Towards Software Sustainability Guidelines for Long-living Industrial Systems

Heiko Koziolek[1], Roland Weiss[1], Zoya Durdik[2], Johannes Stammel[2], Klaus Krogmann[2]

[1]Industrial Software Systems, ABB Corporate Research Ladenburg, Germany
[2]Forschungszentrum Informatik (FZI), Karlsruhe, Germany
heiko.koziolek@de.abb.com

Abstract: Long-living software systems are sustainable if they can be cost-effectively maintained and evolved over their complete life-cycle. Software-intensive systems in the industrial automation domain are typically long-living and cause high evolution costs, because of new customer requirements, technology changes, and failure reports. Many methods for sustainable software development have been proposed in the scientific literature, but most of them are not applied in industrial practice. We identified typical evolution scenarios in the industrial automation domain and conducted an extensive literature search to extract a number of guidelines for sustainable software development based on the methods found in literature. For validation purposes, we map one evolution scenario to these guidelines in this paper.

1 Introduction

Software systems in the industrial automation domain are typically *long-living* systems, i.e., some of them may operate for more than 20 years, because of the investment in the underlying machines and devices. Examples for such systems are distributed control systems to manage industrial processes, such as power generation, manufacturing, chemical production, or robotics systems to automate manual tasks, such as welding, pick&place, or sealing. Such systems are usually implemented with a variety of technologies and have very high requirements for safety, performance, and availability.

Because of their long life-cycles and complicated replacement procedures, industrial software systems are continuously maintained and evolved resulting in high costs, which amount for a large portion of the overall 10 billion dollar market. Thus, industrial automation companies, such as ABB, are interested in creating *sustainable* long-living software systems, i.e., systems that can be cost-effectively maintained over their complete life-cycle. This is a complex challenge, as industrial software systems are continuously subject to new requirements, new standards, failures, and technology changes during their operation time.

While a vast variety of methods for the cost-effective evolution of software systems have been proposed in the academic literature [CHK+01, GG08], it is difficult to select from these methods and to determine upfront whether they can address specific evolution prob-

lems. Ali Babar et al. [BG09] found in a survey that only 24% of the interrogated software architects were aware of scenario-based architecture analysis methods that can be used for the evaluation of evolution scenarios. A state-of-the-art overview and guidelines for software developers are missing, which could increase the willingness to apply the proposed, sophisticated methods in practice. For the industrial automation domain, a mapping from proven methods to typical evolution scenarios is required.

The contributions of this paper are (i) a list of typical evolution scenarios in the industrial automation domain, which can be used by third parties in evolution assessments and (ii) a sketch of initial sustainability guidelines based on an extensive literature survey. For the guidelines, we identified the most mature methods of sustainability and categorized them according to the development phases of a software system. In this paper, we provide a preliminary validation by mapping one sample evolution scenario to these guidelines to show that they are readily applicable in practice. A full validation of the guidelines is planned through several future case studies, which we briefly sketch in this paper.

The rest of this paper is structured as follows: Section 2 lists some re-occurring evolution scenarios in the industrial automation domain. Section 3 summarizes our preliminary guidelines for developing sustainable software systems structured against different development phases. Section 4 provides an initial validation of the guidelines by mapping them to one sample scenario and sketches planned validation activities. Finally, Section 5 discusses related approaches and surveys and Section 6 concludes the paper.

2 Evolution Scenarios in the Industrial Automation Domain

2.1 The Industrial Automation Domain

Industrial automation deals with the use of control systems and information technology to reduce manual work in the production of goods and services. While industrial automation systems originate from manufacturing, now there are many other application scenarios. Industrial control systems are for example used for power generation, traffic management, water management, pulp and paper handling, printing, metal handling, oil refinery, chemical processes, pharmaceutical manufacturing, or carrier ships.

