Sustainability Guidelines for Long-Living Software Systems

Zoya Durdik[*], Benjamin Klatt[*], Heiko Koziolek[†], Klaus Krogmann[*], Johannes Stammel[*] and Roland Weiss[†]
[†]Industrial Software Systems, ABB Corporate Research Ladenburg, Germany
Email: {heiko.koziolek, roland.weiss}@de.abb.com
[*]Research Center for Information Technology (FZI), Karlsruhe, Germany
Email: {zoya.durdik, benjamin.klatt, klaus.krogmann, johannes.stammel}@fzi.de

***Abstract*—Economically sustainable software systems must be able to cost-effectively evolve in response to changes in their environment, their usage profile, and business demands. However, in many software development projects, sustainability is treated as an afterthought, as developers are driven by time-to-market pressure and are often not educated to apply sustainability-improving techniques. While software engineering research and practice has suggested a large amount of such techniques, a holistic overview is missing and the effectiveness of individual techniques is often not sufficiently validated. On this behalf we created a catalog of "software sustainability guidelines" to support project managers, software architects, and developers during system design, development, operation, and maintenance. This paper describes how we derived these guidelines and how we applied selected techniques from them in two industrial case studies. We report several lessons learned about sustainable software development.**

***Index Terms*—Software maintenance; software engineering; guidelines; sustainable development;**

I. Introduction

Software systems in the industrial automation domain are often long-living systems with a life-span of more than 10 years. These systems include a range of products from embedded real-time systems to large-scale distributed control systems. Such systems have to be constructed with special requirements to their design, structure, and extra-functional properties, such as safety, performance, and availability. During their life-cycle, they evolve in response to changes in their environment (i.e., hardware and software), usage profile (e.g., changed workload), and business demands (e.g., new features, changed business processes). Because this may require expensive changes to a system, it is necessary to keep efforts and costs under control during maintenance and evolution.

The term "sustainability", i.e., the ability for cost-efficient maintenance and evolution, is restricted to an economical perspective here and subsumes quality attributes of a software system that impact its maintenance and evolution [1], [2], [3]. Such sustainability properties should be kept in mind during the whole life-cycle of a software system and are especially relevant for *long-living* systems. Design and development decisions are sometimes taken omitting sustainability aspects in favour of time and budget or due to the lack of expert knowledge, leading to accumulation of technical debt [4]. This can lead to increased maintenance costs and introduce a major risk due to insufficient flexibility and quality.

There is already a plethora of approaches for analyzing and solving evolution problems [3]. Problem analysis is for example supported by architecture analysis, software comprehension, and quality indicators. Evolution problems can be resolved by improving architectural structures, reactive elimination, variability strategies, development automation, process improvement as well as knowledge management. Other authors have defined evolution laws [5], [6], classifications of evolution and maintenance types [7], [8], and taxonomies of software changes [9]. Our approach provides a unique perspective as it incorporates all phases of the system's life-cycle and aims at immediate developer support.

In order to facilitate long-living software systems, we have created a catalog of "sustainability guidelines" for ABB project managers, software architects, and developers. It fosters an explicit consideration of sustainability during system design, development, operation, and maintenance. The guidelines consist of selected software engineering methods, techniques and tools with a positive effect on sustainability. The catalog provides condensed method descriptions, information on their industrial validation, supporting tools, potential benefits, connected risks, checklists, and literature references. The catalog is a short and efficient reference to architects and developers.

The overall benefits of the guidelines are i) a structured overview of software sustainability-improving methods, ii) a single point of access to all relevant information, and iii) checklists and risk description for the guideline application in practice. We have applied selected methods and tools from the guidelines in two industrial case studies at ABB. First, we analyzed the architectural sustainability of a reconfigurable controller reference architecture. Second, we conducted a change impact analysis for a third-party middleware integrated in different industrial software systems. The results of these case studies have been used to further improve the guidelines, and system under study.

The rest of this paper is structured as follows: Section II summarizes a systematic literature search for sustainability-improving methods, which preceded the creation of the guidelines catalog. Section III introduces our sustainability guidelines catalog, its structure, target audience, outline and covered topics. The application of selected guidelines follows in Section IV. Finally, Section V reports lessons learned and Section VI concludes the paper.

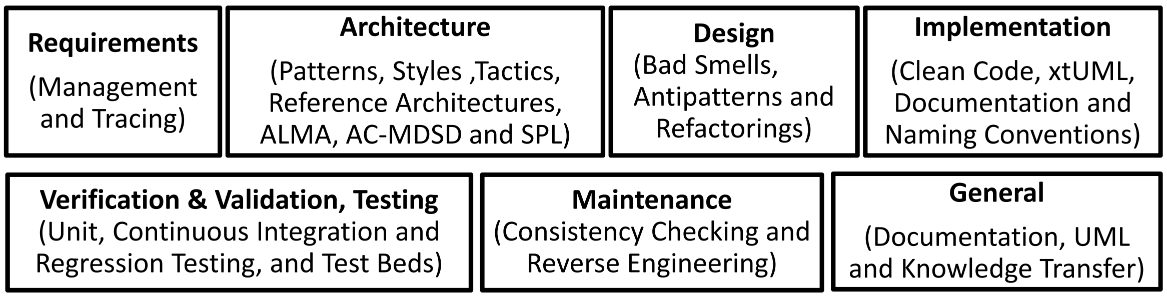

Requirements
(Management and Tracing)
Architecture
(Patterns, Styles ,Tactics, Reference Architectures, ALMA, AC-MDSD and SPL)
Design
(Bad Smells, Antipatterns and Refactorings)
Implementation
(Clean Code, xtUML, Documentation and Naming Conventions)
Verification & Validation, Testing
(Unit, Continuous Integration and Regression Testing, and Test Beds)
Maintenance
(Consistency Checking and Reverse Engineering)
General
(Documentation, UML and Knowledge Transfer)

Fig. 1. Topics covered by the software sustanability guidelines catalog

II. State-of-the-art survey

A. Systematic literature review for sustainability supporting solutions

In order to get an extensive overview of state-of-the-art approaches to support software sustainability we have conducted a systematic literature review. The results of this literature review were published in [3] and later refined for the guidelines document.

