Open access • Proceedings Article • DOI:10.1145/2000259.2000263

Sustainability evaluation of software architectures: a systematic review

Heiko Koziolek

Institutions: Ladenburg Thalmann

Published on: 20 Jun 2011 - Quality of Software Architectures

Topics: Software peer review, Software architecture, Software system, Sustainability and Software

Sustainability Evaluation of Software Architectures: A Systematic Review

Heiko Koziolek[1]
[1]Industrial Software Systems, ABB Corporate Research, Ladenburg, Germany
heiko.koziolek@de.abb.com

ABSTRACT

Long-living software systems are sustainable if they can be cost-efficiently maintained and evolved over their entire life-cycle. The quality of software architectures determines sustainability to a large extent. Scenario-based software architecture evaluation methods can support sustainability analysis, but they are still reluctantly used in practice. They are also not integrated with architecture-level metrics when evaluating implemented systems, which limits their capabilities. Existing literature reviews for architecture evaluation focus on scenario-based methods, but do not provide a critical reflection of the applicability of such methods for sustainability evaluation. Our goal is to measure the sustainability of a software architecture both during early design using scenarios and during evolution using scenarios and metrics, which is highly relevant in practice. We thus provide a systematic literature review assessing scenario-based methods for sustainability support and categorize more than 40 architecture-level metrics according to several design principles. Our review identifies a need for further empirical research, for the integration of existing methods, and for the more efficient use of formal architectural models.

1. INTRODUCTION

Software systems with a life span of more than 15 years must be designed and implemented carefully so that they are prepared for maintenance and evolution. During their life-time such systems inevitably undergo many corrective, adaptive, enhancive, and preventive changes. This is especially pronounced in the industrial automation domain, where software systems are embedded in complex technical hardware/software environments. Software architectures are a major driver for the sustainability (i.e., cost-efficient longevity) and evolvability [12, 73], because they influence how quickly and correctly a developer is able to understand, analyse, extend, test, and maintain a software system. Evaluating and improving the sustainability of a software architecture is thus a major concern for software architects.

QoSA '11 Boulder, USA

While researchers have proposed many scenario-based evaluation methods [25], it is not well understood how they support improving the sustainability of a system. In practice many architects still mainly rely on experience and prototyping to support their design decisions [10]. For implemented architectures, architecture-level code metrics assessing modularization quality can add valuable information to a sustainability evaluation [18], but an overview and systematic validation of such metrics is missing. Thereby, architecture-level code metrics are still sparsely used in practice.

Existing literature reviews for architecture evaluation methods [26, 9, 38, 11] focus mainly on scenario-based methods to evaluate early software architecture designs and do not analyse their suitability for sustainability evaluation. Other surveys [11, 59, 19] provide more breadth but do not include architecture-level metrics either. Reviews of architecture-level metrics cannot be found in literature, as related studies (e.g., [57]) focus on class-level OO metrics (e.g., McCabe [49], Halstead [33], Chidamber [24]) and neglect metrics for higher-level code structures.

The contribution of this paper is a structured literature review on methods and metrics for evaluating the sustainability of software architectures. Our review carefully analyses existing scenario-based methods for their suitability to evaluate sustainability and additionally provides a survey and analysis of more than 40 architecture-level metrics. An integration of scenario-based and metrics-based methods is useful to provide a continuous, pro-active approach towards evolution problem throughout the entire system life-cycle. Our survey is intended to help practitioners to select a method reflecting their specific requirements, and to help researchers to identify gaps and pointers for future work in the existing body of work. Our review also provides the base for a possible integration of both kinds of methods in a combined and even more valuable approach.

The remainder of this paper is as follows [39]: Section 2 defines the most important terms and motivates the need for a new review. Section 3 states our research questions, list the data sources, inclusion criteria and data collections. Section 4 then presents the results of the review, which shall answer the formally stated research questions. Section 5 discusses the results and provides implications for research and practice. Finally, Section 6 concludes the paper.

2. BACKGROUND

This section first defines the terms 'sustainability' (Section 2.1) and 'software architecture' (Section 2.2) and then discusses related surveys (Section 2.3).

2.1 Sustainability

To define the term 'sustainability' in the context of software architecture, we first introduce the notion of a long-living software system:

***Definition** 1.* A software-intensive system is *long-living* if it must be operated for more than 15 years.

Thus, a long-living software system usually needs to operate longer than its technical infrastructure, which for example consists of COTS, middleware, operating systems, and databases. In the industrial automation domain longevity requirements are rooted in the large investments in the controller, network, and field devices of the system. Besides changes of the technical infrastructure, a long-living software system faces new or changing functional and extra-functional customer requirements, changing business strategies, and potentially corrective and preventive maintenance during its life-cycle. The manageable and predictable operation of the system in terms of costs, customer requirements, and technological changes characterizes a sustainable system.

***Definition** 2.* A long-living software system is *sustainable* if it can be cost-efficiently maintained and evolved over its entire life-cycle.

The term 'sustainable' is derived from the Latin word 'sustinere' (tenere: to hold; sus, up) and thus best graps our intended connotation. Here, the term is *not* used in the sense of environment-friendliness as in other domains, but instead in the sense of cost-effective longevity and endurance. We do not use the terms 'maintainability' (ISO/IEC 25000), 'modifiability' [14], or 'evolvability' [15, 19], because they arguably include the notions of longevity and cost-effectiveness only to a limited extent.

The opposite of a sustainable software system is a long-living system that cannot be adapted to changing requirements and environments due to unjustifiable costs or even technical infeasibility. The architecture of a sustainable system may evolve during its life-cycle, but the fulfillment of customer requirements within timing, budget, and quality constraints must be assured.

Sustainability at least comprises the attributes maintainability (i.e., analysability, stability, testability, understandability), modifiability, portability, and evolvability. A sustainable software architecture can be achieved through adherence to design principles (e.g., modularity, separation of concerns, conceptual integrity) throughout the entire life-cycle. It requires pro-active planning for the long life-time of the system, which can be achieved by periodic evaluations of evolution scenarios.

2.2 Software Architecture

In the following, we use the *software architecture* definition of ISO/IEC 42010-2007: "Architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.". Software architecture descriptions consist of multiple *views* (e.g., functional, concurrency, development) dealing with higher-level software structures (i.e., components, modules, subsystems; not classes or methods).

A *scenario* is a brief description of a single interaction of a stakeholder with a system [25]. Scenario-based methods provide techniques for eliciting, documenting, and evaluating software architecture related scenarios against the requirements. Scenarios assessing the sustainability of an architecture are often called change, evolution, or exploratory scenarios [14]. A change scenario may impact multiple components. Undesired and costly *ripple effects* [66] can occur if the change to a component causes changes in dependent components. Thus, loose coupling between system components is a desirable property of a sustainable architecture to avoid such effects.

2.3 Related Reviews

Several authors have provided reviews on scenario-based and metrics-based evaluation methods.

Dobrica and Niemelä [26] classified eight early *scenario-based* evaluation methods for their activities, goals, addressed quality attributes and other criteria. In a similar survey, Barbar and Gorton [9] added several practical classification criteria, such as maturity stage, process support, and resources required. They also pointed out that several of the formerly reviewed methods were already dormant or merged with other methods. The same authors later analysed the state-of-practice in software architecture evaluation [8, 10] and found that scenarios are used by 54% of the polled software architects. Kazman et al. [38] criticized the bottom-up classification of scenario-based methods and thus proposed a new top-down classification, which they used to compare the ATAM [25] and ALMA [14] method. None of these surveys focussed on sustainability or analysed metrics-based methods.

