a list of recommended articles and books in the sidebar "Related References and Further Reading." You could also attend one of the annual conferences or workshops covering this topic, such as the IEEE International Conference on Software Maintenance (ICSM), the IEEE European Conference on Software Maintenance and Reengineering (CSMR), or the International Workshop on Principles of Software Evolution (IWPSE).

Acknowledgments

We thank all the authors who contributed to this special issue as well as the anonymous reviewers and the administrative staff of *IEEE Software*. Our *IEEE Software* mentor, Martin Robillard, contributed his experience and insights into many aspects of this issue's preparation. We're also grateful to editor-in-chief Hakan Erdogmus, lead editor Dale Strok, content editor Brian Brannon, and administrator Mercy Frederickson for promptly and professionally answering the many emails and queries, and helping out when needed. **Yann-Gaël Guéhéneuc is supported by the Fonds Québécois de la Recherche sur la Nature et les Technologies (FQRNT), the Natural Sciences and Engineering Research Council, and the Canada Research Chair Tier II in Software Patterns and Patterns of Software. Part of Juan Fernandez-Ramil's work related to this special issue was funded by the Belgian Fonds de la Recherche Scientifique (FRS—FNRS) through postdoctoral grant number 2.4519.05. Tom Mens is supported by Action de Recherche Concertée AUWB-08/12-UMH19 "Model-Driven Software Evolution" funded by the Ministère de la Communauté française—Direction générale de l'Enseignement non obligatoire et de la Recherche scientifique (Belgium), and by the project Technologies d'Information et de Télécommunication (TIC) co-funded by the European Regional Development Fund and the Walloon Region (Belgium).**

About the Authors

Tom Mens is a professor directing the Software Engineering Lab at the Institut d'Informatique, Faculty of Sciences, Université de Mons. His research interests are in formal foundations and automated tool support for software evolution, as well as in model-driven software engineering. Mens has a PhD in software evolution from Vrije Universiteit Brussel. He's a member of IEEE, the IEEE Computer Society, the ACM, the European Research Consortium for Informatics and Mathematics (ERCIM), and the European Association of Software Science and Technology (EASST). He co-edited *Software Evolution* (Springer, 2008) with Serge Demeyer. Contact him at tom.mens@umons.ac.be.

Yann-Gaël Guéhéneuc is an associate professor in the Department of Computing Engineering and Software Engineering at Ecole Polytechnique of Montreal where he leads the Ptidej (pattern trace identification, detection, and enhancement in Java) team. In 2009, the Natural Sciences and Engineering Research Council awarded him the Research Chair Tier II on software patterns and patterns of software. His research interests include evaluating and enhancing the quality of object-oriented programs by promoting the use of patterns, at the language-, design-, and architectural level. Guéhéneuc has a PhD in software engineering from the University of Nantes. He's a member of IEEE, the European Research Consortium for Informatics and Mathematics (ERCIM), and the Order of the Engineers of Quebec. Contact him at yann-gael.gueheneuc@polymtl.ca.

Juan Fernández-Ramil is a lecturer in the computing department at The Open University, UK, where he's involved in distance teaching of software management and relational databases. His research interests include software metrics, software maintenance productivity, and code decay assessment. Fernández-Ramil has a PhD in computing from Imperial College London. He co-edited *Software Evolution and Feedback: Theory and Practice* (Wiley, 2006) and co-authored *Managing the Software Enterprise: Software Engineering and Information Systems in Context* (Cengage Learning EMEA, 2007). Contact him at j.f.ramil@open.ac.uk.

Maja D'Hondt is a program manager at Imec Belgium, an independent research center for microelectronics and information and communication technology. She's responsible for developing software technology for adaptive resource management in embedded and high-performance systems. Additionally, she's a guest professor at the Vrije Universiteit Brussel. D'Hondt has a PhD in computer science from Vrije Universiteit Brussel. Contact her at maja.dhondt@imec.be.