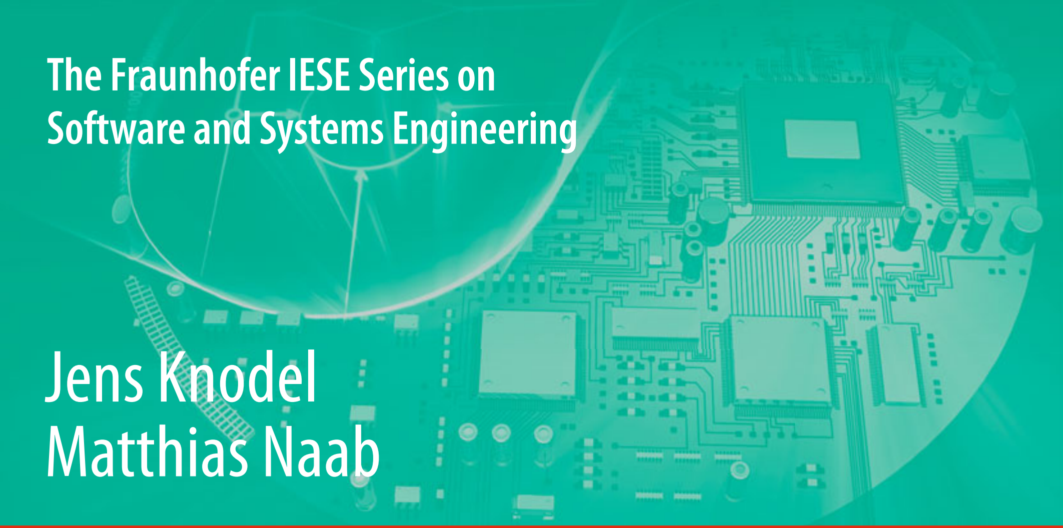
The Fraunhofer IESE Series on Software and Systems Engineering



Jens Knodel
Matthias Naab

Pragmatic Evaluation of Software Architectures

Fraunhofer IESE

Springer

The Fraunhofer IESE Series on Software and Systems Engineering

Jens Knodel · Matthias Naab

Pragmatic Evaluation of Software Architectures

Springer

Jens Knodel
Fraunhofer IESE
Kaiserslautern
Germany

Matthias Naab
Fraunhofer IESE
Kaiserslautern
Germany

ISSN 2193-8199 ISSN 2193-8202 (electronic)
The Fraunhofer IESE Series on Software and Systems Engineering
ISBN 978-3-319-34176-7 ISBN 978-3-319-34177-4 (eBook)
DOI 10.1007/978-3-319-34177-4

Library of Congress Control Number: 2016939924

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG Switzerland

About this Series

Whereas software engineering has been a growing area in the field of computer science for many years, systems engineering has its roots in traditional engineering. On the one hand, we still see many challenges in both disciplines. On the other hand, we can observe a trend to build systems that combine software, microelectronic components, and mechanical parts. The integration of information systems and embedded systems leads to so-called smart ecosystems.

Software and systems engineering comprise many aspects and views. From a technical standpoint, they are concerned with individual techniques, methods, and tools, as well as with integrated development processes, architectural issues, quality management and improvement, and certification. In addition, they are also concerned with organizational, business, and human views. Software and systems engineering treat development activities as steps in a continuous evolution over time and space.

Software and systems are developed by humans, so the effects of applying techniques, methods, and tools cannot be determined independent of context. A thorough understanding of their effects in different organizational and technical contexts is essential if these effects are to be predictable and repeatable under varying conditions. Such process–product effects are best determined empirically. Empirical engineering develops the basic methodology for conducting empirical studies and uses it to advance the understanding for the effects of various engineering approaches.

The series presents engineering-style methods and techniques that foster the development of systems that are reliable in every aspect. All the books in the series emphasize the quick delivery of state-of-the-art results and empirical proof from academic research to industrial practitioners and students. Their presentation style is designed to enable the reader to quickly grasp both the essentials of a methodology and how to apply it successfully.

Foreword

Many activities in software engineering benefit from explicit documentation of the software architecture. Such activities range from cost estimation via make-or-buy decisions to implementing software and deployment, or even the establishment of a software product line or other forms of planned reuse.

However, in many projects, the documentation of software architectures is considered as a burden and is thus avoided.

I think one of the major reasons for this reluctance to invest in software architecture documentation is the absence of feedback: Do I have a good architectural design? Is the architecture documented well enough for my project? As a result, developers are unsure whether the architecture is useful altogether. In fact, the problem of the lack of feedback in architectural design exists regardless of the documentation approach. When attempting to document the architecture, it just becomes more obvious. Anyway, the effects of bad architectural design are also well known: shift of risks to the later phases of software development, lowered productivity, and, most often, unsatisfactory quality of products. The latter, in particular, is not very surprising, as the software architecture is for many IT systems the major factor influencing the perceived quality for customers and developers alike.

Therefore, the role of early evaluation of software architectures is well understood in research. In fact, there exists a large body of proposals on how to evaluate software architectures for various concerns. Most approaches are informal, but automated formal approaches also exist. The benefits of informal approaches are a certain flexibility regarding acceptable types of architectural documentation and a broad range of potential quality impacts of the architecture. However, they require manual effort and depend to a considerable extent on the experience of the evaluation team. In contrast, formalized automated approaches require specific architectural models, but the evaluation is "objective," as it is automated. However, only specific quality concerns are answered.

This book addresses the very important field of software architecture evaluations. It is important for two reasons: First, it provides practical means for evaluating software architectures, which is relevant to the reasons given above. Second, it bundles experiences and how-to guidelines for the manual approaches, which particularly benefit from exactly such experiences and direct guidelines. This book is an invaluable help in bundling the existing body of knowledge and enriching it strongly with real-world project expertise of the authors. It is not only a major step toward the practical applicability of software architecture evaluation but also helps to find good designs right at the beginning, as it helps to avoid design errors in all phases of a software system's lifecycle.

I wish the readers many insights from the abundant experience of the authors and a lot of fun when applying the knowledge gained.

Prof. Dr. Ralf H. Reussner
Chair Software Design and Quality, Karlsruhe Institute of Technology (KIT) and
Executive Director, Forschungszentrum Informatik (FZI), Karlsruhe, Germany

Preface

What is the Point of This Book?

Software architecture evaluation is a powerful means to mitigate risks when making decisions about software systems, when constructing them, when changing them, when considering (potential) improvements, or when reasoning about ways to migrate. Architecture evaluation is beneficial in such cases either by predicting properties of software systems before they are built or by answering questions about existing systems. Architecture evaluation is both effective and efficient: effective, as it is based on abstractions of the system under evaluation, and efficient, as it can always focus only on those facts that are relevant for answering the evaluation questions at hand.

Acknowledging the need for architecture evaluation does not necessarily mean that it has been adopted by practitioners. Although research has disseminated numerous publications about architecture evaluation, a pragmatic and practical guide on how to apply it in one's own context and benefit from it is still missing. With this book we want to share our lessons learned from evaluating software architectures. We do not aim at reinventing the wheel; rather, we present a consolidation of useful ideas from research and practice, adapting them in such a way as to make them applicable efficiently and effectively—in short, we take a pragmatic approach to evaluating software architectures. Where necessary, we will fill in gaps in existing approaches, in particular in the areas of scalability and applicability. Additionally, we aim at interrelating all aspects and techniques of architecture evaluation and creating an understandable and memorable overall picture. We will refer to examples of real architecture evaluation cases from our industrial practice (anonymized due to confidentiality reasons) and provide data on projects.

Why Read This Book?

"The most serious mistakes are not being made as a result of wrong answers. The truly dangerous thing is asking the wrong question."

Peter Ferdinand Drucker

We think that thorough and continuous architecting is the key to overall success in software engineering, and architecture evaluation is a crucial part of architecting. Asking the right questions and knowing about the right techniques to answer is crucial for applying architecture evaluations as a valuable and pragmatic means of technical risk management in software engineering.

To date (i.e., as of February 2016), we have conducted more than 75 architecture evaluation projects with industrial customers in the past decade. In each of these projects, at least one of the authors has been directly or indirectly involved as part of our jobs as architects and consultants at the Fraunhofer Institute for Experimental Software Engineering IESE. Fraunhofer IESE is an applied research institute for software engineering located in Kaiserslautern, Germany. These projects covered a large number of different types of systems, industries, organizational constellations, technologies, modeling and programming languages, context factors, and, of course, a whole spectrum of different evaluation results. Most importantly, we collected a lot of evaluation questions that were asked and operationalized them into actions.

"You can't control what you can't measure."

Tom DeMarco

"Everything that can be counted does not necessarily count; everything that counts cannot necessarily be counted."

Albert Einstein

While scientific literature on architecture evaluation approaches is available, the number of publications on practical experiences is rather limited. The contribution of this book consists of the presentation and packaging of our experiences together with context factors, empirical data, example cases, and lessons learned on mitigating the risk of change through architecture evaluation. Our approach for architecture evaluation (called RATE Rapid ArchiTecture Evaluation) has evolved and been proven successful in many projects over the past decade. We will provide an in-depth description on the ingredients of our approach, but will also tackle the field of architecture evaluation as a whole, as many of our insights and findings are independent of the approach.

After reading this book, the target audiences will be able to take their own steps in evaluating software architecture. By giving comprehensive answers to more than 100 typical questions (including questions we had, questions we heard, and questions our industrial partners had) and discussing more than 60 frequent mistakes and lessons learned, readers will take their first steps on ground paved by more than a decade of the authors' experiences.

Even more importantly, readers will learn how to interpret the results of an architecture evaluation. They will become aware of risks such as false conclusions, fiddling with data, and wrong lines of arguments in evaluations. Last but not least, readers will become confident in assessing quantitative measurement results and will learn when it is better to rely on qualitative expertise. In short, it is important to be aware what counts in architecture evaluation.

The target audience for the experience shared with this book includes both practitioners and researchers. On the one hand, we aim at encouraging practitioners to conduct architecture evaluations by showing the impact and lowering the hurdles to making first attempts on their own by providing clear guidelines, data, and examples. On the other hand, we aim at giving researchers insights into industrial architecture evaluations, which can serve as basis for guiding research in this area and may inspire future research directions.

How Should I Read This Book?

Our book is structured into three parts explaining the background of architecture evaluation, describing concrete evaluation techniques, and offering hints on how to successfully start and institutionalize architecture evaluation in practice.

Part I What is the Point of Architecture Evaluation?
Part II How to Evaluate Architectures Effectively and Efficiently?
Part III How to Apply Architecture Evaluation in Practice?

- For an **executive summary** on one page, please jump directly to question Q.117 on page 148.
- For **architects**, **developers**, or as learning material for **aspiring evaluators**, all three parts are highly relevant.
- For **managers**, mainly Part I and Part III are relevant.
- For a **quick start into an evaluation**, we recommend starting with question Q.117, reading Chaps. 3, 4, and 11, then proceeding directly to the respective checks you want to conduct in Chaps. 5–9.

In order to serve practitioners' needs in the best possible way, the entire book is structured along questions. These questions are organized in a hierarchical and uniform way to cover the big picture of architecture evaluation. For every topic, we also present frequently made mistakes we often encountered in practice and give hints on how to avoid them. Lists containing all questions, all frequently made mistakes, and the lessons learned serve to offer quick guidance for the reader.

In the following, we depict the recurring patterns that guide readers through the book. To a large extent, the chapters follow a uniform structure and are organized internally along questions. The questions are numbered consecutively. Frequent mistakes and lessons learned are visually highlighted, as shown in the following examples.

1.1 What is the point?
→ *This section summarizes the key points of the chapter's topic.*
1.2 How to do it effectively and efficiently?
→ *Here we present detailed descriptions and guidance.*
1.3 What mistakes are made frequently in practice?
→ *This section names typical pitfalls and points out how to avoid them.*

Q.001. Question

Frequently made mistake

→ Question Q.00X (*please read this question for more background information*)

Lesson learned

Acknowledgments

We would like to express our gratitude to our families, our colleagues (past and present), our customers, and the software architecture community.

We thank our families for their continuous support and encouragement in writing this book.

We had the opportunity to work and collaborate with great people at Fraunhofer IESE, where architecture evaluation has been a field of expertise for far more than 10 years now. We would like to thank our institute's directors Peter Liggesmeyer and Dieter Rombach for giving architecture evaluation a high degree of awareness and visibility. We also thank our past and present colleagues at Fraunhofer IESE for collaborating with us in software architecture projects and helping us to evolve our architecture evaluation method. The architecture evaluation projects making up this book would not have been possible without you.

We would like to thank Stefan Georg, Jens Heidrich, Thorsten Keuler, Torsten Lenhart, Mikael Lindvall, Dirk Muthig, Stefan Toth, Marcus Trapp, Carl Worms, and Stefan Zörner for their reviews and valuable feedback on earlier versions of this book. We are especially happy about the foreword by Ralf Reussner. Our special thanks go to Sonnhild Namingha for her excellent proofreading.

More than 75 customer projects were the key enabler of this book—as writing for practitioners is only possible when collaborating with practitioners. The collaboration with companies worldwide gave us lots of practical experience. Although we are not allowed to name them, we would like to thank our customers for their trust and the insights we were able to learn.

Over the years, we had many inspiring discussions with practitioners and researchers all over the world about software architecture in general and architecture evaluation in particular. We would like to thank them for stimulating ideas, shared insights, and their feedback to our ideas.

Finally, we would like to thank our publisher Springer and in particular Ralf Gerstner for his continuous support and his patience on the long journey of getting this book done—at last.

Contents

Table of Questions

Table of Frequent Mistakes and Lessons Learned

Part I
What Is the Point of Architecture Evaluation?

Why Architecture Evaluation? 1

Architecture evaluation is a valuable, useful, and worthwhile instrument for managing risks in software engineering. It provides confidence for decision-making at any time in the lifecycle of a software system. This chapter motivates architecture evaluation by explaining its role and its initiators, and by pointing out its benefits. Architecture evaluation requires investments, but saves time and money (if done properly) by preventing wrong or inadequate decisions.

1.1 What Is the Point?

Q.001. What Is Architecting?

Architecting, in its core essence, is the creative activity of software engineers making principal design decisions about a software system to be built or to be evolved. It translates concerns and drivers in the problem space into design decisions and solution concepts in the solution space.

Architecting is the process of coming up with some kind of solution for some kind of problem. Key to success for the architect is awareness of the problem to be solved and the design of an adequate solution considering given context factors. Consequently, any principal design decision made in advance is an investment in software engineering: the design decisions capture the output of reasoning based on facts, assumptions, and experiences and prescribe an implementation plan to satisfy the desired properties of the software system.

As a matter of fact, every software system has an architecture. The question is whether it is known or not, whether it has been designed proactively and intentionally or has emerged accidentally, and whether the design decisions have been

J. Knodel and M. Naab, *Pragmatic Evaluation of Software Architectures*,
The Fraunhofer IESE Series on Software and Systems Engineering,
DOI 10.1007/978-3-319-34177-4_1

explicitly documented or are only known and communicated implicitly. Therefore, architecting denotes the process of deliberately designing, using, communicating, and governing the architecture of a system.

Q.002. Why Invest in Software Architecture, Which Is Only an Auxiliary Construct in Software Engineering?

Because it saves time and money (if done properly) by preventing repeated trial and error.

The architecture offers an escape from the complexity trap. It systematically provides abstractions to enable coping with the complexity of modern software systems. Common software engineering practice is to use software architectures as the central vehicle for prediction, analysis, and governance over the whole lifecycle of a software system. The architecture enables the architects to reason about the pros and cons of design decisions and alternatives. Any principal design decision, and thus architecting itself as an activity, is always accompanied by the potential risk of being wrong or turning out to be extremely expensive.

The architecture enables making the right trade-offs between functionality and quality attributes. It assures that the technical aspects of realizing business goals are met by the underlying software system. Without investing effort into software architecture, it is unlikely that a software system with adequate quality (e.g., maintainability, performance, security, etc.) can be developed in time and on budget.

Q.003. What Is the Role of Architecture Evaluation in Software Engineering?

Architecture evaluation is a crucial assessment technique for mitigating the risks involved in decision-making. It contributes to the typical assessment questions depicted in Fig. 1.1. Note that architecture evaluation does not aim at answering the question “Is this design (or the decisions leading to it) good or bad?”. It rather evaluates whether the architecture is adequate to address the stakeholder concerns or not. However, having a well-designed architecture is just half the rent. The best-designed architecture does not help at all if it is not properly realized in the source code (the so-called drift between intended and implemented architecture). Hence, architecture evaluation further aims at making sure that the right design decisions have been manifested correctly in the implementation. The architecture is only of value if the implemented system is built as prescribed by the architecture; otherwise, all investments made into architecting become delusive and useless.

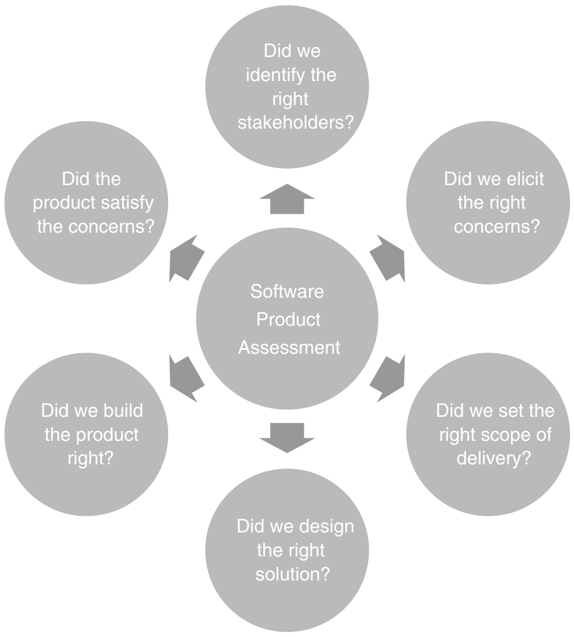

Did we identify the right stakeholders?
Did we elicit the right concerns?
Did we set the right scope of delivery?
Did we design the right solution?
Did we build the product right?
Did the product satisfy the concerns?
Software Product Assessment

Fig. 1.1 Software product assessment questions. © Fraunhofer IESE (2015)

Consequently, the mission of architecture evaluation is twofold: (1) to determine the quality of the (envisioned) software system and (2) to determine the quality of the (auxiliary) artifacts created during architecting or derived from the architecture.

Architecture evaluation supports informed decision-making: It creates awareness of risks and allows understanding potential trade-offs and side effects in decision-making. It aims at delivering quantitative and/or qualitative facts as input to decision-making. The architecture evaluation reveals design decisions (already made or under discussion) to address concrete stakeholders concerns and make them explicit. This enables challenging these design decisions and reasoning about their adequacy for addressing the stakeholder concerns. Risks associated with the design decisions can be made explicit, either to discuss mitigation measures or to find an alternative solution that fits the given context.

Architecture evaluation supports tracking of the realization of decisions made: Architecture evaluation creates awareness of drifts between the intended architecture (as designed) and the implemented architecture (as codified). The implemented architecture is typically not visible directly; rather, it is usually buried deeply in the source code and has to be extracted by means of reverse engineering or architecture reconstruction techniques. Since the architecture describes not only static artifacts of a system at development time, it might be necessary to mine for information from the system behavior at runtime. Risks associated with drift are detected and support the tracking of decisions made.

Q.004. What Are the Benefits of Architecture Evaluation?

The key benefit of architecture evaluation is to deliver guidance—for making business decisions, for making technology decisions, for controlling product quality, and last but not least for managing the evolution and migration of software systems. Any decision comes with the risk of being inadequate: thus, architecture evaluation aims at sustaining the investments made.

The obvious advantage of an architecture evaluation is, of course, the subsequent improvement of the architecture as such. Given the fact that the architecture (or the design decisions made) often determines the success or failure of arbitrary projects (e.g., software construction, software acquisition, software customization, retirement), it is clear that effort should be spent in every project on getting the architecture right. Architecture evaluation helps to avoid running into problems with respect to quantity, quality, budget, and/or time. This stresses the importance of architecture in general and architecture evaluation as a means for assuring its quality in particular. Consequently, the need for a systematic approach for architecture evaluation is obvious—and it is no surprise that architecture evaluation is considered one of the software engineering best practices in both research and industry. Numerous success stories in industry are proof of the crucial role of architecture evaluation in reaping benefits for risk management.

Besides, architecture evaluation provides other positive side effects: Its results can be the basis for improving the documentation of the software system under evaluation. Implemented violations of architecture decisions can be revealed as well and can be refactored afterwards. Furthermore, the organization's awareness for the architecture is raised (at least for the stakeholders directly or indirectly involved in the evaluation).

In summary, the advantages of architecture evaluation are significant. We believe it should be part of the quality assurance portfolio of any software development organization. However, the return on investment for evaluating architecture is only positive if the results can be achieved with adequate effort and if fast feedback is provided regarding the design decisions made with respect to the achievement of stakeholder concerns.

Q.005. Who Should Ask for an Architecture Evaluation?

There is no single stakeholder (see Fig. 1.2) who is typically interested in an architecture evaluation. Depending on the overall situation of the development and the evaluation goals, very different stakeholders may ask for an architecture evaluation.

Stakeholders who own an architecture evaluation may be from the company that develops the software system or from another company. In the same company, the owner of an architecture evaluation may be the architect or the development team itself, or higher development management. In particular, architects of a system should be interested in evaluating their own architectures. However, other

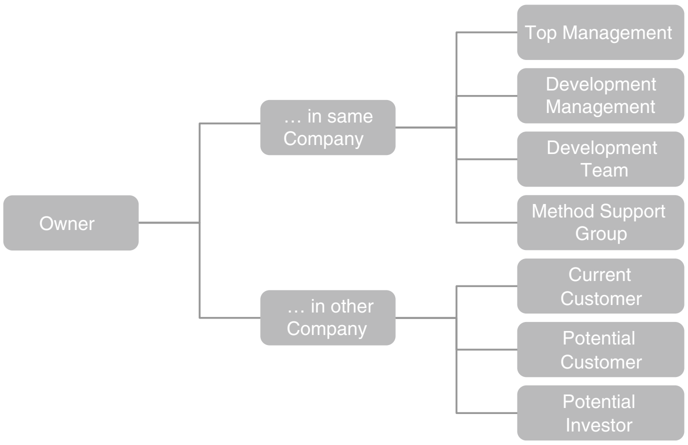


Fig. 1.2 Typical initiators of architecture evaluations. © Fraunhofer IESE (2014)

stakeholders may also be interested: for example, method support groups that intend to introduce and manifest architecture practices for improving overall software development quality. In addition, top management stakeholders may be interested in an architecture evaluation when they have to make decisions about software that will have a far-reaching impact.

On the other hand, stakeholders from external companies might also be interested in architecture evaluations. Potential customers or potential investors might be interested in the quality of a system and how adequate it is for the customer's purposes and requirements in the context of product selection and risk mitigation. Current customers have to make decisions about the future of acquired software systems: Is there enough confidence that the system will evolve in the right direction? What will happen if the customer's own context changes—for example, if its customer base strongly increases? Will the acquired system still be adequate or would it need to be replaced?

Q.006. Who Executes Architecture Evaluations?

Determining who is best suited for an architecture evaluation strongly depends on the goals of the evaluation.

An architecture evaluation can be done by very different people. Software architects who design a system should continuously challenge their own design decisions. Whenever they make a decision, they directly reflect on how well the tackled drivers are fulfilled by the decision they just made as well as those they made earlier. They should also reflect on how the current decision affects other drivers. These continuous evaluations can be augmented by the opinion of other people with direct involvement in decision-making, such as developers or product managers.

As architects are human beings, they tend to overlook deficiencies in their own work. Thus, it is often useful to get neutral people involved in the performance of an architecture evaluation at critical points. These neutral people may be from the same company: Internal auditors perform architecture evaluations as a (regular) quality engineering instrument. Internal auditors may be other architects, technology experts, or experienced developers. They typically have in-depth domain knowledge, are at least partially aware of the architecture/system history, and are potentially familiar with the overall vision and roadmap of the software system, its business goals, and the requirements to be satisfied in the next releases. Internal auditors may face the challenge of being low priority when requesting information or they may be ignored when summarizing the results and pinpointing identified risks.

However, sometimes it is even better to get a completely external opinion from external auditors. External auditors provide an objective evaluation delivered by qualified and experienced people. External auditors often apply a more formal approach for systematically eliciting and collecting information. They have predefined templates for audit presentations and reports that create higher visibility and awareness of the results. They avoid the "prophet in one's own country" symptom. In case of conflicts, external auditors can be neutral towards all involved parties.

Q.007. What Is the Return on Investment for Architecture Evaluations?

In our experience, architecture evaluation is a worthwhile investment that pays off. Any wrong or inadequate decision prevented or any risk avoided may save sufficient time and/or effort to pay for the evaluation. The cost for reversing fundamental or business-critical design decisions typically outweighs the investment required for architecture evaluation.

In more than 75 architecture evaluation projects in the past ten years in which we performed architecture evaluations or were part of a team of external auditors, we received—without any exception—positive feedback about the results and the output. In most cases, the cost for the auditing contract consumed less than one percent of the overall project budget. In most cases, the architecture evaluation results were delivered within a few weeks and were perceived as valuable input to decision-making by the sponsor of the evaluation. Architecture evaluation results are always expected to be delivered immediately as they deal with critical stakeholder concerns. This means that the "time to evaluate" is in many cases a critical factor for the overall success of the architecture evaluation.

In a nutshell, architecture evaluations serve to improve the architecture and mitigate risks. They identify problems, allow early and quick corrective reactions, and provide valuable input to decision-making for future-proof systems. Architecture evaluation results can be achieved rapidly with comparably low effort and provide fast feedback regarding stakeholder concerns. Avoiding just a single risk can already save more time and effort than what is required for conducting the architecture evaluation.

1.2 What Mistakes Are Frequently Made in Practice?

Having no idea about architecture evaluation at all.