Barcelos et al. [11] again classified scenario-based methods but also included a small number of measurement approaches without however analysing their suitability for sustainability analysis. In the same manner, Roy and Graham [59] surveyed scenario-based evaluation methods, but provided only limited comparison of the metrics-based methods. Breivold and Crnkovic [19] provided a broad structured literature review on architecture evolvability, which also included experience-based methods, design methods, and knowledge management techniques but provided no sustainability analysis on the scenario-based and metrics-based evaluation methods.

While there are numerous reviews and discussions on class-level metrics reported in literature [24, 57], there is no systematic review on *metrics-based* software architecture evaluation methods. Sarkar et al. [64] categorized several modularization metrics for higher-level software structures according to their adherence to well-known design principles, such as similarity of purpose, encapsulation, and layering. We reuse these generic principles in our review, but include more metrics. Riaz et al. [57] provided a systematic review of class-level OO metrics [49, 33, 24] and their usefulness for maintenance prediction, but did not take architecture-level metrics into account. Finally, Ducasse et al. [27] surveyed methods and tools for software architecture reconstruction, which can be helpful for the sustainability evaluation of implemented architectures.

3. REVIEW METHOD

3.1 Research Questions

The goal of our study is to review software architecture evaluation methods and metrics for the purpose of assessing their industrial applicability in the context of sustainability analysis from the perspectives of the software architect and

software analyst. From this goal, we derive three research questions:

- **RQ1:** How do scenario-based architecture evaluation methods used in industry support sustainability evaluation?
- **RQ2:** Which architecture-level metrics have been proposed to analyse the sustainability of software architectures?
- **RQ3:** What implications can be derived for the industrial and research communities from the findings?

The motivation for **RQ1** is to analyse the usefulness of existing methods for the specific purpose of sustainability evaluation, as most scenario-based methods were defined for the broader scope of a generic architecture evaluation. The aim is to identify gaps in currently used methods and to analyse the potential of combining scenario-based and metrics-based methods. Concrete sustainability criteria analysed will be described in Section 3.3.

As no systematic reviews on architecture-level metrics currently exist, **RQ2** first asks for an overview and classification of metrics proposed in literature. It is important to note that we restrict our study exclusively at *architecture-level* metrics dealing with subsystems, components, and interfaces. *Class-level* metrics have the potential to analyse and improve the sustainability of a software system as well, but are out of scope of our study.

Finally, **RQ3** aims at a discussion of the findings, potential for future research, and possible improvements and new directions for sustainability evaluation.

3.2 Data sources and search strategy

Search Process: We searched for software architecture evaluation methods in several books [40, 16, 25, 12, 47, 60, 73] and the journal and conference proceedings listed in Tab. 1. We used the following search engines for our reviews: ACM Digital Library, Google Scholar, IEEE Xplore, Elsevier ScienceDirect, and SpringerLink. We searched for term "software architecture" together with the following keywords: bad smells, evolvability, evolution, maintainability, maintenance, qualitative evaluation quantitative evaluation, scenario-based evaluation, metrics, modifiability, modularization, and sustainability.

Journals
ACM Trans. on Software Eng. and Meth. (TOSEM)
Communications of the ACM (CACM)
Elsevier Information and Software Technology (IST)
Elsevier Journal of Systems and Software (JSS)
Elsevier Science of Computer Programming (SCP)
IEEE Transactions on Software Engineering (TSE)
IEEE Computer
IEEE Software
Springer Empirical Software Engineering (EMSE)
Journal on Softw. Tools for Tech. Transfer (STTT)
Springer Software Quality Journal (SQJ)
Wiley Journal of Softw. Maint. and Evolution (JSME)
Wiley Journal of Systems Engineering (JSE)
Journal on Softw. Eng. and Knowledge Eng. (SEKE)

Conferences & Workshops
ACM/IEEE Int. Conf. on Softw. Eng. (ICSE)
ACM Symp. on the Foundations of Softw. Eng. (FSE)
IEEE Int. Symp. on Software Metrics (METRICS)
IEEE/ACM Int. Conf. on Automated Softw. Eng. (ASE)
Europ. Softw. Eng. Conference (ESEC)
IEEE Int. Conf. on Softw. Main. (ICSM)
Europ. Conf. on Software Maint. and Reeng. (CSMR)
Int. Worksh. on Principles of Softw. Evolution (IWPSE)
ERCIM Workshop on Software Evolution (EVOL)
Working IEEE/IFIP Conf. on Softw. Arch. (WICSA)
Int. Conf. on the Quality of Softw. Architecture (QoSA)
Europ. Conf. on Software Architecture (ECSA)

Table 1: Sources: Journals and Conferences

Inclusion and Exclusion Criteria: More than 20 scenario-based architecture evaluation methods have been proposed and categorized in various surveys [26, 9, 38, 10]. We exclude methods for which no industrial application has been reported or which target other quality attributes exclusively. We also exclude methods that are considered dormant [9] (e.g., SAAM, SBAR, SAMMCS), or for which no applications (e.g., case studies) or extensions have been reported for several years.

For metrics-based approaches, we only consider metrics concerning high-level software structures and exclude classical complexity metrics (e.g., McCabe [49], Halstead [33]) or class-level object-oriented metrics (e.g., Chidamber [24], Harrison [34]). We also exclude process metrics or other development metrics as our focus is on software architecture.
Quality Assessment: For quality assessment we include as primary studies only publications from the books cited before and the journals and conferences listed in Tab. 1. For both scenario-based methods and architectural metrics, we checked whether they have been applied in an industrial setting.

3.3 Data Collection

The following describes which data we extracted from the primary studies to evaluate scenario-based methods and metrics for their suitability in the context of sustainability evaluation.
Scenario-based methods: For the scenario-based methods, we can reuse most of the criteria from the former surveys [26, 9, 38] in our context. Each evaluation method serves a specific *goal* and requires a certain *architectural description*. Furthermore *tool support*, *process support*, *knowledge bases*, and the form of *validation* are generic requirements. Some additional, sustainability-specific requirements can be stated as follows:

- **Support for change scenario specification:** A useful method should provide templates for specifying change scenarios and guidelines for finding and eliciting such scenarios. A mature method could provide change scenario patterns from former applications, which may speed up scenario definition. To effectively support change scenario evaluation, a method should offer repeatable techniques for determining the consistency and completeness of change scenarios.
- **Support for analyzing ripple effects:** An ideal method should provide (semi-)automatic support for analyzing ripple effects. The possibility for such analyses is largely determined by the level of formalization of the architecture. Interface descriptions and dependencies between components are required. If possible, the analyzed architectural documentation should provide service effect specifications [56] for individual software components, which determine the control and data flow through components on an architectural level.
- **Support for analyzing variation and extension points:** Variability must be introduced with care as too much variability can complicate the evolution of a system, while too few variability makes a system inflexible [71]. Thus, a sustainability evaluation method should provide means to identify, analyze, and constrain variation and extension points.
- **Support for improving the architecture (heuristics):** A complete analysis method provide recommendations and techniques on how to improve the architecture. This is especially helpful for less experienced architects. An integration of an evaluation method with documented best practices in styles, patterns, and tactics is desirable [12, 38].
- **Support for trade-off analysis:** Optimizing a software architecture for sustainability inevitably leads to trade-offs with other quality attributes. For example, the performance of a sustainable system might be compromised,

if a system uses lots of indirection to ensure low coupling. Thus, a sustainability evaluation method should provide techniques to allow the analysis of trade-offs between sustainability and other quality attributes.