Information sources for the preparation of the literature review included books, journals, dissertations, conferences, workshops, selected podcasts, master's theses, standards, and white papers. All of them were examined for software evolution-related topics. A list of keywords (e.g. "software evolution", "software maintenance", "evolvability", "sustainability") helped to choose the relevant publications. Those selected publications were then manually investigated for potentially relevant approaches. The further processing of those information sources is discussed in detail in Chapter 7 of our previous publication [3][p. 115ff].

Our criteria catalogue for the final selection of development guidelines covered properties such as applicability, relevance, positive/negative perspective, degree of formalization, abstraction level, and development phases of presented approaches. We aimed to have a set of 2-5 approaches per development phase. From this literature review, the most promising approaches were selected to be included in the guidelines document.

B. Sustainability guidelines: Literature classification

The idea of providing guidelines for recurring software engineering problems is not new. Guidelines are for example provided by software development standards (e.g., ISO or IEC), development process models (e.g., V-model or agile methods), and software engineering books (i.e., Sommerville [10]). However, these sources usually provide general software engineering guidelines and do not focus on system sustainability.

We separated the approaches identified in our literature review, into one group of approaches for the *identification and analysis of evolution problems* and another group of approaches for *solving evolution problems*. For each area, we name several approaches representative for the field.

Approaches for identification and analysis of evolution problems were classified as *architecture analysis*, *software comprehension* (including use of historical data), and *quality indicators*. Popular architecture analysis approaches are for example ATAM [11], SAAM [12], and ALMA [13]. State of the art for software comprehension is for example surveyed by Hassan [14], who reviews the field of mining software repositories. Quality indicators are a broad field of qualitative and quantitative approaches, which were surveyed by Kan [15] and Fenton et al. [16], and can be classified based on their scope as product-related, process-related, and project-related metrics. Tool automation is required for quantitative approaches (e.g. ISIS [17]).

Approaches for solving evolution problems include a broad range of complementing strategies and were divided into *static architecture and software system structure*, *reactive elimination of evolution problems*, *variability strategies*, *automation of software development*, *development process*, *knowledge management and documentation*, *team support*, and *software infrastructure*. Recommendations for static architecture and software system structures include design principles [18], patterns [19], and reference architectures [20]. Reactive approaches tackle changes or their consequences. They range from refactorings [21] to migration patterns on the architecture level [22]. Variability can be tackled by generation (of code / configurations) [23] on an abstract level (typically models) or by architectural means such as product lines [24]. Automation of software development comprises software generation [23] as introduced above but also the model driven architecture [25] and model-driven techniques by de-facto standards of the Eclipse ecosystem [26], [27].

Development processes are nowadays influenced by agile methods [28] which propose more flexibility and lower costs for handling of change requests. To ease long-term knowledge management and documentation the enduring presence of knowledge and a complementary team management are crucial to ensure a long-term knowledge for development teams (see e.g. Bommer et al., [29] for an overview on suitable methods). Evolution pressure such as the infrastructure (e.g. hardware or middleware) are dealt with for example in Wolf et al. [30]. This book focuses on different aspects of virtualization, including virtual machines, virtual file systems, virtual storage solutions, and clustering for both Windows and Linux.

This overview on existing approaches targeting software evolution and long-term management provided the base to derive the sustainability guidelines, which are described in Section III, after the related work presented in the next section.

C. Other related work

The majority of methods to handle sustainability deals with evolution aspects of existing software systems, for example, the cost-effective evolution [7], [8], or laws, generalities and management of evolution [5], [6]. To our best knowledge, there are no overarching guidelines conditioning different methods to improve the sustainability of a system.

Chapin et al. [7] proposed a clarifying redefinition of the types of software evolution and software maintenance to improve the sustainability of a system already during its initial development. They propose a classification of software evolution and software maintenance types, based on the maintainers activities. The classification distinguishes changes of i) the software, ii) the documentation, iii) the properties of the software, and iv) the customer-experienced functionality.

Godfrey and Buckley [9] discuss differences and relations between software evolution and maintenance. They create a taxonomy of software changes. The taxonomy is based on characterizing the mechanisms of change and the factors that influence these mechanisms. The goal of this taxonomy is to provide a framework that positions concrete tools, formalisms and methods within the domain of software evolution. However, they do not propose a solution, but rather analyse change and evolution properties.

Mens et al. surveyed software refactoring [31] and provided a list of challenges in software evolution [31]. Germain et al. discuss the past, present, and future of software evolution [8]. They provide definitions of evolution and maintenance terminology as well as a comparison with biological evolution.

Bass et al. [32] list a number of modifiability tactics for software architectures. They focus on modifiability and do not include requirements or implementation related issues.

Rozanksi and Woods [33] 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. The approach of defining tactics to handle specific evolution challenges is also included in our sustainability guidelines and primary covered in the architecture related sections.

Seacord et al. [1] discuss differences between software maintenance and sustainability, overview existing sustainability measures, and propose additional measures and sustainability assessment.

Beside that the vast approaches only concentrate on one of the phases of system life-cycle, it is difficult to select from these methods and to determine upfront whether they can address specific evolution problems. Moreover, it is not always clear if methods are suitable for long-living systems and how well the methods have been validated on practice. The effort required for the introduction into the project and potential benefits, especially considering sustainability, are seldomly explained.

III. Sustainability Guidelines

In this section we introduce our sustainability guidelines, their general idea, target audience, outline and covered topics. We explain our approach on how to navigate through the guidelines as well as how to select appropriate ones. We further provide an excerpt from one of the guideline-chapters as an example.

A. General Idea and Target of the Guidelines

The sustainability guidelines support the explicit consideration of sustainability during system design, development, operation, and maintenance. Their objective is to support the problem analysis and decision making with active incorporation of sustainability aspects. The guidelines are based on an extensive literature survey [3] and represent the condensed state-of-the-art approaches with respect to methods and tools. They cover design and analytical, as well as proactive and reactive techniques for sustainability-aware software engineering. The approaches have been selected with focus on industrial applicability and reasonable efforts for learning and application, especially because only few of them can be automated.

The guidelines aim to make software architects and developers aware of state-of-the-art methods, their potential benefits and risks. We expect them to be familiar with basic concepts of software engineering, but do not assume any specific knowledge of sustainability or one of the included guidelines. In addition to software architects and developers, the guidelines are intended to be considered by project leads, requirements engineers and testers, who shall be aware of sustainability and shall keep track of it during their work.

