Measuring Architecture Sustainability

Heiko Koziolek, Dominik Domis, Thomas Goldschmidt, Philipp Vorst
ABB Corporate Research Germany, Industrial Software Systems Program
Contact: heiko.koziolek@de.abb.com

Abstract

Many aspects influence the economic sustainability of a software architecture, such as modularization, technology choices, and design decisions facilitating evolutionary changes. Relevant information is spread across many artifacts and the software architects. An approach to sustainability measurement focusing on a single artifact or perspective is likely to neglect important factors. At ABB, we are measuring and tracking the architecture sustainability of a large-scale industrial control system under development. We have applied a multi-perspective approach called MORPHOSIS. It focuses on requirements, architecture design, and source code. It includes evolution scenario analysis, architecture compliance checks, and tracking of architecture-level code metrics. This article reports our experiences from tracking the selected sustainability measures for two years.

Keywords: Software Architectures, Software Quality/SQA, Maintenance measurement, Product metrics

Introduction

In many application domains software systems are maintained and evolved over decades. For example, in industrial automation, longevity is a necessity as industrial devices have long life-cycles. Software architectures are a major factor for sustainability (i.e., the economical longevity) of large-scale systems, influencing maintenance and evolution costs vastly. It is thus desirable to measure the sustainability of a software architecture in order to derive refactoring actions and avoid poor evolution decisions.

Information concerning sustainability is spread across requirements, architecture design documents, technology choices, sources code, system context, and the implicit knowledge of the software architects. Many aspects influence economic sustainability, such as design decisions facilitating evolutionary changes, adherence of good modularization practices, and technology choices. An approach focusing on a single artifact or perspective is likely to neglect important factors.

At ABB, we are measuring and tracking the architecture sustainability of a large-scale industrial control system currently under development. It is a distributed system based on Microsoft technologies and includes a layered architecture. A former version of the system grew to several million lines of code and suffered from architecture erosion and high maintenance costs [1]. We use a multi-perspective approach called MORPHOSIS [2], which intends to avoid such a situation. It focuses on requirements, architecture design, and source code. It includes evolution scenario analysis, scoring of technology choices, architecture compliance checks, and tracking of architecture-level code metrics. This article reports our experiences from tracking the selected sustainability measures for two years.

Perspectives of Architecture Sustainability

Software architecture sustainability should be viewed from multiple perspectives including change-prone requirements, technology choices, architecture erosion, and modularization best practices.

Volatile **requirements** disregarded in architecting may lead to poor design decisions. As stated by Parnas [3], software modules should be designed around potentially changing requirements to limit ripple effects, i.e. changes to modules triggering changes to other modules. This may allow renovating a system by localized module replacement. For example, security requirements are frequently changing for industrial software systems [4]. Encapsulating security-related concerns into modules can limit according ripple effects. Thus, one measure for architecture sustainability is the degree to which an architecture is prepared for the change of volatile requirements [5].

Technology choices during design are another important dimension of architecture sustainability. The chosen frameworks, third-party components, and programming languages are significant sustainability factors. If built on a fashionable but transient technology whose support is stopped soon, a system may later need expensive renovations. For example, in the nineties ABB incorporated Visual Basic 6 into several products, which was subsequently discontinued by Microsoft, and required costly replacements. A second measure for architecture sustainability is thus the expected longevity of included technologies [6].

When being evolved, long-living software system frequently suffer from **architecture erosion**. Then, the implementation violates architectural constraints, such as prescribed module dependencies or separation of concerns. This situation can render a whole architecture design invalid as it becomes economically infeasible to replace modules. The architecture is no longer useful to understand a system on a higher abstraction level. For example, architectural analyses at ABB [1] revealed layering violations and unwanted component dependencies in long-term evolved, large-scale industrial software, which prompted costly architecture refactoring. Thus, a third measure for architecture sustainability is the degree of architecture erosion.