- **Support for legacy systems:** While many scenario-based methods target early design stages of a software system, the use case of assessing legacy systems is much more common in practice, especially for long-living software systems. Thus the interoperability of a method with architecture reconstruction tools is desirable [27].
- **Support for existing artifacts and tools:** To facilitate a broader adoption of any architecture analysis method, an integration with existing artifacts and tool chains is essential. For example, if a method provides analytical tools, they should be able to process existing design document (e.g., in UML). As architecture evaluation relies on the functional and extra-functional requirements of a system, interfacing with respective requirements management tools is desirable.
- **Return on investment (ROI):** Empirical evidence of a significant cost/benefit ratio is essential for any method to achieve broad adoption in industry. To better characterize any method, it should be clear how many resources are required and what kinds of results can be expected.

We also collected data about supplemental approaches and tools for the scenario-based methods, as well as metrics for ranking evolution scenarios.

Architectural Metrics: Architectural metrics often measure the modularization quality of a system under the assumption that a good modularization leads to better understandability, analysability, and maintainability and thus sustainability. We extracted the *name* of each metric, its *abbreviation*, an *intuitive description*, and the *required inputs* from the primary studies. The latter helps to analyse whether the metric can be determined automatically using tools. Additionally, we categorized the metrics according to common modularization design principles [64]:

- **Similarity of purpose:** This principle states that functions and data structures serving similar purposes or aiming at common goals should be grouped in a single module, package, or subsystem. It implies high cohesion within such a module.
- **Encapsulation:** Information hiding improves understandability and analysability. Modules should have an explicitly defined, restricted API through which all inter-module call traffic should be routed.
- **Independent compilability, extendibility, testability:** These principles allow modules to grow in parallel and to be tested independently. The possibility for an independent evolution of modules is important for cost-effective development.
- **Acyclic dependencies:** As cyclic dependencies negate many of the benefits of modularization, they should be avoided where possible. In layered architectures, control flow should be directed only from upper layers to lower layers but not vice versa.
- **Size:** While there are no universally agreed module sizes, it is plausible that modules should neither be too large nor to small. A uniform size distribution is desirable to improve the maintainability of a system. This however can hardly be planned when implementing systems from scratch, but is rather a desirable property when clustering legacy systems.

3.4 Data Synthesis

Data synthesis involves collating and summarizing the results of the included primary studies. We chose a descriptive synthesis and display the information extracted from the primary studies in tables.

4. RESULTS

This section reports on the results for research questions RQ1 (Section 4.1) and RQ2 (Section 4.2). RQ3 will be tackled in Section 5.2.

4.1 Scenario-based Methods (RQ1)

Research question RQ1 asks for the suitability of current scenario-based methods for sustainability evaluation. We found more than 20 scenario-based evaluation methods in literature. Our exclusion criteria limit the scope to ATAM [25] and ALMA [14], because these are the only active methods that have been applied in a number of industrial case studies in different domains [9]. For example, SAAM [25] is no longer supported by its creator as it has been superseeded by ATAM. Additionally, its various derivates are dormant or have been merged into other methods [9]. PASA and SALUTA have been applied in industry but target performance and usability exclusively.

4.1.1 ATAM

The goals of ATAM [25] are to identify trade-offs between different quality attributes for a system and reveal sensitivity points in an architecture. ATAM involves presenting and discussing the architectural design in a 2 day workshop attended by the system's major stakeholders. The participants define scenarios (i.e., regular use-cases, growth, and exploratory scenarios) to evaluate the architecture against and discuss the technical constraints of the system. Concerning tool support, there is a web-based tool available [44], which is however not regularly used. There are no specific modeling tools targeted by ATAM, but the method requires a logical and module view of the architecture.

ATAM supports sustainability evaluation as follows: the definition of *change scenarios* is assisted with the quality attribute scenario template and procedures for a quality attribute workshop (QAW) to determine them. The method description explicitly mentions that growth and exploratory scenarios, which can represent change scenarios, should be described. Stakeholders have to determine *ripple effects* manually, as the granularity of architectural description used in ATAM does not allow a formal analysis. There is no special support for analysing *variation and extension points*. ATAM offers some sustainability *improvement* recommendations in the form of change-oriented architectural styles and so-called modifiability tactics. For *trade-off analysis*, the stakeholders rank different quality attributes in a so-called utility tree. To support analysing *legacy systems*, the SEI proposed the DALI workbench, which however has not been applied on systems greater than 50 KLOC and seems to be dormant since 2001. There is no special support for existing *tool chains*. The industrial maturity has been proven in more than 20 industrial case studies [25, 13, 17]. The effort for applying ATAM is estimated between 30-70 person days [25], but it was never attempted to quantify its

benefits.

Evaluation: ATAM was not specifically designed for sustainability evaluation, but more generically for assessing risks and trade-offs between quality attributes. Nevertheless, it offers many helpful techniques in the context of sustainability evaluation (e.g., utility trees trading off sustainability with other attributes, quality attribute scenarios to document expected changes, abstract modifiability tactics). Other authors have noted that ATAM does not allow for in-depth analysis of scenarios [38], because of the strict timing constraints of the workshop. However, identified sustainability risks could be further evaluated with other methods after ATAM. In general ATAM is not perceived as a light-weight evaluation method, because it requires to conduct expensive workshops with all major stakeholders.

4.1.2 ALMA

The goals of ALMA [14] are to predict maintenance effort, to assess risks, or to compare candidate architecture w.r.t. to modifiability. There is no specific workshop described with the method but the required information is collected by an analyst through stakeholder interviews. Based on a software architecture description the analyst elicits and evaluates change scenarios for the system together with the stakeholders (e.g., architects, developers, customers). Maintenance effort is predicted by estimating the expected extent of code changes per evolution scenario in lines of code. Risk analysis involves weighting the impact of evolution scenarios, and candidate architecture comparison is based on calculating an overall score for implementing the evolution scenarios for all candidates.

ALMA supports sustainability evaluation as follows: a top-down and bottom-up technique for finding *change scenarios* is sketched and the authors suggest partitioning the scenarios into equivalence classes. There is no documentation template provided. Determining *ripple effects* is explicitly foreseen but relies on the experience of the involved architects and developers, which can be misleading [42]. There is no support for analysing *variation and extension points* and there is no guidance on how to *improve* the architecture after evaluation. *Trade-off analysis* is not in the scope of ALMA, which exclusively focuses on modifiability. The architecture of a *legacy* system has to be extracted manually or using other methods into the required views and there is no *tool* support. The authors have applied ALMA in seven industrial case studies [14] until 2004, but no third party applications are known and effort estimations for a *ROI* calculation are missing.