Different types of software applications are used in industrial automation:

- Domain specific applications: e.g., control loops according to IEC 61131-3, typically below 50 KLOC
- Embedded software applications: e.g., running on controllers (e.g., ARM) interacting with custom hardware, programmed in C/C++, Assembler; OS: VxWorks, Linux, QNX, typically below 500 KLOC
- Large-scale software systems: e.g., distributed control systems (DCS), programmed in C++, C#, and Java; OS: Windows, Linux, millions lines of code

2.2 Evolution Scenarios

Industrial software systems have high requirements on reliability, safety, performance, security, and usability. Redundancy is often used on various levels to ensure reliability. Safety certifications are required in some domains for the controller software (e.g., according to IEC61508), which implies very high development standards. Performance issues are dangerous, because many systems are real-time critical, thus, missed deadlines imply a failure of the system and may harm human beings.

During their potentially more than 30 years life-cycles, these kinds of systems undergo various changes. Based on our experience with various software systems at ABB, we have identified a number of typical evolution scenarios. These scenarios highlight typical changes to the system and could be generalised to evolution scenario patterns. Future software systems could be evaluated with respect to these evolution scenarios. We have classified typical evolution scenarios for these systems according to ISO/IEC14765 into corrective, adaptive, perfective, and preventive scenarios. The following provides a selection of these scenarios.

Perfective scenarios modify a software product after delivery to improve for example the functionality, performance, or reliability. Some of the most frequent scenarios are listed below.

- new functionalities and services: customers continuously demand new features and functionality for industrial software (e.g., alarm management, history logging, optimizations)
- integration of third party components: special new functionality (e.g., certain data analysis functions) are sometimes already supported by third party vendors, whose components need to be integrated into an industrial system. Thus, writing adapters for these components is a common evolution scenario.
- integration of third party applications: industrial systems at customer sites often consist of software products from different vendors. Ensuring the interoperability between different applications is thus a continuous concern for industrial systems.
- integration of third party systems: industrial systems are often connected to manufacturing execution systems (MES) and/or enterprise resource planning systems (ERP). Their evolution often requires a corresponding evolution of the industrial software system
- safety certification: some parts of an industrial control system (e.g., controller devices) require an increasing safety certification (e.g., moving from IEC61508 SIL2 to SIL3) in certain application domains. For updated devices also the safety cases have to be updated or rebuilt.
- performance/scalability improvements: supporting more complex applications and processing more signals (e.g., measured in the maximum number of I/Os) from field devices is a latent requirement for industrial systems. For several software systems at ABB, a main feature of former revisions was the support for more devices and/or different devices.
- usability improvements: besides improving the layout of graphical user interfaces, also new functionality for improved usability is added to an industrial system, e.g., support

for multi-touch devices or multi-user support for engineering tools.
- security improvements: with the advent of service-oriented and web-based industrial control systems the amount of security issues has increased dramatically. Thus, there is a need to continuously supply security patches and enhancements to prevent attackers to exploit security weaknesses in the systems. The Stuxnet computer worm is a recent, highly publicized example, which required updated existing DCS systems.

Adaptive evolution scenarios refer to modification of a software product performed after delivery to keep a software product usable in a changed or changing environment. Examples of such scenarios are:

- new industry standards: industrial systems rely on multiple standards (e.g., OPC, IEC 61850), which are periodically revised or newly introduced. Compliance to these standards is thus a competitive factor and a re-occurring evolution scenario.
- migration to a new GUI framework: human machine interfaces are critical components in distributed control systems to allow operators efficiently running a plant. Replacing older frameworks (e.g., Visual Basic with WPF or Java AWT with SWT) is a re-occurring scenario and can have significant impacts on a system.
- migration to a new middleware: the communication between software components is often managed by a middleware, especially for distributed systems. New technologies or industry standards (e.g., OPC UA) require migration (e.g., from a COM-based communication to web services).
- support for new operating systems: operating systems typically have much shorter update cycles than industrial software systems. As industrial systems are often tightly integrated with operating systems, respective updates of the OS can require many modifications to the industrial software system.
- support for virtualization: for distributed control systems, new virtualization techniques allow saving server nodes and thus costs. Due to the real-time capabilities of the software special measures to support virtualization are required.
- support for multi-core processors: the general trend towards multi-threaded software is also present in the industrial domain. Increasing performance requirements due to more complex industrial processes lead to a desire to parallelize the software systems where possible to exploit multi-core CPUs.
- updated controller and field devices: many new devices are developed during the lifecycle of an industrial system. Their support is often critical to stay competitive.
- new network standards: while new network standards (e.g., Industrial Ethernet, WirelessHART) enable more efficient and fail-safe communication, they also often induce high migration efforts for existing systems