If stakeholders do not know what an architecture evaluation is (i.e., if they are not aware of its benefits, procedures, and limitations), they are unable to use architecture evaluation as an instrument for quality assurance. This means risks may remain uncovered and crucial design decisions may be realized without critical reasoning and challenging of the decision.

This book and in particular Chap. 3 will allow the reader to learn about the key issues and characteristics of architecture evaluation. Stakeholders can use the detailed descriptions and sample cases used throughout this book to match them to their particular situations. We consider architecture evaluation to be a crucial instrument for assessing software products and believe that it should be part of the quality assurance portfolio of every software development organization.

→ Questions Q.003, Q.004, Q.014, Chaps. 3 and 4.

Not being able to engage in an architecture evaluation for a particular case.

Stakeholders might be aware of the power of architecture evaluation and currently have a concrete case for architecture evaluation at hand, but they might not be able to engage in the architecture evaluation. The advocates for an architecture evaluation might not be able to convince management, or technical stakeholders might not be willing to participate.

→ Questions Q.028, Q.029, Q.030, Q.100, Q.101.

Not having the skills or the right people to execute an architecture evaluation.

Stakeholders willing to go for an architecture evaluation might face the challenge of identifying a skilled auditor for the task. Instead of starting immediately with a good enough candidate, they might spend too much time and effort on selecting an auditor. We experienced more than ten cases with long preparation phases prior to the actual architecture evaluation where the patient eventually died on the way to the hospital (before we could start evaluating, the project was canceled due to inadequate architecture). Being successful in evaluating architectures requires skills and experiences and is driven by the stakeholders' level of confidence. To introduce architecture evaluation in an organization, one should think big, but start small. Incremental and iterative rollout of architecture evaluation promises to be the path to success if accompanied by coaching of architecture evaluation skills.

→ Questions Q.006, Q.098, Q.113.

2 What Is the Background of Architecture?

We will sketch the big picture of architecting in this chapter—it is far more than just a phase for modeling boxes and lines between requirements and implementation. Today's complex software systems require continuous architecting over the whole lifecycle of the software system. We will place particular emphasis on industrial practice, where architecting can never be done in the ideal way but where one has to continuously make deliberate tradeoffs.

2.1 What Is the Point?

Q.008. What Causes Complexity in Software Engineering and Architecting?

The output of software engineering is mainly the source code representing the solution ideas translated into algorithms and data structures, which eventually are built into an executable form. What sounds like a simple, straightforward problem-solving process, in fact turns out to be a real engineering challenge for any non-trivial problem—because complexity strikes back.

Complexity arises because of many reasons:

- **Modern software systems are inherently complex**. Understanding the problem to be solved can become difficult, in particular for legacy systems, which have a history of design decisions made in the past—potentially forgotten, unknown, and inadequate for today's requirements, and causing technical debt.
- **Managing the stakeholders and sponsors can be complicated**. Elicitation of the actual requirements from stakeholders can become difficult and conflict-laden. Drift between the assumed and the actual requirement may lead to inadequate design decisions.

J. Knodel and M. Naab, *Pragmatic Evaluation of Software Architectures*,
The Fraunhofer IESE Series on Software and Systems Engineering,
DOI 10.1007/978-3-319-34177-4_2

- **The solution itself turns out to be a complex conglomerate of system artifacts**. The implementation typically consists of hundreds and thousands of source code entities and grows continuously. Obviously, it is not feasible to manage this efficiently on the source code level. Drift between the actually implemented architecture and the intended architecture may cause subsequent design decisions to be made on the wrong basis.
- **The collaboration of humans in teams or distributed organizations requires the exchange of information**. The resulting need for sharing adds another dimension of complexity. Design decisions and their rationales have to be documented explicitly, otherwise they get lost during such exchanges.
- **Tools and technologies are another source of complexity**. Tools and technologies promise to be beneficial in supporting engineering, but their drawbacks are often discovered only after the decision to use them has been made. It not easy to reverse such decisions once the tool or technology has become an integral part of the system.
- **Change is the inevitable characteristic of any successful software system**. As software is “soft”, it is often assumed that arbitrary changes can be made. In principle, this is of course correct, but the effort required and the time needed to accommodate the change can be enormous and hence in practice, implementing changes is often economically infeasible. Any change entails the risk of making wrong or inadequate design decisions.

Thus, making adequate design decisions is the most crucial thing in software engineering. In common software engineering practice, software architectures are used as the central vehicle for making design decisions explicit as well as for prediction, analysis, and governance in the lifecycle.

Architecture provides the possibility to work with abstractions, which promises to be less effort-intensive than realizing every solution idea in code and seeing afterwards whether it works or not or which alternative works best. We believe that architectures should be used as the central vehicle for prediction, analysis, and governance. Of course, the benefits gained by architecting have to outweigh the costs created by the investments into architecting. If done properly, we are able to consciously design the software with the architecture as the conceptual tool to manage complexity in software engineering. At the same time, we can use the architecture as a vehicle to communicate the vision of the system, to make the right decisions at the right time, and to save a lot of rework. Thus, architecting avoids the trial-and-error approach of just writing source code without having a master plan.

In fact, it is commonly agreed that “every system has an architecture”. This statement is definitely correct, as all design decisions made for an implemented system are made somehow, somewhere, by someone. However, such an implicit architecture has limited value, as it can’t be “used explicitly” for any purpose. This holds true independent of the quality of the implicit architecture, even if the architecture perfectly fulfills all requirements of the software. Thus, the mission of architecting is to provide explicit information on everything that is important for solving current and/or future anticipated challenges regarding the engineering, operation, or execution of the software. This information enables sound decision making by the architect or any other stakeholder with the power to decide (e.g., management, project lead, product owner) or with the need to use this information for other engineering tasks (e.g., development, testing, operation).

Q.009. What Drives Architecting?

The desired properties of the software system originate from stakeholders. Stakeholders are those individuals or groups who can directly or indirectly affect, or be affected by, the software system. Stakeholders have concerns, which shape the product and thus drive the architecting process. Concerns in the context of the architecture are requirements, objectives, intentions, or aspirations that stakeholders have for the software system, as defined in (ISO/IEC 42010 2011). For any software system, there is always a multitude of stakeholders with almost indefinite concerns and limited time and budget to produce the software system. So architects have to focus on what really matters: how to accomplish business goals, how to achieve key functional and quality requirements, and how to handle given constraints. These are things that the architect must take care of. Architecture drivers capture these important, significant, missing, or still unknown requirements and complex exceptional cases, aggregate large numbers of similar (types of) or repeating requirements, and consolidate different stakeholders' opinions and concerns. The design decisions must define solution concepts that satisfy the architecture drivers considering the given context factors (please refer to question Q.013 for more details), and thus also satisfy the most important stakeholder concerns.

Q.010. How Does Architecting Work?

Architecting translates concerns and drivers in the problem space into decisions and manifestations in the solution space. To be successful with architecting, we first have to understand the problem space and then we have to make design decisions in the solution space and finally manifest the decisions made by coding the algorithms and data structures of the software. Architecting has different facets:

- **Predictive**: It is used to make predictions and avoid costly trial and error.
- **Prescriptive**: It is used to prescribe solutions to be realized in a uniform way.
- **Descriptive**: It is used to build abstractions and make complexity manageable.

We recommend the following key activities for effective architecting in practice (see Fig. 2.1), during new construction as well as maintenance of legacy systems. These are not one-time activities but permanently accompany a system.

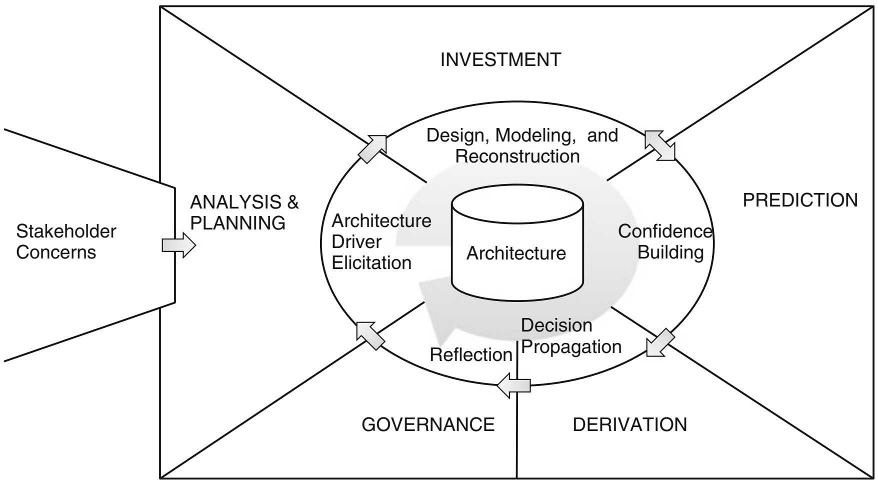


Fig. 2.1 Architecture-centric engineering: key activities. © Fraunhofer IESE (2012)

- **Engage the architecture for a purpose**: Define an explicit mission for architecting, plan effort and time to be spent on accomplishing the mission, track progress, and orchestrate the architecting work.
- **Elicit architecture drivers**: Know and manage your stakeholders, elicit their concerns and figure out what is important to whom and why, negotiate and resolve conflicting concerns, and consolidate concerns into architecture drivers. Understand the scope of the current architecture driver: be aware of what can be influenced and what is fixed for the current architecture driver, and what might change or could be changed.
- **Design (or reconstruct) a solution**: invest in being creative and designing a solution that is adequate and appropriate for satisfying the architecture driver—not more and not less; be explicit and document the architecture; ideally describe what the code itself does not; document the rationale for design decisions and discarded alternatives; reconstruct design from existing systems, if necessary; reuse.
- **Build confidence in the design**: Make early predictions on quantity and quality based on abstractions (before manifesting the decisions made), reason about design alternatives and finally accept one (and discard the others), and convince the stakeholders of the adequacy of the architecture to satisfy their concerns.
- **Propagate decisions and exploit the design**: Communicate design decisions to subsequent engineering steps and derive system artifacts from the architecture to manifest design decisions in the source code.
- **Reflect about the decisions made and govern the architecture**: Collect feedback on the code and technology level about the decisions made; adapt and refine, if necessary, and sustain the manifestation of design decisions by assuring compliance of the implementation with architecture.

- **Embrace change**: Anticipate changing and new concerns and handle unforeseen concerns (they are inevitable); evolve the architecture when changes occur; seek to improve past decisions, if necessary; iterate over the activities as listed above.

In theory, architecting sounds simple and straightforward, but in practice it is far more complicated.

Q.011. Why Is Architecting Complicated in Practice?

At any stage, drift happens between explicit and implicit information—accidentally or intentionally (see Fig. 2.2). Mastering this drift is the major challenge for architecting in practice.

Architecture and consequently the resulting software system have to follow the concerns of the stakeholders. Not being aware of a drift in concerns might result in investing design and engineering effort for the wrong items (i.e., concerns not intended by the stakeholders). However, drift is inevitable when architecting in practice. Stakeholders as the sources of architecture drivers may change their minds and may change their view on certain aspects of the system over time. Architects themselves may deviate from the information they made explicit in the documentation. Last but not least, developers may diverge from the decisions made by the architect.

Thus, mastering the drift requires continuous architecting over the whole lifecycle of the software system. The architect needs to promote the architecture and constantly gather information about stakeholders' concerns and their manifestation

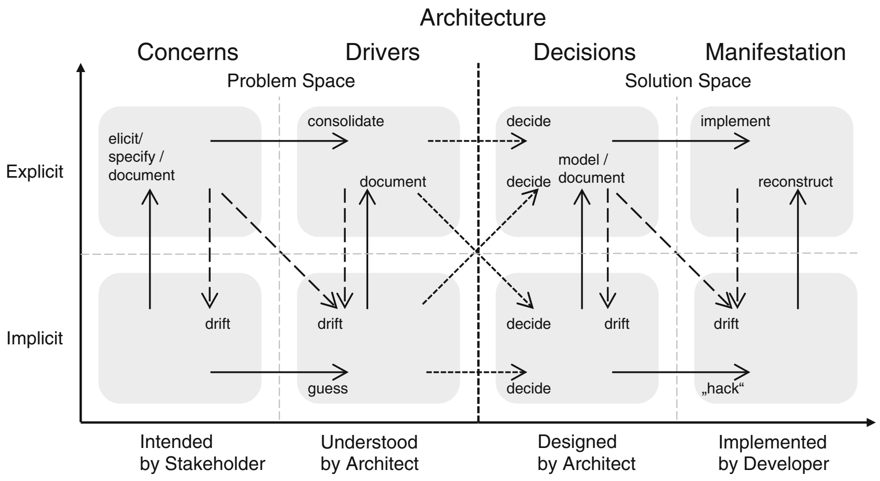


Fig. 2.2 Drift in architecting. © Fraunhofer IESE (2013)

in the source code—and, of course, handle any drift that is detected. Mastering (or failing to do so) determines the success of architecting and sustains the return of architecting investments over the lifecycle of the software system.

Figure 2.2 depicts the stages of architecting—covering concerns and drivers in the problem space and decisions and manifestations in the solution space. Note that each concern spans one architecting area of interest; of course, the areas of interest can overlap or may even be almost identical for different concerns. The stages in architecting deal with implicit information by default and we invest effort to make the information explicit. Having information about the "what & how" in an explicit form allows exploiting the architecture for manifold purposes and hence getting benefits from the investments made into the architecture. Explicit information can be communicated and is useful for humans who perform tasks and make decisions; hence, it serves every stakeholder, including the architects themselves.

Every stakeholder pursues targets and intentions with a software system. Such targets can be quite diverse, ranging from functional or quality requirements via business goals to technical, organizational, legal, or even political constraints. When stakeholders specify these concerns and they get documented, they are made explicit. In part, this is the classical task of requirements engineering, and the output of requirements engineering can be shared with the architect. But typically, not all stakeholders are covered by requirements engineering; in particular internal technical stakeholders (e.g., developers, testers, and maintainers) are often omitted. In these cases, the architect has to elicit the stakeholders' concerns. The same is true in cases of missing or incomplete requirements. The goal is to identify architecture drivers, meaning the things the architect must take care of. Note that it is not the goal of the architect to substitute the requirements engineer and produce a full specification of the software system. However, merely knowing what every stakeholder strives for is not enough. Competing concerns need to be consolidated, prioritized, and aligned with the overall strategy when engineering the software system.

The architect captures his understanding of the stakeholders' concerns by formulating architecture drivers. Drivers are the consolidated output of architecting in the problem space—representing accepted and approved concerns. Thus, architecting in the problem space transforms concerns into drivers, potentially including refinements and/or modifications, which have been negotiated and agreed on during interaction between the architect and the stakeholders.

In case the architects do not receive explicit information from the stakeholders, the only thing they can do is guess. Guessing might be guided by experiences from the past or assumptions about the domain of the problem space. Such educated guesses need to be documented in order to become explicit output of architecting in the problem space.

Ideally, every design decision would be explicitly documented prior to implementation. In practice, this would be extremely time-consuming and effort-intensive, and thus infeasible. Additionally, this is only possible in theory: there is always the need for information and feedback from implementation work. Therefore, we need to define the scope of the architecture regarding what we design now and what others or we will design later (i.e., we leave it open for now). In practice, design decisions are made throughout the whole lifecycle of the software system. Some design decisions have far-reaching effects and are hard to change (in later phases or future projects). So it is crucial to make these architecture decisions consciously during the lifecycle, to get these decisions right, and to model or document them explicitly.

For every engineering iteration, architects need to at least address the key requirements and quality attributes and be able to explain how they are addressed. Furthermore, they should have enough confidence that these can actually be achieved. Then they can propagate design decisions and assign implementation work units to developers. Such work units are decomposed frames of the software system derived from the architecture. They are constrained by the decisions made by the architect but leave space to refine these design decisions and to implement algorithms and data structures. Finally, an architect should be able to reflect the decisions made based on feedback from the code level and control the manifestation of these decisions.

Every decision not made with an architectural intention would otherwise be made at a later point in the lifecycle of the software system by someone else, who in the worst case might not even be aware of making the decision. Such (often implicit) decisions are manifested in the source code but it is not easy to control, evaluate, and reverse them. Reconstructing them can become a tedious task for the architect and is more often than not infeasible given common effort and time constraints.

Q.012. How Do I Bridge the Gap Between “What & How”?

Bridging the gap between problem space and solution space requires creativity and experience in design and decision making. Unfortunately, it is not possible to automatically derive a solution based on the statement of a problem. Hence, in design we rely on best practices and sound principles such as abstraction, separation of concerns, encapsulation, and uniformity. In addition, we establish an architecting approach that is tailored to the characteristics of the software system, in particular software engineering factors arising from the context, in which architecting and consequently architecture evaluation take places (engineering, execution, or operation context).

Q.013. What Are Context Factors for Architecting and for Evaluating Architectures?

Having the right information at hand is the prerequisite for making the right decisions in architecting the software system. Obviously, we need information about the architecture of the software system itself, but also about its context. The same holds true when evaluating the architecture.

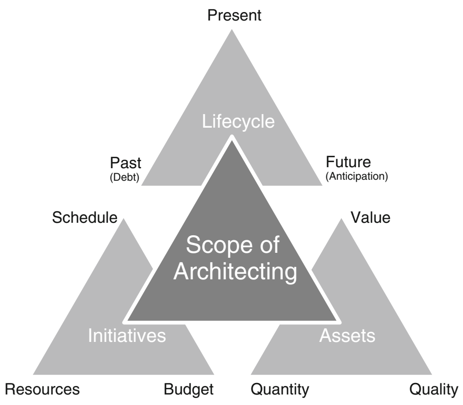


Fig. 2.3 The Bermuda triangle of architecting. © Fraunhofer IESE (2015)

The context factors—we call it the Bermuda Triangle of architecting—are depicted in Fig. 2.3. For any activity, architects have to be aware of the software system's current position within the Bermuda Triangle (and note that a change in any factor depicted in the triangle affects the other factors). The challenge for architecting in general is to balance this architecture equilibrium:

- **Design decisions have to acknowledge the overall lifecycle of the software system**. In architecting, many decisions to be made at present require anticipating future change scenarios (which might emerge or not). All of these decisions have to be made in the light of the history—the previous decisions made in the past. Ideally, past decisions would have anticipated the future perfectly, but in reality, this never happens. So the architecture has to tackle the debt of the past—inadequate decisions originating due to changing requirements or wrong assumptions due to workarounds made because of high workload or time pressure.
- **Design decisions have to acknowledge the constraints imposed by the initiatives driving software engineering**. Construction, maintenance, evolution, and retirement of software systems are software engineering efforts driven by humans and are associated with a certain schedule, budget, and availability of human resources. Architects are limited in their creativity by these boundaries (of course they can try to influence these constraints). Many ideas sound appealing at first glance; however, their realization is not possible under the given constraints.
- **Design decisions have to acknowledge the scope of the assets to be delivered per iteration**. Architecting is not an end in itself. It is an auxiliary artifact in software engineering to facilitate software engineering. Architects should always keep in mind that there is a demand for assets (source code, executables) to be delivered—and that the architecture as such is of minor interest for users, customers, or sponsors. Assets have a business value (typically decreasing over

time) and are delivered with a scope for each iteration—comprising a defined amount of functionality exhibiting a certain degree of quality in use.

To enable sound decision making at any point in the software lifecycle, knowledge is required. Knowledge is the result of a learning process of a human being. It can be seen as a function of (task-related) information, experience, skills, and attitude at a given moment in time. Thus, knowing the context of architecting is crucial for decision making and evaluating the impact of design decisions.

2.2 What Mistakes Are Frequently Made in Practice?

Doing no architecting at all.

Development organizations with no architects or no clear responsibilities for architecting run the risk of not meeting functional and quality requirements. Our experiences show that an organization might succeed in bringing releases to the market at the beginning. However, after some time, there is a turning point and decisions that were made early, but which were not thoroughly thought through, strike back: Maintenance costs might explode; developers might spend more and more effort on bug fixing and testing rather than on implementing new features; or countermeasures such as restructuring might fail because the organization lacks the abstractions of the architecture as a communication and decision vehicle. An architecture evaluation can still provide input to decision making. However, prior to the actual evaluation, a reconstruction phase has to take place to uncover the architecture that is relevant for the current question. Note that reconstructed excerpts are typically far from complete but nevertheless constitute a good starting point for further architecture re-documentation.

→ Questions Q.002, Q.008.

Having no explicit architecture documentation.

Making design decisions and documenting them in models, views, perspectives, and diagrams enables explicit sharing of information. If not made explicit, the information remains in the mind of the architect in the best case; in the worst case, if the architect has left, the information is lost as well. An architecture evaluation (with neutral auditors) provides explicit information in terms of reports or presentations for the areas under evaluation. Such parts can be a starting point for turning more and more information from implicit to explicit. Note that it is not

economically possible to make all information explicit; however, we recommend investing into explicit architecture documentation, at least to a certain extent.
→ Question Q.011.

Not being aware of the architecting Bermuda Triangle.

If architects are not aware of the context factors, they might get lost in the Bermuda Triangle of architecting. Ideally, they would meet the sweet spot, but in practice they fail. The scope for architecting in the iteration is too ambitious, too trivial, too large, or too small; design decisions are made too late or too early, for too many or too few things. An architecture evaluation can reveal wrong assumptions about the context factors of design decisions. When evaluating the architecture, the auditors shed light on the context factors for the area of interest currently under evaluation. This may lead to insights about design decisions that were made based on wrong assumptions or ignoring certain constraints imposed by context factors.
→ Questions Q.008, Q.013.

Not being aware of drift in architecting.

Drift in the problem space might lead architecting into the wrong direction, whereas drift in the solution space might make the abstractions provided by the architecture unreliable, delusive, and in the worst case useless. Architecture evaluation is a quality assurance instrument that creates awareness of drift by eliciting and revisiting stakeholder concerns to be evaluated. By checking the compliance between the intended and the implemented architecture and challenging the architect against the documented concepts, drift may become apparent. Architecture evaluation helps to mitigate the risk of drift.
→ Question Q.011.

Not being aware that drift is often unavoidable.

It can be observed that most solution concepts need iterations of revisions as soon as they are getting implemented. Typically, drifting from the designed concepts cannot be avoided (often directly at the start of implementation). Architecting requires updating the documentation of the respective solution concepts directly or at a later point in time.
→ Question Q.011.

What Is Architecture Evaluation? 3

In this chapter, we will present what architecture evaluation is and what it consists of. We will break down the overall method of architecture evaluation into five clearly delineated checks: (1) checking the integrity of the drivers, (2) checking the solution adequacy of an architecture, (3) checking the architecture documentation quality, (4) checking the compliance of the implementation with the architecture, and (5) checking the code quality in general. We will introduce confidence levels as a response to the risk-driven idea of architecture evaluation: we only want to invest as much as needed to gain enough confidence. We will show how evaluation results can be interpreted, aggregated, and represented to senior management by mapping them to a color-coded rating scale.

3.1 What Is the Point?

Q.014. What Is the Mission of Architecture Evaluation?

The mission of architecture evaluation is to determine the quality of the (envisioned) software system and the quality of the (auxiliary) artifacts created during architecting or derived from the architecture.

Architecture evaluation aims at achieving confidence in the (architectural) design decisions made about the software system with respect to an evaluation question. It summarizes all activities aimed at answering critical concerns regarding the software system, its environment, its evolution (debts from the past and anticipated needs for the future), and its artifacts documenting and manifesting the design decisions.

J. Knodel and M. Naab, *Pragmatic Evaluation of Software Architectures*,
The Fraunhofer IESE Series on Software and Systems Engineering,
DOI 10.1007/978-3-319-34177-4_3

The evaluation of the (envisioned) system quality analyzes the following questions:

- How well are the stakeholders' requirements (driving the architecture) understood and agreed on?
- How well are the solution concepts and design decisions of the architecture suited to adequately addressing the requirements?
- How well are the solution concepts manifested in the implementation?

The evaluation of the artifact quality (documentation, models, source code) analyzes the following questions:

- How well is the documentation of the architecture structured and how consistent is it?
- How well is the source code of the software system structured and how readable is it?

Thus, architecture evaluation requires several checks and ranges from requirements via the architecture to the implementation/system level:

- Check for ambiguities, unclarities, or drift in stakeholder concerns and derived architecture drivers.
- Check for flaws and issues in solution concepts and identify inadequate architecture decisions.
- Check for problematic deficiencies and inconsistencies in the system's architecture documentation.
- Check for drift between a system's intended architecture and the architecture as realized in the implementation.
- Check for anomalies in the source code with respect to best practices, quality models, style guides, and formatting guidelines.

Q.015. What Does Our Architecture Evaluation Method Consist of?

Our approach to architecture evaluation applies a set of checks—driver integrity check, solution adequacy check, documentation quality check, architecture compliance check, and code quality check—to deliver answers with a certain level of confidence to the stakeholder concerns.