Evaluation: ALMA was specifically designed for modifiability evaluation, which is closely related to sustainability evaluation. It offers some helpful techniques for change scenario elicitation but still relies heavily on the experience of the involved stakeholders as there is no guidance on how to improve the architecture. Some experience reports with ALMA offer interesting insights to sustainability evaluation [41, 42]: During change scenario elicitation architects are biased towards the scenarios they already had in mind when designing the architecture. Determining the components affected by a change scenario is often straight-forward, but determining ripple effects is not. Often, stakeholders miss important change scenarios during architectural evaluation. Furthermore, the architecture and the change scenarios may be based on an incorrect initial requirements specification, which then invalidates them. In conclusion, support for architectural improvements, trade-off analysis, and extension points would be desirable for ALMA besides tool support.

4.1.3 Supplemental methods and applications

The following supplemental approaches are not among the primary studies of our systematic review for various reasons, but could be combined with ATAM and ALMA for improved sustainability analysis and are therefore listed here for completeness:

- Kazman et al. [37] combined ATAM and CBAM, and added a more in-depth model-based analysis for selected risks to form the "Analytic Principles and Tools for the Improvement of Architectures" (APTIA) method. They also used the method in a case study to analyse the variation points of an architecture.
- Shen and Madhavji [68] describe the "Evolutionary Scenario Development Method" (ESDM), which includes an elaborate template for change scenario specification and shall be used in combination with ATAM.
- Olumofin and Misic [52] introduce the "Holistic Product Line Architecture Assessment" (HooPLA) method extending ATAM for SPL evaluation and provide a qualitative analytical treatment of variation points using scenarios.
- Breivold and Crnkovic [20] propose the "Architecture Evolvability Analysis" (AREA) method, which evaluates change stimuli against fine-grained sustainability attributes (e.g., analysability, testability) and defines respective refactorings and test cases. The method was prototypically applied for assessing a single change scenario of an industrial system.

In addition to the proposed extensions to scenario-based evaluation methods, there are several case study reports in literature [48, 32, 17].

4.1.4 Metrics for Scenario Ranking

Several authors have also proposed metrics to rank evolution scenarios and use them for maintenance effort predictions:

- Avritzer and Weyuker [7] created a list of potential project issues based on architectural reviews and ranked them to get a simple project risk metric. These issues are however only loosely related to the structure of the system.
- Paulish and Bass [54] decompose evolution scenarios into smaller tasks and asked developers for the task efforts.
- Liu and Wang [43] propose two metrics (Impact On the Software Architecture (IOSA), Adaptability Degree of Software Architecture (ADSA)) based on probabilities for evolution scenarios and their impact in terms of affected lines of code or function points. Tarvainen [72] applies these metrics in a case study.
- Stammel and Reussner [69] propose the KAMP tool, which lets architects decompose evolution scenarios into smaller change actions based on a formal architectural model and uses summarized implementation effort estimations for scenario ranking.
- Anwar et al. [6] compute the "maintenance effort" per evolution scenario based on probability weights and estimated LOC using COCOMO II.

In conclusion scenario ranking methods combine classical cost estimation techniques (e.g., COCOMO, Function Points) with evolution scenarios to enable early maintenance effort predictions.

4.1.5 Conclusions RQ1

Table 2 summarizes the sustainability support of our primary studies ATAM and ALMA. While ATAM is more refined and offers more features, ALMA is more specifically designed for sustainability evaluation. In practice, a combination of ATAM, ALMA, and the supplemental methods and scenario-ranking approaches is advisable. It is common that the methods are not used exactly as they are documented [10], but that different parts of them are recombined as needed. Nevertheless, existing scenario-based methods do not provide systematic analysis of ripple effects, integration with reverse engineering tools, or knowledge management support and could benefit from more formal models.

Name	ATAM	ALMA
General criteria		
Goals	Sensitivity & tradeoff-analysis	Change impact analysis, predicting maintenance effort
Architectural Description	Process, data-flow, uses, physical & module view	Any
Process support	Comprehensive	Limited
Tool support	n/a	n/a
Knowledge repository	Recommended	n/a
Validation	>20 industrial case studies	7 industrial case studies (none since 2004)
Sustainability criteria		
Change scenario specification	Quality attribute scenario template	Top-down, bottom-up method, no template
Ripple effect analysis	Manual, based on experience	Manual, based on experience
Variation/extension point analysis	n/a	n/a
Architecture improvement	Tactics, Styles	n/a
Trade-off analysis	Based on utility tree with preferences	n/a
Legacy systems	No support	No support
Existing artifacts and tools	No explicit support	No explicit support
Return on Investment	Costs: 30-70 person days Benefits: not quantified	Costs: unknown Benefits: not quantified

Table 2: Comparison of scenario-based evaluation methods

4.2 Metrics-based Methods (RQ2)

Concerning **RQ2** this section summarizes architecture-level metrics suites for software architectures. We briefly describe each approach (Section 4.2.1) and provide a categorization in Tab. 3. Furthermore, we list some recent approaches for defining and detecting architecture bad smells, which are closely related to architecture-level metrics (Section 4.2.2).

4.2.1 Architecture-level Metrics Suites

Most work in the area of architecture-level metrics derives from the module concept described by Parnas [53] and the notions of coupling and cohesion [70, 74]. Software complexity metrics [49, 33] as well as class-level object-oriented metrics [24, 57, 1] are out of scope for our study. Our systematic review extracted the following metrics suites and approaches from literature:

M1 Briand et al. [22, 21] provide a generic formalization for metrical notions independent of any programming paradigm, which includes coupling and cohesion besides complexity-based and class-based metrics. The authors also formally define coupling and cohesion between modules.

M2 Lakos [40] defines a metric called Cumulative Component Dependency (CCD), which is the sum of required dependencies by a component within a subsystem. Derived metrics are the average component dependency (ACD) and the normalized CCD (NCCD). They can be determined by tools such as SonarJ or STAN.

M3 Mancoridis et al. [45, 51] introduce a clustering tool called BUNCH, which tries to optimize the proposed modularization quality metric. This metric is based on a partitioned module dependency graph and computed by the difference of the average inter- and intra-connectivity of the partitions.

M4 Allen and Khoshgoftaar [3, 4, 2] propose an information-theory based approach to define coupling and cohesion metrics. Opposed to the former count-based measures, their definitions are based on the entropy in a module interconnection graph, which accounts for patterns in the relationships. They found that the information-theory based metrics were able to make finer distinctions than the count-based metrics.

M5 Martin [47] defines metrics for software packages, i.e., groups of related classes (e.g., java packages, C++-projects, low-level modules). These include package afferent coupling, efferent coupling, abstractness, instability, distance from main sequence, and package dependency cycles. Several tools support measuring these metrics (e.g., JDepend, CppDepend, STAN).

M6 Sant'anna et al. [62] propose a set of 11 concern-driven metrics to measure the modularity of a software system. An architectural concern is defined as a partition of system components with a common goal (e.g., GUI, persistence, distribution). Examples for the metrics are concern diffusion (i.e., counts the number of components or interfaces for a given concern), coupling between concerns (i.e., counts for a concern how components relate itself and other concerns), or interface complexity. The authors compare the modularization of aspect-oriented and non aspect-oriented systems using their metrics in three case studies.