Finally, if a system implementation starts to violate **best practices for software modularization**, this indicates decreasing architecture sustainability. These best practices for example concern acyclic dependencies, layering organization, API usage, encapsulation, concern dispersion, and testability [7]. While architecture erosion usually refers to violations of the explicitly prescribed architecture, such best practices are sometimes not explicitly encoded in architecture documents. Numerous architecture-level code metrics have been proposed (see sidebar “Architecture-level Code Metrics”). A fourth measure of architecture sustainability is thus the degree to which a software system is within desirable thresholds of such architecture-level code metrics.

Architecture-level Code Metrics

Most work in the area of architecture-level metrics derived from the module concept described by Parnas[1] and the notions of coupling and cohesion[2]. We categorized more than 40 architecture-level code metrics in another article[3].

For example, Lakos[4] defined a metric called Cumulative Component Dependency (CCD), which is the sum of required dependencies by a component within a subsystem: CCD provides a numerical measure of the module coupling in a system, where low values represent better maintainability and testability. Derived metrics are the average component dependency (ACD) and the normalized CCD (NCCD). They can be determined by tools such as SonarJ or STAN.

Martin[5] defined several metrics for software packages, i.e., groups of related classes (e.g., Java packages, C++ projects). These include afferent coupling, efferent coupling, abstractness, instability,

distance from main sequence, and package dependency cycles. For example, the distance from main sequence measures how usable and maintainable a module is. Several tools support these metrics (e.g., CppDepend, STAN).
Sarkar et al.[6] created a set of twelve API-based and information-theoretic metrics for modularization quality. The metrics rely on the definition of APIs between modules, module size thresholds, and concept term maps.
Sangwan et al.[7] introduced 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 package or module level.
Bouwers et al.[8] proposed a metric called Component Balance, which combines the number of components and their relative sizes.

References

1. D. Parnas. "On the criteria to be used in decomposing systems into modules". Communications of the ACM, 15(12):1053–1058, 1972.
2. W. P. Stevens, G. J. Myers, and L. L. Constantine. "Structured design". IBM Syst. J., 13:115–139, 1974.
3. H. Koziolek. "Sustainability evaluation of software architectures: a systematic review". In Proc. 7th ACM SIGSOFT Int. Conference on the Quality of Software Architectures (QoSA'11), pp. 3-12, 2011.
4. J. Lakos. "Large-scale C++ software design". Addison-Wesley, 1996.
5. R. C. Martin. "Agile Software Development: Principles, Patterns, and Practices". Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.
6. 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.
7. 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.
8. E. Bouwers, J. Correia, A. van Deursen, and J.Visser. "Quantifying the analyzability of software architectures", Proc. 9th Working IEEE/IFIP Conference on Software Architecture (WICSA'11), IEEE Computer Society.

The measures for architecture sustainability described so far mainly refer to requirements, architecture design, and source code. There are further indirect measures for architecture sustainability, such as the documentation quality and the development process maturity. Another important factor is the sustainability of the development organization, after which a software architecture is often modeled. Organizational changes may compromise architecture sustainability if, for example, teams working on specific modules are restructured. However, these indirect and organizational measures for architecture sustainability are out of the scope of this article.

For the analyzed industrial control system by ABB, we first assessed volatile requirements and technology choices with an evolution scenario analysis. Then we prepared for architecture erosion by setting up architecture compliance checks. Finally we set up a metrics dashboard to track modularization best practices in the source code. These steps are detailed in the following.

Evolution Scenario Analysis

To analyze the architecture's sustainability regarding volatile requirements and technology choices, we created evolution scenarios at the architecture level, i.e., changes regarding important interfaces and components in the system. Such scenarios can be assessed for their impact on the architecture

and ranked according to their likelihood to give a measure on the architecture sustainability. It allows identifying sustainability sensitivity points and preparing architectural mitigation measures, such as creating abstraction layers to decouple the system from expected changes.