Corrective evolution scenarios refer to reactive modifications of a software product performed after delivery to correct discovered problems. In a former case study [KSB10], we analysed the bug tracking system of a large industrial control system and found that many bugs are still found by customers after release. The development of such a complex system cannot produce failure free software. While we could attribute failure reports to specific subsystems, we have not yet categorized the failure reports to derive bug-related

evolution scenarios, which remains future work..

Preventive evolution scenarios are modifications of a software product after delivery to detect and correct latent faults in the software product before they become effective faults. This may for example relate to refactoring or architecture improvements. From our current perspective in the industrial automation domain these activities similar to the activities in other domains.

3 Software Sustainability Guidelines

To tackle the evolution scenarios described in the former section and to ensure their cost-efficient implementation, we conducted a comprehensive literature survey [DKK+11] on analytical and corrective methods and tools for evolution issues. Our intent is to support developers in the industrial automation domain to be aware of state-of-the-art methods and their potential benefits and risks. This may allow a proactive approach towards sustainability and avoid firefighting evolution problems.

Our literature search targeted renowned journals and conference proceedings as well as books. We identified different kinds of analytical and constructive approaches. We extracted the most promising approaches based on our subjective judgment, which relied on the published industrial case studies and tool support, and summarized them in a guidelines document.

The guidelines document contains approx. 20 different approaches and is structured along the software development phases (e.g., requirements, architecture, implementation), but implies no waterfall model. The phases merely improve the orientation of the reader and are aligned with internal ABB guidelines for software engineering.

Our method descriptions focus on the relevance for sustainability and motivate the use of a method from the perspective of a developer or architect who wants to tackle an evolution scenario. We additionally coarsely estimated the learning and application effort of each method and provided references to respective tools. In order to focus on sustainability guidelines without providing generic software engineering best practices, we conducted several stakeholder interviews.

In the following, we briefly summarize selected methods and tools for each phase and discuss some of their risks (Fig. 3).

Sustainable Requirements: Documenting, prioritizing, analyzing, and tracing functional and non-functional requirements to an industrial software system is an important prerequisite for sustainability. Due to the complexity and longevity of industrial systems, tool support (e.g., DOORS, CaliberRM, HP Quality Center) is essential. Well managed requirements preserve the knowledge about the design rationale of a system and ease decision tracking. Requirements tracing is helpful for long-term development with changing personal as it preserves the knowledge about why certain design decisions were made or why certain parts of the code exist. Additionally, each requirement should be analysed for its potential impact on sustainability as early as possible to be able to improve the upfront

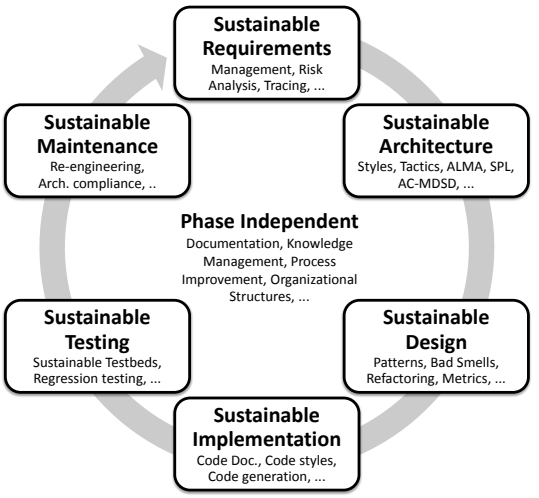

Figure 1: Structure of Software Sustainability Guidelines

design of a system.