Figure 3.1 provides a conceptual overview of our approach for architecture evaluation, which is called RATE (Rapid ArchiTecture Evaluation). It is not intended to be yet another architecture evaluation method. Rather, it is a compilation and collection of best practices of existing evaluation approaches tailored towards pragmatic (or rapid) application in industry. We derived this approach from

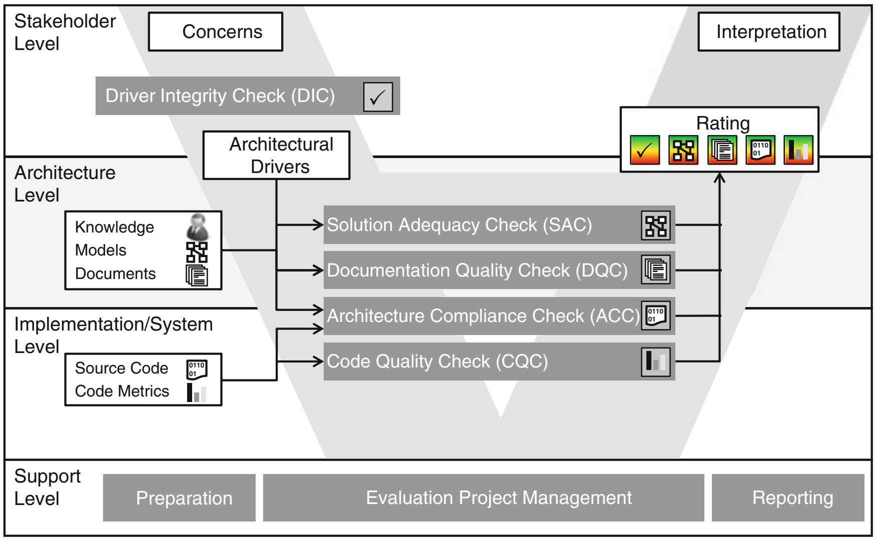


Fig. 3.1 The Rapid ArchiTecture Evaluation Approach (RATE). © Fraunhofer IESE (2011)

experiences in projects with industry customers (Knodel and Naab 2014a, b) and consolidated lessons learned and concrete hints. For more background information on the underlying architecture evaluation methods, we refer to the respective (academic) publications, for instance (Clements et al. 2001).

RATE comprises five checks, with each check serving a distinct purpose. All checks follow the same principle of work: to reveal findings aimed at confirming/improving the system's quality and/or the artifact's quality. The checks are always applied for concrete stakeholder concerns.

The Driver Integrity Check (DIC) serves to check the integrity of stakeholders' concerns in its manifestation as an architecture driver.	
Goal	Consolidate Stakeholder Concerns
Key Findings	Deviations, Inconsistencies, Ambiguities
In	Architecture Drivers (Business Goals, Constraints, Quality Attributes / Non-Functional Requirements, Key Functional Requirements)

Stakeholders have concerns about a software system. The DIC analyzes the integrity of these concerns and checks whether there is agreement about the stakeholders' concerns. It sets the goals for other architecture evaluation checks (and determines whether or not additional checks should be performed). It balances and negotiates these potentially conflicting concerns with respect to business goals, constraints, external quality (runtime), internal quality (development time), and preparation for anticipated changes. Thus, the driver integrity check consolidates different stakeholder concerns into architecture drivers. Moreover, it structures and formalizes the set of drivers for the architecture (sometimes also called architecture-significant requirements) with the help of templates and requires explicit stakeholder approval (e.g., agreement and signing off on the driver) before proceeding with other checks.

The Solution Adequacy Check (SAC) serves to check how well solution concepts (relevant excerpts of the architecture) are suited to satisfying the stakeholders' concerns.	
Goal	Satisfy Architecture Drivers
Key Findings	Inadequacies, Risks, Assumptions, Trade-offs
In	Architecture (Design Decisions, Solution Concepts)

The SAC determines whether an architecture permits or precludes the achievement of targeted functional requirements and particularly quality requirements [for an overview of frequently checked quality attributes, see (Bellomo et al. 2015)]. However, it does not guarantee achievement because activities further downstream obviously have an impact. The Solution Adequacy Check allows reasoning about the advantages, drawbacks, and trade-offs of a solution concept (or alternative solution concepts) for satisfying requirements or changing requests at hand. Hence, it provides valuable input to decision-making by creating awareness of flaws in the design. It enables identification of potentially problematic design decisions, early prediction based on abstraction, and comparison among alternatives, and can be the starting point for timely responses and corrections to improve the architecture. The Solution Adequacy Check is applied specifically for the software product under evaluation and comes with the downside that it is a largely manual task causing significant effort and leading to mainly qualitative results only.

The Documentation Quality Check (DQC) serves to check the documentation of solution concepts and adherence to documentation best practices.	
Goal	Enable Understanding and Satisfy Best Practices
Key Findings	Deficiencies, Violations of Best Practices
In	Architecture Documentation (Models, Documents)

The DQC assures that the documentation of the architecture makes it possible to understand the solution concepts. It determines whether or not readers will find the right information to gain knowledge, understand the context, perform problem-solving tasks, and share information about design decisions made. Additionally, the documentation (reports, presentations, models) is inspected with respect to criteria such as consistency, readability, structuredness, completeness, correctness, extensibility, and traceability.

The Architecture Compliance Check (ACC) serves to check the manifestation of the solution concepts in the source code and/or in the executables of the system.	
Goal	Realize Solution Concepts Consistently in the Code
Key Findings	Violations (Structural & Behavioral)
In	Source Code & Executables

The ACC aims at assuring the compliance of an implementation (with its so-called implemented architecture) or of the running system with the intended architecture and its rules on code structure and system behavior. Only if the architectural concepts are implemented compliantly does the architecture have value as a predictive and descriptive instrument in the development process. Compliance checking [see Knodel et al. 2006; Knodel and Popescu 2007, or Knodel 2011] typically requires architecture reconstruction (reverse engineering of code or instrumentation of runtime traces) to collect facts about the system, from the source code as well as from the running system. By mapping the actual and the intended architecture onto each other, this compliance can be evaluated. This is a task that strongly benefits from tool support due to the large number of facts typically extracted.

The Code Quality Check (CQC) serves to check the implementation regarding adherence to coding best practices and quality models.	
Goal	Enable Understanding and Satisfy Best Practices
Key Findings	Anomalies to Best Practices and Quality Models
In	Source Code, Configuration Files

The CQC aims at numerically quantifying quality properties of the source code. Many integrated development environments (IDEs) support a wide range of code quality metrics, either natively or through plugins or external tools. Such tools include certain thresholds or corridors that are recommended not to be exceeded as best practice. Violations are reported for the offending code elements. The aggregation by these environments of code-level metrics into system level metrics is often not more sophisticated than providing descriptive statistics (i.e., average, mean, maximum, minimum, total). Assessors typically either need to craft their own aggregations or use expert opinions to provide accurate evaluations or to establish quality models for one or more quality attributes. This is typically done by mapping a selection of low-level metrics to those quality attributes using sophisticated aggregation techniques.

The code quality check is not really an architecture evaluation technique. However, we integrated it into our overall method as we found over time that it is necessary to thoroughly answer the evaluation questions posed by stakeholders.

Q.016. What Determines the Scope of an Architecture Evaluation?

Any decision we make during the design and evolution of a software system entails the risk of being wrong or inadequate for its intended purpose. For every design decision, we have a gut feeling about our confidence regarding how well the design decision will satisfy the goals of architecting in its context. More often than not, this feeling might be wrong. Furthermore, changing requirements and drift over time, as discussed above, challenge the design decisions made in the past.

Architecture evaluation can increase confidence in such cases by identifying risks and flaws in the driving requirements, the solution concepts of the architecture, or the resulting implementation. However, not all design decisions are equally important; some are more crucial for success than others. In those cases where the confidence level as such is not (yet) acceptable or where we need to have more evidence to convince stakeholders, we conduct an architecture evaluation. It will lead either to increased confidence in the decisions made or to findings about flaws calling for redesigning the architecture.

Chapter 1 presented many concerns and questions that can be answered with architecture evaluation: Depending on the criticality of the questions, the confidence level differs within an architecture evaluation. For some design decisions, it is fairly easy to check them off, while others require a thorough analysis of the software architecture and its underlying implementation or even the running system to arrive at a conclusive answer.

Consequently, architecture evaluation is triggered by the stakeholders' concerns for which the architectural solution concepts and their implementation (if available) are checked. The criticality of the concern determines the depth to which the concerns will be analyzed (i.e., the level of confidence).

Q.017. What Are the Basic Confidence Levels in Architecture Evaluation?

The level of confidence indicates the quality of the analysis for each of the five checks introduced above. It expresses, for instance, the degree to which the requirements driving the architecture have been clearly understood, the degree to which the architecture adequately addresses the requirements, or the degree to which the implementation consistently realizes the architecture. Thus, the higher the confidence, the lower the risk of having made a wrong design decision.

We use distinct levels to express confidence (see Fig. 3.2 for an overview). Each confidence level has certain characteristics (applicability, input, effort), which differ for the five checks introduced above (see Chaps. 5–9, respectively).

The basic confidence levels for **artifact quality** are as follows:

- **Believed**: The lowest level of confidence is pure belief. This means that the quality of the artifact is accepted without further analysis. No dedicated effort is

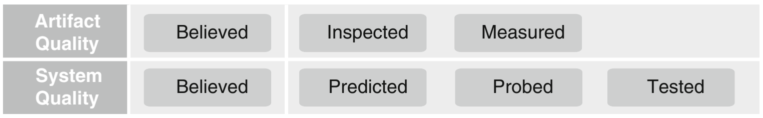


Fig. 3.2 Architecture evaluation: confidence levels

spent on raising findings and investigating their impact (spanning concerns, driver, solution concept, and source code). The work of the architect regarding these design decisions is trusted. This level of confidence applies to the vast majority of design decisions. In most cases, it will be just fine because the concerns were well known, the solution concepts were suited to the architecture drivers, and the implementation is compliant with the intended design. Leveraging the knowledge, skills, competencies, and experiences of the architect yields adequate design decisions.

- **Inspected**: The artifact quality is inspected in each of the checks in order to reveal findings (positive and/or negative). The inspections typically follow a structured, well-defined process with defined roles and activities in order to examine the degree to which the manifestation satisfies the target. Inspectors may be the architects who are actually responsible for the design, internal peers not directly involved in the architectural design (e.g., architecture boards, other architects, quality assurance groups, or method groups within the same organization), or an external, independent third party (consultants, research organizations, etc.).
- **Measured**: Measurements provide quantitative numbers and data by calculating certain characteristics of the artifacts of the software system under evaluation. Measurements apply formally defined formulas to perform the calculations (e.g., metrics, style guides, formatting guidelines, etc.).

The basic confidence levels for **system quality** are as follows:

- **Believed**: The lowest level of confidence is again pure belief, meaning that it is assumed that the resulting software system will be adequate; no further analysis is done.
- **Predicted**: Prediction means using existing knowledge, current information, and historical data to reason about the future. Confidence is achieved by using the available resources to predict properties of the software system. The better the underlying data and model are applied for prediction, the higher the level of confidence achieved. Predictions comprise estimations performed by topic experts (educated guessing based on individual experiences and expertise), simulations based on the execution of a formally described model (often leaving room for some uncertainty in the parameters or in the data set of the model), or heuristics based on data and fact-based (sets of) rules where evidence is provided from other software systems that they are relevant for some kind of issue

(because of their general nature, the heuristics are likely to be valid for the software systems under evaluation, too).

- **Probed**: A probe indicates an exploratory action to technically try out and spike a solution (or a potential solution concept under evaluation). A detailed and thorough investigation serves to study context factors, examine assumptions about environment, concepts, and technologies, and explore scaling factors. In short, it serves to mitigate risks. Probes may use mock-ups to explore an idea or a concept with low effort in a simplified or simplistic manner, representatives of the actual software system (e.g., earlier versions or variants, or similar parts of other systems sharing the design or the same technology stack), or prototypes to learn by example by providing the implementation of a crucial part or aspect of the solution concept. Probes provide high confidence as the solution concept realized can be explored in full detail, with the target technology, and its adequacy can be trialed.
- **Tested**: Testing executes the software system to evaluate certain aspects of interest. It is checked whether a given functionality works properly with acceptable quality. Additionally, measurements about the running software system can be taken. The systematic, reproducible act of testing can be done either in a lab environment or in the field (production environment). Lab environments provide a playground for testing with (artificial) data and a (similar or identical) infrastructure as in the field. Testing in the field provides ultimate confidence as the quality of the software system is analyzed in the production environment.

Q.018. What Is the Outcome of an Architecture Evaluation?

The application of each of the five checks (see above) will produce a set of findings in the object under evaluation with respect to the evaluation question. Findings represent positive and negative results revealed by the check performed. They may include risks identified, assumptions made, scaling factors, limitations, trade-offs, violations, etc. All findings can be pinpointed to a concrete part of the object under evaluation. The results of each check always have to be interpreted in the light of the overall questions triggering the architecture evaluation, and in the context and the environment of the software system under evaluation. This interpretation is typically not easy and straightforward and hence requires experience in architecting. Accordingly, the outcome of each check is a list of findings.

Q.019. How to Interpret the Findings of an Architecture Evaluation?

The interpretation of the findings is crucial in order to benefit from the overall architecture evaluation results. Architecture evaluation reveals positive and negative findings about both the system quality and the artifact quality. The interpretation of the findings is context-dependent, based on the underlying evaluation question, the software system under evaluation, and the nature of a finding. In particular in case of negative findings, it is the starting point towards deriving improvement actions.

The nature of findings (see Fig. 3.3) is characterized along two dimensions: the cause of the finding and the baseline on which the finding is based. Causes can either be based on product properties such as the inherent flaws or weaknesses of the software system or technology and their misusage, or on human capabilities and human limitations in terms of coping with complexity (i.e., comprehensibility) and dealing with continuous change in the evolution of the software system (which is the inevitable characteristic of any successful system). The baseline describes the criteria that are the basis for stating the finding. These can range from universal criteria (applicable in the same form to any other software system, e.g., ISO standards or metrics) to individual criteria (tuned specifically to the software system under evaluation, e.g., meeting product-specific performance requirements). In between there may be domain-specific or organization-specific criteria, which have limited applicability compared to universal criteria.

Architecture evaluations may reveal a number of findings whose nature differs (see question Q.096 for an example of the nature of findings for maintainability). Some evaluation questions may need to be evaluated individually for the software system under evaluation, while others may use general-purpose rules (these thus rules have wider applicability and may be checked across many software systems). Typically, findings caused by technologies in use and limitations of human capabilities are more general. They serve to detect risks with respect to best practices, guidelines, misusage, or known pitfalls in the technologies. In practice, it is much easier to aim for such causes because tools can be bought that come, for instance, with predefined rules or standard thresholds and corridors for metrics. They are easy to apply and make people confident that they are doing the right thing to mitigate

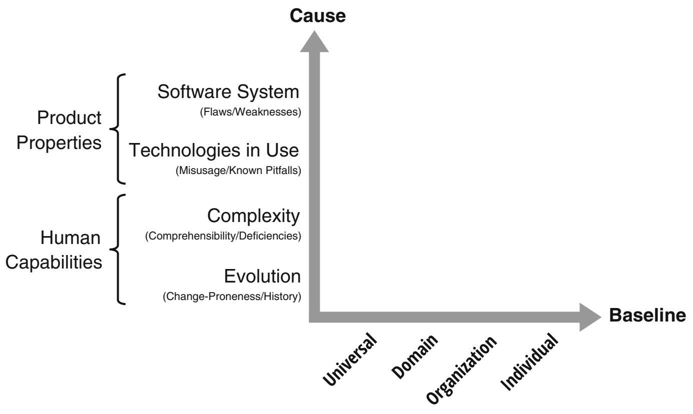


Fig. 3.3 Architecture evaluation: nature of findings. © Fraunhofer IESE (2014)

risks. The downside is that such general-purpose means often do not fit to the evaluation questions driving the architecture evaluation, but this is not discovered. Organization-specific techniques take the specific guidelines of the domain, the development organization, the specific development process, or the company culture into account. However, they do not take the concrete product properties into account. In simple cases, organization-specific techniques could be tailored from instruments for detecting universal causes based on experiences made in the company. Being product-specific implies being concerned with all the aspects that make the software system unique and thus need to be applied to the product. What is common sense for testing (no one would just run technology-related test cases without testing the product-specific logic of the software system) is not the case for architecture evaluations and source code quality checks. Here many development organizations, tool vendors, and even some researchers claim to be product-specific with a general-purpose technique realized in a quality assurance tool that can be downloaded from the Internet. However, these general-purpose techniques can merely lead to confidence regarding the avoidance of risks with respect to the technology or the limitations of human capabilities.

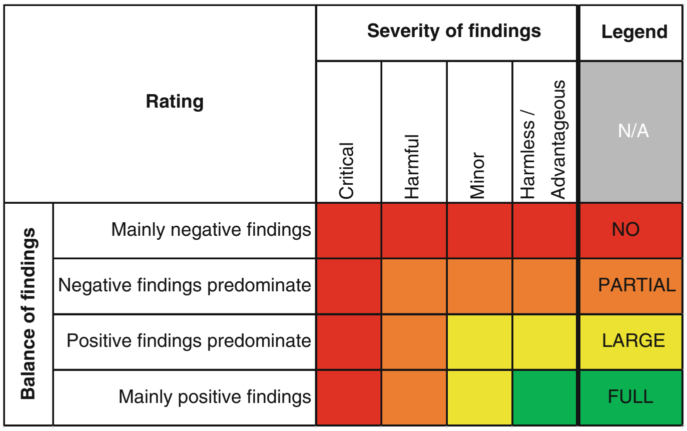
If done properly, architecture evaluation does indeed combine product-specific and general-purpose aspects and hence enables reasoning about the evaluation question at hand. This is what makes architecture evaluation an extremely valuable instrument for mitigating risks in the lifecycle of a software system.

Q.020. How to Aggregate the Findings of an Architecture Evaluation?

While the findings are detailed by their nature, it is also important to provide an overview and enable comparability of checks over time and of several candidates undergoing the same checks. We recommend rating every check result on two simple four-point scales. All findings in summary are considered to provide the aggregated rating. The higher the score, the better the rating of the manifestation object and the less critical the risks.

- **Severity of Findings** expresses the criticality of the findings aggregated over all findings per goal. Importance ranges over (1) critical, (2) harmful, (3) minor, and (4) harmless/advantageous.
- **Balance of Findings** expresses the ratio of positive versus negative findings aggregated per goal. It ranges over (1) only or mainly negative findings, (2) negative findings predominate (3) positive findings predominate, and (4) only or mainly positive findings.

The combination of both scales (i.e., the mathematical product of the two factors, see Fig. 3.4) determines the overall score for the check-based expression of the target achievement of the manifestation object:



Rating		Severity of findings				Legend
		Critical	Harmful	Minor	Harmless / Advantageous	⬜ N/A
Balance of findings	Mainly negative findings	🟥	🟥	🟥	🟥	🟥 NO
	Negative findings predominate	🟥	🟧	🟧	🟧	🟧 PARTIAL
	Positive findings predominate	🟥	🟧	🟨	🟨	🟨 LARGE
	Mainly positive findings	🟥	🟧	🟨	🟩	🟩 FULL

Fig. 3.4 Architecture evaluation: rating and legend. © Fraunhofer IESE (2015)

- **N/A Not Applicable (GRAY)**: This means that the goal achievement has not (yet) been checked. The check might have been deferred or discarded due to missing information, effort and time limitations, or unavailable stakeholders.
- **NO Target Achievement (RED)**: This means there are issues, flaws, or strong arguments against the goal achievement. Fixing these major weaknesses will cause serious effort for repair or require fundamental rethinking.
- **PARTIAL Target Achievement (ORANGE)**: This means that significant flaws or risks have been identified, but we expect that they can be removed with modest effort.
- **LARGE Target Achievement (YELLOW)**: This means that there are no major objections to goal achievement. However, some details may require further refinement or elaboration.
- **FULL Target Achievement (GREEN)**: This means there is full goal achievement. No findings or only harmless or advantageous findings have been revealed.

While such a simple, traffic-light-inspired scale is important for summarizing the evaluation results and for presenting the evaluation results to management and other stakeholders, there is always a need for keeping the more detailed results at hand. The rating should be based and justified on a detailed and differentiated explanation of the findings. This is necessary to justify the rating in critical situations and to make the rationale of the rating persistent. Furthermore, it serves to derive action items for starting improvement actions.

Q.021. What Are the Limitations of Architecture Evaluation?

Despite being a powerful instrument, architecture evaluation has limitations, too.

- **Architecture can only be evaluated indirectly**: The architecture that is actually interesting is the one that is implemented. However, this is not really tangible and thus abstractions and their documentations are used. However, there is always the risk of inadequate information.
- **Architecture is an abstraction**: That architecture is an abstraction has advantages and disadvantages at the same time. It has advantages in the sense that the complexity can only be handled via abstractions. It has disadvantages in the sense that whenever we leave something out, this still has an impact on the final system, which just cannot be evaluated.
- **Absolute architecture evaluation is typically not possible**: The result of an architecture evaluation is mostly not a quantified measure or degree of goal achievement. Rather, it is a set of findings that require interpretation.
- **Architecture evaluation requires cooperation**: It is very hard and inefficient to evaluate an architecture without cooperating architects as all the information has to be collected and reconstructed from documentation and code.
- **Architecture evaluation cannot guarantee quality**: The details of the implementation, e.g. specific algorithms, also have a strong impact on achieving quality attributes.

Q.022. What Is a Good Metaphor for Architecture Evaluation?

Consider a route guidance system as a metaphor for an architecture evaluation. Imagine that after driving on the road for three consecutive hours (a running project), you are almost halfway to reaching your destination (the release). The route guidance system (the architecture evaluation) can tell you whether you will reach the destination on time (within budget) or not. It gives you a plan regarding which roads to follow (static information) and is aware of construction work and traffic jams (dynamic information). But in the end, the route guidance system is able to give only recommendations and make the driver aware of risks (warnings). The final result as to whether or not you will arrive at the scheduled time significantly depends on your decisions as a driver (taking a rest, ignoring the recommendations) and on your car (the underlying technology).

3.2 What Mistakes Are Frequently Made in Practice?

Doing no architecture evaluation at all.

Although architecture evaluation has found its way into industry (Babar and Gorton 2009; Knodel and Naab 2014a, b; Bellomo et al. 2015), many crucial decisions are still being made with regard to software without getting the relevant architectural facts from an architecture evaluation. And there are still many software engineers and managers who have not heard about architecture evaluation as a means for identifying and mitigating risks.

→ Questions Q.003, Q.004, Q.014.

Reducing the evaluation results to the traffic lights only.

Providing aggregations of the evaluation results is necessary, in particular for communication and presentation. However, these aggregations only offer a limited view and there are always complex and detailed findings behind the aggregated results that have to be considered. The essence of architecture evaluation is to provide guidance for decision-making and for the upcoming work. Thus, the right interpretation of the findings is important but also challenging.

→ Question Q.018, Q.019, Q.020.

Focusing on architecture metrics or code metrics only.

Architecture evaluation with a strong focus on the fulfillment of requirements is effort-intensive. Evaluating only metrics with a tool is rather easy. Thus, practitioners often tend to measure general properties of an architecture (such as coupling metrics), which are a rough indicator of internal quality properties such as maintainability, but they completely fail to evaluate the adequacy for most architecture drivers.

→ Questions Q.015, Q.017, Q.096, Q.097.

Evaluating only the quality of the documentation.

Architecture is more than a bunch of diagrams and documents. Sometimes, architecture evaluation is reduced to the review of architecture documents with a focus on document properties such as readability, consistency, or traceability.

Although it is important to have high-quality documentation in place, it is not enough to check the documentation quality in isolation.

→ Question Q.015.

Evaluating whether the architecture is "state-of-the-art".

Using state-of-the-art technologies seems to be a promising goal for many practitioners. Thus, the goal of an architecture evaluation is sometimes to check whether the architecture is "state-of-the-art". In such cases, an evaluation against the architecture drivers is typically missing completely.

→ Question Q.014, Q.046.

4 How to Perform an Architecture Evaluation?

There are typical indicators that an architecture evaluation would be beneficial. Architecture evaluation is typically conducted as a project, answering specifically formulated evaluation goals. We will show the big picture of how an evaluation project can be set up and structured, including which stakeholders to involve and how to manage their expectations. We will offer support for estimating the effort for an architecture evaluation and the actual project management. Finally, we will share experiences on the interpretation of evaluation results and describe how to structure a concluding management presentation that accurately conveys the evaluation results and presents recommendations.

4.1 What Is the Point?

Q.023. When Should an Architecture Evaluation Be Conducted?

There is no single point in time when an architecture evaluation should be done. Rather, there are several points in time throughout the whole lifecycle of software products that benefit from architecture evaluation.

- Before and during architecture design work, the architecture drivers should be challenged and checked for completeness and consistency.
- Regular checks of the adequacy of an architecture under design should be done right after making the decisions or in order to compare and select architecture alternatives. Architecture evaluation aims at predicting properties of the system under design.

J. Knodel and M. Naab, *Pragmatic Evaluation of Software Architectures*,
The Fraunhofer IESE Series on Software and Systems Engineering,
DOI 10.1007/978-3-319-34177-4_4

- At certain milestones, e.g., before sending numerous implementation teams to work after a ramp-up phase, the architecture should be evaluated.
- During implementation work, compliance between the intended architecture and the implemented architecture should be checked continuously.
- Before making major decisions about the acquisition, migration, or retirement of components or whole systems, the architecture should be evaluated.
- During larger maintenance, integration, or migration projects, which need architectural guidance, the respective target architecture has to be evaluated in terms of adequacy and the subsequent development activities have to be monitored in order to check for architecture compliance.

Q.024. What Are the Steps to Follow When Performing Architecture Evaluations?

Figure 4.1 depicts an overview of the steps for performing architecture evaluations. The following sections and chapters will describe these steps in detail.

Q.025. How to Define Evaluation Goals?

Architecture evaluation is never an end in itself. It should always be aimed at achieving clear and concrete evaluation goals. The following list shows frequent categories and examples of evaluation goals:

- **Business**-related evaluation goals
  - We are planning to roll out our product in 6 months: Is the overall quality of the architecture adequate for achieving our business goals?
  - Which subcontractor should be selected considering their product portfolio?