To elicit scenarios, we applied an extended version of the ALMA method [5]. Another popular method is ATAM [8], which we ruled out since we were not able to conduct a stakeholder workshop. In our case, we first interviewed eight domain experts for top-down scenario elicitation according to ALMA. The domain experts worked on similar products and had a background in software engineering. Besides their experiences from former systems, they also pointed to business trends and technical trends which could require changes in the system under analysis. In total, this analysis yielded short descriptions of 31 evolution scenarios that were categorized according to multiple criteria, such as the kind of change (e.g., 13 perfective and 18 adaptive scenarios), the source of the change, and the potentially affected subsystems. We also requested an initial ranking from the domain experts on the likelihood and expected impact of each scenario to aid prioritization.

For bottom-up scenario elicitation, we interviewed managers, architects, and developers in the development unit. We analyzed requirements specification, architecture documentation, and initial parts of the source code. For example, we traced specific evolution scenarios to requirements to enable impact analysis in case the requirement would be changed. Additionally, we used dependency analyzer tools, such as NDepend[1] and CppDepend, to trace evolution scenarios to the code and reveal potential ripple effects. Thus, for several scenarios we could state the directly affected amount of source code and the indirectly affected modules due to dependencies. We documented numerous criteria per evolution scenario, such as change history, architecture sensitivity points, occurrence probability, and potential risks involved in the change (details in [2]).

After the top-down and bottom-up scenario elicitation, seven evolution scenarios had a medium to high assessment for the combination of impact and occurrence probability over the next five years. We analyzed these seven in detail. Table 1 lists these scenarios anonymously with a condensed assessment statement. A more detailed scoring was documented in our internal assessment report.

For example, in 2011, the operating system implementation for one of the subsystems was expected to change in the near future. This change was already foreseen: A portability layer had been added to affected subsystem. However, by dependency analysis with CppDepend we found that some parts of the code already circumvented the layer in 2011. We recommended automatically checking the compliance of the code to the portability layer in the build process.

Other scenarios concerned changes in system workload, hardware components, platforms, APIs, and GUI. Although not all of these scenarios are highly likely to occur, some of the less likely ones would considerably impact the system. Several of these scenarios were extrapolated from the evolution histories of similar former systems.

[1] http://www.ndepend.com

Evolution Scenario	Assessment 2011	Assessment 2012	Assessment 2013
Change of operating system implementation	Highly likely, but limited impact (preparation done)	Change occurred, limited impact	No further changes expected in the near future
Increase of system workload	Likely, high impact	Internal testing done, impact on architecture lower than expected	Still likely, but no additional measures implemented
Integration of a new hardware component	Likely, limited impact	Change occurred, project started	Under development
Change of the deployment platform	Scenario recognized, but no immediate preparation desired	Scenario refined, likelihood increased	Large research project started
Change of a client-facing plug-in interface	Unlikely, but impact would be high	Did not occur, still possible	Did not occur, still possible
Change of the third-party middleware implementation	Medium likelihood, limited impact	Did not occur, still possible	Did not occur, still possible
GUI framework change	Unlikely, but impact would be high	Research project executed	Planned as an extension

Table 1: Periodic evolution scenario assessment as a measure for architecture sustainability

We reassessed the evolution scenarios from 2011 again in 2012 and 2013. We found that two of the 2011 scenarios actually occurred in 2012. The anticipated changes of three other scenarios were being mitigated by corresponding test, refinement, and research activities (see Table 1). The remaining evolution scenarios did not occur, but are still possible. We were not able to identify additional evolution scenarios to the ones from 2011, suggesting that we were successful in covering the most important potential architectural changes to the system.

Architecture Compliance Checking

As a measure to prevent architecture erosion, we decided to check the system's implementation against the prescribed architecture. The system under study has a layered architecture, where only certain dependencies between modules are allowed. Violated dependency rules can severely increase maintenance costs, as they negate the benefits of modularization and complicate independent module compilability, extensibility, and testability [7].

Dependency violations usually stem from developers who are not aware of the prescribed architecture or work under time pressure. The violations do not have an immediate impact on the system functionality and are therefore sometimes neglected. Additionally, the architecture documentation may be out of sync with the implementation. There are several tools for architecture compliance checking and enforcement (e.g., SARTool, Bauhaus, DiscoTect, Symphony, Lattix) which can check dependency rules, e.g. at build time. This way, violations show up early, do not accumulate and can be fixed during regular refactoring.