For this, the guidelines are designed to provide an overview and a quick introduction into the particular topics to decide about their relevance for the current project. Software architects and developers can refer to these guidelines while working on a new system or during the evolution of an existing one. They can browse through the relevant approaches and even if they are familiar with an approach, checklists and risk estimations provide valuable information on recommendations and possible pitfalls.

B. Guidelines Sections and Structure

The approaches are grouped by software life cycle phases: Requirements, Architecture, Design, Implementation, Validation and Verification, and Maintenance. These groups are followed by a general group of approaches, which are relevant for multiple lifecycle phases. Note that the aforementioned software life cycle phases do not imply a waterfall development process but serve as an intuitive structure for the document.

The overview of topics covered by the current version of the guidelines is presented on Figure 1. Each guideline section (requirements, architecture, etc.) contains several selected approaches.

In order to quickly grasp the essence of each guideline, the description follows a common description template, that fits on one page, as presented on Figure 2. The header contains meta information, such as application and learning efforts, relevance for evolution, the addressed problem and existing validation of the approach.

The efforts required to introduce a guideline are classified as follows:

- *High:* A satellite-project or distinct resources are required.
- *Medium:* An introduction during project execution is possible but should be considered explicitly in the project plan.
- *Low:* An introduction can be done by a single person in reasonable time. Implementation requires little overhead.

The provided effort of each guideline is complemented with a short explanation of the estimation.

The efforts required to learn a guideline are classified as follows:

- *High:* Requires to understand and learn complex, highly abstract or large amounts of material, often accomplished by a complex tooling.
- *Medium:* Requires some teaching or experience and getting familiar with several tools.
- *Low:* The basic concept is simple and most of the tools are familiar.

Furthermore, each guideline is provided with a note on its relevance to sustainability, as well as problems that are targeted. The validation is distinguished into general validation experience and ABB-internal validation and expresses the maturity of the approach for its application in practice. In both cases, the validation is either strong ("+"), neutral ("0"), or weak ("-").

The body section of each guideline provides an introduction to the individual topic, sources of further information and guidance for its application. A short description gives an overview and reflects the major aspects of the approach the guideline is about. For further reading, each guideline provides a primary literature reference, which can be used for detailed understanding of the topic and answering of conceptual or implementation questions. If tools are available to support the application of a guideline, the most relevant or ABB recommended ones are listed in a specific tool section. For the advantages and risks of each guideline, two separate sections have been introduced for these. The information about each guideline, is always documented with a strong focus on sustainability. In the same way, the usefulness and risks are described from a system evolution point of view.

C. Guideline Navigation and Selection

We provide two strategies to identify suitable approaches from the guidelines: based on a specific development phase (life cycle phase) and based on an evolution scenario. While the first strategy is intuitive due to the nature of software product development, we discuss the second one in more detail in this section.

Providing relevant and compressed information in a compact document, which does not frighten users because of its length, was an important goal during the creation of the guidelines. To provide guidelines which address actually relevant evolution scenarios we interviewed several developers and engineers at ABB [34].

During the interviews, evolutions scenarios were collected and prioritised. The most relevant evolutions scenarios (prioritized based on how many times a scenario was mentioned in the interviews) were clustered into broader scenarios to represent typical evolution cases. For example, the two separate scenarios replace hardware "A" and "B" were generalised to "update hardware execution environment". Those scenario clusters are specifically addressed in the guidelines document. In the guidelines, an evolution scenario is a brief description of an anticipated change because of modified requirements, failures, changing technical infrastructure, or internal maintenance activities.

The most common scenarios of each cluster were used to build a table with the scenarios on the vertical axis and the guideline chapters on the horizontal axis. As the complete table can not be provided in this paper due to the limited space, Table I provides an exemplary excerpt of it. With a specific scenario in mind, the reader can identify the representing scenario in the table head, and find relevant sections marked in the specific column. The relevance of a guideline for a specific scenario is marked as high ("+"), moderate ("0"), or irrelevant ("-").

Scenario / Chapter	**Enhance or add in-house components**	**Support for processing larger amounts of data**	**...**
Requirements			
Requirement management	+	+	
Sustainable requirements tracing	+	0	
Architecture			
Patterns / Reference Architectures / Tactics / Styles	+	+	
Architecture-Level Modifiability Analysis (ALMA)	0	—	
Software Product Lines (SPL)	0	0	
...			
...			

TABLE I
NAVIGATION TABLE FOR APPROACH LOOK UP AND SELECTION BY EVOLUTION SCENARIO (EXCERPT)

D. Excerpt of a Guideline Chapter

All guidelines are documented according to a common template shown in Figure 2. In this section we present the Architecture-Level Modifiability Analysis (ALMA) guideline. The ALMA approach [13] is used to assess the modifiability of a software on an architectural level by analyzing the impact of change scenarios with a series of interviews with key stakeholders.

The header of this guideline captures the estimated application and learning efforts as well as the rationales for these estimation. ALMA is rated to require a medium application effort, because it is applied manually by an analysis team, but requires only a minimum upfront training. The learning effort is also rated as medium.

Beside the focus of the architectural level, the header describes the relevance of ALMA as high due to the support of identifying changes with potentially high implementation costs. The short description summarises the major steps to conduct ALMA. This includes indentifying potential change scenarios, clustering and prioritising them, and analysing the architecture's support of the most relevant scenarios.

The section about further readings states the major publication about ALMA by Bengtsson et al. [13].

Finally, the template discusses advantages and risks of the guideline. In case of ALMA, the advantages include the potential prevention of long-term costs and the applicability to validate implementation costs upfront in early design phases. The support of make-or-buy-decisions is also described. However, there are also documented risks such as missing analysis results due to missing change scenarios, and unnecessary analysis overhead due to the irrelevant scenarios. Furthermore, the analysis resides on an abstract architectural level, and lacking in the examination of modifiability on the code level, which might be missed. Beside content regarding risks, a too high effort might result from involving too many stakeholders or an inefficient application of the ALMA process.