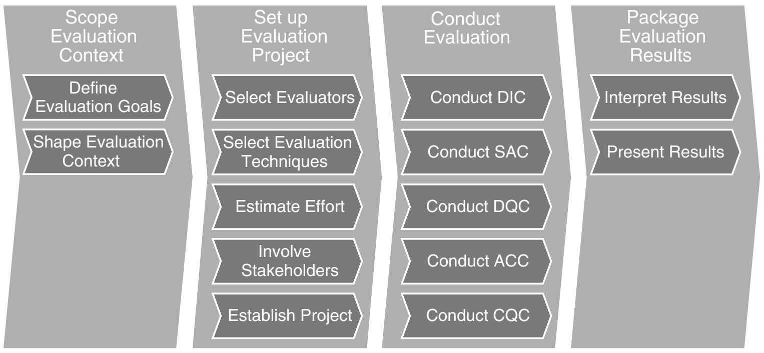


Fig. 4.1 Steps of an architecture evaluation

  - What does the adoption of a new paradigm such as cloud computing mean for our own products and business?
  - Is the current product portfolio a sound basis for expanding to a new market?

- **Quality**-related evaluation goals [related to quality attributes, see (ISO 25010, 2011)]
  - Will system A be capable of serving 20 times the number of users served today (scalability)?
  - Is the newly designed architecture able to greatly reduce the response times of the system (performance)?
- **Technology**-related evaluation goals
  - Is technology X a good candidate for data storage in our software system?
  - What is the impact of technology Y on our system and what needs to be done in order to replace it with technology Z?
- **Evolution**-related evaluation goals
  - Is the architecture of our system still adequate after 15 years or does it require significant changes in order to further serve our business needs?
  - Which changes to our system are necessary in order to make it fit for a completely new class of features?

Please note that architecture evaluations can often give only partial answers to the evaluation questions. Further evaluations might include economic and organizational aspects.

The definition of evaluation goals should be done cooperatively between the architecture evaluation owner, the architects responsible for the system under evaluation, and the auditors conducting the evaluation.

Q.026. How to Shape the Context of an Architecture Evaluation Project?

Based on the evaluation goals for the project, the context that has to be taken into account has to be clarified:

- **Systems in and out of scope**: Which systems need to be considered in the evaluation and which can be seen as out of scope? Which systems have to be evaluated with respect to their inner details, and which systems are only interesting with respect to their outer interfaces?
- **Organizations in and out of scope**: Which organizations or organizational units have to be involved?
- **Stakeholders in and out of scope**: Which stakeholders should be involved or should at least be represented in some form in the project?

Q.027. How to Set up an Architecture Evaluation Project?

To set up an architecture evaluation project, the following aspects have to be discussed and clarified:

- **Selection of evaluators for the evaluation project**
  External evaluators: If one or more of the following conditions is true, an external evaluator should be involved: in particularly critical situations; if neutrality is absolutely essential; if there is discussion about quality between a supplier and a customer; if no adequate architecture evaluation knowledge is available internally; if no architect is available internally with enough time; if the evaluation project should be conducted in a more visible manner.
  Internal evaluators (e.g., architects from other projects or cross-cutting architecture departments): If the factors above don't apply, an internal evaluator is adequate and could help to do the evaluation faster and with less lead time.
- **Selection of the right combination of evaluation techniques**
  Depending on the evaluation goals and the status and context of the software development regarding the system under evaluation, a set of evaluation techniques has to be selected. The evaluation technique categories are introduced in Chap. 3 and explained in detail in Part II.
- **Determination of the rough effort to be spent**
  The effort to be spent on architecture evaluation depends on several factors, such as the criticality of the situation, the required confidence in the results, or the complexity of the organizational situation. Questions Q.101, Q.103, Q.104, and Q.107 discuss this in more detail.
- **Involving stakeholders**
  Architecture evaluation is an activity that is not done only by the evaluators. Rather, it requires significant involvement of stakeholders (e.g., all types of stakeholders for eliciting architecture drivers or the system's architects for discussing the architecture). It is important to involve these stakeholders early. They have to understand the goals and the approach of the architecture evaluation. In particular, it is important to demonstrate management commitment to the architecture evaluation and to actively manage the expectations of all involved stakeholders. Since architecture evaluation strongly depends on a cooperative and open climate, continuous communication with the stakeholders is essential.
- **Establishing a project with rough time planning and resources**
  Unless it is a very small and informal internal activity, architecture evaluation should be viewed as a project that is actively managed. The key reason is that this increases the perceived importance and commitment. The availability of the stakeholders required for an architecture evaluation is typically very limited, and thus it is important to schedule specific meeting dates as early as possible. Often, architecture evaluation projects are expected to be conducted very quickly. From

the owner's perspective, this is understandable as he needs facts and answers to strategic questions. However, the involvement of various stakeholders and sometimes even several organizations implies the need for more time. Thus, sound scheduling and expectation management is important for the persons conducting the architecture evaluation.

Q.028. Who Should Be Involved in an Architecture Evaluation?

In principle, all relevant stakeholders should get involved in the architecture evaluation.

Typical stakeholders include architects, developers, project managers, product managers, customers, users, operations staff, testers, maintainers. Depending on the type of development organization and process, the responsibilities and role names might vary. E.g., in agile development contexts, there might be no dedicated architect, but hopefully there is somebody who knows the architecture very well. If stakeholders are not accessible directly (e.g., the user), representatives could be used as a substitute.

Q.029. How to Involve Stakeholders in Architecture Evaluation Projects?

In an architecture evaluation project, relevant stakeholders need to be identified first. This may sound easier than it actually is in practice. Stakeholders are often widely distributed in an organization; they are hard to catch due to their tight schedules; and they might not agree right away that they need to be involved in an architecture evaluation project since they are not interested in the technical details.

It has to be made clear that architecture evaluation evaluates against the concerns voiced by the stakeholders and the corresponding drivers derived from these. This is a pressing argument for being part of the architecture evaluation. Of course, not all wishes are realistic and it is necessary to agree on a set of architecture drivers before diving into the details of the evaluation. That is, the stakeholders also need to be involved in negotiation activities, which have to be actively guided and moderated by the evaluation team.

Stakeholders should be involved directly from the start of an architecture evaluation project. We always conduct the initial kickoff meeting together with all stakeholders and explain to them the goals and, in particular, the procedure of the evaluation project. They can ask all their questions to feel really involved.

During the course of the architecture evaluation project, they have to be informed and their expectations need to be managed. This can be done with regular emails stating the current status of the project, sending them intermediate results to comment on, or through intermediate and final presentations. The architects and developers responsible for the system under evaluation should be involved even closer: When there are findings as results of evaluation workshops, these should be summarized and shared with the architects and developers to check whether everything was understood correctly.

Q.030. Why Manage Stakeholders' Expectations?

It is very important for evaluators to manage the expectations of evaluation owners and stakeholders. This requires clearly communicating what has been done as part of an architecture evaluation and how to interpret the outcome of an architecture evaluation.

It is mandatory to explicitly state the confidence level that applies to the findings revealed and the ratings derived in each check. This allows conscious interpretation of the results by the stakeholders.

Furthermore, it is always necessary to keep the evaluation question and the context of the evaluation owner in mind in order to answer the questions in a way that is most beneficial for him or her. Just providing facts and findings is not enough. Architecture evaluation provides the evaluation owners and stakeholders with the information they asked for in the way that helps them to understand it best and thus enables them to make (keep, rethink, change) decisions.

Q.031. How to Conduct an Architecture Evaluation Project?

The following meetings have proven to be meaningful in most evaluation projects:

- **Kickoff meeting**
  In the kickoff meeting, all involved stakeholders should come together. The evaluation owners should state their expectations and the background. In the meeting, the evaluation goals, the evaluation approach, the schedule, and the results to be expected should be presented.
- **Working meetings**
  The working meetings are conducted based on the needs of the evaluation techniques selected. Example working meetings are stakeholder workshops to elicit architecture drivers, meetings for the reconstruction and discussion of the architecture, meetings for the discussion of code details, and so on. Evaluators should always be well prepared in order to optimally use the time invested by the stakeholders.
- **Result presentation and touchdown meeting**
  An architecture evaluation project should be officially closed with a meeting where the evaluation results are presented to the evaluation owner and the other

stakeholders. How to interpret the results and how to present them in such a meeting will be discussed in the next questions. Depending on the project, the presentation of intermediate results might be relevant (e.g., to further focus upcoming evaluation activities, to have early results and improvement suggestions that could be incorporated before a fixed release date). The result presentation and touchdown should always be used to discuss the findings and recommendations and the derived actions. Very often, the group of people attending a result presentation is not meeting each other in the same group composition otherwise.

Besides the actual evaluation work, an architecture evaluation project needs managerial and supportive activities, too:

- **Preparation**
  All activities have to be prepared by the evaluators in order to run effectively and efficiently as the stakeholders involved typically have tight schedules. Further preparation may concern the selection, acquisition, and installation of tools, which might take considerable time as well.
- **Audit management**
  An architecture evaluation is typically a project and has to be managed as a project. That is, it is necessary to schedule and organize all meetings, collect and distribute material, and define and assign tasks to evaluators and stakeholders. In our experience, architecture evaluation projects are performed because of their importance often on a tight schedule and often require re-planning due to the unavailability of stakeholders on short notice.
- **Reporting**
  An architecture evaluation has to produce results. These results have to be presented in a way that fits the expectations of the evaluation owner. Typical forms are reports and presentations. In our experience, writing reports that are too long does not pay off—condensed reports are much better. We highly recommend a final result presentation with the stakeholders (see Question Q.033).

Meaningful milestones for structuring an architecture evaluation project depend on the evaluation techniques used. The following milestones directly reflect the typical checks of our approach. However, a specific project setting could also be organized in two or three iterations in order to deliver first results of each check as early as possible and then refine them.

- Architecture drivers elicited
- Solution adequacy checked
- Documentation quality checked
- Architecture compliance checked
- Code quality checked.

Q.032. How to Interpret the Evaluation Results?

The interpretation is crucial for benefiting from the evaluation results. It is the bridge to recommendations and subsequent actions.

The results of an architecture evaluation have to be interpreted in light of the evaluation questions and the concrete context of the software system under evaluation. This interpretation is typically not easy and requires experience in architecting. Even when there are quantitative results (e.g., the number of architecture violations from compliance checks), the interpretation of the results remains a difficult step. Due to the nature of software architecture and software architecture evaluation methods, the evaluation results often cannot be fully objective and quantifiable. In the case of a solution adequacy check, rating the adequacy of an architecture for a set of architecture drivers or a single driver often does not lead to any quantification at all. The results are rather the identified risks, assumptions, limitations, and trade-offs, and the overall rating of adequacy.

It is very important for evaluators to manage the expectations of evaluation owners and stakeholders and to clearly communicate these. For instance, it is not possible to establish standard thresholds for the number of acceptable architecture violations. Rather, it is always necessary to keep the goals and the context of the evaluation owner in mind to answer the questions in a way that is most beneficial for him or her. Just providing facts and findings is not enough.

Q.033. How to Present Evaluation Results?

The presentation of results and recommendations to evaluation owners such as senior management and other stakeholders has to be done carefully in order to steer development activities into the right direction. To facilitate giving understandable presentations, we decided to depict the outcome using traffic light colors. This is done for all types of checks and different levels of detail: e.g., to show the adequacy of the architecture for single drivers but also aggregated for the complete check. From our experience, the following aspects should be covered in a result presentation (see more details in the upcoming chapters):

- Overview of the results to give an overall impression (see Fig. 4.2)
  (While this sounds too early, our experience is that the audience wants to know what is going on directly in the meeting. Then it is easier to listen to the details and to see how they contribute to the big picture. We are not telling a story that has to be thrilling—rather we give our stakeholders the information they asked for in the way that helps them to understand it best.)
- Evaluation goals and questions
- Evaluation approach

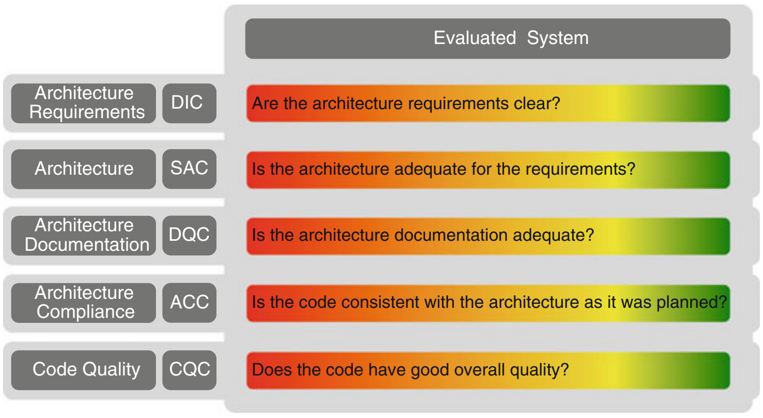

Evaluated System
Architecture Requirements
DIC
Are the architecture requirements clear?
Architecture
SAC
Is the architecture adequate for the requirements?
Architecture Documentation
DQC
Is the architecture documentation adequate?
Architecture Compliance
ACC
Is the code consistent with the architecture as it was planned?
Code Quality
CQC
Does the code have good overall quality?

Fig. 4.2 Evaluation result overview template (colour figure online). © Fraunhofer IESE (2015)

- Involved stakeholders/persons
- Inputs used and outputs produced
- Architecture drivers evaluated (overview, examples, prioritization)
- Evaluation results (if applicable)
  - Driver integrity check results
  - Solution adequacy check results
  - Documentation check results
  - Compliance check results
  - Code metrics results
- Rating of the severity of the situation
- Information on the confidence level achieved and potential needs for actions to get more confidence
- Recommendations derived from the evaluation results
  - Clustered along areas of recommended actions
  - Very rough estimates on effort needed for the recommendations.

4.2 What Mistakes Are Frequently Made in Practice?

Having no clear goals for an architecture evaluation.

Having no clear goals for evaluating an architecture might lead to severe drawbacks. Effort might be spent on the wrong concerns; too much effort might be spent on the design and on decisions that do not require that much attention (analysis paralysis) or too little effort might be spent on items that would require more confidence (underestimation). Always start with a clear and approved formulation of the evaluation goals.

→ Question Q.025.

Having no systematic approach for an architecture evaluation.

Having no or no systematic approach for an architecture evaluation may cause additional effort due to a lack of approach maturity. Furthermore, ad hoc approaches may cause omissions in the conduction of the checks and may lead to limited exploitation of the results.

→ Questions Q.015, Q.017, Q.024, Q.031.

Selecting the wrong evaluation setup for the concrete context.

The selection of an evaluator and adequate setup of the evaluation approach are decisive factors for the success of an architecture evaluation. Only if the evaluation approach is adequate for the goals and the required confidence can the project deliver the necessary answers. Evaluators have to have the necessary expertise, availability, and neutrality.

→ Questions Q.024, Q.026.

Conducting the architecture evaluation without the necessary strictness.

If an architecture evaluation project is not given the necessary attention, it will nearly always be overruled by other important activities. If the project is not strictly planned upfront with clear milestones and meeting dates, important stakeholders will not be available and the project cannot proceed.

An architecture evaluation project needs the attention and commitment of the evaluation owner and the stakeholders to be involved. The project needs careful planning (and re-planning) and strictness regarding its conduction.

→ Questions Q.024, Q.026, Q.029, Q.031.

Inadequately interpreting the evaluation results.

Interpreting the results of an architecture evaluation is not an easy task. Whether a solution for requirements is adequate or whether the trade-offs made are acceptable cannot be judged objectively. Thus, the interpretation has to take many situational factors as well as the involved people into account. Even when quantitative data is available, such as from a compliance analysis, the interpretation of severity is not straightforward.

→ Questions Q.032, Q.033.

Inadequately presenting the evaluation results to the stakeholders.

An architecture evaluation often has to serve as input for decision makers who do not know about all the technical details of an architecture. On the other hand, an architecture evaluation often has to look at all the technical details in order to come up with fact-based results. Thus, it is very important to elaborate and present the results in a form that is adequate for the audience.

→ Question Q.033.

Evaluating one's own architecture on-the-fly.

Human beings tend to overlook problems in their own work. Thus, they often go over it very quickly as they assume all the details are clear. This often leads to superficial architecture evaluations that do not reveal severe problems.

→ Questions Q.006, Q.017.

Prophet in one's own country syndrome.

Performing an architecture evaluation with internal auditors may lead to the problem that nobody listens to their results (or does not listen anymore)—they are perceived as a prophet in their own country. In such cases, it might be an option to involve an external party or to consider our experience in convincing management.

→ Questions Q.006, Q.017, and Chap. 11.

Part II
How to Evaluate Architectures Effectively and Efficiently?

5 How to Perform the Driver Integrity Check (DIC)?

The goal of the Driver Integrity Check (DIC) is to get confidence that an architecture is built based on a set of architecture drivers that is agreed upon among the stakeholders. We will show how to work with stakeholders and how to reveal unclear architecture drivers or those on which no agreement exists. Architecture drivers are expressed using the well-known architecture scenarios. The activity is based on classical requirements engineering aimed at compensating for not elicited requirements and aggregating a large set of requirements into a manageable set for an architecture evaluation (Fig. 5.1).

5.1 What Is the Point?

Q.034. What Is the DIC (Driver Integrity Check)?

Stakeholders of the software system under evaluation have concerns. Concerns can be considered a moving target: they are influenced by the current stakeholders' perceptions of the software system as well as by other factors and by experiences stakeholders make; they change over time; and their priority might be different at different points in time. Most importantly, there are always many stakeholders, with different concerns not (yet) aligned and potentially resulting in a conflict of interest.

Concerns shape the product and drive the architecting activities in general, and are thus crucial for evaluating the architecture, too. The objective of the DIC is to set the goals for the architecture evaluation. This means for the assessor to elaborate with all relevant stakeholders on the questions that are critical to them at a given moment in time. Investigating these concerns in detail and delivering informed, fact-based, and well-grounded responses to the questions is the key to being successful with architecture evaluations. The objective of the DIC is to set the focus in

J. Knodel and M. Naab, *Pragmatic Evaluation of Software Architectures*,
The Fraunhofer IESE Series on Software and Systems Engineering,
DOI 10.1007/978-3-319-34177-4_5

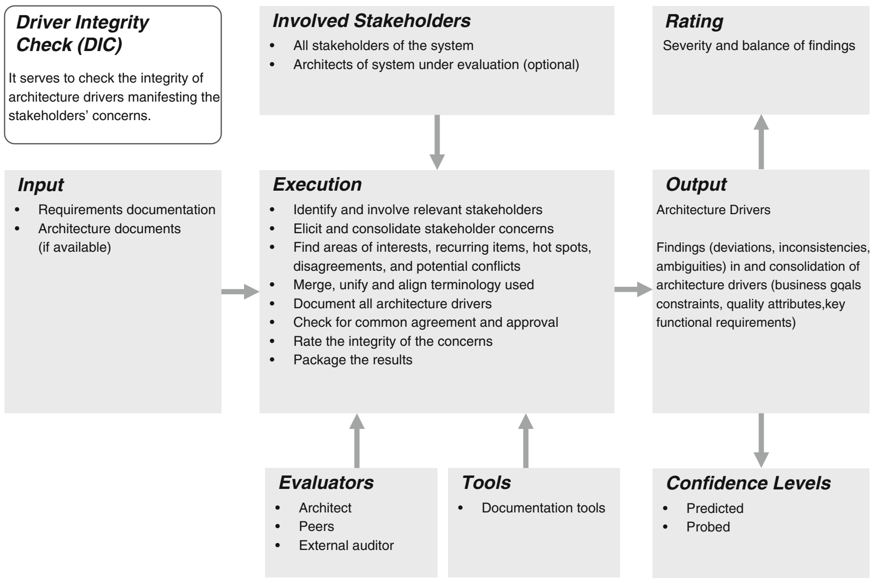

Driver Integrity Check (DIC)
It serves to check the integrity of architecture drivers manifesting the stakeholders' concerns.
Involved Stakeholders
• All stakeholders of the system
• Architects of system under evaluation (optional)
Rating
Severity and balance of findings
Input
• Requirements documentation
• Architecture documents (if available)
Execution
• Identify and involve relevant stakeholders
• Elicit and consolidate stakeholder concerns
• Find areas of interests, recurring items, hot spots, disagreements, and potential conflicts
• Merge, unify and align terminology used
• Document all architecture drivers
• Check for common agreement and approval
• Rate the integrity of the concerns
• Package the results
Output
Architecture Drivers
Findings (deviations, inconsistencies, ambiguities) in and consolidation of architecture drivers (business goals constraints, quality attributes,key functional requirements)
Evaluators
• Architect
• Peers
• External auditor
Tools
• Documentation tools
Confidence Levels
• Predicted
• Probed

Fig. 5.1 DIC overview

the jungle of stakeholder concerns and requirements. Having a clear view of the evaluation goals (i.e., the underlying questions that cause trouble to the stakeholders or make them feel uneasy) turns architecture evaluation into a worthwhile exercise. This goal orientation helps to make the evaluation effective and efficient and allows focusing on the most pressing issues in subsequent checks.

To achieve alignment among assessors and stakeholders, the concerns are formalized into architecture drivers. We apply a template for documenting such drivers and use them as a basis for discussions with the stakeholders. We recommend proceeding with an architecture evaluation only if there is agreement on the architecture drivers.

Ideally, the execution of a DIC would be straightforward: inspect the documentation of requirements and architecture, distill the set of architecture-relevant drivers currently of interest for the architecture evaluation, make sure they are still valid and up-to-date, and, last but not least, achieve agreement on the drivers among the stakeholders. However, in practice this process is far more complex. Documentation does not exist (anymore), is not up-to-date (was written years ago and has not been updated since), or its level of detail is not appropriate, ranging for instance between the extremes of being too little (e.g., three pages) or too much (e.g., thousands of pages). Moreover, even more challenging is the fact that stakeholders typically have different concerns and opinions about the priority of architecture drivers.

The DIC serves to deliver explicit and approved input for the subsequent checks. Consequently, we need to elicit from the stakeholders what their current concerns for the software system under evaluation are. Such concerns may relate to technologies, migration paths, quality attributes, key functional requirements, and constraints. In detail, the DIC aims at:

- **Compensation** for missing or yet unknown requirements for the software system under evaluation, and in particular the analysis of complex exceptional requirements that may be underrepresented in the requirements documentation. Here the DIC draws attention to concerns that are important for the architecture evaluation.
- **Aggregation** of large numbers of similar (types of) or repeating requirements with little or no architecture relevance. Here the DIC raises the abstraction level and places emphasis on those concerns that cause an impact on the architecture of the software system.
- **Consolidation** of different stakeholder opinions and concerns (business vs. technical) and balancing the focus of the investments of the architecture evaluation between (1) clearing technical debt of the past, (2) resolving current challenges, and (3) anticipating and preparing for future changes/needs. Here the DIC places the focus and the priority on the most pressing concerns.
- **Negotiation** in case of conflicting interests among stakeholders or conflicting priorities of concerns [e.g., of external quality (run time) and internal quality (development time)]. Here the DIC aligns the conflicting stakeholder concerns and achieves shared awareness.

Q.035. Why Is the DIC Important?

The DIC is important because it delivers clear, explicit, and approved information about the areas of interest for the architecture evaluation at the time of the evaluation. As concerns drift over time and as an architecture evaluation is always performed relative to the concerns, it is crucial to perform the DIC before checking solution adequacy or architecture compliance. This enables efficient and effective use of the time and effort allotted to the evaluation.

The DIC sets the questions for further checks within the architecture evaluation. It elaborates and documents the questions at a given point in time and thus counteracts drift in stakeholders' concerns. It delivers a clear view on business goals (of the customer organization, the development organization, the operating organization), quality attributes of the software system (system in use or under development), key functional requirements (functions that constitute unique properties or that make the system viable), and constraints (organizational, legal, technical, or with respect to cost and time).

The DIC serves to clarify whether or not additional checks should be conducted, if and only if there is agreement about the stakeholders' concerns. If there is no agreement, we recommend reiterating over the concerns instead of wasting effort on evaluating aspects that may prove irrelevant or superfluous later on.

Q.036. How to Exploit the Results of the DIC?

The results of the DIC are documented and architecture drivers are agreed upon. Their main purpose is their use as input in subsequent checks of an architecture evaluation. In addition, the results might be used for designing an architecture (in case the driver has not been addressed yet), for raising awareness among all stakeholders involved regarding what is currently driving the architecture design, for detailing quality attributes by quantifying them, and for making different implicit assumptions explicit and thus discussable.

Moreover, the DIC serves to increase the overall agreement of the product's architecture-relevant requirements and may justify the need for architecture (evaluation) by revealing a large number of disagreements or architecture drivers still neglected at that point.

5.2 How Can I Do This Effectively and Efficiently?

Q.037. What Kind of Input Is Required for the DIC?

Inputs to the DIC are stakeholder information (if available), existing documentation (if available), and a template for documenting architecture drivers (mandatory).

- Stakeholder information provides input about the roles and responsibilities of the people in the organizations participating in the architecture evaluation. Knowing who is involved, who is doing what, and who is reporting to whom allows identifying stakeholders and thus enables eliciting their concerns.
- Existing documentation (documents and presentations about requirements, architecture, release plans, etc.) provides inputs in two ways. On the one hand, it is a viable source of information for extracting concerns about the architecture under evaluation, and on the other hand, it enables the assessors to prepare for the evaluation by getting familiar with the software system under evaluation, domain-specific concepts, and, of course, the architectural design.
- A template for architecture drivers allows structured and formalized notation of the consolidated concerns. Please refer to the question about the output of the DIC for the template we recommend. The template may be customized and adopted to the concrete situation where it is used. The stakeholders should be informed about the structure and the content types of the template in order to be able to read and work with the template.