For the system under study, we based architecture compliance checks on given dependency rules defined in UML diagrams. The architects had specified allowed dependencies between layers and modules and for example did not permit lower-level layers to call upper-level layers. We specified corresponding dependency rules, e.g. between .NET assemblies, in the declarative Code Query

Language (CQL) from the tools NDepend (for C#) and CppDepend (C++). Based on fact databases extracted from source code, these tools produce dependency graphs and design structure matrices (DSM) enabling developers to visually check for violations. We integrated the rule checks into the system's weekly build process.

The tools initially identified multiple violations of the layering structure in the source code. One violation was an undesired dependency from a lower to an upper architecture layer. Developers had assigned classes to wrong modules and were then able to resolve the violation. Additionally, classes from an upper layer directly accessed platform modules, although the architecture prescribed routing all platform calls through an intermediate layer. We also experienced dependency violations that resulted from an outdated architecture model after source code redesigns.

Architecture Metrics Tracking

To complement evolution scenario analysis and architecture compliance checking, we created a dashboard to track best practices for modularization using architecture-level code metrics. Clean modularization reduces system complexity, allows faster system understanding, and enables easier replacement of modules during system evolution. It thus contributes to a system's longevity [7]. We applied the Goal/Question/Metric (GQM) approach (details in [2]), broke down sustainability into modifiability, reusability/layering, modularity, and testability, and selected the subsequent metrics that covered these aspects:

- Cyclic Dependency Index (to improve modifiability): percentage of modules with namespace dependency cycles
- Well-sized Methods Index (to improve modifiability): percentage of well-sized (<30 lines of code) methods in the code base.
- Distance from Main Sequence (to improve modifiability): fraction of modules that are either concrete and stable (i.e., difficult to maintain) or abstract and unstable (i.e., difficult to use)
- Module Interaction Stability Index (to improve layering): percentage of modules that depend on modules with higher instability
- Layer Organization Index (to improve layering): extent to which module dependencies skip adjacent layers
- Normalized Testability Index (to make testing more efficient): extent to which modules are independently testable
- Module Interaction Index (to improve API usage): effectiveness of how a module's API functions are used
- API Function Usage Index (to improve API usage): averaged percentage of the ratio of non-API functions to API functions
- Module Size Uniformity Index (to limit the sizes of modules): extent of heterogeneously sized modules
- Module Size Boundedness Index (to limit the sizes of modules): extent of module sizes differing from a maximum size threshold
- State Access Violation Index (to reduce internal variable usage): extent to which state variables are accessed directly across module boundaries
- Association-Induced Coupling (to reduce internal variable usage): extent of coupling between modules due to class association

These metrics are formally defined in literature (e.g., [9],[10]) but have not been widely used in practice. All metrics are normalized between zero and one, where one is the best achievable value. Some of the metrics conflict with each other, i.e., blindly optimizing for one metric leads to the decrease of the other metric. For instance, optimizing Module Size Uniformity with only two large modules of similar sizes and poor inner structures yields worse Module Size Boundedness.

There was no tool available providing these metrics. We thus implemented a calculation tool and reporting dashboard. NDepend and CppDepend provide basic statistics, such as the lines of code per module, classes, and methods as well as raw module dependencies, stability, and abstractness indices of modules. We added several custom CQL queries to provide intermediate results for calculating the metrics. Our tool chain imports the XML output into an Excel workbook. Spreadsheets combine the advantages of user familiarity, existing data import functions (e.g., from Team Foundation Server), and reporting facilities (e.g., Kiviat diagrams). We calculate the architecture-level metrics with Excel macros.