Several *risks* are associated with sustainable requirements capturing. Requirements can be incomplete, contracting, or unstructured. Their impact on system sustainability is often not systematically analysed. Many future requirements during system evolution cannot be foreseen. Trade-offs between maintainability requirements and requirements for other quality attributes (e.g., performance, reliability, safety) require a thorough analysis. Especially for long-living systems, there is a danger that the requirements themselves are not maintained and updated during the evolution of the system, which then complicates further evolution.

Sustainable Architecture: Several heterogeneous approaches are known for designing sustainable software architectures. Scenario-based architecture evaluation methods (e.g., ATAM [CKK02], ALMA [BLBvV04]) can be used to elicit evolution scenarios from various stakeholders and to discuss their impacts on software components and reveal potential ripple effects through the architecture. Change-oriented architectural styles and patterns (e.g., layering, pipe-and-filter, plug-ins) and architectural modifiability tactics (e.g., localizing changes, deferring binding time [BCK03]) help to prepare the architectural design for evolution. Even more profound, software product lines help to manage system variability, while architecture-centric model-driven development is a potential mean to cope with platform adaption.

Risks associated to sustainable software architecture design are for example undocumented design rationale or too much flexibility. Missing design rationale complicates the evolution of a system and can potentially be mitigated through architecture knowledge management [BDLvV09]. While too few flexibility in the architecture limits evolution options, too much flexibility increases architectural complexity and thus maintainability (i.e., understandability, analyseability, modifiability).

Sustainable Design: Methods for improving the design (i.e., more low level structures,

code-related) of an industrial software system towards sustainability are the application of design patterns, refactoring, and code metric monitoring. Many design patterns [GHJV95] increase the flexibility and extensibility of code (e.g., abstract factory, bridge, decorator) and thus improve sustainability on a small scale. Bad smells are critical code areas which might incure high maintenance costs. They can be mitigated semi-automatically through refactoring tools [FBBO99]. To detect undesired structures in the code there are several tools (e.g., Findbugs, SISSy). Further tools (e.g., ISIS) monitor and aggregate certain code and development metrics. While these methods are largely standard software engineering best practice, they need organizational support.

Design *risks* are associated with creating an untouchable design, wrong abstraction level, or misleading metrics. Design principles should not be changed on a daily basis, but still be regularly assessed and adapted where appropriate. Too low design abstraction levels require substantial efforts during evolution, while too high abstraction levels might miss some important information. Design metrics and laws are not fixed and can require deviations in rare cases, which should be documented. Small-scale sustainability requires a special design and regularly planned refactorings.

Sustainable Implementation: During implementation, the sustainability of a system can be increased by writing clean code, documenting code, and/or code generation. Clean code adheres to specific coding guidelines, assists understandability, and thus lowers the effort for maintenance tasks. In the same manner, code documentation and naming conventions can improve understandability. Tools for codestyle conformance checks are for example Checkstyle or FxCop. Executable UML has found initial application in the industry and the corresponding code generation may lead to saved implementation efforts and higher code quality thus increasing sustainability.

Some implementation-related *risks* are poor code quality, final version thinking, and the use of unproven technologies. Furthermore, due to the longevity of industrial systems, their initial development environment tools may get lost, which then complicates further changes.

Sustainable Testing: While software testing is important for overall software quality, for sustainable systems continuous integration testing, regression testing, and sustainable testbeds have special importance. Continuous integration testing implements continuous processes of applying quality control. Regression testing checks whether new bugs are introduced during maintenance activities or while fixing bugs. Reusable unit tests facilitate change, simplify integration, and serve as way of documentation. A sustainable test-bed can for example rely on virtualization techniques to emulate older hardware, operating systems, and development environments.