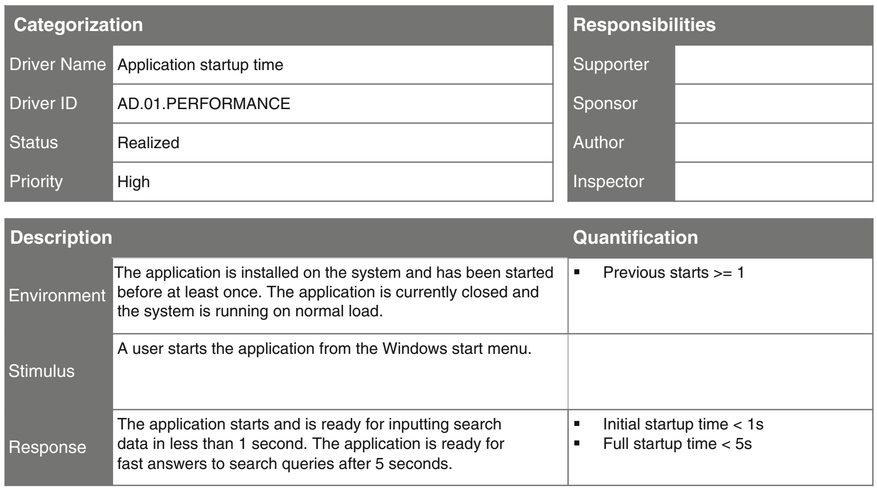
Q.038. How to Execute the DIC?

The DIC applies a structured approach consisting of the following steps:

- **Identify** the stakeholders who are relevant and important for the software system under evaluation.
- **Involve** the stakeholders and make sure that they are available during the DIC. If they are unavailable, the use of personas might help to guess architectural concerns (be aware of the risk of guessing wrong when using personas instead of talking to the real stakeholders).
- **Elicit** stakeholder concerns for each stakeholder identified in workshops, face-to-face interviews (of single persons or a group), or video or phone conferences (whatever is the most applicable instrument for elicitation in the given context of the evaluation).
- **Consolidate** the stakeholders' concerns over all stakeholder interviews. Find areas of interests, recurring items, hot spots, disagreements, and potential conflicts. Merge, unify, and align the terminology used.
- **Document** all architecture drivers using the template. Please refer to the question about the output of the DIC for the template we recommend.
- **Check** for common agreement and approval on architecture drivers by offering them for review. Discuss feedback with the stakeholders and mediate in case of conflicts. Be a neutral moderator of the discussion. Raise attention to potential trade-offs that might be acceptable for all involved stakeholders. Achieve agreement on the priorities of the individual architecture drivers.
- **Refine** the documentation of the architecture drivers and make amendments, if necessary. Iterate over the updated set of architecture drivers, if necessary, and check again for agreement.
- **Rate** the integrity of the concerns (please refer to the question about how to rate the results of a DIC).
- **Package** the results of the DIC and report the findings to the owner of the architecture evaluation to get a decision on whether or not to continue with subsequent checks.

Q.039. What Kind of Output Is Expected from the DIC?

The output of a DIC is a set of prioritized architecture drivers. Such drivers are documented using templates [for instance, see Fig. 5.2, adapted from the architecture scenario template of (Clements et al. 2001)]. The template consists of a set of fields for organizing and tracking information and the actual content. Please note that the template is mainly to be seen as a form of support. It does not have to be followed strictly. We often note down drivers merely as a structured sequence of



Categorization		Responsibilities	
Driver Name	Application startup time	Supporter	
Driver ID	AD.01.PERFORMANCE	Sponsor	
Status	Realized	Author	
Priority	High	Inspector	

Description		Quantification
Environment	The application is installed on the system and has been started before at least once. The application is currently closed and the system is running on normal load.	▪ Previous starts >= 1
Stimulus	A user starts the application from the Windows start menu.	
Response	The application starts and is ready for inputting search data in less than 1 second. The application is ready for fast answers to search queries after 5 seconds.	▪ Initial startup time < 1s ▪ Full startup time < 5s

Fig. 5.2 DIC example result. © Fraunhofer IESE (2011)

sentences (oriented along the content of the template). This is easier to write (given enough experience) and easier to read. We separated the field Quantification in order to remind writers of scenarios that quantification is very helpful in creating precise architecture drivers.

- *ID* and a representative *Name* identify an architecture driver.
- *Status* indicates the current processing status of an architecture driver (e.g., elicited, accepted, rejected, designed for, implemented, evaluated).
- *Responsibilities* can assign several responsibilities around the scenario to concrete stakeholders (e.g., the persons who up brought the driver and support it, are financially responsible and sponsor it, wrote the driver down, or evaluated the architecture with respect to the driver). This can be adapted individually to the needs in a concrete project.
- *Environment* describes the concrete context in which the architecture driver is relevant and where the stimulus arrives. If possible, provide quantifications.
- *Stimulus* describes a certain situation that happens to the system, respectively the architecture, and which requires a certain response. If possible, provide quantifications. The stimulus can arrive in the running system, e.g. in the form of user input or the failure of some hardware, or the stimulus can arrive in the system under development, e.g. in the form of a change request. If possible, provide quantifications.
- *Response* describes the expected response of the system, respectively the architecture, when the stimulus arrives. If possible, provide quantifications.

Q.040. What Do Example Results of the DIC Look like?

Figure 5.2 depicts the documentation of a sample architecture driver using the template described above.

Q.041. How to Rate the Results of the DIC?

We rate the driver integrity for each architecture driver derived. All findings (i.e., disagreements, deviations, inconsistencies, ambiguities) are considered in total and then aggregated by assigning values on the two four-point scales (severity of the findings and balance of the findings). The higher the score, the better the degree of driver integrity for the architecture driver.

The combination of both scales (i.e., the mathematical product of the two factors) determines the overall driver integrity for each architecture driver:

- **N/A** means that the driver integrity of the architecture driver has not (yet) been checked.
- **NO Driver Integrity** means there is strong disagreement among the stakeholders (conflicting concerns or priorities), or between stakeholders' concerns and the architecture driver specified by the assessor.
- **PARTIAL Driver Integrity** means that the architecture driver consolidates the stakeholders' concerns to some extent, but that parts of the driver need further elaboration before getting approval from the stakeholders.
- **LARGE Driver Integrity** means that the stakeholders have no major objections and approve the architecture driver in principle; some details may require further refinement or elaboration.
- **FULL Driver Integrity** means there is shared agreement among stakeholders and assessors about the architecture driver and the driver has been approved by the stakeholders.

Q.042. What Are the Confidence Levels in a DIC?

The procedures of a DIC ideally result in agreement about the relevant, critical, and important drivers of the software system under evaluation. If no agreement is reached and depending on the criticality of the driver, it might be necessary to invest into additional means to predict or probe the driver with a prototype to make sure that the stakeholders share the same understanding regarding what the software system shall achieve. Creating such prototypes consumes significant more effort than just inspecting, but delivers higher confidence. Figure 5.3 schematically depicts the confidence levels for the DIC.

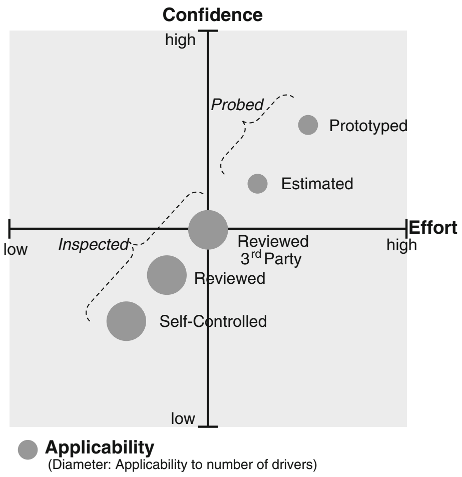


Fig. 5.3 DIC confidence levels. © Fraunhofer IESE (2015)

Q.043. What to Do with the Findings of the DIC?

The findings of a DIC consist of a list of open issues that require the stakeholders' attention, clarification, or conflict resolution. We recommend revising the architecture drivers until the conflicts have been resolved before conducting other checks. For conflict resolution between stakeholder parties, the following strategies may apply:

- **Convincing**: one party convinces the other; both parties eventually agree.
- **Compromising**: new alternative or trade-off is accepted by the parties in conflict.
- **Voting**: the alternative with the most votes by all stakeholders involved wins over the other options.
- **Variants**: conflict is not resolved, but different variants co-exist (parameterized) and all variants eventually get evaluated separately.
- **Overruling**: a (third) party with a higher organizational rank decides and enforces the decision over the conflicting party.
- **Deferring**: decision-making is delayed and the conflicted architecture driver will not be evaluated further (for the time being).

The documentation of the architecture drivers is considered a living work product, which is updated as soon as new drivers emerge, the results of the DIC are compiled, or the findings of the DIC are addressed.

Q.044. What Kind of Tool Support Exists for the DIC?

Performing the DIC mainly comprises manual activities to be performed by the assessors. Only the documentation of the architecture drivers can be supported by tools. Here we use the tooling that is already in place at the company, which ranges from modeling tools (capturing the drivers as first-class model elements and using the template as part of the description of the model element), office tools (adopting the template in documents, slide sets, or spreadsheets), or wikis (adopting the templates in dedicated pages).

Other than that, no special tools are available for the DIC, except for general-purpose tools for sharing and version management.

Q.045. What Are the Scaling Factors for the DIC?

Scaling factors that increase the effort and time required for performing a DIC include:

- Number of organizations involved
- Distribution of organization(s)
- Number of stakeholders involved
- Number of evaluation goals
- Size of the software system
- Criticality of the architecture evaluation.

5.3 What Mistakes Are Frequently Made in Practice?

Evaluating against unclear architecture drivers.

Architecture drivers are the foundation of the evaluation. In practice, architecture drivers are often not clear, not commonly agreed on, or they are too abstract to be useful for evaluation. We think that the DIC is one of the most crucial steps for making the overall architecture evaluation effective. The DIC provides a clear view on the most important concerns of the stakeholders and allows prioritizing.

→ Questions Q.035, Q.038 and Q.039.

Waiting too long to achieve driver integrity.

Sometimes stakeholders and assessors have a hard time achieving agreement on particular architecture drivers, or conflict resolution strategies consume too much time. Do not wait too long to achieve driver integrity. Defer the driver until agreement has been reached, but continue doing subsequent checks of other drivers (where driver integrity has already been achieved). It is better to start off with 80 % of the architecture drivers than to delay the entire architecture evaluation endeavor until perfection has been reached. In addition, in most cases, the Pareto principle applies here, too.

→ Questions Q.098 and Q.102.

How to Perform the Solution Adequacy Check (SAC)? 6

The main goal of the Solution Adequacy Check (SAC) is to check whether the architecture solutions at hand are adequate for the architecture drivers identified and whether there is enough confidence in the adequacy. We present a pragmatic workshop-based approach that is based on ideas of ATAM. We provide guidance for the documentation of evaluation results, such as the discussed architecture decisions and how they impact the adequacy of the overall solution. We also provide concrete templates and examples and show how evaluation results and specific findings can be rated and represented (Fig. 6.1).

6.1 What Is the Point?

Q.046. What Is the SAC (Solution Adequacy Check)?

There is no good or bad architecture—an architecture always has to be adequate for the specific requirements of the system at hand. Checking this adequacy is exactly the mission of the SAC. It is performed in nearly all architecture evaluation projects and is often used synonymously with architecture evaluation.

The SAC requires a sound set of architecture drivers as input, as generated by the DIC. The architecture drivers (often represented as architecture scenarios) can be used to structure the SAC: For each architecture driver, an independent SAC is possible, the results of which can be aggregated into the overall result.

The SAC works across "two worlds": requirements in the problem space and architecture in the solution space. There is no natural traceability relation between requirements and architecture. Rather, architectural decisions are creative solutions, which are often based on best practices and experiences, but sometimes require completely new approaches. This has an impact on the solution adequacy check: It

J. Knodel and M. Naab, *Pragmatic Evaluation of Software Architectures*,
The Fraunhofer IESE Series on Software and Systems Engineering,
DOI 10.1007/978-3-319-34177-4_6

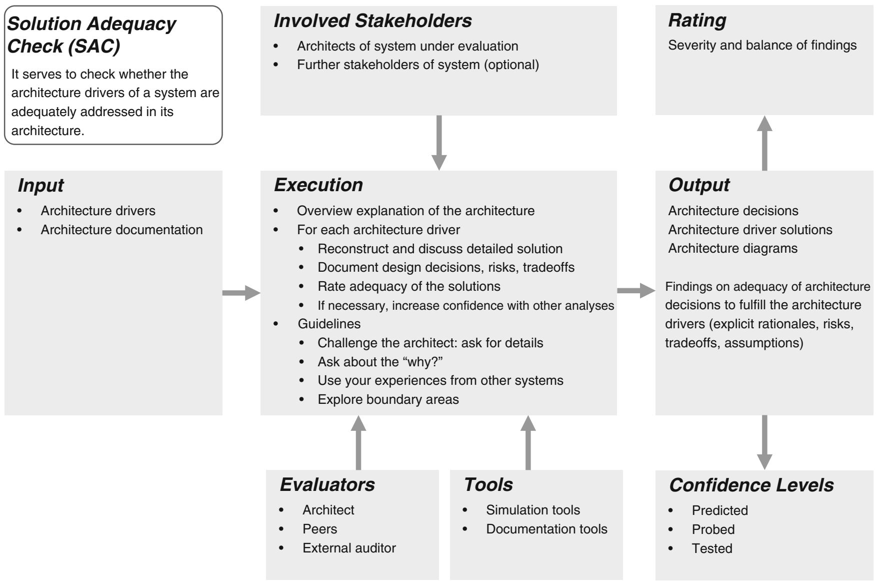

Solution Adequacy Check (SAC)
It serves to check whether the architecture drivers of a system are adequately addressed in its architecture.
Involved Stakeholders
Architects of system under evaluation
Further stakeholders of system (optional)
Rating
Severity and balance of findings
Input
Architecture drivers
Architecture documentation
Execution
Overview explanation of the architecture
For each architecture driver
Reconstruct and discuss detailed solution
Document design decisions, risks, tradeoffs
Rate adequacy of the solutions
If necessary, increase confidence with other analyses
Guidelines
Challenge the architect: ask for details
Ask about the "why?"
Use your experiences from other systems
Explore boundary areas
Output
Architecture decisions
Architecture driver solutions
Architecture diagrams
Findings on adequacy of architecture decisions to fulfill the architecture drivers (explicit rationales, risks, tradeoffs, assumptions)
Evaluators
Architect
Peers
External auditor
Tools
Simulation tools
Documentation tools
Confidence Levels
Predicted
Probed
Tested

Fig. 6.1 SAC overview

offers limited opportunities for direct tool-supported analyses and is rather an expert-based activity.

The goal of the SAC is to get the confidence that the solutions are adequate. As architecture is always an abstraction, it typically does not allow for ultra-precise results. Thus, it should be made clear throughout an architecture evaluation which level of confidence needs to be achieved and what this means in terms of investment into evaluation activities. Confidence levels are no exactly (pre-) defined levels; rather, their aim is to establish a common understanding of the confidence that needs to be obtained or has been obtained regarding an architecture evaluation.

More clarification is needed regarding what talking about the adequacy of "an architecture" means:

- An **architecture is not a monolithic** thing: It consists of many architecture decisions that together form the architecture. In the SAC, architecture drivers and architecture decisions are correlated. An architecture decision can support an architecture driver; it can adversely impact the driver; or it can be unrelated.
- The SAC is done to support **decisions about the future**. This can mean that only an architecture (or parts of it) has been designed and it should be made sure that the architecture is appropriate before investing into the implementation. This can also mean that a system is already implemented, for example by a third-party provider, and it should be made sure that the system fits the current and future requirements of a company. Some properties of a system, such as its performance (in particular its response time), can be judged well by looking at

the running system, but only if tests can be conducted representing all relevant parameters. Other quality attributes such as development time quality attributes can be judged much better by evaluating the architecture of a system. Whenever it is not possible to observe properties in the running system or in local parts of the implementation, architecture becomes the means to provide the right abstractions for evaluating system properties.

- Looking at the lifecycle of a software system, the **architecture can mean different things**: If the system is not implemented yet, it most likely means the blueprint for building the system. If the system is already implemented, it can mean the decisions as manifested in the code, the original blueprint from the construction phase, or a recently updated blueprint. Which architecture to take as the foundation for the SAC depends on the concrete context of the evaluation project and on the evaluation goals.

Q.047. Why Is the SAC Important?

The main goal of checking the adequacy of architectural solutions is to avoid investing a lot of implementation effort until it can be determined whether the architectural solutions are really adequate. In that sense, the SAC is in investment made to predict at a higher level of abstraction (predicting at the architecture level instead of testing at the implementation level) whether certain solutions are really adequate. The SAC can thus support questions from many levels: business-level questions with far-reaching effects as well as rather low-level technological questions. The SAC can be seen as a risk management activity (in particular the identification of risks arising from wrong architectural decisions or architectural mismatches).

Additionally, the SAC can provide further benefits:

- Revealing inadequacies that did not exist in earlier times but that arose due to architecture drivers changing over time
- Making implicit decisions and trade-offs clear and known to everybody (often, decision-making is rather implicit and the consequences are not considered so much)
- Revealing aspects that were not considered well enough: Where are gaps in the argumentation; which decisions are not thoroughly considered?
- Increasing awareness of and communication about the architecture in a software company
- Increasing the architectural knowledge of the involved persons.

Q.048. How to Exploit the Results of the SAC?

The results of the SAC are mainly the basis for well-founded decisions. Which decisions to make depends on the evaluation questions that triggered the architecture evaluation. Potential decisions could be: (1) The architecture is a sound basis for the future and should be realized as planned. (2) The architecture is not adequate and has to be reworked. (3) The architecture does not provide enough information, thus the level of confidence achieved is not high enough. More focused work has to be spent to create the required confidence.

Another stream of exploitation is based on the further benefits described in the previous section: exploiting the improved knowledge and communication regarding the architecture in order to achieve higher quality of the products.

6.2 How Can I Do This Effectively and Efficiently?

Q.049. What Kind of Input Is Required for the SAC?

Key inputs for the SAC are:

- **Architecture drivers**: They are typically the output of the DIC, but may already be available from other requirements engineering activities. One useful form of representation are architecture scenarios.
- **Architecture under evaluation**: This architecture may be more or less accessible: Sometimes the architecture is already made explicit in terms of models and/or documents, sometimes it is completely implicit or only in the minds of people. In order to be able to assess the adequacy of an architecture, it has to be explicit. Thus, many architecture evaluation projects have to include reconstruction activities, which extract architectural information from the source code or from people's minds. In our experience, there was not a single project that provided architecture documentation which was sufficient for directly performing a solution adequacy check. In practical terms, this means that some rough reconstruction of the key architectural aspects must be performed upfront and that the details must be reconstructed when discussing how architecture scenarios are fulfilled.

Q.050. How to Execute the SAC?

ATAM (Architecture Tradeoff Analysis Method) (Clements et al. 2001) is probably the best-known method for solution adequacy checks. It describes in great detail how to collect and prioritize architecture drivers and how to evaluate an architecture against them. In particular, it also gives detailed advice on how to organize an architecture evaluation and which organizational steps to propose. We recommend the book on ATAM for the details; here, we will only provide some brief guidance and experiences for conducting the SAC. Our proposed approach is less strict than ATAM in several aspects in order to react to constraints that we often encountered in practice:

- We do not require all the stakeholders to attend all the time (although this would often be useful).
- We do not require the architecture to be documented upfront (documentation was insufficient in almost all of our evaluation projects). Rather, we try to compensate for and reconstruct missing documentation in the evaluation.
- We simplify the process of eliciting the architecture drivers.
- We extended/slightly modified the template for describing how a certain architecture driver is addressed.
- We keep the workshops lightweight by not using the templates for drivers, decisions, and solutions in the workshops. In the workshops, facts are informally noted down by the evaluators and later consolidated in the respective templates. Doing differently distracts the workshop members from the real evaluation work and is not advisable.

The key idea behind techniques for the Solution Adequacy Check is to gain confidence in solutions by taking a detailed look at particular architecture drivers and to use the expertise of (external) people to assess the adequacy of the architecture.

An established way of organizing a Solution Adequacy Check is to conduct workshops with at least the following participants: (1) the people evaluating the architecture and (2) the architects who designed the system under evaluation. Additionally, further stakeholders can be valuable, in particular those who stated architecture drivers. These stakeholders often have experienced that certain solutions did not work and can thus help to reveal potential problems.

Figure 6.2 gives an overview of the procedure of an SAC evaluation workshop. At the beginning of the workshop, the architects of the system introduce the architecture

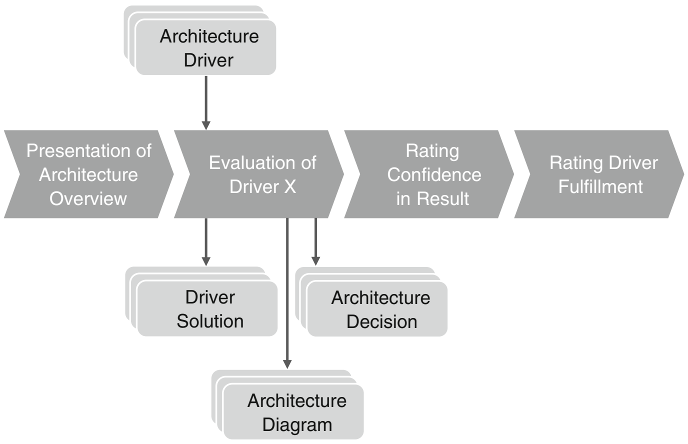


Fig. 6.2 SAC procedure. © Fraunhofer IESE (2014)

to the other participants to give an overview of the architecture under evaluation. This typically takes one to three hours. Depending on the availability and capability of the architects, this can be done with more or less guidance by the evaluators.

The basic procedure of a solution adequacy check workshop consists of going through the architecture drivers according to their priorities and discussing how adequate the architecture is in terms of fulfilling the respective driver. For every driver, the architectural decisions that are beneficial for the scenario or that hamper the fulfillment of the driver are discussed and documented. To reveal these decisions, the evaluators need their expertise to identify which architectural aspects are affected by a driver (data aspects, deployment aspects, …), and they need experience to judge whether a set of decisions would really fulfill the requirements. For each driver, all the decisions and their pros and cons are documented and it is described as a sequence of steps how the architecture or the system is achieving the fulfillment of the driver. Additionally, the reviewers have to maintain an overview of the discussion of other drivers and the respective architecture decisions, as these might also be relevant for other scenarios. More details about the produced outputs will be described later.

While it is important to maintain a very constructive and open atmosphere during an architecture evaluation, it is in the nature of this exploration to continuously challenge the architects by asking questions like:

- How did you address this particular aspect?
- What happens in this particular case?
- How does this relate to the decisions explained for the other scenario?
- Why did you design it like that and not the other way around?

The goal to keep in mind when asking such questions is: Is the architecture adequate? The evaluator has to judge this based on his experience, and it is not possible to provide clear guidelines regarding how to make this judgment. However, a good guideline for evaluators is the architecture decomposition framework (ACES-ADF) of Fraunhofer IESE (Keuler et al. 2011), as it provides a quick overview of relevant architectural aspects. Yet, not all the aspects of the ADF are relevant for the fulfillment of each scenario. For scenarios expressing runtime qualities, the runtime aspects in the ADF are more important, and the same is true for development time. Of course, there is always a close connection between runtime and development time, and quite often trade-offs can be identified between runtime scenarios and development time scenarios: optimizing for performance often adversely impacts maintainability and vice versa.

When discussing architecture drivers in a solution adequacy check, the first drivers take quite a long time (up to several hours) as many details of the overall architecture (in addition to the initial overview) have to be asked and explained. Later, evaluating the drivers becomes faster and finally it sometimes only takes minutes to refer to architecture decisions discussed before.

Q.051. What Kind of Output Is Expected from the SAC?

The output of the discussion of architecture scenarios is organized in three connected types of output (see Fig. 6.3, also showing relationships and cardinality). The evaluators consolidate the facts and findings of the workshop afterwards and use the templates to structure the information.

- **Architecture decisions**, documented in the **Decision Rationale Template** (see Fig. 6.4). Architecture decisions can be related to several architecture drivers and can positively and negatively impact these drivers. The template can also be used to document discarded decisions.
  - *ID* and a representative *Name* identify an architecture decision.
  - *Explanation* describes an architecture decision.
  - *Pros* summarize reasons in favor of the decision.
  - *Cons and Risks* summarize reasons that would rather speak against the decision.
  - *Assumptions* express what was assumed (but not definitely known) when making the decision.
  - *Trade-Offs* describe quality attributes, drivers, and other decisions that are competing with this decisions.
  - *Manifestation Links* are pointers to architecture diagrams, in which the architecture decision is manifested.