By the time of writing, we have tracked the architecture-level metrics for almost two years. Figure 1 exemplarily shows trends of selected metrics for the industrial system under study. We only included the most interesting metrics in the figure. The omitted metrics did not reveal any interesting results for the system under study. The Module Size Uniformity Index (MSUI, Figure 1, upper left) increased slowly over the two years, similarly to the observations of other long-living systems [1], [7]. For the system under study, several modules had been defined in the beginning as code stubs thus had small sizes lowering MSUI. The MSUI increased over time because these stubs were filled with implementations. While the MSUI has not yet required restructuring actions, this index can become more important as the system grows and enters the maintenance phase.

The Module Interaction Stability Index (MISI) as well as the Cyclic Dependency Index (CDI) show only limited variations (Figure 1, upper right). These indices are computed based on dependencies between higher-level code modules, which do not change as often as module sizes. The MISI increased after approximately half the measurement period, when one of the modules was removed from the system. This altered the interaction stability. After some more time the index again decreased when a new dependency was introduced, which is negative from a structural viewpoint. The CDI increased after half the measurement period. At that point, approximately 1000 lines of code and one cyclic module dependency were removed from the code base. Later a new cyclic dependency was introduced, lowering the CDI again. However, both the MISI and the CDI had comparably high values.

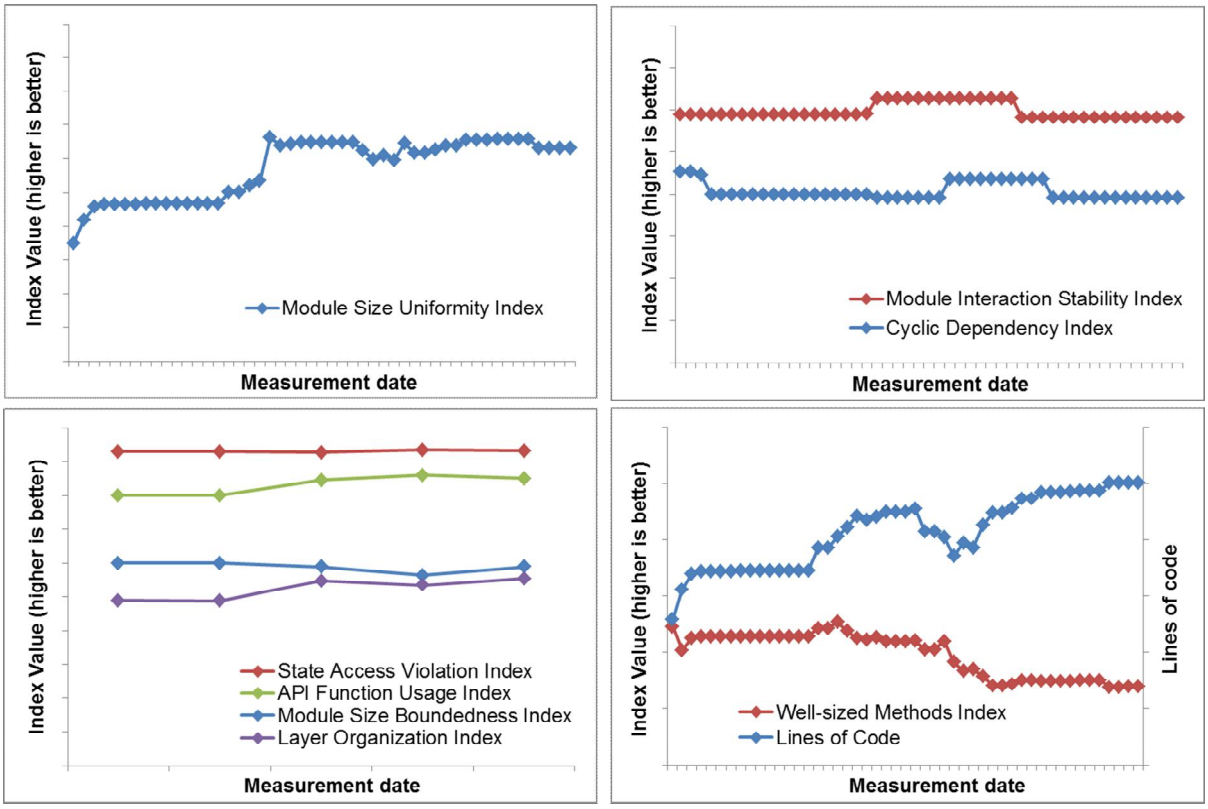

Figure 1: Trends of architecture-level code metrics from the case study as a measure for architecture sustainability