M7 Sarkar et al. [64] create a set of 12 API-based and information-theoretic metrics for measuring modularization quality. The metrics rely on the definition of APIs between modules, module size thresholds, and concept term maps and explicitly exclude any object-oriented features. Each metric is defined between 0 and 1 where higher values are better. Some metrics are derived from the works of Lakos [40] and Martin [47]. The metrics were applied on a number of open source systems (e.g., Apache, MySQL, Mozilla) as well as a 12 MLOC commercial system [65], but the authors do not provide their tools publicly.

M8 Sarkar et al. [63] extend their former work on module metrics with 9 additional metrics concerning object-oriented relationships (e.g., inheritance, association) between higher-level modules in large OO-systems. These metrics for example measure the extent of the fragile base-class problem or inheritance relations between higher-level modules. They were measured for eight open source systems between 30 KLOC and 2.5 MLOC, where a human

#	Source	Abbr.	Name	Description	Required Input	Tool
Similarity of Purpose						
M7	Sarkar2007	CDM	Concept Domination Metric	Non-uniformity of the distribution of concepts	List of concepts, frequency of occurrences per mod.	Proprietary
M7	Sarkar2007	CCM	Concept Coherency Metic	Amount of mutual information between mod./concept	List of concepts, entropy for concepts	Proprietary
M7	Sarkar2007	APIU	API Function Usage Index	Percentage of API functions used by other modules	API definition, # calls to API	Proprietary
M6	Sant'anna2007	CDAC	Concern Diffusion over Arch. Components	Counts the components realizing an arch. concern	Mapping of components to architectural concerns	
M11	Sethi2009	CS	Concern Scope	Amount of design decisions influenced by a concern	Design decisions, concerns	
M11	Sethi2009	CO	Concern Overlap	Amount of design decisions infl. by multiple concerns	Design decisions, concerns	
Encapsulation						
M1	Briand1996	RCI	Ratio of Cohesive Interactions	Ratio of potential/known data declarations interactions	Module dependencies	
M1	Briand1996	IC	Import Coupling	Extend to which a module depends on externals	# imports per module	
M1	Briand1996	EC	Export Coupling	Interactions between internal/external data decl.	Module dependencies	
M3	Mancoridis1998	MQ	Modularization Quality	Diff. of inter- and intra-connectivity of subsystems	Module dependency graph, clusters	Bunch
M5	Martin2003	Ca	Afferent Couplings	# packages depending on classes in a package	Class dependencies	JDepend
M5	Martin2003	Ce	Efferent Couplings	# packages the classes of a package depend on	Class dependencies	JDepend
M6	Sant'anna2007	CLIC	Comp.-level Interlacing Betw. Concerns	Counts components sharing concerns	Mapping of components to architectural concerns	
M6	Sant'anna2007	LCC	Lack of Concern-based Cohesion	Counts the number of concerns by a component	Mapping of components to architectural concerns	
M7	Sarkar2007	MII	Module Interaction Index	Percentage of calls routed through APIs	Module and API definition	Proprietary
M7	Sarkar2007	NC	Non-API Function Closedness Index	Percentage of functions classified API or non-API	API definition	Proprietary
M7	Sarkar2007	IDI	Implicit Dependency Index	Percentage of explicit module dependencies	# Implicit module dep. (e.g., global variables, files)	Proprietary
M8	Sarkar2008	BCFI	Base class fragility index	Extent of base-class fragility in the system	Classes, ancestors, inherited methods, depend.	Proprietary
M8	Sarkar2008	IC	Inheritance-based intermodule coupling	Fraction of classes in other mod. defined by inherit.	Module definition, inheritance dependencies	Proprietary
M8	Sarkar2008	NPII	Not-programming-to-interfaces Index	Percentage of calls to to root interfaces	Interface definitions, call dependencies	Proprietary
M8	Sarkar2008	AC	Association-induced coupling	Percentage of class associations to other modules	Module definition, associations	Proprietary
M8	Sarkar2008	SAVI	State Access Violation Index	Extend of intermodule access to internal state	Module definition, state accesses	Proprietary
M10	Anan2009	IEAS	Entropy of an architectural slicing	Amount of information encoded in a arch. layer	Mapping of modules to layers, dependency graph	
M10	Anan2009	ASC	Architecture Slicing Cohesion	Ratio of intra- and intermodule coupling	Mapping of modules to layers, dependency graph	
M11	Sethi2009	DV	Decision Volatility	Stability of a decision decision ag. ext. influences	Design decisions, env. impact, impact scope	
Compilability, Extendibility, Testability						
M2	Lakos1996	CCD	Cumulative Component Dependency	Sum of component dependencies in a subsystem	Component dependency graph for a subsystem	SonarJ
M2	Lakos1996	ACD	Average Cumulative Comp. Dependency	CCD divided by components in subsystem	Component dependency graph for a subsystem	SonarJ
M2	Lakos1996	NCCD	Normalized Cumulative Comp. Dependency	CCD divided by CCD of a binary dependency tree	Component dependency graph for a subsystem	SonarJ
M4	Allen2001	COUM	Coupling of a module	Amount of information in intermodule-edges graphs	Module dependency graph	
M4	Allen2001	ICM	Intramodule coupling of a module	Amount of information in intramodule-edges graph	Module dependency graph	
M4	Allen2001	COHM	Cohesion of a module	Amount of information in intramodule coupling	Module dependency graph	
M5	Martin2003	A	Abstractness	Ratio of abstract classes to total classes in package	Class definitions in a package	JDepend
M5	Martin2003	I	Instability	Ratio of efferent to total coupling [I=Ce/(Ce+Ca)]	Class dependencies	JDepend
M5	Martin2003	DMS	Distance from the Main Sequence	Perpendicular dist. of a package from the line A + I = 1	Abstractness and Instability	JDepend
M7	Sarkar2007	MISI	Module Interaction Stability Index	Percentage of module depending on stable layers	Mapping of modules to layers, fan-in, fan-out	Proprietary
M7	Sarkar2007	NTDM	Normalized Testability Dependency Metric	Percentage of module independent testing	Test dependencies between modules	Proprietary
M8	Sarkar2008	PPI	Plugin Polution Index	Amount of superfluous code in a plugin module	Extension API, abstract methods in plugins	Proprietary
M11	Sethi2009	CI	Change impact	Amount of design decisions changed during evolution	Design decisions, evolution scenario	
M11	Sethi2009	IL	Independence Level	System perc. changeable under stable design rules	Independent module set in augmented constr. netw.	
Acyclic Dependencies						
M5	Martin2003	PDC	Package Dependency Cycles	Cyclic dependencies between packages	Package dependency graph	JDepend
M7	Sarkar2007	Cyclic	Cyclic Dependencies Index	Extent of cyclic dependencies between modules	Module dependency graph	Proprietary
M7	Sarkar2007	LOI	Layer Organization Index	Cyclic dependencies between layers	Mapping of modules to layers, dependency graph	Proprietary
M9	Sangwan2008	XS	Excessive Structural Complexity	Cyclic dependencies violation times amount of dep.	Module dependency graph	Structure101
Size						
M7	Sarkar2007	MSBI	Module Size Boundness Index	Deviation of module sizes from a threshold	Lines of code per module, optimal module size	SourceMonitor
M7	Sarkar2007	MSUI	Module Size Uniformity Index	Distribution of module sizes	Lines of code per module	SourceMonitor

Table 3: Architecture-level software metrics potentially useful in the context of sustainability evaluation

modularization achieved significantly higher values than a randomized modularization based on assigning classes arbitrarily to modules.