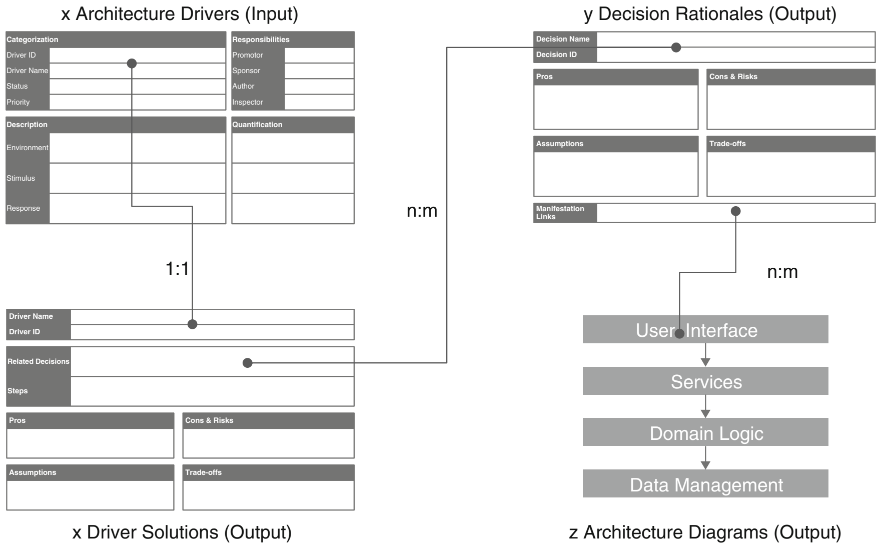
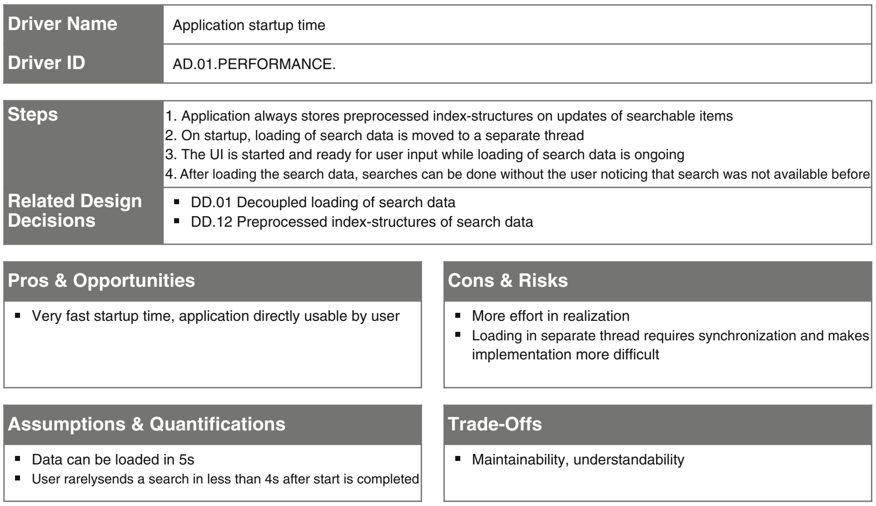


Fig. 6.3 Relationships—drivers, solutions, decisions, and diagrams. © Fraunhofer IESE (2012)



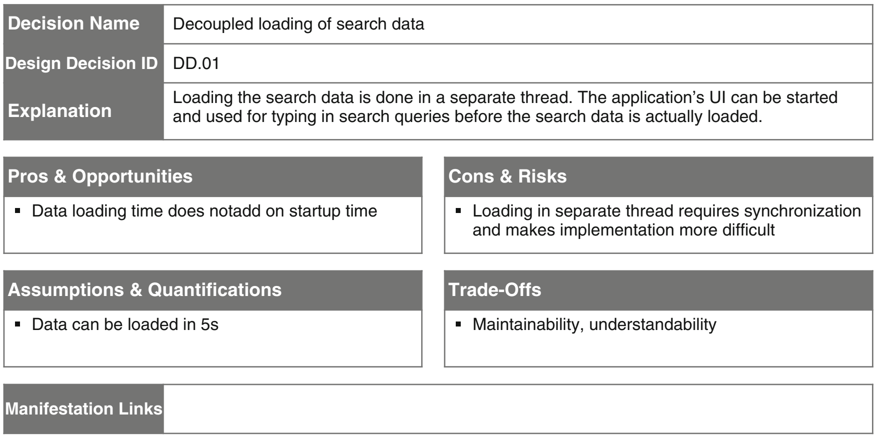
Driver Name	Application startup time
Driver ID	AD.01.PERFORMANCE.

Steps	1. Application always stores preprocessed index-structures on updates of searchable items 2. On startup, loading of search data is moved to a separate thread 3. The UI is started and ready for user input while loading of search data is ongoing 4. After loading the search data, searches can be done without the user noticing that search was not available before
Related Design Decisions	▪ DD.01 Decoupled loading of search data ▪ DD.12 Preprocessed index-structures of search data

Pros & Opportunities	Cons & Risks
▪ Very fast startup time, application directly usable by user	▪ More effort in realization ▪ Loading in separate thread requires synchronization and makes implementation more difficult

Assumptions & Quantifications	Trade-Offs
▪ Data can be loaded in 5s ▪ User rarelysends a search in less than 4s after start is completed	▪ Maintainability, understandability

Fig. 6.4 SAC example results—driver solution template. © Fraunhofer IESE (2012)



Decision Name	Decoupled loading of search data
Design Decision ID	DD.01
Explanation	Loading the search data is done in a separate thread. The application's UI can be started and used for typing in search queries before the search data is actually loaded.

Pros & Opportunities	Cons & Risks
▪ Data loading time does notadd on startup time	▪ Loading in separate thread requires synchronization and makes implementation more difficult

Assumptions & Quantifications	Trade-Offs
▪ Data can be loaded in 5s	▪ Maintainability, understandability

Manifestation Links	

Fig. 6.5 SAC example results—decision rationale template. © Fraunhofer IESE (2012)

- **Architecture driver solutions**, documented in the **Driver Solutions Template** (see Fig. 6.5). This summarizes and references everything that is relevant for the solution of a specific architecture driver.

  – *ID* and *Name* refer to the architecture driver that is being addressed.
  – *Related Decisions* refer to all architecture decisions that contribute to the architecture driver or adversely impact it.

  - *Steps* describes an abstract sequence of steps regarding the way the system and its architecture address the architecture driver if the related architecture decisions are used.
  - *Pros* summarize aspects that contribute to achieving the architecture driver.
  - *Cons and Risks* summarize aspects that adversely impact the achievement of the architecture driver or that are risky in terms of uncertainty.
  - *Assumptions* express what was assumed (but not definitely known) when making the decisions for addressing the architecture driver.
  - *Trade-offs* describe quality attributes, drivers, and other decisions that are competing with this architecture driver.

- **Architecture Diagrams**, documented in any convenient notation for architectural views. In architecture diagrams, architecture decisions are manifested and visualized.

Q.052. What Do Example Results of the SAC Look Like?

Figures 6.4 and 6.5 show examples of the driver solution template and of the decision rationale template for the architecture driver introduced in Fig. 5.2. According to Fig. 6.3, multiple architecture decisions might be related to the driver, but only one is fully described. Additionally, there might be architecture diagrams, which are not necessary for the examples shown.

Q.053. How to Rate the Results of the SAC?

We rate the solution adequacy for each architecture driver that is evaluated. All findings (i.e., risks, assumptions, trade-offs, missing confidence) are considered and then aggregated by assigning values on the two four-point scales (severity of the findings and nature of the findings). The higher the score, the better the solution adequacy for the architecture driver. The combination of both scales determines the overall solution adequacy for each architecture driver:

- **N/A** means that the solution of the architecture driver has not (yet) been checked. It can also mean that the check was not possible as the architecture driver was stated but not agreed upon.
- **NO Solution Adequacy** means there are major weaknesses in the solution or no solution may even be provided for the architecture driver.
- **PARTIAL Solution Adequacy** means that the architecture driver is addressed but there are still weaknesses and risks that require further clarification or architectural rework.

- **LARGE Solution Adequacy** means that the architecture driver is generally well addressed but with minor weaknesses or risks.
- **FULL Solution Adequacy** means there is confidence that the architecture driver is well addressed by the architecture decisions.

Q.054. What Are the Confidence Levels in an SAC?

While the evaluation procedure as described above provides important results with limited investment, it sometimes cannot provide the confidence that is needed for the results (Fig. 6.6). A good example are performance requirements: Imagine a system has to respond to every request within 0.1 s. The whole architecture is explained and for a reasonably complex system and known technologies you might have an idea whether the response time is realistic, but this is not a guarantee. However, if the system is not very simple, it is probably not possible to get high confidence that the response time is really achieved. In particular when new technologies come into play in which the architects do not have any experience yet, there is no chance to judge whether the requirements will be fulfilled. This is particularly true for performance requirements, but may also occur for other types of requirements.

In such cases, extended techniques for architecture evaluation are necessary. In the case of unknown technologies, prototyping should be used to gather first data about the properties of these technologies. This could mean building a skeleton of the system with a realization of the relevant architectural decisions in order to measure for example response times. Another possibility to gain more confidence are simulation-based approaches (e.g. Becker et al. 2009; Kuhn et al. 2013): Simulation is useful for expressing complex situations but always requires experience in the form of calibration data in order to align the simulation model with the real behavior of the resulting system.

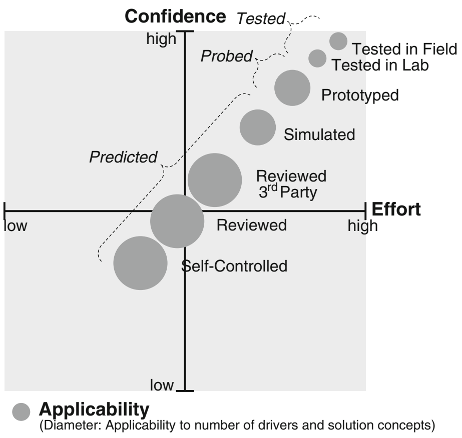


Fig. 6.6 SAC confidence levels. © Fraunhofer IESE (2015)

Q.055. What Kind of Tool Support Exists for the SAC?

As described above, the real evaluation part of the SAC is a strongly expert-based activity. Thus, the only tool support for this type of activity can support the evaluating experts in organizing information and notes. During the evaluation workshop, tools are needed that allow very quick note-taking. These may be plain text tools or mind-mapping tools that allow structuring the gathered information very quickly. Office tools such as text processing and presentation tools are useful for structuring the consolidated information and sharing it with the stakeholders.

More sophisticated tools come into play when the level of confidence in the evaluation results has to be increased. Example tools are prediction and simulation tools for certain quality attributes [e.g., for performance and reliability (Becker et al. 2009)]. Another approach can be to quickly prototype architectural ideas, which can be supported by tools such as IDEs and cloud environments, which allow quick trials of new technologies.

Q.056. What Are the Scaling Factors for the SAC?

The key scaling factor for the SAC is the number of scenarios that are evaluated. The prioritization of the architecture drivers makes it clear that the scenarios with the highest priorities are checked first. Additionally, it can be agreed to select further scenarios mandatory for evaluation according to other criteria (e.g., a certain number of scenarios for development time quality attributes, whose priority might not be so high depending on the voting stakeholders).

Our approach is to determine a fixed time for the evaluation (1 or 2 workshop days have proven to be appropriate). As long as time is left, scenarios are discussed. In our experience, we managed to evaluate an average of 10–25 architecture drivers in one to two days.

Of course, this leads to a number of remaining scenarios that are not evaluated in detail. However, our experience shows that the first 10–25 evaluations of architecture drivers approximate the full evaluation result very well. Thus, we have pretty high confidence that after two workshop days, a summarizing evaluation result can be presented.

Further scaling factors that increase the effort and time required for performing the SAC include:

- Number of organizations involved
- Distribution of organization(s)
- Number of stakeholders involved
- Number of evaluation goals
- Size of the software system
- Criticality of the architecture evaluation.

Q.057. What Is the Relationship Between the SAC and Architecture Metrics?

Another, quite popular, aspect of architecture evaluation is to use architecture level metrics to assess the quality of the architecture (Koziolek 2011). Architecture metrics try to capture general rules of good design. For example, they measure aspects such as coupling and cohesion of modules. Although the name might suggest otherwise, architecture metrics are typically measured on the source code. Architecture is used as an abstraction of the source code, and thus mainly more abstract properties of modules and the relationships between modules are checked.

The key difference between architecture metrics and the SAC is that architecture metrics do not evaluate against a product-specific evaluation goal but against metric thresholds and interpretation guidelines, which have been determined before in other settings to express aspects of software quality. Architecture metrics are, like nearly all other metrics, typically measured with the help of tools. As they work on the source code, they have to deal with a large amount of information that needs to be processed.

6.3 What Mistakes Are Frequently Made in Practice?

Being too superficial in the evaluation.

Many architecture evaluation results turn out to be too superficial if one looks at the details. Some of the reasons for this are: missing experience of the evaluators, trying to be very polite and challenging the architects too little, not covering all necessary architectural aspects.

→ Questions Q.017, Q.051 and Q.054.

Distracting the architecture evaluation by focusing too much on templates.

Templates are helpful for presenting results in a structured form. But working with the templates in workshops where most of the participants are not used to the method and to the templates can be very distracting. We propose working as informal and focused on the real evaluation content as possible in the workshop. It is the duty of the evaluators to make sure that all the necessary content is discussed and collected. Then it can be persisted in structured templates afterwards.

→ Questions Q.051 and Q.052.

Losing the good atmosphere due to the evaluation.

Architecture evaluation benefits from an open and constructive atmosphere as the information has to be provided mainly by the architects of the evaluated system. Since architecture evaluations now and then originate from critical situations, there is the risk of losing the good atmosphere. It is the task of the evaluators to have a feeling for the criticality of the situation and to preserve the open and constructive climate.

→ Question Q.050.

Losing the overview over the number of drivers and decisions.

During the evaluation workshops, the evaluators have to maintain an overview over a large number of architecture drivers and decisions that contribute positively or negatively to the drivers. As stakeholders and architects rarely have time, evaluators cannot spend much time on writing and organizing notes. Rather they have to mentally organize all information almost on the fly and still maintain an overview of the previous discussion. This is particularly important in order to reveal inconsistencies in the discussion of architectural solutions for different architecture drivers ("This morning you said the communication protocol should work like this to achieve … Now you are saying …").

→ Questions Q.050, Q.051 and Q.052.

Improperly dealing with levels of confidence.

Mistakes happen in two directions: Overcautious people will sometimes try to get level of confidence that is too high and thus too costly for drivers where this is not necessary. This strongly increases the time and budget consumed for an architecture evaluation. On the other hand, people often fail to accept that in a scenario-based review, it might just not be possible to achieve the necessary level of confidence.

→ Question Q.054.

Replacing the SAC with automated measurement of architecture metrics.

Measuring architecture metrics and seeing this as sufficient for an architecture evaluation is tempting: It can be widely done with tool support and does not consume much expert time. Unfortunately, it does not tell much about the adequacy of an architecture for the architecture drivers. We strongly encourage everyone to clearly look at the evaluation goals and at the evaluation techniques that can be used to achieve them.

→ Questions Q.054, Q.055 and Q.096.

7 How to Perform the Documentation Quality Check (DQC)?

The main goal of the Documentation Quality Check (DQC) is to check how adequate the architecture documentation is for its audience and purposes. The evaluation checks both the content and the representation of the architecture documentation. Thinking from the perspective of the audience and considering the purposes of the documentation helps to rate the adequacy of the documentation. In addition, principles of good documentation, such as structure, uniformity, or traceability, can be checked. These principles are determined by the mental capabilities of the readers and can be used across domains and system types (Fig. 7.1).

7.1 What Is the Point?

Q.058. What Is the DQC (Documentation Quality Check)?

Architecture documentation is made for people, not for being processed by computers. Its intention is to describe aspects about a system that the code does not contain or expose. It offers targeted and quick access to the key ideas and decisions behind a system. Here, we will not provide profound insights into architecture documentation but refer the reader to the dedicated literature (Zörner 2015; Clements et al. 2010).

As for any documentation, two main aspects of architecture documentation should be checked in the DQC:

- **Content** of the architecture documentation
- **Representation** of the architecture documentation.

DOI 10.1007/978-3-319-34177-4_7

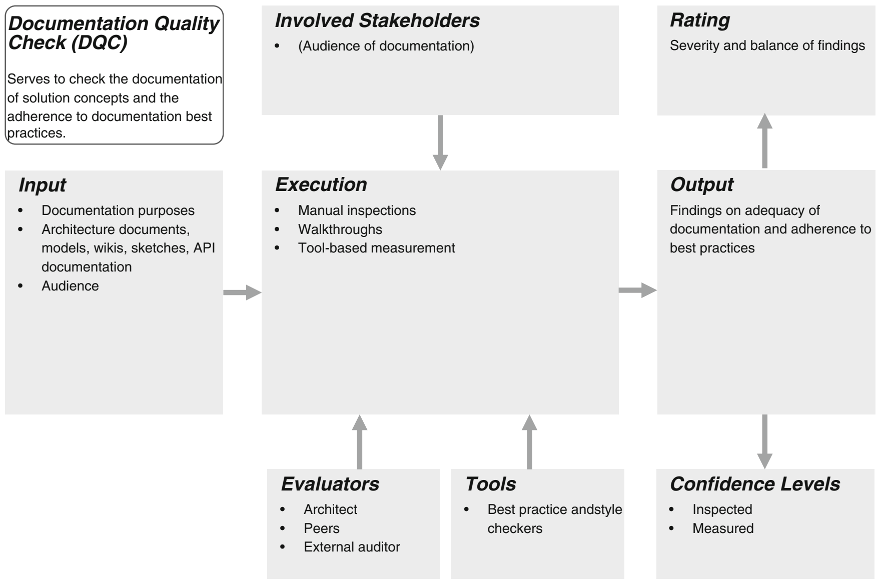

Documentation Quality Check (DQC)
Serves to check the documentation of solution concepts and the adherence to documentation best practices.
Involved Stakeholders
(Audience of documentation)
Rating
Severity and balance of findings
Input
Documentation purposes
Architecture documents, models, wikis, sketches, API documentation
Audience
Execution
Manual inspections
Walkthroughs
Tool-based measurement
Output
Findings on adequacy of documentation and adherence to best practices
Evaluators
Architect
Peers
External auditor
Tools
Best practice andstyle checkers
Confidence Levels
Inspected
Measured

Fig. 7.1 DQC overview

Regarding **content**, two things are important to consider as background information when checking architecture documentation quality:

- Who is the **audience**?
  The main audience are, of course, the developers. However, there are other important stakeholders of architecture documentation, such as management, marketing, and testers. It is fairly obvious that these target groups have different needs in terms of architecture documentation. These differences are related to content and focus, representation, and level of detail. Schulenklopper et al. (2015) describes with great examples how architecture documentation can be tailored to different audiences and supports the notion that it might pay off to invest into different representations of the architecture documentation, although this mostly means effort for manual creation.
  The audience can typically be characterized further and should be known to the architects writing the architecture documentation. For example, the need for architecture documentation depends on the knowledge of developers as one factor: Do they know the domain well? Do they know the architectural implications of the technologies used? Do they know other similar systems of the company? These characteristics might lead to strongly different needs for architecture documentation and an evaluator of the quality of this documentation has to take them into account.

- What is the **purpose** of the architecture documentation?
  One key purpose of architecture documentation is to convey the main ideas and concepts and to enable efficient and effective communication among stakeholders. Developers should get the right information to conduct their implementation tasks; testers should get information about things that are important to test and about their testability. Managers should understand the main risks and mitigation strategies and the implications of the architecture on schedules and on the staffing situation.

Regarding **representation**, general best practices of good documentation (Zakrzewski 2015) apply for architecture documentation as well. In the following, a selection of important best practices is presented:

- Adherence to best practices such as architecture documentation view frameworks [e.g., SEI viewtypes (Clements et al. 2010), 4 + 1 views (Kruchten 1995), arc42 (Starke and Hruschka 2015), Fraunhofer ACES-ADF (Keuler et al. 2011)].
- Internal and external consistency, uniformity: Is the information provided consistently across different areas, diagrams, naming of elements, and documents?
- Structuredness: Is the information structured in a form that supports the construction of a mental model and that breaks the information down in a meaningful way?
- Readability, Understandability, Memorability: Is the language understandable and adequate for the target audience? Is the document easy to read and follow? Are the diagrams well organized and are recurring aspects depicted in a similar layout? Are the diagrams clear and not overloaded?
- Completeness: Is the information complete in the sense that relevant questions can be answered by the document and the system can be understood based on the document?
- Adequate notation: Is the notation adequate for the intended audience? Management needs different notations and descriptions and another level of detail than developers.
- Traceability within and across documents: Can related aspects (and there are many relations in an architecture documentation) be identified and easily navigated to?
- Extensibility: Is the documentation created in a way that allows updating and distribution of new versions of the documentation (very important to have a low barrier for updating the documentation)?

Checking the quality of the documentation comes down to the question: Is the documentation adequate for the audience and their purposes? A given architecture documentation does not have to address all potential audiences and purposes. Rather, it should clearly show what it addresses and how. Reading the architecture documentation of real projects often exposes deficiencies because the writer (probably the architect) did not explicitly think about the audience and the purposes of the documentation. In such cases, the document is rather optimized from a writing perspective than from a reading perspective. It makes little sense to spend effort on such work.

Q.059. Why Is the DQC Important?

The DQC assures that the documentation of the architecture enables comprehension of the solution concepts. It determines whether or not readers will find the **right information** in order to gain knowledge, understand the context, perform problem-solving tasks, and share information about design decisions. Additionally, the documentation (reports, presentations, models) is inspected with respect to **information representation** criteria such as consistency, readability, structuredness, completeness, correctness, uniformity, extensibility, and traceability. The rules for high-quality information representation are mainly determined by human capabilities and limitations.

Missing quality in architecture documentation can lead to many problems, including:

- Communication problems in the team, as no uniform idea of the system exists
- More time required by developers to understand their tasks and the context of the tasks
- Difficulties for new developers to get a quick introduction to the system
- Lack of uniformity in the implementation
- Quality problems in the resulting system as solution concepts are not realized adequately
- Lack of possibilities to analyze the existing system and to plan necessary changes.

The good news with respect to problems with the quality of architecture documentation is that these problems can be overcome with relatively little cost if the architecture is known at least to an architect or developer. Compared to other problems found in architecture evaluations, low-quality or missing architecture documentation is something that is easy to fix. Many important things can be written down even in just one day, and within two or three weeks, a comprehensible architecture documentation can be re-documented, if missing. The effort for creation can be scaled quite well and can be dedicated to the most important information.

Q.060. How to Exploit the Results of the DQC?

The results of the DQC can be used directly to improve the documentation of a system. Additionally, it can improve the understanding of the audience, the purposes, and the resulting readers' needs. The DQC can improve and converge a development team's common understanding of good documentation.

7.2 How Can I Do This Effectively and Efficiently?

Q.061. What Kind of Input Is Required for the DQC?

The obvious input for the DQC is the architecture documentation to be checked. It can be provided in any form that is available and found in practice:

- Architecture documents
- Architecture models
- Architecture wikis
- Architecture sketches
- API documentation
- Etc.

Typically, such documentation artifacts can simply be collected. In systems with some history, there is no often uniform location and it may not even be clear which documents are up-to-date and which are not. This will lead to some more analysis on the hand; on the other hand, it is a direct finding of the DQC.

The other input that is mostly not so obvious is the clarification of the audience and the purpose of the architecture documentation. This has to be done by the evaluators in cooperation with the responsible architects and developers. If a system has undergone a certain history of development and maintenance, the current audience and purposes might also have drifted away from the initial ones. This is only natural as the system and its surrounding development activities change over time.

Q.062. How to Execute the DQC?

Checking whether the architecture documentation is adequate for its audience and purposes is a highly manual effort. It requires manual inspection by an experienced architect who can quickly understand the content of the documentation and can put himself into the role of the audience. Well-known inspection techniques such as perspective-based reading (Basili et al. 1996) can be applied.

An additional possibility is to conduct walkthroughs through the documentation with representatives of the documentation's audience and to conduct interviews with these representatives in case they already had to work with the documentation in the past.

To check adherence to best practices, checklists can be used. The list of characteristics found in Question Q.058 can be used as a starting point and can be refined if needed. In part, adherence to such best practices (e.g., traceability in case of well-structured models) can be supported with tools.

The amount of existing architecture documentation that can be used as input for the DQC strongly differs in practice. For many software systems, no or not much architecture documentation exists to be checked. At the other end of the spectrum, in particular in environments that are more restricted and require documentation for instance regarding safety regulations, large amounts of documentation exist. Most of the time, the goal of the DQC is not to identify every single deviation from a best practice; rather, the goal is to check the overall quality and to show significant findings with the help of examples. Thus, if the amount of documentation is too large, it might be sufficient to pick parts of the architecture documentation at random and to make sure that adequate coverage is achieved.

Q.063. What Kind of Output Is Expected from the DQC?

The DQC mainly provides qualitative findings related to the aspects described above:

- Adequacy to communicate architectural information to a certain audience for certain purposes
- Adherence to best practices that make the documentation understandable and memorable for the audience.

The output can be represented in the following way:

- Stating the general impression and supporting it with examples (findings across a wide range)
- Stating further findings that are rather exceptions or intensifications of the general impression (e.g., violations of best practices such as traceability, uniformity, etc.).

Q.064. What Do Example Results of the DQC Look Like?

These example results stem from a real architecture evaluation project and are anonymized. The stages of the rating are simplified to positive or negative findings only.

Positive findings:

- Overall, good and comprehensive documentation
- Extremely detailed, well-structured, well-described model
- Strong focus on functional decomposition
- Mapping to code: very clear on higher level
- Good support for concrete development tasks
- Extremely detailed

- Well maintained (regarding development, stabilization, maintenance)
- Well structured (regarding logical parts, architectural views, hierarchical decomposition, data objects, …)
- Good introductions and background descriptions
- Good explanations of diagrams and architecture elements
- Good coverage of architectural aspects in views: data, functions, behavior, deployment
- Diagrams: mostly good layout and easy to read.

Negative findings:

- Concrete architecture decisions are not made very explicit and thus the rationale is often not so clear
- Partially missing uniformity
- Less focus on architecture decisions and solutions for requirements
- Mapping to code: sometimes not so clear in details
- Difficult to understand the overall system
- Missing linkage between architectural solutions and architecture drivers
- Detailed diagrams are sometimes overloaded and hard to read
- Sometimes missing uniformity in the model structure (different substructures, different naming of packages).

Q.065. How to Rate the Results of the DQC?

The rating is done according to the scheme introduced in Sect. 3.1 for rating the nature and severity of the findings. The rating can be done for the complete architecture documentation or in a more fine-grained manner for single artifacts such as documents or models.

- **N/A** means that the documentation quality for a criterion has not (yet) been checked.
- **NO Documentation Quality** indicates that major problems with the architecture documentation have been found. Significant amounts of effort and strong rework of the documentation concept are necessary.
- **PARTIAL Documentation Quality** means that a substantial number of deficiencies has been found in the documentation. These deficiencies endanger the usefulness of the documentation and require significant improvement.
- **LARGE Documentation Quality** means that only manageable deficiencies have been identified. The existing anomalies should be addressed explicitly and the estimated effort for fixing these fits into the next evolution cycle.