Testing *risks* are associated with practically limited testing coverage, costs for testing, and the evolution of tests. Formal verification might be effective for some smaller safety-critical systems, but requires checking the underlying verification assumptions.

Sustainable Maintenance: Many typical maintenance activities (e.g., refactoring, checking code-metrics, applying patterns) are also part of the design and implementation phase. Hence, these activities are not repeated in this paragraph. Additional maintenance techniques relevant for sustainability and applied in industry are architecture consistency

checking and reverse engineering. Tools for checking the architectural consistency (e.g., Lattix, ndepend) can help to avoid architecture erosion, i.e., the situation that the implementation violates architectural constraints (e.g., strict layering between subsystems). Reverse engineering is applied on legacy systems, for example when no architectural design documentation is available. Reverse engineered higher-level structures improve understandability and may allow to estimate the impact of system modifications.

Risks associated with maintenance are for example architecture erosion and a lost overview. Architecture erosion might lead to a violation of a system's initial requirements and design principles and is thus undesirable. On the other hand, the implementation should not become a slave to the architecture, which might get outdated due to a changing environment or stakeholder requirements. Without a complete system overview, maintenance activities with a narrow focus can further decrease maintainability.

Sustainable Phase-independent Methods: Some approaches are cross-cutting and concern all development phases. These include for example process improvements, organizational support, sustainable documentation as well as knowledge management. Sufficient system documentation can prevent the loss of knowledge in a long-living system due to a change of personal. Many empirical studies have found a positive influence on system maintenance and evolution activities due to good documentation. Examples for documentation tools are JavaDoc, Doxygen, and Wikis. Design rationale can be handled through architecture knowledge management tools [TAJ+10] and thus supports traceability and maintenance activities.

4 Validation

We have not yet performed an extensive validation of our sustainability guidelines against the described evolution scenarios. However, this section discusses our validation goals (Section 4.1), maps one selected evolution scenario to the guidelines (Section 4.2), and sketches planned case studies (Section 4.3).

4.1 Validation Goals

The validation of our sustainability guidelines (i.e., evidence of benefits both technically and financial) is an essentially complex endeavor. First, the usability of the guidelines document itself needs to be validated. Second, the proposed methods must be validated through application and collection of experience.

To validate the *usability* of the guidelines, we propose a qualitative study first among a selected number of ABB architects and developers and then also among other developers from the industrial automation domain. Developers should be asked, whether the guidelines reflect the necessary amount of information to decide on the application of a method and whether the guidelines can be mapped to their concrete problems. As an initial step in this direction, we provide a mapping of a concrete evolution scenario to the guidelines in

Section 4.2.

Validating the *applicability* and usefulness of the recommended methods themselves is much more complicated than assessing the usability of the guidelines document. Replicated empirical studies would be necessary for the recommended methods to get trustworthy results. Validation criteria are whether evolution costs can be reduced (e.g., fewer modification necessary), whether software quality increases (e.g, lower number of failure reports), and whether maintenance tasks can be carried out faster (e.g., lower time to fix a bug). A complete validation is out of scope for our research in the short term. However, it is possible to refer to literature for already documented industrial case studies, and furthermore, we plan a number of smaller case studies to initially assess some selected methods (Section 4.3).

Empirical studies and experience reports about selected sustainability methods can be found in literature. For example, Sommerville et al. [SR05] have evaluated a maturity model for *requirements* engineering and tried to determine the impact on the quality of the resulting software product. In the area of software *architecture*, there are for example more than 20 industrial case studies reported for ATAM [CKK02] and 7 case studies for ALMA [BLBvV04]. Sustainability guidelines for software *design* can rely on empirical studies on refactoring [MT05] and the impact of applying design patterns [PULPT02]. Graves et al. [GHK+01] analysed the impact of different strategies for regression *testing*.