M9 Sangwan et al. [61] introduce the complexity measurement framework Structure 101, which uses a metric called excessive structural complexity (XS). It is computed as the product of the degree of cyclic dependencies violations (metric 'tangled') and a multi-level complexity metric ('fat'), which can also be determined on the package or module level.

M10 Anan et al. [5] propose an approach that is similar to Allen and Khoshgoftaar [4] and use information-theory to measure the coupling between modules. The authors compute the entropy of an architectural slicing (i.e., a module layer) and condense the values to a metric called "architecture maintainability effort". The metrics were applied for a number of artificial module dependency graphs.

M11 Sethi et al. [67] base their metrics not on source code, but instead on an augmented constraint network and design structure matrix derived from a higher-level UML component diagram, which also captures architectural concerns and design rules. They define metrics such as decision volatility (i.e., impact of environmental conditions on a design decision) and concern overlap. They evaluate the metrics on eight object-oriented and aspect-oriented releases of a software product line and find for example that the aspect-oriented design is better in accommodating optional features, but also leads to a higher design volatility.

Besides mapping the metrics to design principles as in Tab. 3, they could also be classified according to their required inputs. Some metrics can be determined by simply analysing source code, while others require additional inputs, e.g., concept maps, design decisions, or module size thresholds. Some metrics are similar or overlapping (e.g., MISI is derived from Martin's Instability, NTDM is an adaption of CCD).

4.2.2 Architecture Bad Smells

Several authors have proposed to transfer the idea of "bad smells" in code [29] to higher-level software structures, where they are called "architecture smells" [58]. It is also conceivable to deduce a condensed metric for sustainability by weighting a number of found architectural smells. They are typically used in the evaluation of implemented architectures [18].

Kazman and Burth [36] propose the so-called "Interactive Architecture Pattern Recognition" (IAPR) approach for detecting architecture smells. The approach is supported by a tool for diagnosis and exploration. In a case study, the authors demonstrate the detection of cyclic dependencies, layering violations, communication constraint violations, classes with high fan-out or high fan-in. The approach can be considered a precursor to recent architecture smell investigations, but was not followed-up by the authors.

Roock and Lippert [58] provide an extensive catalog of architecture smells. They include smells in dependency graphs (e.g., static cycles), inheritance hierarchies (e.g., parallel inheritance), packages (e.g., too large), subsystems (e.g., overgeneralization), and layers (e.g., upward references). They

also list tools for locating architecture smells and describe best practices on how to perform large-scale refactorings.

Garcia et al. [30] describe four architectural smells (i.e., connector envy, scattered parasitic functionality, ambiguous interfaces, and extraneous adjacent connectors) and discuss their quality impacts and trade-offs. They regard tool support for smell detection as future work.

4.2.3 Conclusions RQ2

Our review has provided a list of more than 40 architecture-level metrics, which could assist sustainability evaluation of implemented architectures. A mix of these metrics as well as the architecture bad smells should be identified for a given project, as no single metric is able to characterize the overall sustainability of the implementation. The metrics can be combined with class-level metrics [57], process metrics, and other development metrics [35] and can be monitored during system evolution (e.g., as done in the ISIS approach [55]). Normalizing the metrics between 0 and 1 [64] improves understandability. Most metrics are currently based on plausibility and have not been systematically validated empirically. Their concrete value towards sustainability improvement is thus still unknown.

5. DISCUSSIONS

5.1 Principle findings

Our survey analysed the capabilities of scenarios-based methods and architecture-level metrics for evaluating the sustainability of software architectures. While there are many *scenario-based methods* proposed, only ATAM and ALMA have so far been used repeatedly in industrial case studies in different domains. Since ALMA (published in 2004), no new method was validated extensively. ALMA too lacks newer experience reports. Practitioners still mostly rely on experience and prototypes instead of following a scenario-based method step-by-step [10]. Nevertheless applying the methods likely results in more sustainable software architectures. For an effective sustainability architecture review, the best techniques from ATAM, ALMA, and supplemental methods should be combined.

Besides the plethora of class-level OO metrics, there is now also a growing number of metrics on the architectural level. These mostly focus on the modularization quality of an architecture and require an implementation for measurement. A comprehensive set of metrics was proposed by Sarkar et al. [64, 63], but tool support is still missing. There is potential to define metrics based on architecture documentation (e.g., UML component diagrams [67]). Desired values for the metrics as well as systematic empirical evaluations are mostly missing. Nevertheless, reviewers use metrics informally during the evaluation of implemented architectures [18].

No end-to-end method for sustainability evaluation from requirements to maintenance is available.

5.2 Implications for research/practice (RQ3)

Based on our review, we identified the following implications for research and practice:

- **More empirical research needed:** both scenario-based methods and architecture-level metrics should be evaluated more thoroughly and repeatedly in empirical studies especially regarding their return on investment. This should extend the current amount of controlled experiments, field studies, and surveys [28]. Besides a sound scientific assessment of benefits and drawbacks, empirical studies can also be helpful to overcome the limited application of the analysed approaches in practice [10], as they might be used to advocate the methods better. A return on investment quantified through empirical studies could greatly speed up their adoption in practice. Another open question is how much effort should be invested upfront in the definition of evolution scenarios since the evolution of a system is often unpredictable [42] and thus might lead to wasted efforts.
- **Integration of methods:** while the combination of different scenario-based methods or the combination of multiple metrics can be helpful, there is also potential for combining the qualitative and quantitative approaches formerly presented. As quantitative methods are usually only applicable on implemented architectures, combined methods should target legacy systems. Initial attempts at combined methods have been made by Briand and Wüst [23] (SAMM + coupling measures) and Kazman et al. [37] (ATAM + rate monotonic analysis and variability analysis). Methods from ALMA for change scneario elicitation could be combined with ATAM techniques for trade-off analysis. A metrics assessment report evaluation step could be integrated into the scenario-based approaches. Additionally, architecture recovery tools [27] or automated improvement tools [46] could be integrated with the methods.
- **Effective use of formal models:** scenario-based evaluations are usually based on early, informal architecture models. Thus, tool-supported analysis (e.g., for ripple effects, undesired dependencies, other non-functional properties) is hardly feasible at this stage. As more formal models require a higher specification effort, methods and tools to rapidly construct and analyse such models would be desirable. Modeling tools could provide aids to ask the architects precisely for the required information for a certain analysis (e.g., more refined interface specifications or dependencies) given an early architectural model. Modelling notations and tools should support iterative refinements based on the successively increasing available information during architectural design.
- **Codify experiences:** the outcome of most scenario-based evaluation methods is still largely determined by the experience of the participants. Patterns and tactics are a way to encode design knowledge and make good design practice available even to inexperienced architects. Knowledge bases should be created to capture and reuse the experiences from former sustainability evaluations. Also architecture-level metrics could be accompanied with a catalogue or guidelines for architecture-level refactorings aiming at improving specific metric values.
- **Explore other approaches:** Using scenarios during early design stages and metrics during implementation and maintenance seem to be the most popular methods to improve software architecture evolution. Other approaches (e.g., based on simulation or other techniques) are not explored extensively. Garlan et al. [31] propose a method and tool called AEvol for assisting software architects in designing and analysing architectural evolution paths. Mens et al. [50] propose an approach and tool called Evolve to link architectural descriptions to im-

plementation artifacts and making the evolution intrinsic to the architectural description. These approaches still need refinement and maturation but provide pointers for future directions of developing sustainable software architectures.