- **FULL Documentation Quality** means no or only few weaknesses were found in the documentation. Overall, the documentation is well suited for its purposes and follows documentation best practices.

Q.066. What Are the Confidence Levels in a DQC?

According to the techniques for performing the DQC, measurement and inspection are the categories of the checks. Inspections can be performed by different evaluators as explained above, resulting in different levels of confidence. Adherence to best practices can generally be measured with little effort if the best practices can be described formally. However, most architecture documentations are not that formal and thus there is limited applicability, although the tools could be applied to a large number of documentations due to the high degree of automation (Fig. 7.2).

Q.067. What Kind of Tool Support Exists for the DQC?

The DQC is mainly a manual activity, in particular in terms of checking the adequacy of the architectural information provided to the audience for their purposes. When it comes to checking adherence to best practices that can be formalized (e.g., the maximum number of elements per diagram, the presence of traceability links between elements, etc.), tool support is possible. Such tool support is mainly

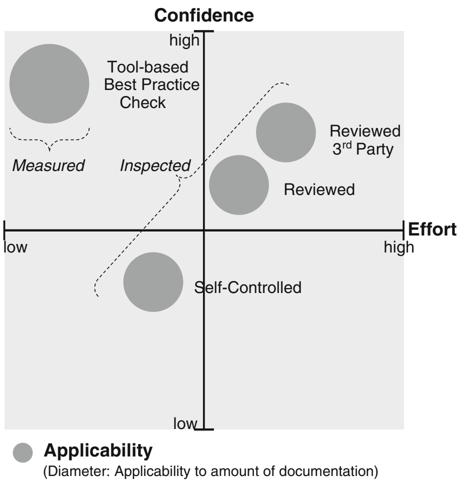


Fig. 7.2 DQC confidence levels. © Fraunhofer IESE (2015)

not available out of the box in architecture documentation tools. However, modeling tools often provide extensibility APIs that allow creating custom checks and searches that enable automatic measurement of deviations from the desired best practices.

Q.068. What Are the Scaling Factors for the DQC?

The most important scaling factor is the amount of architecture documentation available. This partly depends on the system's size and complexity, but mainly on the willingness and discipline of the architects creating the documentation. As there is no clear boundary between architecture and fine-grained design, there is also no such boundary in documentation. There is partial overlap and the DQC has to inspect both types of documentation, if they exist. If documents are extremely scattered or hard to find and to correlate, this also has an impact on how much of the documentation can be evaluated. If architecture models exist, their navigability and structuredness are important scaling factors. Finally, the experience of the evaluators in reading and evaluating documentation is a scaling factor that determines how much documentation can be checked.

Often the amount of architecture documentation is rather limited and does not cause problems for the DQC. If a huge amount of documentation is available, the key factor for scaling the effort for the DQC is to focus on the overview and then select further information at random, aiming at representative coverage.

7.3 What Mistakes Are Frequently Made in Practice?

Mixing up the results of the SAC and those of the DQC.

Checking the adequacy of architecture solutions and how they are documented are two fundamentally independent things. As architecture is intangible and mostly only accessible from people's minds or from documentation, the temptation exists to mix up the quality of the architecture and that of the architecture documentation.

→ Questions Q.051 and Q.063

Checking only documentation best practices and not the suitability for the audience.

As described in this chapter, the DQC has to consider the content of the architecture documentation and its representation. The content is determined by the audience and the purposes, while the representation can be guided by general best practices of good architecture documentation.

→ Questions Q.058, Q.062 and Q.063

8 How to Perform the Architecture Compliance Check (ACC)?

The main goal of the Architecture Compliance Check (ACC) is to check whether the implementation is consistent with the architecture as intended: only then do the architectural solutions provide any value. Nevertheless, implementation often drifts away from the intended architecture and in particular from the one that was documented. We will show typical architectural solutions that are well suited to being checked for compliance. Compliance checking has to deal with large amounts of code and thus benefits from automation with tools. Not all violations of architecture concepts have the same weight: we provide guidance for the interpretation of compliance checking results (Fig. 8.1).

8.1 What Is the Point?

Q.069. What Is the ACC (Architecture Compliance Check)?

The objective of architecture compliance checking is to reveal where the consistency between the solution concepts and the resulting source code is no longer given. We distinguish two kinds of violations of the intended architecture: structural violations exhibit a (part of a) solution concept that has a counterpart in the source code not realized as specified, whereas behavioral violations indicate the same in a running instance of the software system. Almost all implementations (at least those we analyzed in the past decade) exhibit significant structural or behavioral violations. The best solution concepts of the designed (intended) architecture do not help if they are not reflected properly in the source code (the implemented structural architecture) or the running system (the realized behavioral architecture). As architecture is an abstraction to allow making predictions about a software system,

J. Knodel and M. Naab, *Pragmatic Evaluation of Software Architectures*,
The Fraunhofer IESE Series on Software and Systems Engineering,
DOI 10.1007/978-3-319-34177-4_8

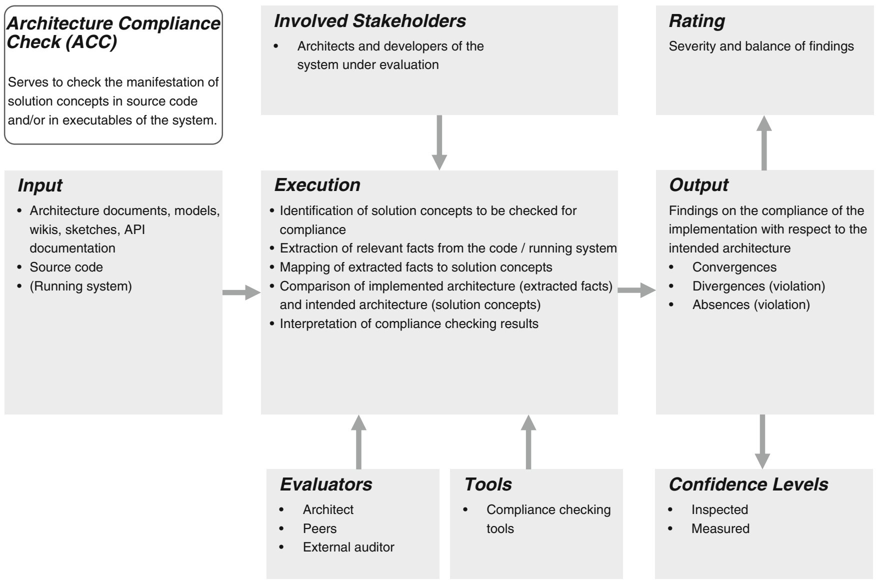

Architecture Compliance Check (ACC)
Serves to check the manifestation of solution concepts in source code and/or in executables of the system.
Involved Stakeholders
• Architects and developers of the system under evaluation
Rating
Severity and balance of findings
Input
• Architecture documents, models, wikis, sketches, API documentation
• Source code
• (Running system)
Execution
• Identification of solution concepts to be checked for compliance
• Extraction of relevant facts from the code / running system
• Mapping of extracted facts to solution concepts
• Comparison of implemented architecture (extracted facts) and intended architecture (solution concepts)
• Interpretation of compliance checking results
Output
Findings on the compliance of the implementation with respect to the intended architecture
• Convergences
• Divergences (violation)
• Absences (violation)
Evaluators
• Architect
• Peers
• External auditor
Tools
• Compliance checking tools
Confidence Levels
• Inspected
• Measured

Fig. 8.1 ACC overview

these predictions only have any value if the implemented system is built and behaves as prescribed by the architecture. In cases where coding fixes an inadequate architecture, the architecture documentation becomes useless if not updated and respective predictions have no value at all.

Thus, the goal of compliance checking is to check whether the architecture has been implemented in the source code as intended. The implemented architecture is not directly visible from the source code. Rather, it is typically buried deeply in the source code and has to be extracted by means of reverse engineering activities to collect facts about the system. As the architecture describes not only static artifacts of a system at runtime, it might not even be enough to extract information from the source code, but information might also be needed from the running system. Checking for other architectural aspects getting codified in some form (e.g., procedures for (continuous) building, linking, and testing the system, rules for configuration files and deployment descriptors, guidelines for using technologies) might be worthwhile, but may be effort-intensive and time-consuming as these typically lack tool support. Thus, considerations about the required level of confidence have to drive which and how much compliance checking is needed. If necessary and economically feasible, we construct additional checks for such cases. If and only if architectural concepts are realized compliantly, the architecture keeps its value as an abstraction used as a predictive and descriptive instrument in software engineering.

Q.070. Why Is the ACC Important?

The ACC and implementation evolve independently and at different speeds. Already during the initial system development and during maintenance there is the threat of having drift. Violations of the structure or behavior of the software system violate the intended solution concept defined at the architectural level. Reasons for having drift include: developers are working within a local, limited scope, while the architecture is balanced from a global viewpoint; time pressure on engineers; developers are not aware of the right implementation; violating the architecture is easier in individual cases; the architecture does not allow realizing a certain requirement or a requested change; technical limitations require violating the architecture.

An analysis of industrial practice covering various software systems distributed across diverse application domains such as embedded systems or information systems revealed that there was not even a single system that the developers implemented in full compliance with the architecture. On the contrary, all analyzed systems featured substantial structural violations (see Knodel et al. 2006; Lilienthal 2015). Other researchers confirm that the lack of compliance is a practical problem in industrial practice; for instance (see Murphy et al. 2001; Bourquin and Keller 2007; Rosik et al. 2008). However, not only industrial software systems lack compliance: open source software systems face the same problem. The most prominent example here is probably the Mozilla web browser, where Godfrey and Lee (2000) observed significant architecture decay within a relatively short lifetime; the browser was still under development after a complete redesign from scratch. Another prominent study is reported in Garlan and Ockerbloom (1995), where architectural mismatches resulted in a number of issues (e.g., excessive code, poor performance, need to modify external packages, need to reinvent existing functionality, unnecessarily complicated tools), which eventually hampered successful reuse of components. Further empirical studies show that lack of compliance negatively affects the effort for realizing evolutionary changes in a software system and the quality of such tasks (Knodel 2011).

Lack of compliance bears an inherent risk for the overall success of the development organization: The architecture as a communication, management, and decision vehicle for stakeholders becomes unreliable, delusive, and useless. Decisions made on the basis of the architecture are risky because it is unclear to which degree these abstractions are actually still valid in the source code. Hence, structural violations seriously undermine the value of the architecture. It is unclear whether the development organization will meet the essential demands of the requested functionality delivered while meeting effort, quality, and time constraints for the software system under development. Even worse is the long-term perspective during maintenance and evolution, which was already observed by Lehman and Belady (1985), who states that "an evolving program changes, its structure tends to become more complex". The source code surpasses innovations designed in terms of the architecture and can prevent their introduction. Because all decisions made to obtain the goals were derived from the architecture, the imperative need for architecture compliance becomes apparent.

Moreover, knowing about violations does not remove them from the source code. The later violations are revealed, the more difficult their removal. The development organization can decide between two fundamental options. None of them is really appealing to the development organization because each has substantial drawbacks:

- **Ignore the lack of compliance**: This option increases the technical debt for the development organization and has severe negative impacts, e.g., the architecture documentation becomes delusive and useless; it gets harder to meet quality goals not met yet; reuse of components may fail; and eventually projects may get canceled due to quality problems caused by or very negatively affected by architecture violations.
- **React to and repair lack of compliance**: This option requires unplanned, additional effort for fixing the architectural violations. For instance, tasks like analyzing violations [e.g., the inspection of six violations required four hours (see Lindvall et al. 2005)], communication (e.g., a single 2-day workshop with 10 engineers consumes 20 person-days), and coding projects for repairing architectural violations (e.g., approx. 6 person-months of effort spent to fix 1000 violations, assuming that fixing one distinct violation consumes roughly 1 h).

Q.071. How to Exploit the Results of the ACC?

In practice, the ACC often reveals a large number of architecture violations: in some of our evaluation projects, we found more than tens of thousands of violations! These architecture violations are, of course, not all different. They often follow similar deviation patterns, which became necessary due to a certain functionality that needed to be implemented, or they are the result of a developer not knowing the intended architecture.

Architecture violations may also require being treated in different ways in order to resolve them. Depending on the type of the violation, the architects and the engineer have to decide which of the violations require (1) refactoring in the small (a set of rather simple code changes to fix the violations), (2) refactoring in the large (larger and more complex restructuring of the source code to realize the intended solution concepts of the architecture compliantly), or which violations are (3) indicators of a systemic misunderstanding hampering the achievement of the architecture driver. The latter requires great dedicated effort for redesigning the architecture and fix the issue. Quite on the contrary, the results of the ACC might also lead to (4) changes in the architecture while the implementation remains the same. In cases of wrong assumptions or previously unknown technical constraints, the code proves the architecture to be wrong and reveals the need to adapt the model and documentation to the facts established by the implementation.

In both cases, we strongly recommend taking the initiative to ensure traceability between the architecture and the source code. Architectures have to be implemented as they were intended. Otherwise, their value disappears and causes technical debt, as mentioned above. Thus, the results of the ACC can be used directly to improve the implementation of a software system.

8.2 How Can I Do This Effectively and Efficiently?

Q.072. What Kind of Input Is Required for the ACC?

The inputs to architecture compliance checking depend on whether structural compliance checking or behavioral compliance checking is applied. The architecture (or rather the solution concepts) need to be evaluated as well as the respective counterparts in the software system, either the source code for structural checking or runtime traces for behavioral checking. Figure 8.2 depicts example and typical models of solution concepts (the arrows depict uses dependencies of modules): (a) depicts a typical technical layer structure; (b) depicts a recurring internal structure of a service; (c) depicts the separation of customizations, a shared core and framework; (d) depicts the organization of functionality in a system. Please note that these solution concepts can be orthogonal to each other and the implementation might have to comply with all of them at the same time. Further examples of inputs as well as psychological backgrounds can be found in Lilienthal (2015).

Q.073. How to Execute the ACC?

The ACC typically comprises the following steps:

- Identify and describe the architectural concepts, structures, or behavior that should be checked for compliance in the software system. This step is mainly driven by the evaluation question at hand and performed manually by the evaluator and the architect.

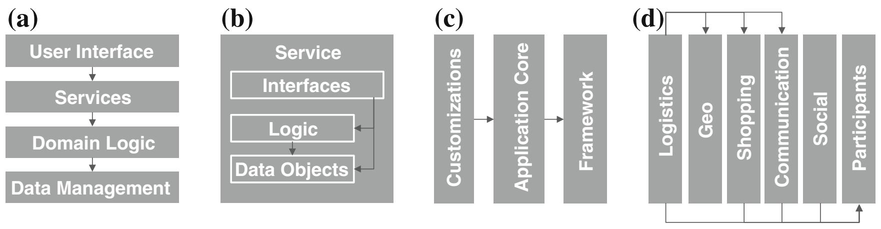


Fig. 8.2 ACC solution concept input examples. © Fraunhofer IESE (2013)

- Extract relevant facts from the source code or from runtime traces using reverse engineering or reconstruction techniques (typically performed with and only scaling due to tool support). It is important to tailor and scope the reverse engineering method and tools to the identified solution concepts in order to minimize the reverse engineering effort.
- Map the extracted facts to the elements of the solution concepts (done manually, typically with tool support) and lift elements to the same level of abstraction.
- Conduct the compliance checking, which will identify deviations between the intended architecture and the implemented architecture (these are called architecture violations).
- Interpret the compliance checking results: How many architecture violations were found? Can they be classified? What is their severity? How much effort is estimated to remove the architecture violations? Is it worthwhile removing the architecture violations?

The ACC is often performed in an iterative manner, starting with a high-level model and a coarse-grained mapping of model elements to code elements. The mapping sounds straightforward, but in fact is non-trivial. Especially for aged or eroded systems or in cases where the original architects and developers are no longer available, it can become a tedious task requiring many iterations. Often, there are also findings during these iterations that lead to an adjustment of the intended architecture, just because the realized architecture is more adequate and there has been no feedback from development to the original architecture documentation.

Q.074. What Kind of Output Is Expected from the ACC?

Architecture compliance is always measured based on two inputs, the intended architectural plan and the actual reality manifested in the software system. The output is a collection of so-called violations: violation is an architectural element or a relationship between elements that has a counterpart in the system artifacts (source code or running system), which is not realized as specified. From this definition, we can derive three distinct results types:

- **Convergence** is an element or a relation that is allowed or was implemented as intended. Convergences indicate compliance, i.e., the reality matches the plan.
- **Divergence** is an element or a relation that is not allowed or was not implemented as intended. Divergences indicate violations, i.e., the reality deviates from the plan.
- **Absence** is an element or a relation that was intended but not implemented. Absences indicate that the elements or relations in the plan could not be found in the implementation or may have not been realized yet.

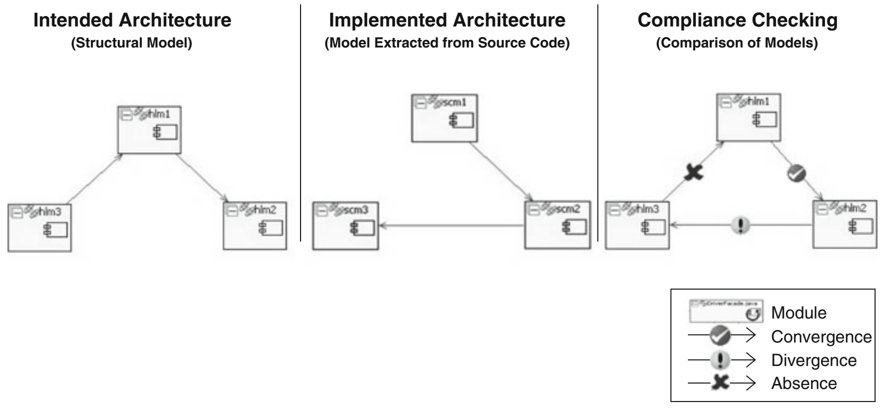


Fig. 8.3 ACC Output of structural checking. © Fraunhofer IESE (2006)

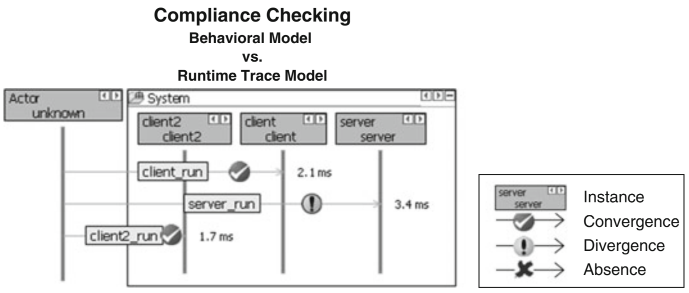


Fig. 8.4 ACC output of behavioral checking. © Fraunhofer IESE (2009)

As stated above, we distinguish between structural and behavioral compliance checking. Consequently, the output looks differently, see Figs. 8.3 and 8.4, respectively.[1] Figure 8.3 depicts a structural model and exemplifies the three result types. The source code model has been lifted to the same level of abstraction based on a mapping provided. This enables comparison of the two models.

Figure 8.4 shows the comparison of a behavioral model versus a trace generated from an instrumented run of the software system. In the example depicted, the "server_run" invocation is a divergence from the specified protocol because it should have been invoked before "client_run". Such a trace provides exactly one execution of the software system under evaluation, and the challenges are similar as in testing. Achieving full code coverage is hardly possible for any non-trivial software system.

[1]Note that all screenshots in this chapter were produced with the research prototype Fraunhofer SAVE (Software Architecture Visualization and Evaluation).

Q.075. What Do Example Results of the ACC Look Like?

Figure 8.5 depicts an example result of a structural ACC for a layered architecture. It is an excerpt of checking an industrial software product line for measurement devices where the ACC was institutionalized as a means for ensuring compliance between the reference architecture and the source code of all product instances (see Kolb et al. 2006; Knodel et al. 2008) for more details on the case study).

Q.076. How to Rate the Results of the ACC?

We rate architecture compliance for each solution concept manifested in the software system. All findings (i.e., convergences, divergences, absences) are considered in total and then aggregated by assigning values on the two four-point scales (severity of the findings and balance of the findings). The higher the score, the better the degree of architecture compliance for the solution concept.

- **N/A** means that the architecture compliance for a solution concept has not (yet) been checked.
- **NO Architecture Compliance** indicates a systemic misunderstanding that has been manifested in the code. It affects the fulfillment of architecture drivers and requires great dedicated effort for correction. Another possibility is even worse: no counterparts were found on the source code level for the architectural solution concept (e.g., see Fig. 8.6 for an example where the architects proclaimed having a layered architecture and the visualization of the extracted facts from the source code revealed the chaos and disorder depicted in Fig. 8.6).

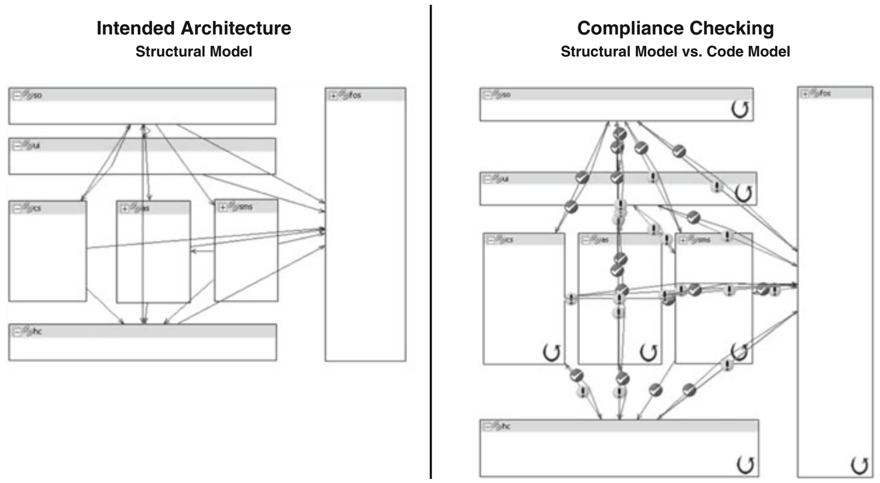


Fig. 8.5 ACC example results: compliance check for layered architecture. © Fraunhofer IESE (2008)

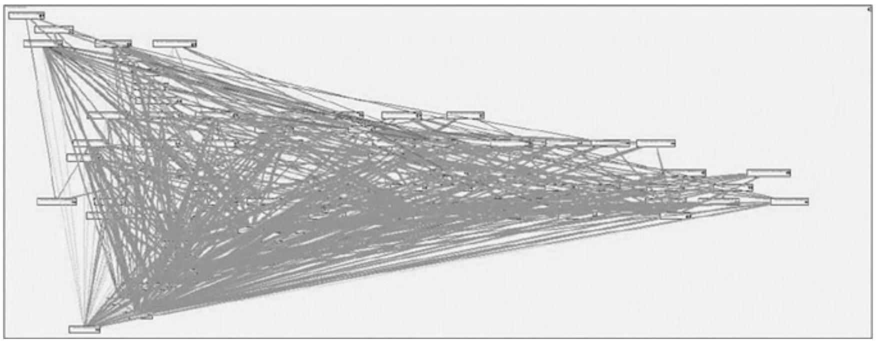


Fig. 8.6 ACC example results: visualization of critical gap in layered architecture. © Fraunhofer IESE (2008)

- **PARTIAL Architecture Compliance** means that there is a large gap between the solution concept and the source code. The lack of compliance does not break the architecture but the number of violations is drastically high. As a consequence, the impact on the achievement of some architecture drivers is harmful or detrimental. The estimated effort for fixing these violations does not fit into the next evolution cycle; rather, fixing the violations requires dedicated effort for redesigning, restructuring, and refactoring.
- **LARGE Architecture Compliance** means that there is a small or medium gap between the solution concept and the source code. The lack of compliance does not break the architecture but has a significant adverse impact on the achievement of some architecture drivers. The existing violations should be addressed explicitly and the estimated effort for fixing these does fit into the next evolution cycle.
- **FULL Architecture Compliance** means there are no or almost no violations in the source code (short distance to the architectural solution concepts). However, having no violations at all is unrealistic for non-trivial software systems; there will always be exceptions for good reasons (technical limitations, optimizations of quality attributes, etc.). It is rather important to have a low number of violations (e.g., less than one percent violations of all dependencies) that are known explicitly and revisited regularly to keep them under control.

Q.077. What Are the Confidence Levels in an ACC?

The procedures of the ACC deliver as output architecture violations (divergences and absences). Tool-based compliance checking enables analysis over large code bases. Ideally, all kinds of architecture violations could be detected by tools, but in practice, tools mainly focus on structural dependencies (a few tools also provide basic support for behavioral dependencies). However, architecting is more than just

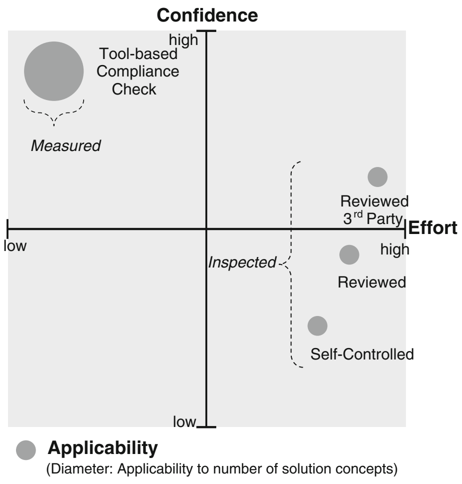


Fig. 8.7 ACC confidence levels. © Fraunhofer IESE (2015)

dependency management and not all implications of an architectural solution concepts can be translated into a rule that can be processed by a tool. For these cases, inspections are the means to achieve confidence. Their applicability is rather limited due to the size and complexity of the source code for any non-trivial system. Figure 8.7 schematically depicts the confidence level for the ACC.

Q.078. What Kind of Tool Support Exists for the ACC?