IV. Sustainability Guidelines Application

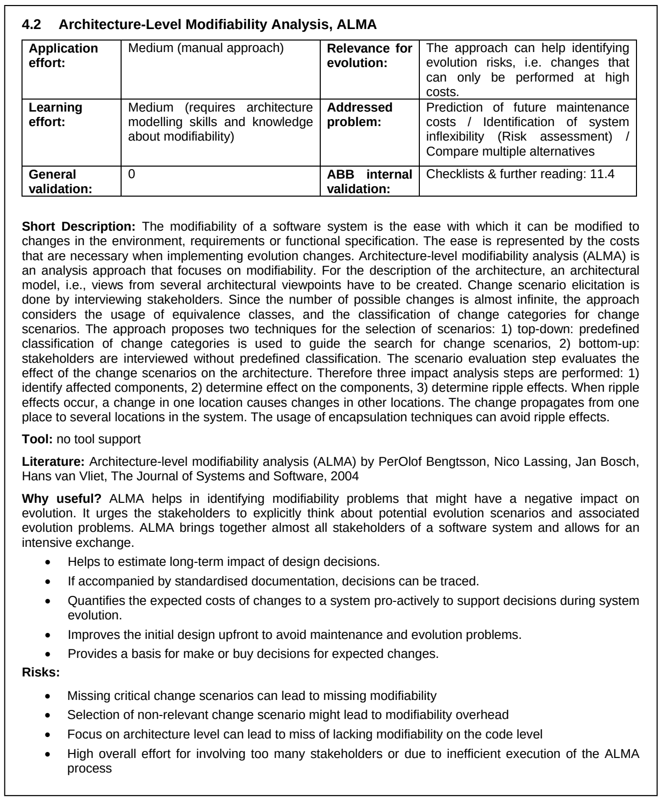
We applied selected sustainability guidelines from our catalog to two case studies to improve the systems under study. These case studies are not intended to formally validate the applicability of the guidelines, which would require a longitudinal study to be able to quantify their effect on a software system's evolution. Rather the case studies gave hints on the applicability of selected sustainability-improving methods and helped to mature and refine the guidelines themselves.

In the first case study (Section IV-A), we compared a legacy version of ABB's so-called PCS (Protection Controller System) with a newly designed version called RCRA (Reconfigurable Controller Reference Architecture) in collaboration with the system's chief engineer.

In the second case study (Section IV-B), we analyzed the impact of evolving third-party components on a communication middleware for industrial devices used at ABB.

A. Case study 1: Architecture evolution scenario analysis

The first case study consisted of two parts: (1) Checking whether a subset of the guidelines would be applicable to the new RCRA architecture, and (2) Evaluation of the PCS and RCRA target architectures using ALMA, one of the approaches highlighted in the sustainability guidelines.



4.2 Architecture-Level Modifiability Analysis, ALMA

Application effort:	Medium (manual approach)	**Relevance for evolution:**	The approach can help identifying evolution risks, i.e. changes that can only be performed at high costs.
Learning effort:	Medium (requires architecture modelling skills and knowledge about modifiability)	**Addressed problem:**	Prediction of future maintenance costs / Identification of system inflexibility (Risk assessment) / Compare multiple alternatives
General validation:	0	**ABB internal validation:**	Checklists & further reading: 11.4

Short Description: The modifiability of a software system is the ease with which it can be modified to changes in the environment, requirements or functional specification. The ease is represented by the costs that are necessary when implementing evolution changes. Architecture-level modifiability analysis (ALMA) is an analysis approach that focuses on modifiability. For the description of the architecture, an architectural model, i.e., views from several architectural viewpoints have to be created. Change scenario elicitation is done by interviewing stakeholders. Since the number of possible changes is almost infinite, the approach considers the usage of equivalence classes, and the classification of change categories for change scenarios. The approach proposes two techniques for the selection of scenarios: 1) top-down: predefined classification of change categories is used to guide the search for change scenarios, 2) bottom-up: stakeholders are interviewed without predefined classification. The scenario evaluation step evaluates the effect of the change scenarios on the architecture. Therefore three impact analysis steps are performed: 1) identify affected components, 2) determine effect on the components, 3) determine ripple effects. When ripple effects occur, a change in one location causes changes in other locations. The change propagates from one place to several locations in the system. The usage of encapsulation techniques can avoid ripple effects.

Tool: no tool support

Literature: Architecture-level modifiability analysis (ALMA) by PerOlof Bengtsson, Nico Lassing, Jan Bosch, Hans van Vliet, The Journal of Systems and Software, 2004

Why useful? ALMA helps in identifying modifiability problems that might have a negative impact on evolution. It urges the stakeholders to explicitly think about potential evolution scenarios and associated evolution problems. ALMA brings together almost all stakeholders of a software system and allows for an intensive exchange.

- Helps to estimate long-term impact of design decisions.
- If accompanied by standardised documentation, decisions can be traced.
- Quantifies the expected costs of changes to a system pro-actively to support decisions during system evolution.
- Improves the initial design upfront to avoid maintenance and evolution problems.
- Provides a basis for make or buy decisions for expected changes.

Risks:

- Missing critical change scenarios can lead to missing modifiability
- Selection of non-relevant change scenario might lead to modifiability overhead
- Focus on architecture level can lead to miss of lacking modifiability on the code level
- High overall effort for involving too many stakeholders or due to inefficient execution of the ALMA process

Fig. 2. Examplary chapter of sustainability guidelines

In the first part, a subset of potentially applicable guidelines had been selected based on the recommendations within the guidelines document itself. These guidelines were applied to the architecture and design artefacts provided by the RCRA development team, that also cover implementation aspects expressed in UML class diagrams of the technical documentation of RCRA.

The general effort to familiarise the project participants with the system architecture equals approximately 12 person days (4 persons with 3 days per person). During this phase we collected (1) experiences on the application of the subset of the guidelines complemented by (2) findings about RCRA as opposed to the guidelines (e.g., violation of principles proposed by the guidelines), and (3) findings which go beyond the sole application of the guidelines (findings that are specific to the RCRA and cannot be generalized in the guidelines).

Our findings about the RCRA were summarised and provided as a report containing sustainability recommendations, where each recommendation is linked to the corresponding parent sustainability guideline. An overview of these recommendations is provided in Figure 3.