5.3 Strength and weaknesses of this review

Our review is the first review on architecture evaluation including both scenario-based and measurement-based approaches. It thus provides a more holistic perspective. Compared to former reviews, we use a more restricted scope (i.e., sustainability evaluation) and include more recent studies. Our survey includes architecture-level metrics, which were formally not reviewed systematically.

However, some limitations are inherent to our review. We use very selective inclusion criteria and potentially exclude promising studies, which have not been evaluated thoroughly. Our survey might be biased towards to more popular evaluation methods and metrics also since we focus on a restricted set of renowned books, journals, and conference proceedings. We have not applied the methods and metrics ourselves and can thus not be sure of their applicability.

6. CONCLUSIONS

This paper provided a systematic review on scenario-based evaluation methods and architecture-level metrics for the sustainability of software architectures. We analysed the suitability of existing methods for sustainability analysis and assembled a list of more than 40 architecture-level metrics. We discussed implications for practice and research.

Practitioners can use our review to tailor their own sustainability evaluation method based on the referenced methods and tools. Researchers can identify gaps in the body of work and create systematic sustainability evaluation methods.

Our review identifies a need for more empirical studies on architecture evaluation. Scenario-based methods should better validate their potential return on investment and metrics require a validation of their relevance for sustainability. Additionally the existing methods and metrics should be combined in integrated approaches that offer even more benefits. Using more formal architecture models could provide more automated analyses during architecture analyses.

7. REFERENCES

[1] J. Al Dallal and L. C. Briand. An object-oriented high-level design-based class cohesion metric. *Inf. Softw. Technol.*, 52:1346–1361, December 2010.

[2] E. Allen, S. Gottipati, and R. Govindarajan. Measuring size, complexity, and coupling of hypergraph abstractions of software: An information-theory approach. *Software Quality Journal*, 15(2):179–212, 2007.

[3] E. B. Allen and T. M. Khoshgoftaar. Measuring coupling and cohesion: An information-theory approach. In *Proceedings of the 6th International Symposium on Software Metrics*, pages 119–, Washington, DC, USA, 1999. IEEE Computer Society.

[4] E. B. Allen, T. M. Khoshgoftaar, and Y. Chen. Measuring coupling and cohesion of software modules: An information-theory approach. In *Proceedings of the 7th International Symposium on Software Metrics*, METRICS '01, pages 124–, Washington, DC, USA, 2001. IEEE Computer Society.

[5] M. Anan, H. Saiedian, and J. Ryoo. An architecture-centric software maintainability assessment using information theory. *J. Softw. Maint. Evol.*, 21:1–18, January 2009.

[6] S. Anwar, M. Ramzan, A. Rauf, and A. Shahid. Software Maintenance Prediction Using Weighted Scenarios: An Architecture Perspective. In *Proc. Int. Conf. on Information Science and Applications (ICISA'10)*, pages 1–9. IEEE, 2010.

[7] A. Avritzer and E. Weyuker. Metrics to assess the likelihood of project success based on architecture reviews. *Empirical Software Engineering*, 4(3):199–215, 1999.

[8] M. A. Babar, L. Bass, and I. Gorton. Factors influencing industrial practices of software architecture evaluation: an empirical investigation. In *Proc. 3rd. Int. Conf. on the Quality of Software Architectures (QoSA'07)*, QoSA'07, pages 90–107, Berlin, Heidelberg, 2007. Springer-Verlag.

[9] M. A. Babar and I. Gorton. Comparison of scenario-based software architecture evaluation methods. In *Proc. 11th Asia-Pacific Software Engineering Conf.*, APSEC '04, pages 600–607, Washington, DC, USA, 2004. IEEE Computer Society.

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

[11] R. Barcelos and G. Travassos. Evaluation approaches for software architectural documents: a systematic review. In *Ibero-American Workshop on Requirements Engineering and Software Environments (IDEAS'06)*, 2006.

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

[13] L. Bass, R. Nord, W. Wood, and D. Zubrow. Risk themes discovered through architecture evaluations. Technical Report CMU/SEI-2006-TR-012, Software Engineering Institute (SEI), 2006.

[14] P. 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.

[15] S. Bode and M. Riebisch. Impact evaluation for quality-oriented architectural decisions regarding evolvability. In *Proc. 4th Europ. Conf. on Software Architecture (ECSA'10)*, volume 6285 of *LNCS*, pages 182–197. Springer, 2010.

[16] J. Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. Addison-Wesley Professional, 2000.

[17] N. Boucké, D. Weyns, K. Schelfthout, and T. Holvoet. Applying the atam to an architecture for decentralized control of a transportation system. In C. Hofmeister, I. Crnkovic, and R. Reussner, editors, *Quality of Software Architectures*, volume 4214 of *Lecture Notes in Computer Science*, pages 180–198. Springer Berlin / Heidelberg, 2006.

[18] E. Bouwers, J. Visser, and A. Van Deursen. Criteria for the evaluation of implemented architectures. In *Proc. 25th IEEE Int. Conf. on Software Maintenance (ICSM'09)*, pages 73–82. IEEE, 2009.

[19] H. Breivold and I. Crnkovic. A Systematic Review on Architecting for Software Evolvability. In *21st Australian Software Engineering Conference*, pages 13–22. IEEE, 2010.

[20] H. Breivold, I. Crnkovic, R. Land, and M. Larsson. Analyzing Software Evolvability of an Industrial Automation Control System: A Case Study. In *3rd Int. Conf. on Software Engineering Advances (ICSEA'08)*, pages 205–213. IEEE, 2008.

[21] L. Briand, S. Morasca, and V. Basili. Property-based software engineering measurement. *IEEE Trans. on Softw. Eng.*, 22(1):68–86, 1996.

[22] L. C. Briand, S. Morasca, and V. R. Basili. Measuring and assessing maintainability at the end of high level design. In *Proc. Int. Conf. on Sofw. Maintenance (ICSM'93)*, pages 88–97, Washington, DC, USA, 1993. IEEE Computer Society.

[23] L. C. Briand and J. Wüst. Integrating scenario-based and measurement-based software product assessment. *Journal of Systems and Software*, 59(1):3–22, 2001.

[24] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20:476–493, June 1994.

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

[26] L. Dobrica and E. Niemelä. A survey on software architecture analysis methods. *IEEE Trans. on Softw. Eng.*, 28(7):638–653, July 2002.

[27] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Trans. Softw. Eng.*, 35:573–591, July 2009.

[28] D. Falessi, M. A. Babar, G. Cantone, and P. Kruchten. Applying empirical software engineering to software architecture: challenges and lessons learned. *Empirical Software Engineering*, 15(3):250–276, 2010.

[29] M. Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[30] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In *Proc. 13th European Conf. on Software Maintenance and Reengineering (CSMR'09)*, pages 255–258. IEEE, 2009.