4.2 Mapping the Guidelines to a Sample Scenario: Third party components

The integration of third party components into automation systems is a typical evolution scenario. When looking at the different life- and evolution-cycles of automation systems and third party components, developers have to ensure that the integration is done in a way that future independent changes are easily possible without costly rework. For example, middleware frameworks evolve significantly over the lifetime of a process automation system, e.g. Java EE had 5 major revisions since its first release in 1999.

In order to ease typical development tasks, we ranked the relevance of recommended activities in our guidelines for every evolution scenario. In this section we briefly show this ranking for the evolution scenario of *integrating a third party component* and map it to a concrete development task. Only those phases and activities with high relevance are included.

Requirements Activity / Guideline: Management / Document why the 3rd party component is selected and which alternatives exist.

Mapping: Based on the performance and connectivity requirements of the automation system, a communication middleware was selected after a thorough technology evaluation.

Activity / Guideline: Tracing / Capture where the 3rd party component is assembled and deployed.

Mapping: The used capabilities of the middleware component are traced from requirements to system components down to implementation artifacts.

Architecture Activity / Guideline: Patterns, Styles, Tactics / Clean encapsulation of the component; if uncertainties are implied by 3rd party component: make it a variation point by use of a pattern (e.g. Factory).

Mapping: The usage of the middleware has been isolated in dedicated communication components, e.g. connect components for realizing connectivity through standardized automation protocols.

Design Activity / Guideline: OO-Metrics / Check encapsulation of 3rd party components; strict interface communication.

Mapping: A code inspection tool is used to monitor usage of public API calls of the middleware.

Testing Activity / Guideline: Unit Testing, Integration / Test API, test system integration with 3rd party components.

Mapping: Conformance tests for the middleware are used to ensure its quality; unit tests are in place to find functional regressions, and system tests find quality differences, e.g. performance degradation.

Maintenance Activity / Guideline: Architecture consistency checking / Check strong dependencies to component and existence of clear interfaces.

Mapping: Advanced static analysis tools (Lattix, Klocwork) are used to enforce conformance to the system architecture. The automation system has to be portable to several operating systems, thus the usage of the middleware has been restricted to isolated components, which is checked as well.

4.3 Planned Case Studies

While the former mapping provides an initial validation of the usefulness of our sustainability guidelines, we plan further case studies to evaluate a small number of the recommended methods. We identified three evolution scenarios from ABB products, which shall be assessed and analysed post-mortem.

The first case study shall apply the ALMA method [BLBvV04] to compare two software architecture designs of a controller firmware for their support of sustainability. The second case study analyses the evolution problems of a third party component in a process control system using the SISSy code analysis tool searching for bad smells and sustainability impeding hot spots. The third case study shall redocument architectural design rationale using a tool for software architecture knowledge management [BDLvV09].

5 Related Work

Several surveys in the area of software evolution can be found in literature. Chapin et al. [CHK+01] propose a redefinition of the types of software evolution and software

maintenance. A classification of software evolution types is presented by distinguishing changes of (1) the software, (2) the documentation, (3) the properties of the software, and (4) the customer-experienced functionality.

Godfrey and Buckley et al. [BMZ+05] create a taxonomy of software change based on characterizing the mechanisms of change and the factors that influence these mechanisms. The ultimate goal of this taxonomy is to provide a framework that positions concrete tools, formalisms and methods within the domain of software evolution.

Mens et al. surveyed software refactoring [MT05] and provided a list of challenges in software evolution [MWD+05]. Germain [GG08] discuss the past, present, and future of software evolution. They provide definitions of evolution and maintenance terminology as well as a comparison with biological evolution.