For example, the "Copy-Paste-Instantiation" sustainability recommendation proposes to extract a framework out of the RCRA architecture instead of planned copy-paste code reuse. The "Framework and Environment Distinction" sustainability recommendation suggests to improve the layered architecture by extracting the infrastructure layer, which is technology dependent, out of the reusable core of the RCRA architecture.

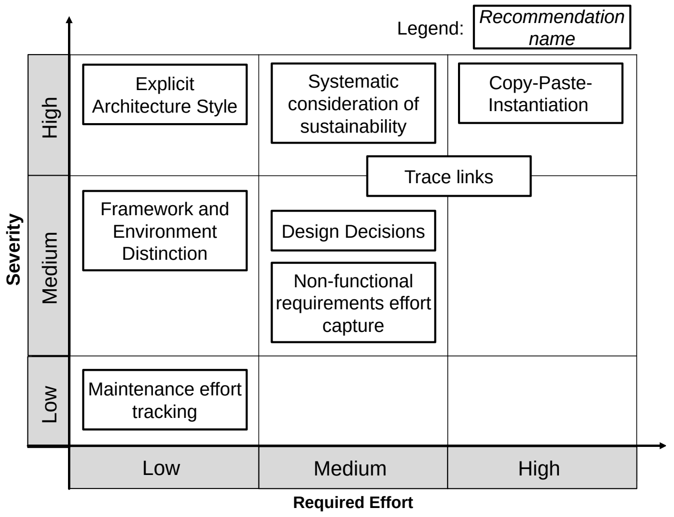
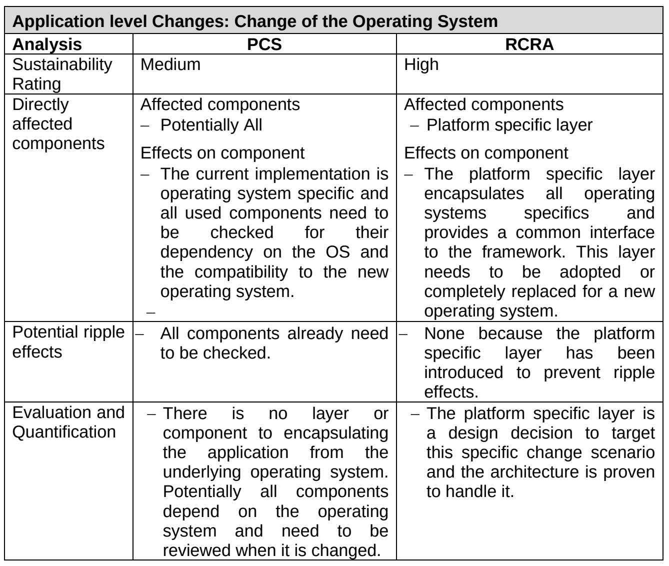

Legend:
Recommendation name
Severity
High
Medium
Low
Explicit Architecture Style
Systematic consideration of sustainability
Copy-Paste-Instantiation
Trace links
Framework and Environment Distinction
Design Decisions
Non-functional requirements effort capture
Maintenance effort tracking
Low
Medium
High
Required Effort

Fig. 3. Severity and effort classification of the derived recommendations for the RCRA system



Application level Changes: Change of the Operating System		
Analysis	**PCS**	**RCRA**
Sustainability Rating	Medium	High
Directly affected components	Affected components – Potentially All Effects on component – The current implementation is operating system specific and all used components need to be checked for their dependency on the OS and the compatibility to the new operating system. –	Affected components – Platform specific layer Effects on component – The platform specific layer encapsulates all operating systems specifics and provides a common interface to the framework. This layer needs to be adopted or completely replaced for a new operating system.
Potential ripple effects	– All components already need to be checked.	– None because the platform specific layer has been introduced to prevent ripple effects.
Evaluation and Quantification	– There is no layer or component to encapsulating the application from the underlying operating system. Potentially all components depend on the operating system and need to be reviewed when it is changed.	– The platform specific layer is a design decision to target this specific change scenario and the architecture is proven to handle it.

Fig. 4. An excerpt of the ALMA analysis

These recommendations were classified according to the estimated required efforts and severity of impact on sustainability of the improvement suggestions from the recommendations. The estimated required effort is classified as *high*, *medium*, and *low*, similarly to the efforts required to introduce a guideline provided in the Section III-B.

The impact severity on sustainability of the improvement suggestions is classified as:

- *High impact on the sustainability:* shall be implemented if possible, otherwise high probability of noticeably negative consequences on sustainability.
- *Medium impact on the sustainability:* has no severe, but some negative consequences on sustainability if not implemented.
- *Low impact on the sustainability:* has few to no consequences on sustainability if the recommendation is not implemented.

In the second part of the case study, we executed ALMA to compare the modifiability of the PCS and the RCRA systems. According to Bengtsson et al. [13], for the RCRA system we applied ALMA at an early stage of the development and focused on the modifiability but not the maintainability of the system. Furthermore, we assessed the current status of the RCRA and the PCS systems from an external perspective. We did this from a viewpoint based on the system's structure, including the components, externally visible properties and the relationships among them. Bengtsson et al. defined "Prediction of future maintenance costs", "Identification of system inflexibility", and "Compare multiple alternatives" as three possible goals for an ALMA analysis. We limited the analysis to the cost prediction and inflexibility identification.

The generic, system-independent evolution scenarios mentioned in Section III-C were prioritized and the scenarios with the highest priority were analysed in depth. A table with results of the analysis for PCS and RCRA was provided per change scenario in order to enable a direct comparison. As shown in the exemplary Table II, each table contains sustainability ratings for each system, which quantifies the support for the individual scenario according to the available documentation are. Possible ratings are:

- *High*: The scenario is explicitly considered in the architecture.
- *Medium*: The scenario is not explicitly considered but no blocking issues have been identified.
- *Low*: Issues that might block this scenario have been identified in the architecture and documentation.

Furthermore, each table contains the list of directly affected components (in the example table there is only one component called "Platform specific layer") that are followed by the identified potential ripple effects, and evaluation and quantification of the scenario based on the affected components and identifies ripple effects.

The final results of the ALMA analysis are summarized in Table II, where each of the selected scenarios is listed with its subjective sustainability rating for PCS and for RCRA by our analysis team. It can be concluded that the new RCRA is better prepared for several of the analysed change scenarios.