[31] D. Garlan, J. Barnes, B. Schmerl, and O. Celiku. Evolution styles: Foundations and tool support for software architecture evolution. In *Proc. IEEE/IFIP Working Int. Conf. on Software Architecture (WICSA'09)*, pages 131–140. IEEE, 2009.

[32] B. Graaf, H. van Dijk, and A. van Deursen. Evaluating an Embedded Software Reference Architecture - Industrial Experience Report. In *9th European Conf. on Software Maintenance and Reengineering (CSMR'05)*, pages 354–363. IEEE, 2005.

[33] M. H. Halstead. *Elements of Software Science*. Elsevier Science Inc., New York, NY, USA, 1977.

[34] R. Harrison, S. Counsell, and R. Nithi. An Evaluation of the MOOD Set of Object-Oriented Software Metrics. *IEEE Trans. on Softw. Eng.*, 24(6):491–496, 1998.

[35] S. H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, Boston, MA, USA, 2nd edition, 2002.

[36] R. Kazman. Assessing architectural complexity. In *Proc. 2nd EUROMICRO Conf. on Software Maintenance and Reengineering (CSMR'98)*, pages 104–112. IEEE, 1998.

[37] R. Kazman, L. Bass, and M. Klein. The essential components of software architecture design and analysis. *Journal of Systems and Software*, 79(8):1207–1216, 2006.

[38] R. Kazman, L. Bass, M. Klein, T. Lattanze, and L. M. Northrop. A basis for analyzing software architecture analysis methods. *Software Quality Journal*, 13(4):329–355, 2005.

[39] B. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. ST5 5BG, UK Version 2.3, School of Computer Science and Mathematics, Keele University, UK, 2007.

[40] J. Lakos. *Large-scale C++ software design*. Addison-Wesley, 1996.

[41] N. Lassing, P. Bengtsson, H. van Vliet, and J. Bosch. Experiences with alma: architecture-level modifiability analysis. *J. Syst. Softw.*, 61:47–57, March 2002.

[42] N. Lassing, D. Rijsenbrij, and H. van Vliet. How well can we predict changes at architecture design time? *J. Syst. Softw.*, 65:141–153, February 2003.

[43] X. Liu and Q. Wang. Study on application of a quantitative evaluation approach for software architecture adaptability. In *5th Int. Conf. on Quality Software, 2005 (QSIC'05)*, pages 265–272. IEEE, 2006.

[44] P. Maheshwari and A. Teoh. Supporting atam with a collaborative web-based software architecture evaluation tool. *Sci. Comput. Program.*, 57:109–128, July 2005.

[45] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proc. Int. Conf. on Softw. Maitenance (ICSM'99)*, ICSM '99, pages 50–, Washington, DC, USA, 1999. IEEE Computer Society.

[46] A. Martens, H. Koziolek, S. Becker, and R. H. Reussner. Automatically improve software models for performance, reliability and cost using genetic algorithms. In *Proc. 1st Joint WOSP/SIPEW Int. Conf. on Perf. Eng. (ICPE'10)*, pages 105–116, New York, NY, USA, 2010. ACM.

[47] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[48] M. Matinlassi. Evaluating the portability and maintainability of software product family architecture: terminal software case study. In *4th Working IEEE/IFIP Conf. on Software Architecture (WICSA'04)*, pages 295–298. IEEE, 2004.

[49] T. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, pages 308–320, 1976.

[50] T. Mens, J. Magee, and B. Rumpe. Evolving software architecture descriptions of critical systems. *Computer*, 43:42–48, May 2010.

[51] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Trans. Softw. Eng.*, 32:193–208, March 2006.

[52] F. Olumofin and V. Misic. A holistic architecture assessment method for software product lines. *Information and Software Technology*, 49(4):309–323, 2007.

[53] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[54] D. Paulish and L. Bass. *Architecture-centric software project management: A practical guide*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2001.

[55] N. Rauch, E. Kuhn, and H. Friedrich. Index-based Process and Software Quality Control in Agile Development Projects. http://goo.gl/RxNXJ, 2008.

[56] R. Reussner, I. Poernomo, and H. Schmidt. Reasoning about software architectures with contractually specified components. In *Component-Based Software Quality*, volume 2693 of *LNCS*, pages 287–325. Springer, 2003.

[57] M. Riaz, E. Mendes, and E. Tempero. A systematic review of software maintainability prediction and metrics. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 367–377, Washington, DC, USA, 2009. IEEE Computer Society.

[58] S. Roock and M. Lippert. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. John Wiley & Sons, 2005.

[59] B. Roy and T. Graham. Methods for Evaluating Software Architecture: A Survey. Technical Report 545, Queen's University at Kingston, Ontario, Canada, Kingston, 2008.

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

[61] R. S. Sangwan, P. Vercellone-Smith, and P. A. Laplante. Structural epochs in the complexity of software over time. *IEEE Softw.*, 25:66–73, July 2008.

[62] C. Sant'Anna, E. Figueiredo, A. Garcia, and C. Lucena. On the modularity of software architectures: A concern-driven measurement framework. In F. Oquendo, editor, *Software Architecture*, volume 4758 of *Lecture Notes in Computer Science*, pages 207–224. Springer Berlin / Heidelberg, 2007.

[63] S. Sarkar, A. C. Kak, and G. M. Rama. Metrics for measuring the quality of modularization of large-scale object-oriented software. *IEEE Trans. Softw. Eng.*, 34:700–720, September 2008.

[64] S. Sarkar, G. M. Rama, and A. C. Kak. Api-based and information-theoretic metrics for measuring the quality of software modularization. *IEEE Trans. Softw. Eng.*, 33:14–32, January 2007.

[65] S. Sarkar, S. Ramachandran, G. Kumar, M. Iyengar, K. Rangarajan, and S. Sivagnanam. Modularization of a large-scale business application: A case study. *IEEE Software*, pages 28–35, 2009.

[66] N. F. Schneidewind. The state of software maintenance. *IEEE Trans. Softw. Eng.*, 13:303–310, March 1987.

[67] K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant'Anna. From retrospect to prospect: Assessing modularity and stability from software architecture. In *Proc. IEEE/IFIP Working Int. Conf. on Software Architecture (WICSA'09)*, pages 269–272. IEEE, 2009.

[68] Y. Shen and N. Madhavjim. ESDM-A Method for Developing Evolutionary Scenarios for Analysing the Impact of Historical Changes on Architectural Elements. In *Proc. 22nd IEEE Int. Conf. on Software Maintenance (ICSM'06)*, pages 45–54. IEEE, 2006.

[69] J. Stammel and R. Reussner. KAMP: Karlsruhe architectural maintainability prediction. In *Proc. 1st Workshop des GI-Arbeitskreises Langlebige Softwaresysteme (L2S2)*, pages 87–98. Citeseer, 2009.

[70] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Syst. J.*, 13:115–139, June 1974.

[71] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, 35:705–754, July 2005.

[72] P. Tarvainen. Adaptability evaluation of software architectures; a case study. In *Proc. 31st Int. Conf. on Computer Software and Applications Conference (COMPSAC'07)*, volume 2, pages 579–586. IEEE, 2007.

[73] R. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing, 2009.

[74] E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, 1979.