Rozanksi and Woods [RW05] define an "evolution perspective" in their architectural framework "viewpoints and perspectives". They discuss how architectural changes can be described and characterized and list some abstract evolution tactics, but exclusively focus on the software architecture. Bass et al. [BCK03] list a number of modifiability tactics for software architectures. They focus on modifiability and do not include requirements or implementation related issues.

6 Conclusions

This paper addresses the problem of developing sustainable long-living software systems in the industrial automation domain. Based on our experience in this domain, we provided a list of generic evolution scenarios, which can be used in future evolution evaluation studies. Additionally, we have performed an extensive literature search and extracted a number of sustainability guidelines for different development phases. We initially sketched the validation of these guidelines in this paper.

Our sustainability guidelines shall mainly help practitioners in the industrial automation domain to develop software with higher quality and lower evolution costs. The guidelines provide an overview and evaluation of selected, matured approaches, and a list of associated risks. Researchers can also rely on these guidelines and identify potential issues for future research.

In our future work, we will extend and improve the guidelines and apply them to three small case studies to understand the value of a selected number of the recommended methods.

References

[BCK03] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.

[BDLvV09] Muhammad Ali Babar, Torgeir Dingsyr, Patricia Lago, and Hans van Vliet, editors. *Software Architecture Knowledge Management: Theory and Practice*. Springer, 1st

edition, August 2009.

[BG09] M.A. Babar and I. Gorton. Software architecture review: The state of practice. *Computer*, 42(7):26–32, 2009.

[BLBvV04] P.O. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet. Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software*, 69(1-2):129–147, 2004.

[BMZ+05] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5):309–332, 2005.

[CHK+01] N. Chapin, J.E. Hale, K.M. Khan, J.F. Ramil, and W.G. Tan. Types of software evolution and software maintenance. *Journal of software maintenance and evolution: Research and Practice*, 13(1):3–30, 2001.

[CKK02] P. Clements, R. Kazman, and M. Klein. *Evaluating software architectures: methods and case studies*. Addison-Wesley Reading, MA, 2002.

[DKK+11] Zoya Durdik, Heiko Koziolek, Klaus Krogmann, Johannes Stammel, and Roland Weiss. Software Evolution for Industrial Automation Systems: Literature Overview. Technical Report 2011-2, Faculty of Informatics, Karlsruhe Institute of Technology (KIT), January 2011. ISSN: 2190-4782.

[FBBO99] Martin Fowler, Kent Beck, John Brant, and William Opdyke. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman, 1999.

[GG08] M.W. Godfrey and D.M. German. The past, present, and future of software evolution. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 129–138. IEEE, 2008.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*, volume 206. Addison-wesley Reading, MA, 1995.

[GHK+01] T.L. Graves, M.J. Harrold, J.M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(2):184–208, 2001.

[KSB10] Heiko Koziolek, Bastian Schlich, and Carlos Bilich. A Large-Scale Industrial Case Study on Architecture-based Software Reliability Analysis. In *Proc. 21st IEEE International Symposium on Software Reliability Engineering (ISSRE'10)*, pages 279–288. IEEE Computer Society, November 2010.

[MT05] T. Mens and T. Tourwé. A survey of software refactoring. *Software Engineering, IEEE Transactions on*, 30(2):126–139, 2005.

[MWD+05] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *8th Int. Workshop on Principles of Software Evolution (IWPSE'2005)*, pages 13–22. IEEE, 2005.

[PULPT02] L. Prechelt, B. Unger-Lamprecht, M. Philippsen, and WF Tichy. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *Software Engineering, IEEE Transactions on*, 28(6):595–606, 2002.

[RW05] N. Rozanski and E. Woods. *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley Professional, 2005.

[SR05] I. Sommerville and J. Ransom. An empirical study of industrial requirements engineering process assessment and improvement. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):85–117, 2005.

[TAJ+10] A. Tang, P. Avgeriou, A. Jansen, R. Capilla, and M. Ali Babar. A comparative study of architecture knowledge management tools. *Journal of Systems and Software*, 83(3):352–370, 2010.