B. Case study 2: Third-party component handling

The second case study focused on a particular evolution scenario, namely the "change impact due to third-party components". In this scenario, the guidelines should assist software architects during selection of third-party components to be incorporated into a system by assessing the quality of the third-party component and ensuring a sustainable way of integration.

In order to investigate some of our sustainability guidelines we applied them on a communication middleware component for industrial devices used by ABB. One special property of this middleware component is the availability of source code and bugtracker information.

Scenario	**PCS**	**RCRA**
Change of Third Party Component	Medium	Medium
UI Changes: Replacement of the User Interface Technology	High	High
Functionality level changes: I) Enhance in-house components	Low	Medium
Functionality level changes: II) Add new in-house components	Medium	Medium
Data level changes: Support for processing larger amounts of data	Low	Medium
Application level Changes: Change of the Operating System	Medium	High
Application Level Changes: Support for Virtualization	Low	Medium
Hardware Level Changes: Exploit Multi-Core Processors	Low	High
Hardware level Changes: Deploying components in cloud platforms	n/a	n/a

TABLE II
SUMMARY OF THE ALMA ANALYSIS

We used the look-up table of Section 2.1 of the guidelines document to select appropriate sustainability-improving techniques for this scenario. This table provides particular guideline recommendations concerning third-party components. In this study we limited the analysis to the life-cycle phases architecture and design. In the following, we explain how we interpreted and applied the selected guidelines.

With respect to the architecture phase the guidelines document proposes to "update architectural documentation to include third-party components, acquire comprehensive documentation from third-party component vendor". We followed this guideline by gathering documentation of the case study system from the vendor. This included the source code of three releases of the case study system, as well as user documentation and access to the system's bug tracker. Moreover, we created an architecture overview aligned with the architecture of the client code.

Furthermore, the guidelines document recommends to ensure a clean encapsulation of the third-party component. For this we identified the API parts of the third-party component which are used by client code. We also established a continuous update of the usage information based on static code analysis in order to monitor the encapsulation.

Other guidelines for the architecture phase propose i) risk assessment with information about the third-party component, ii) analysis of ripple effects, and iii) an impact prediction in case that the third-party component has to be replaced or updated. We interpreted the proposed risk assessment as analysing the dependencies of the client code on the third-party component and within the third-party component which might lead to instability in case of third-party evolution.

In order to address these three guidelines, we used static code analysis to investigate the following aspects:

- Dependency overview within the subsystem structure and dependency progress over time
- Identification of explicit and hidden semantic changes within the third-party system
- Estimation of change impact on self-implemented code based on dependency structure

We used the tool SISSy[1], which derives code dependencies and compares the dependency evolution over multiple versions of the software system. We started with a dependency overview that showed absolute numbers of dependencies within the subsystems and packages of the third-party component as well as absolute dependency numbers from client code to the third-party component covering three versions.

Although the absolute numbers already gave a rough impression of the dependency structure and on the change impact potential, the assessment of risks and potentially required efforts connected with evolution were difficult to determine. Since high dependency numbers indicate a high manual effort for investigation in case of evolution, we established a tool-supported analysis process for continuous stability investigation and change impact analysis.

This approach investigates the stability of the third-party component in terms of explicit signature changes and hidden semantic changes inside the third-party component. As a result, we retrieved locations of third-party code that were actually modified during evolution over three given versions. Then, the dependency chain was extended and filtered to identify dependent client code potentially affected by third-party evolution. Hence, we identified client code which is dependent on actual modified third-party code. This approach enabled a more straightforward and focused search and better understanding of critical dependencies.

We also selected guidelines from the design phase. For example, a guideline recommends the investigation of "bad smells" (also called anti-patterns or problem patterns). Hence, we conducted a problem pattern analysis on three versions of the middleware component again using the SISSy tool. The problem pattern analysis of SISSy is able to calculate a benchmark to compare problem pattern statistics with a reference project set. We provide an extract of the analysis results in the following.

Table III shows for three system releases the absolute numbers of detected problem pattern instances in relation to an appropriate reference value (such as number of classes, number of files, etc.; reference values stem from [35]). Definitions of the detected problem pattern types can be found in the online tool documentation[2]. In the next step the problem pattern counts were normalized to 1000 lines of code and mapped to statistical intervals of a reference project set. Table IV shows the number of problem pattern types (for three system releases) assigned to the statistical intervals of the reference project set. For example, five problem pattern types of Release 1 lie in the statistical range of "Minimum To Lower Quartile", which means that with respect to these five problem pattern types the system is better than three quarter of the reference projects.

[1]http://www.sqools.org/sissy/
[2]http://www.sqools.org/sissy/documentation/

Range	Release 1	Release 2	Release 3
Minimum To Lower Quartile	5	5	5
Lower Quartile To Median	3	2	2
Median To Upper Quartile	4	5	5
Upper Quartile To Maximum	10	10	10
Over Maximum	1	1	1

TABLE IV
NUMBER OF PROBLEM PATTERN TYPES PER REFERENCE INTERVAL

In addition to the tables we used a box-plot for each problem pattern type to visualize the ranking of each release compared to the reference projects. Figure 5 shows an example boxplot for problem pattern type "Refused Bequest". In this example all three releases are located between Lower Quartile and Median which means that they are worse than one quarter of the reference projects and better than half of the reference projects.

The boxplox gives easy to understand feedback on the quality of analysed source code. One can derive a tendency whether a given software implies maintainability and thus sustainability risks. For example, a source code project with more bad smells than 95% of all reference projects indicates poor quality with respect to the detected bad smell).

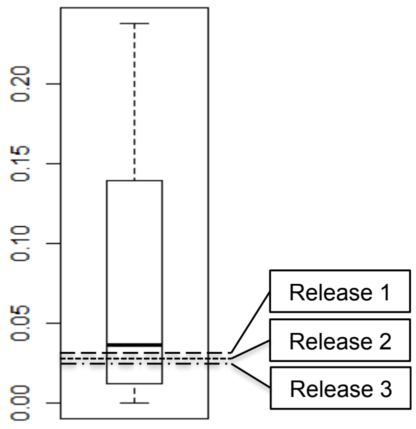


Fig. 5. Example box-plot visualization for problem pattern type "Refused Bequest"

In conclusion, we were able to select and apply guidelines in a straightforward way for the given change scenario. Some interpretation and adaption of the guidelines with respect to the particular project context were necessary. However, the more general setup of the guidelines enables the application of guidelines in a broad and flexible way.