On the lower left part of Figure 1, the trends of four additional architecture-level metrics are visualized. The State Access Violation Index shows high and stable values as such violations were rare in the code base. A slightly increasing trend is visible for the API Function Usage Index, which could again be traced back to the progressing state of the implementation. The Module Size Boundedness Index was rather stable with a decreasing trend. Finally, the Layer Organization Index penalizes cyclic dependencies over layer boundaries and showed a positive trend.

The lower right part of Figure 1 shows another notable trend on code design level. The trend for lines of code of one subsystem is shown in blue. Figure 1 also depicts an index measuring the percentage of well-sized methods in the code base. The trend for the Well-sized Methods Index is decreasing. The developers introduced multiple large methods into the system and did not refactor them. This indicates a negative trend on code design level affecting sustainability.

Lessons Learned

When reassessing the evolution scenarios after one and two years, we found that some of them had actually occurred, while the scenarios that did not occur were still valid for the near future. The probabilities and impacts of some scenarios needed to be updated due to changing assumptions. Contrary to former studies [11], we did not identify new likely evolution scenarios. While some components had been replaced or extended to cover new features, the overall architecture remained stable. Therefore, we learned that the scenario analysis findings were not as volatile as in former studies and that the invested efforts for deeper analysis and requirements tracing paid off. The architects acknowledged that the analysis gave them a frame of reference for the technical risks they previously considered informally.

Checking the dependency rules for architecture compliance in 2012 and 2013, we did not find new dependency violations in the source code. The developers now checked architecture compliance regularly and fixed according problems after code reviews. The architecture compliance checks created a higher awareness for the architecture specification. However, the maintenance of the dependency rules remains a challenge. Currently, new dependency rules need to be specified manually in the CQL language in addition to the specification in the UML model, which creates overhead. It is useful to automate this step to save maintenance efforts.

Setting up our measuring dashboard for the system under study yielded some interesting effects. Our early reports triggered the developers to schedule respective refactoring sessions. A couple of months later the developers had cleaned up the source code for a number of class-level metrics. The developers had restructured all classes with a cyclomatic complexity exceeding 20 or more than 20 methods and additionally assigned all classes to namespaces. Thus, the code quality improved on design level simply because a measurement instrument was in place.

Although the architecture-level code metrics have not yet led to major restructurings, they are regularly monitored by managers and architects and support architecture review meetings. We learned that it is impractical to optimize each metric to the optimal value of one. Thus, there is still a need to define target thresholds for the metrics. Currently, the stakeholders rather use the metrics to show relative sustainability improvement or decline. The importance of the architecture-level metrics is expected to increase once the system enters maintenance and evolves.

Our integrated approach of scenario analysis, compliance checks, and metrics tracking yielded several synergies between the activities. During scenario analysis, we used the same dependency checking tools as for the compliance checks to measure the impact of an evolution scenario and to identify ripple effects. The evolution scenario analysis findings were additionally used to prioritize the tracking of the dependency violations so that more critical sensitivity points received more attention. The dependency rules for compliance checking were condensed into an architectural metric and integrated into the metrics dashboard, so that they are considered in the context of other metrics. Overall, the approach had a perceived good cost-benefit ratio: Scenarios analyses and architecture compliance checks could be set up in a matter of several days. The effort for establishing a metrics framework was considerably higher (4-5 person months), but the metrics and tools can be reused for other systems.

Conclusions

The takeaway message from our study is that architecture sustainability needs to be measured and controlled from multiple perspectives. This includes mining different sources of architecture information, ranging from humans to highly automated code reporting. Focusing on scenario analysis would not prepare for architecture erosion, while focusing on architecture-level code metrics would neglect changing requirements and technologies.