V. LESSONS LEARNED

This section lists several lessons learned by applying the guidelines in the case studies described in Sections IV-A and IV-B.

A. Applying sustainable software engineering requires practical instructions.

Instructions on sustainable software engineering should be easy to transfer to the daily life of developers and software architects. Lowering the application barriers is promising to encourage the use of software engineering methods which support sustainability.

B. Many sustainability problems can already be mitigated by good software engineering practice

A range of sustainability problems found in current practice stem from the lack of applying known software engineering best practices (e.g., modular design, coding standards, regular code reviews, documentation). They do not require further software engineering research, but rather education and discipline on the developer side. When extending and refining the current guidelines in the future, this lesson should be incorporated to explore how developers can be efficiently trained and then enabled to work in a way favorable for sustainability.

C. Lacking motivation for sustainable development

Although those practices are well-known, they are sometimes neglected during the development due to the time and budget pressure from one side, and low enforcement and limited motivation from the other side. Immediate incentives for sustainable development are sometimes missing, which leads to the accumulation of technical debt. However, we believe that the incentives may be reinforced through activities, such as an explicit documentation of the quality and sustainability goals, enforced by a regular review process.

D. High costs for sustainable development

Sustainable systems cost more and require additional effort and investment, as demonstrated by the two case studies. Qualified people who are proficient in good software engineering principles should be hired. They should be given time and means to cleanly design and develop high quality software. Moreover, there must be a business decision to invest for more sustainable software. However, the estimation on the return of investment and the right measure for the sustainability is still an open and is a project-specific challenge.

E. High importance of concrete sustainability goals

Precise sustainability goals, requirements, and scenarios help in choosing the right method to support the systems sustainability. Otherwise there is the risk of wrong trade-off decisions between alternative guidelines to be applied. For example, a requirement for a "system to be operated for 10 years" is too abstract, but a requirement to "manage dependencies on third-party components" supports choosing the right sustainability methods. Without explicitly stated sustainability goals and scenarios, any sustainability improving measure likely result in under- or over-engineering of the system

F. Sustainability of the sustainable development

Paradigm shifts, such as technology game changers, might make sustainability activities obsolete.For example, substantial architectural changes (e.g., a change to multi-core computing, a change to mobile technology or a change to cloud

Problem Pattern	Release 1	Release 2	Release 3	Reference Value
General Parameter	40 / 6602	38 / 7319	38 / 7608	number of methods
Long Parameter List	24 / 6602	27 / 7319	28 / 7608	number of methods
Permissive Visibility Attribute	1769 / 3129	1877 / 3341	1876 / 3379	number of attributes
Refused Bequest (Implementation)	8 / 746	8 / 778	8 / 794	number of classes
Variables Having Const Value	28 / 3129	36 / 3341	36 / 3379	number of attributes
Complex Method	502 / 6602	535 / 7319	545 / 7608	number of methods
Interface Bypass	181 / 746	251 / 778	252 / 794	number of classes
Mini Class	9 / 746	14 / 778	15 / 794	number of classes
Overloaded File	79 / 671	72 / 690	75 / 699	number of files
Polymorphic Calls In Constructor	125 / 875	168 / 952	169 / 980	number of constructors
Violation Of Data Encapsulation	716 / 746	760 / 778	769 / 794	number of classes
Attribute Overlap	2 / 3129	10 / 3341	10 / 3379	number of attributes
Dead Method	254 / 6602	258 / 7319	263 / 7608	number of methods
God Class (Attribute)	3 / 746	3 / 778	3 / 794	number of classes
God Class (Method)	1 / 746	1 / 778	1 / 794	number of classes
God Method	59 / 6602	67 / 7319	67 / 7608	number of methods
Ignored Abstraction	10 / 746	11 / 778	11 / 794	number of classes
Inconsistent Operations	3 / 6602	5 / 7319	5 / 7608	number of methods
Dead Attribute	28 / 3129	19 / 3341	23 / 3379	number of attributes
Import Chaos	74 / 671	45 / 690	47 / 699	number of files
Informal Documentation	2878 / 6602	3428 / 7319	3658 / 7608	number of methods
Misleading Naming Files	119 / 671	133 / 690	133 / 699	number of files
Violation of Naming Convention	6982 / 671	7794 / 690	8079 / 699	number of files

TABLE III
PROBLEM PATTERN SUMMARY

computing) can render previous sustainability efforts useless. Technology changes cannot be anticipated too far ahead.

To cope with this an alternative strategy is an evolutionary approach, which is based on the incremental and iterative refinement in conditions when requirements or context are either not (well-)known or change rapidly. An example of such an evolutionary approach is the evolutionary architecture practice. It proposes incremental and iterative refinement of software architecture in context of unknown or rapidly changing requirements.

G. Trade-offs in applying different guidelines

The guidelines applicability check in the "RCRA" scenario required more time to find out which guidelines are feasibly applicable for the given context then we have expected, due to the fact that we were new to the RCRA. The application of the guidelines often involved a trade-off between separate guidelines due to the connected costs and risks, whereby the sustainability shall be kept in mind. For example, the architecture issues are more important than the implementation, while the implementation issues are likely to be eliminated easier.

Finally, in the case study we decided to follow a problem- and scenario-driven sustainability improvement instead of software development life cycle style partitioning in phases. We assume that the problem- and scenario-driven sustainability improvement better fits projects that are already beyond their initial stage. And vice versa, the software development life cycle style partitioning in phases is more applicable for the projects at their initial development stage.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we presented "sustainability guidelines" that provide a guide for explicit consideration of sustainability during system design, development, operation, and maintenance. The guidelines contain selected software engineering approaches with a focus on sustainability. Approaches are structured according to software lifecycle phases for easier navigation, and are followed by a general group of approaches, that are relevant for multiple lifecycle phases. The template description is used to provide a systematic overview for each approach. template description contains information approach validation, supporting tools, targeted problems, benefits, connected risks and literature for further reading. Whereby, the guidelines provide a short practice-oriented reference for a system architect or developer about the sustainability-relevant software engineering topics and approaches.

The guidelines applicability have been successfully validated on an industrial case study involving a real system RCRA (Reconfigurable Controller Reference Architecture). The results of the case study have been evaluated and improvements both for the guidelines and for RCRA system were derived.

For the case study a subset of applicable guidelines has been selected based on the recommendations within the guidelines document itself. The selected guidelines could be successfully applied, the improvement remarks mainly focused on collection of primary literature sources and rework of requirements engineering guidelines due to not being specific enough.

The RCRA system already complied to many points of the guidelines. However, we also identified possible improvements especially about the explicit documentation of design and decisions.

In our future work, we are planning to apply the guidelines within business units within ABB. We also plan to extend the guidelines with additional sustainability-relevant approaches and to port guidelines from a word document into a easier accessible and editable WiKi form.

Acknowledgment

The authors would like to thank the RCRA project team by ABB for the active assistance in the case study and provided feedback.

References

[1] R. Seacord, J. Elm, W. Goethert, G. Lewis, D. Plakosh, J. Robert, L. Wrage, and M. Lindvall, "Measuring software sustainability," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, 2003, pp. 450 – 459.

[2] H. Koziolek, "Sustainability evaluation of software architectures: a systematic review," in *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS*, ser. QoSA-ISARCS '11, 2011, pp. 3–12.

[3] J. Stammel, Z. Durdik, K. Krogmann, R. Weiss, and H. Koziolek, "Software Evolution for Industrial Automation Systems: Literature Overview," Karlsruhe, Germany, Karlsruhe Reports in Informatics 2011 - 2, 2011.

[4] W. Cunningham, "The wycash portfolio management system," in *Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)*, ser. OOPSLA '92. New York, NY, USA: ACM, 1992, pp. 29–30. [Online]. Available: http://doi.acm.org/10.1145/157709.157715

[5] M. M. Lehman and J. F. Ramil, "Software evolutionbackground, theory, practice," *Information Processing Letters*, vol. 88, no. 12, pp. 33 – 44, 2003.

[6] M. Lehman and J. Ramil, "Rules and tools for software evolution planning and management," *Annals of Software Engineering*, vol. 11, pp. 15–44, 2001.

[7] N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *Journal of Software Maintenance: Research and Practice*, vol. 13, no. 1, p. 3, 2001. [Online]. Available: http://portal.acm.org/citation.cfm?id=371697.371701

[8] M. W. Godfrey and D. M. German, "The Past, Present, and Future of Software Evolution," in *Proc. 24th Int. Conf. on Software Maintenance*, 2008. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.143.8949

[9] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, "Towards a taxonomy of software change:Research Articles," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, no. 5, p. 309, 2005. [Online]. Available: http://portal.acm.org/citation.cfm?id=1090744.1090746

[10] I. Sommerville, *Software Engineering*. Pearson Studium, 2007.

[11] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The architecture tradeoff analysis method," in *Engineering of Complex Computer Systems, 1998. ICECCS '98. Proceedings. Fourth IEEE International Conference on*, 10-14 1998, pp. 68 –78.

[12] R. Kazman, G. Abowd, L. Bass, and P. Clements, "Scenario-based analysis of software architecture," *Software, IEEE*, vol. 13, no. 6, pp. 47 –55, nov 1996.

[13] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet, "Architecture-level modifiability analysis (ALMA)," *Journal of Systems and Software*, vol. 69, no. 1-2, pp. 129–147, 2004.

[14] A. Hassan, "The road ahead for mining software repositories," *Frontiers of Software Maintenance, 2008. FoSM 2008*, pp. 48 – 57, 2008.

[15] S. H. Kan, *Metrics and Models in Software Quality Engineering (2nd ed.)*. Addison-Wesley Longman, 2002.

[16] N. E. Fenton and S. L. Pfleeger, *Software Metrics A Rigorous and Practical Approach*. PWS Publishing Company, 1997.

[17] andrena objects AG, "Isis," http://www.andrena.de/node/160, last retrieved 2012-06-01.

[18] R. C. Martin, *Clean Code A Handbook of Agile Software Craftmanship*. Prentice Hall, 2009.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[20] AUTOSAR-Konsortium, "Automotive open system architecture, http://www.autosar.org." [Online]. Available: http://www.autosar.org

[21] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[22] W. Hasselbring, R. Reussner, H. Jaekel, J. Schlegelmilch, T. Teschke, and S. Krieghoff, "The dublo architecture pattern for smooth migration of business information systems: an experience report," in *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, 23-28 2004, pp. 117 – 126.

[23] K. Czarnecki and U. W. Eisenecker, *Generative Programming*. Addison-Wesley, 2000.

[24] J. Bosch, *Design and Use of Software Architectures Adopting and evolving a product-line approach*. Addison-Wesley, 2000.

[25] OMG, "MDA Guide Version 1.0.1," *http://www.omg.org/cgi-bin/doc?omg/03-06-01*, 2003. [Online]. Available: http://www.omg.org/cgi-bin/doc?omg/03-06-01

[26] M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen, *Model-Driven Software Development*. John Wiley & Sons, 2006.

[27] Eclipse Foundation, "Eclipse modelling framework (emf)," http://www.eclipse.org/emf/, last retrieved 2012-06-01.

[28] O. Salo and P. Abrahamsson, "Agile methods in european embedded software development organisations: a survey on the actual use and usefulness of extreme programming and scrum," *Software, IET*, vol. 2 , Issue:1, pp. 58 – 64, 2008.

[29] C. Bommer, M. Spindler, and V. Barr, *Softwarewartung*. dpunkt.verlag, 2008.

[30] C. Wolf and E. M. Halter, *Virtualization: from the desktop to the enterprise*. Apress; 1 edition, 2005.

[31] 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)*, 2005, p. 13. [Online]. Available: http://portal.acm.org/citation.cfm?id=1108137

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

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

[34] H. Koziolek, R. Weiss, Z. Durdik, J. Stammel, and K. Krogmann, "Towards Software Sustainability Guidelines for Long-living Industrial Systems," in *Proc. of Soft. Eng. 2011 (SE2011), 3rd W. "Long-living Software Systems (L2S2)"*, 2011.

[35] O. Seng, F. Simon, and T. Mohaupt, *Code Quality Management*. dpunkt Verlag, Heidelberg, 2006.