Our MORPHOSIS method for measuring architecture sustainability can be applied with manageable efforts. Still, the evolution of a complex software system can be planned only to a limited extent due to rapid changes in the IT world. Besides methods focusing on technical factors, also organizational and human factors are important.

For short-term future work, we will refine MORPHOSIS by applying it to other ABB systems, which may improve the scenario analysis templates and lead to a revised set of architectural metrics. We

will also compare our findings on architecture metrics tracking to current empirical studies to gain an overall understanding of their usefulness [12]. Longitudinal studies are required that correlate software maintenance costs with the architectural metrics to enable quantitative cost-benefit analyses. Another direction is to explore the augmentation of MORPHOSIS with constructive methods for planning software architecture evolution [13].

References

[1] T. Kettu, E. Kruse, M. Larsson, and G. Mustapic, "Using Architecture Analysis to Evolve Complex Industrial Systems," in *Proc. Workshop on Architecting Dependable Systems V*, 2008, vol. 5135, pp. 326–341.

[2] H. Koziolek, D. Domis, T. Goldschmidt, P. Vorst, and R. J. Weiss, "MORPHOSIS: A Lightweight Method Facilitating Sustainable Software Architectures," in *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, 2012, pp. 253–257.

[3] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

[4] D. Dzung, M. Naedele, T. Von Hoff, and M. Crevatin, "Security for industrial communication systems," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1152–1177, 2005.

[5] 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, Jan. 2004.

[6] A. Jansen, A. Wall, and R. Weiss, "TechSuRe - A Method for Assessing Technology Sustainability in Long Lived Software Intensive Systems," in *37th EUROMICRO Conference on Software Engineering and Advanced Applications*, 2011, pp. 426–434.

[7] S. Sarkar, S. Ramachandran, G. S. Kumar, M. K. Iyengar, K. Rangarajan, and S. Sivagnanam, "Modularization of a Large-Scale Business Application: A Case Study," *IEEE Software*, vol. 26, no. 2, pp. 28–35, Mar. 2009.

[8] P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional, 2001, p. 368.

[9] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall.

[10] 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.*, vol. 33, no. 1, pp. 14–32, 2007.

[11] N. Lassing, D. Rijsenbrij, and H. van Vliet, "How well can we predict changes at architecture design time?," *Journal of Systems and Software*, vol. 65, no. 2, pp. 141–153, Feb. 2003.

[12] E. Bouwers, A. van Deursen, and J. Visser, "Evaluating usefulness of software metrics: an industrial experience report," in *Proc. International Conference on Software Engineering (ICSE 2013)*, 2013, pp. 921–930.

[13] D. Garlan, J. M. Barnes, B. Schmerl, and O. Celiku, “Evolution styles: Foundations and tool support for software architecture evolution,” in *Proc. Joint Working IEEE/IFIP Conference on Software Architecture (WICSA2009)*, 2009, pp. 131–140.

Authors

Heiko Koziolek is a Principal Scientist with the Industrial Software Systems program of ABB Corporate Research, Germany. He is ABB’s Global Research Area Coordinator for Sustainable Software Architectures. His research interests include performance engineering, software architecture, model-driven software development and empirical software engineering. Koziolek has a PhD in computer science from the University of Oldenburg, Germany.

Dominik Domis is a Scientist at the ABB Corporate Research Center in Germany. He focuses on architectural approaches for the integration and systematic reuse of industrial software systems. Dominik holds a PhD in computer science from the University of Kaiserslautern, Germany.

Thomas Goldschmidt is a Principal Scientist at the ABB Corporate Research Center in Germany. He focuses on domain-specific language engineering and software architectures in the automation domain. Thomas holds a PhD in computer science from the Karlsruhe Institute of Technology, Germany.

Philipp Vorst is a Scientist with ABB Corporate Research in Germany. His research interests include software architecture methods with applications in automation. Philipp holds a PhD degree in computer science from the University of Tübingen, Germany.