Tools provide automation support for (1) the reverse engineering steps, (2) the actual compliance checking, and (3) visualizing and navigating the results and the source code. Reverse engineering is an activity that typically requires manual as well as automated steps. Manual steps are necessary to direct the analysis to the relevant facts and to exclude irrelevant information. Automated steps are necessary to handle the sheer amount of data that comes with millions of line of source code. Visualization provides means for navigating, filtering, and processing such large amounts of information.

The ACC typically puts the greatest emphasis on structural aspects (i.e., dependencies among source code modules), which are obviously very relevant for quality attributes such as maintainability, extensibility, and so on. The ACC is used less frequently, but is nevertheless also important, for the behavior of the software system (i.e., whether the executables of the system act as prescribed in the architecture, e.g., adherence to protocols). Here the running system has to be instrumented to gather traces about the invocations made, their order, their timing, and the data processed. The evaluation of such traces sometimes requires individual development of scripts for preprocessing to distill the architectural aspects currently under consideration.

Commercial, open-source, and academic tools[2] are available for automating compliance checking. Googling for tools for “architecture compliance checking”, “architecture conformance checking”, or “architecture reconstruction tools” will lead to prominent tool vendors and consulting services, while the survey of (Pollet et al. 2007) provides an overview of academic research on architecture reconstruction, partly advanced, partly not applicable to industrial systems of such a scale.

Q.079. What Are the Scaling Factors for the ACC?

The ACC has to deal with large amounts (typically millions of lines) of source code or huge runtime traces of system executions (millions of invocations). The processing of such large amounts of data can only be achieved with adequate tool support. The size of the system, respectively the code base, is one of the scaling factors for the ACC, as the larger the model extracted from the source code or from runtime traces, the likelier it becomes for compliance checking tools to run into scalability or performance problems when visualizing or computing the results. Then automated checks become difficult or require separation of the models into sections in order to enable computation of the results.

Another influence factor is the heterogeneity of the code base. Large software systems are often composed of source code implemented in several programming languages or scripting languages. This heterogeneity of languages may constrain the selection of ACC tools as they typically support only a limited number of languages. Furthermore, extensive usage of framework, middleware, and other technologies (e.g., for dependency injection, management of other containers, communication) may affect the results of the ACC, as dependencies may be hidden by the framework (see, e.g., Forster et al. 2013).

The expertise of evaluators in using the tools and understanding the code base is another scaling factor for the ACC. ACC tools are typically made for experts, so some of the tools on the market score with feature richness but lack usability and ease of learning. Working with visualizations of larger code bases (navigation, filtering, zooming, creating abstractions, digging for relevant details, and in particular layouting the information to be visualized) is a huge challenge in itself.

In addition, the availability of experts (architects and developers) to provide inputs to the solution concepts and mapping them to the source code is crucial for the ACC. If architecture reconstruction is required first (i.e., in the event that information was lost, the software system is heavily eroded, or experts are no longer available), the application of the ACC will require tremendously more effort and time.

[2]For instance, see tools such as the CAST Application Intelligence Platform, Structure101, hello2morrow’s Sotograph and sotaarc, NDepends, Axivion Bauhaus, Lattix Architect, or Fraunhofer SAVE.

8.3 What Mistakes Are Frequently Made in Practice?

Simply removing architecture violations is not enough.

Typically, if there is a large number of architecture violations, architecture adequacy is not given (anymore) either. Thus, before worrying about architecture compliance, it is necessary to improve the architecture in terms of adequacy for the requirements.

Considering compliance checking as a one-time activity.

Architecture compliance checking requires regular and repeated applications in order to lead to fewer violations over time. Typically, the point in time when compliance checking is conducted is usually late in product development. However, we were able to observe in several cases that the cycles between two compliance checking workshops became shorter over time. In some cases, it has even been integrated into continuous build environments to further reduce compliance checking cycle times and apply it early in product development, even if only partial implementations are available. Compliance checking has been able to cope with the evolution of the architecture and the implementation. The compliance checking results serve to provide input to the continuous refinement and improvement of the architecture. This is one prerequisite for starting strategic discussions (e.g., investment into reusable components, anticipation of future changes, planning and design for reuse).

Not refining the scope of compliance checking over time.

The initial application of compliance checking typically aims at checking coarse-grained solutions concepts such as the usage of a framework, basic layering, or fundamental design patterns. Initial checks often reveal a high number of architecture violations, which are then subject to refactoring or restructuring. In repeated analyses, the lower number of violations indicates that these solution concepts have become compliant to a large degree. But then no detailed concepts are checked, which might be a risk. We recommend refining the analysis scope of compliance checking by also checking the detailed dependencies on the subsystem and/or component level once issues with coarse-grained solution concepts have been fixed.

9 How to Perform the Code Quality Check (CQC)?

The main goal of the Code Quality Check (CQC) is to gather data about the source code base. As such, the CQC is not a direct part of the architecture evaluation. However, reasoning about quality attributes (in particular maintainability) requires the CQC results in order to make valid statements about the software system under evaluation (Fig. 9.1).

9.1 What Is the Point?

Q.080. What Is the CQC (Code Quality Check)?

The implementation is one of the most important assets of a development organization, possibly determining its overall success in delivering a software system with the requested functionality while meeting effort, quality, and time constraints. Producing the implementation is an activity executed by a number of (teams of) developers. These developers write source code statements in order to translate solution concepts defined in the architecture into algorithms and data structures. The size of the code base can range from a few thousand lines of code to many millions lines of code. Due to the size of software systems and their inherent complexity, it is obviously not feasible to manage software development efficiently on the source code level (hence, we need architecture that provides abstraction, enabling us to keep control over complexity). The situation gets even worse over time, as observed in Lehman's laws of software evolution (see Lehman and Belady 1985) on "continuing growth", "increasing complexity", and "declining quality". Complementary to a sound architecture, the code itself has to be of high quality in order to stay maintainable (Visser et al. 2016).

© Springer International Publishing Switzerland 2016
J. Knodel and M. Naab, *Pragmatic Evaluation of Software Architectures*,
The Fraunhofer IESE Series on Software and Systems Engineering,
DOI 10.1007/978-3-319-34177-4_9

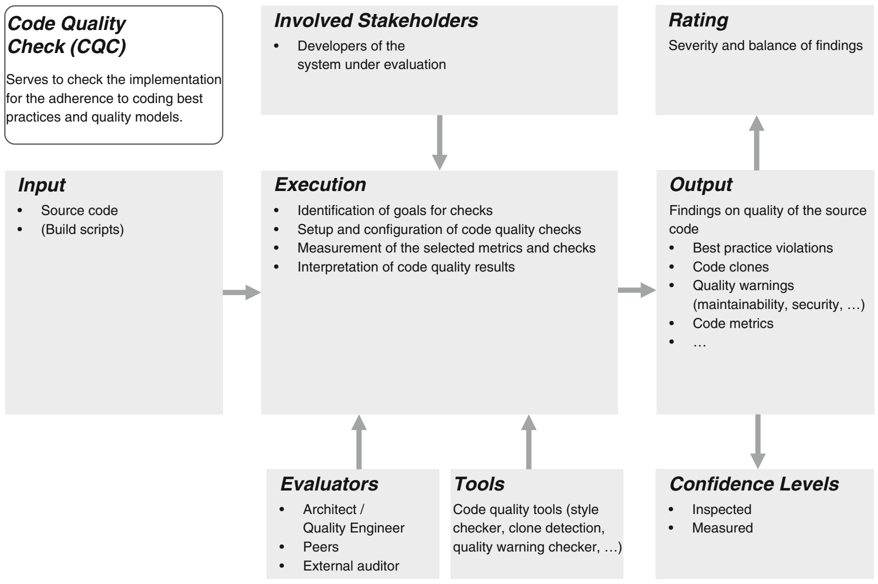

Code Quality Check (CQC)
Serves to check the implementation for the adherence to coding best practices and quality models.
Involved Stakeholders
• Developers of the system under evaluation
Rating
Severity and balance of findings
Input
• Source code
• (Build scripts)
Execution
• Identification of goals for checks
• Setup and configuration of code quality checks
• Measurement of the selected metrics and checks
• Interpretation of code quality results
Output
Findings on quality of the source code
• Best practice violations
• Code clones
• Quality warnings (maintainability, security, …)
• Code metrics
• …
Evaluators
• Architect / Quality Engineer
• Peers
• External auditor
Tools
Code quality tools (style checker, clone detection, quality warning checker, …)
Confidence Levels
• Inspected
• Measured

Fig. 9.1 CQC overview

The CQC analyzes the source code of a software system to reveal anomalies with regard to best practices, quality models, coding and formatting guidelines in order to counteract the declining quality symptom. The underlying model is about the human capabilities to process large and complex information (i.e., the source code). Because the mental capabilities of human beings to manage larger parts of the overall system are limited, source code metrics, for instance, can measure whether the length of a method or class does not exceed a certain threshold or whether the complexity [e.g., the cyclomatic complexity as specified by McCabe (1976)] does not exceed given thresholds.

Code quality checks as such are hence no direct means of architecture evaluation. However, they are often applied to complement the information gathered in architecture evaluations to provide a complete picture. For example, the quality attribute maintainability covers many different aspects: In the SAC, the adequacy of an architecture for supporting certain anticipated changes can be checked by assessing how large the impact of a change and thus the effort to execute it would be. With architecture level metrics, it can be checked to which extent rules of good design are adhered to, which allows making at least some general statements about maintainability. While these two aspects are mainly aimed at predicting the actual effort required to execute a change, code level metrics can cover another aspect of maintainability: the readability and understandability of the source code (see also Question Q.096).

Numerous tools for computing metrics and/or analyzing adherence to various coding and formatting rules exist for the CQC. Using such tools bears the risk of being overwhelmed by numbers and data as some of the tools compute dozens of different metrics and check for adherence to hundreds of rules. This huge amount of numbers (in some organizations even recalculated by the continuous build environment for every commit) might have the effect of not seeing the forest for the trees. Interpreting the numbers and the output lists generated by the tools and mining them for architecture relevance is the key challenge of the CQC.

Q.081. Why Is the CQC Important?

Change is the inevitable characteristic of any (successful) software system. Changing the code base requires the major part of the total effort spent on software engineering (see Boehm 1981). At least since the late 1970s we have known from the studies of (Fjelstad and Hamlen 1983) that codifying and resolving a change is only half the battle; roughly the other half is spent on program (re-) comprehension. These findings make it obvious why it is important to invest into the architecture as a means for navigating larger code base and into instruments for improving code quality.

The CQC is nevertheless a crucial instrument for detecting anomalies in the source code that negatively affect the understanding of code quality. Anomalies are derived from universal baselines and their calculation is supported by tools. Fixing the anomalies can significantly improve code quality. This results in better understanding of the code base by architects and developers. The underlying assumption is that optimizing the source code with respect to general-purpose measurements will facilitate the implementation of change requests and thus improve the productivity of the team and the overall maintainability of the software system. Many empirical studies on software maintenance provide evidence that this assumption is generally true. However, exceptions prove the rule and it is dangerous to trust in pure numbers.

Q.082. How to Exploit the Results of the CQC?

The results of a CQC can be used directly to improve the implementation of a software system. Anomalies can be fixed or monitored over time. Common metrics and coding best practices or team-specific coding guidelines can improve the overall understanding of the code base. This makes the development organization more robust towards staff turnover and integration of new development team members. Additionally, the results of a series of CQCs can be used to define team-specific coding guidelines.

9.2 How Can I Do This Effectively and Efficiently?

Q.083. What Kind of Input Is Required for the CQC?

The mandatory input for the CQC is obviously the source code of the software system under evaluation. Additionally, the CQC requires configuring the thresholds of coding rules and best practices for the detection of anomalies. Here, either general-purpose rules or thresholds can be used, or dedicated quality models are defined and calibrated for the CQC. While the former are rather easy to acquire and in many cases come with the tool, the latter often require fine-tuning to the specific context factors of the software system under evaluation.

Q.084. How to Execute the CQC?

The CQC typically comprises the following steps:

- Select a tool for computing the code quality (make sure the tool supports the programming languages of the software system under evaluation).
- Configure and tailor the tool for your own context (e.g., select metrics and rules and define thresholds).
- Conduct the CQC by applying the tool to the code base of the system.
- Interpret the CQC results: How many anomalies were found? Can they be classified? What is their severity? How much effort is estimated to remove the anomalies? Is it worthwhile removing the anomalies?

The CQC is often performed in an iterative manner, starting with a limited set of rules or best practices to which adherence is checked. In particular, the configuration of thresholds when a check is firing often requires time and effort in order to turn raw data into useful, meaningful, and understandable information. Typically, a lot of coarse-grained reconfiguration takes place in the beginning (e.g., in or out), while fine-grained adaptations are performed later on (adapting thresholds, refining rules, etc.).

In general, the CQC opens a wide field of different tools and possibilities to check for, ranging from style guides via general coding to technology best practices. Some typical purposes for executing a CQC are listed below:

- **Code Quality Metrics** aim to numerically quantify quality properties of the source code. Well-known examples of code quality metrics include the complexity defined by (McCabe 1976), the Chidamber and Kemerer suite of object-oriented design metrics, for instance including Depth of Inheritance Tree, Coupling Between Objects, see Chidamber and Kemerer (1994), and the suite of Halstead metrics including Halstead Effort and Halstead Volume, see Halstead (1977). Many integrated development environments (IDEs, e.g., Eclipse, VisualStudio) support a range of code quality metrics, either natively or through plugins. Typically, the supporting tools include certain thresholds or corridors that should not be exceeded. For the offending code elements, violations are reported, similar to bugs detected with heuristics.

- The aggregation of code-level metrics into system-level metrics by these environments is often not more sophisticated than providing descriptive statistics (i.e., mean, maximum, minimum, total). Assessors typically need to craft their own aggregations or use expert opinion to provide accurate assessments.
- **Code Quality Models** establish generally applicable quantifications for evaluating one or more quality attributes of a software product. This is typically done by mapping a selection of low-level metrics to those quality attributes using relatively sophisticated aggregation techniques.
- Several models that operationalize the maintainability quality characteristic (as defined by the ISO 25010 standard (ISO 25010 2011) or its predecessor, ISO 9126) are available, see Deissenboeck et al. (2009), Ferenc et al. (2014), for instance the SIG maintainability model (Heitlager et al. 2007) for the purpose of assessing and benchmarking software systems. This model seeks to establish a universal measurement baseline to provide benchmarks and enable comparability (see Baggen et al. 2012). Some code quality tools for developers also include quality models as plugins. For example, the SonarQube tool supports the SQALE quality model (see Mordal-Manet et al. 2009). Some quality models are not universal, but require tailoring according to a structured method like QUAMOCO (Wagner et al. 2015). Also, methods exist for constructing individual quality models through goal-oriented selection of metrics, such as the GQM (see Basili and Weiss 1984 or more recently Basili et al. 2014).
- **Clone Management** is aimed at the detection, analysis, and management of evolutionary characteristics of code clones (see Roy et al. 2014). Code clones are segments of code that are similar according to some definition of similarity, according to (Baxter et al. 1998). Tools apply universal rules to detect clones in the source code.
- **Bug Detection Heuristics** aim at finding bugs in the code through static code analysis or symbolic execution. There are various tools for static code analysis, with some of them able to detect potential defects. Tools such as Coverity (see Bessey et al. 2010) and Polyspace for C/C++, or FindBugs (see Ayewah et al. 2007) for Java can be used by developers to identify locations in the program code where bugs are likely to be present. Some of these tools support standards intended to avoid bug patterns that are universal to particular programming languages (e.g., MISRA-C, see MISRA 2004).
- Though such tools typically are not intended to support assessment at the level of an entire system, one can apply them for this purpose. However, there is no broadly accepted method for aggregating numerous code-level bug findings into a unified quality indicator at the system level. When using bug detection tools for this purpose, assessors need to carefully craft the appropriate aggregation for their specific situation. In practice, we have observed that badly chosen aggregations lead to inaccurate assessments.

- **Code Reviews and Inspections** are structured methods for identifying potential problems in the source code by manually reading and reviewing the code. They are typically not conducted on the level of an entire system, but rather on the level of a set of code changes. Typically, the guidelines used for the inspection or review refer to design patterns, naming conventions, and style guides specific to an individual system. Recommender tools support code reviews by proposing source code excerpts to be reviewed, revisiting items under review, highlighting changes made by others or within a certain time frame, or tracking comments on findings.
- **Style Guide Enforcements** aim at checking the conformance of the source code with respect to universal or organization-specific coding guidelines, see Smit et al. (2011). Tools such as PMD or Checkstyle support such checks, which are typically available as plugins to an IDE. As in the case of bug detection and code metrics, we have observed in practice that aggregation to the system level for assessment purposes is problematic.

Q.085. What Kind of Output Is Expected from the CQC?

The output of the CQC is basically a data set capturing all values for all entities that have been analyzed. Typically, most tools highlight the anomalies in the data set or provide certain features for navigating, filtering, aggregating, and visualizing the findings. For instance, the screenshot in Fig. 9.2 uses aggregations (upper left), tree maps (upper right), doughnut charts (lower left), and time series data (lower left) to present the computed data.

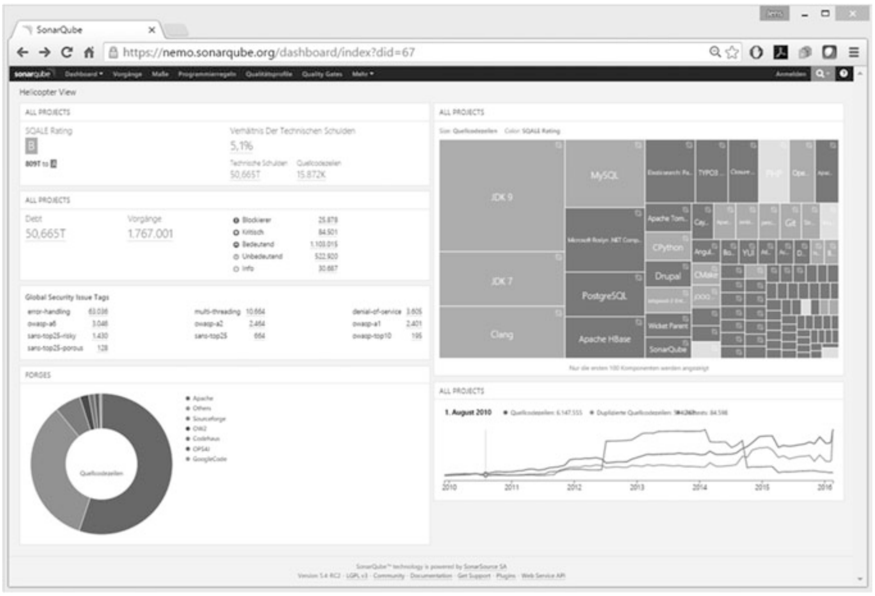
Q.086. What Do Example Results of the CQC Look Like?

Figure 9.2 depicts an example visualization of a tool for conducting code quality checks using SonarQube, representing different metrics in numbers and with adequate visualizations. Other tools for checking code quality produce similar results.

Q.087. How to Rate the Results of the CQC?

We rate the code quality for each criterion (i.e., each metric, rule, or best practice) that has been computed by the CQC analysis tool. The combination of both scales determines the overall code quality:

- **N/A** means that the code quality for a criterion has not (yet) been checked.
- **NO Code Quality** indicates major parts of the code base exceed the thresholds that have been defined for the criterion at hand.



SQALE Rating B
809T to A
Verhältnis Der Technischen Schulden 5,1%
Technische Schulden 50,665T Quellcodezeilen 15.872K

Debt 50,665T Vorgänge 1.767.001

Blockierer	25.878
Kritisch	84.501
Bedeutend	1.103.015
Unbedeutend	522.920
Info	30.687

Global Security Issue Tags

error-handling	63.036	multi-threading	10.664	denial-of-service	3.605
owasp-a6	3.046	owasp-a2	2.464	owasp-a1	2.401
sans-top25-risky	1.430	sans-top25	664	owasp-top10	195
sans-top25-porous	128				

Fig. 9.2 CQC output (screenshot with SonarQube) (screenshot taken from Nemo, the online instance of SonarQube dedicated to open source projects, see https://nemo.sonarqube.org/)

- **PARTIAL Code Quality** means for some parts of the source code, the thresholds defined and the impact of the anomalies is considered harmful. The estimated effort for fixing these anomalies does not fit into the next evolution cycle; rather, dedicated effort for refactoring is required to fix the anomalies.
- **LARGE Code Quality** means that only limited anomalies were found with respect to the defined criterion. The existing anomalies should be addressed explicitly and the estimated effort for fixing them does fit into the next evolution cycle.
- **FULL Code Quality** means there are no or only few anomalies (e.g., condoned exceptions).

Q.088. What Are the Confidence Levels in a CQC?

The CQC procedures deliver numbers and anomalies on the various aspects being analyzed (see Question Q.081). Being able to formulate and define a relevant rule to check for is an effort-intensive and time-consuming endeavor, which has to be undertaken individually for each rule that is defined. Tools then automate the execution of the rule against the code base, which comes at almost zero cost. Most out-of-the-box tools are able to produce numbers and come with a lot of built-in checks (e.g., lines of code, McCabe complexity, depth of inheritance tree, etc.). Deriving useful information

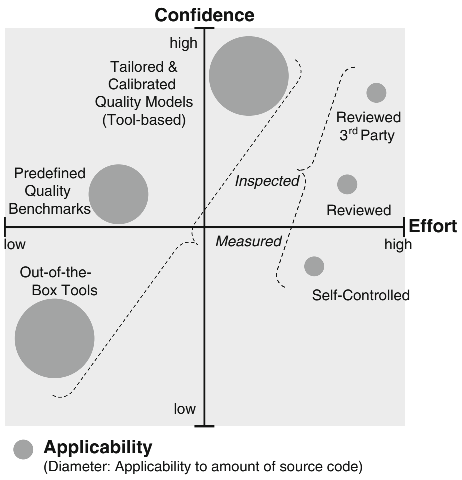

Confidence
high
low
Effort
low
high
Tailored & Calibrated Quality Models (Tool-based)
Predefined Quality Benchmarks
Out-of-the-Box Tools
Reviewed 3rd Party
Inspected
Reviewed
Measured
Self-Controlled
Applicability
(Diameter: Applicability to amount of source code)

Fig. 9.3 CQC confidence levels. © Fraunhofer IESE (2015)

based on these built-in checks is not trivial (i.e., if a method with more than 100 lines indicates an anomaly, what about another method with 99 lines of code?). Consequently, confidence in these basic built-in checks is rather low.

Using tailored quality models (pulling data from built-in checks, configured and adapted to one's own context and calibrated with one's own code base at hand) is an approach for gaining useful information; hence confidence is high. A predefined quality benchmark maintained and evolved by an external organization capable of running arbitrary code analyses on unknown code bases is an intermediate solution. Here the risk might be that the heterogeneous zoo of programming and scripting languages as well as technologies and frameworks used might not be fully supported by the benchmark. Code inspections can lead to rather high confidence. Their applicability is rather limited due to the size and complexity of the source code for any non-trivial system. Figure 9.3 schematically depicts the confidence level for the CQC.

Q.089. What Kind of Tool Support Exists for the CQC?

Tools for the CQC are crucial in order to be able to handle any code base of a non-trivial size. There are numerous tools[1] for the CQC, both commercial and open source. The tools differ with regard to the measurement criteria (metrics, rules, best practices, etc.), the formulas they use for computing the data, the features for visualizing and handling the data, and the way they are integrated into the

[1]For instance, see SciTools Understand, Grammatech Codesonar, Coverity, Klocwork FrontEndart QualityGate, Codergears JArchitect, CQSE Teamscale, semmle Code Exploration, or open source tools like SonarQube, PMD, Findbugs or various Eclipse plugins.

development environment or the tool chains used by the development organization. The list of tools for static code analysis[2] gives a first good, but incomplete overview of available tools.

Some of the tools available for the CQC support integration into continuous build systems, enabling recomputing for any commit that has been made by any developer. This also allows plotting trend charts that show how certain values have changed over time.

Depending on the tools used, the various ways to visualize the results can differ a lot. Many tools also support extension by means of custom-made plug-ins or custom-defined metrics or rules.

Q.090. What Are the Scaling Factors for the CQC?

The CQC face similar scaling factors as the ACC (see Question Q.079): the size of the system, the large amounts of data to be processed, and the heterogeneity of the code base.

The expertise of evaluators in using the tools and understanding the code base is also an important scaling factor for the CQC. Knowing the formulas for computing metrics and the definitions of the rules as implemented in the CQC tool at hand is crucial. Note that even for a well-known metric like Lines of Code there are many possible ways of counting the lines. Again working with visualizations of metrics of larger code bases (navigation, filtering, zooming, digging for relevant details, and in particular layouting the information to be visualized) is also a huge challenge for the CQC.

9.3 What Mistakes Are Frequently Made in Practice?

Focusing on source code measurement only and believing that's enough.

We experienced several times that measurement programs collecting tons of metrics (e.g., lines of code, cyclomatic complexity) had been established in customer companies. Management was confident that they were controlling what could be measured. However, most of the time, the interpretation of the measurement results was not connected to the architecture. Thus, the measurement results were more or less useless in the context of architecture evaluations. Product-specific means for risk mitigation require much more in-depth analysis and more thorough

[2]See https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis.

understanding of the current system and software architecture in general. Thus, it often seems to be the easiest solution to buy a metric tool, with the disadvantages described above.

Positioning certain metric values as goals in an incentive system.

Who wants certain metric values will finally get them. If metrics are used to control the work of developers and certain values have to be achieved, the developers will find ways to achieve them. However, this often leads to the opposite of what was planned. A good example from practice: If the length of a method is restricted to 25, developers might split the method into 2 parts, called part 1 and part 2.

Part III
How to Apply Architecture Evaluation in Practice?

What Are Example Cases of Architecture Evaluation? 10

An architecture evaluation approach can be illustrated best with examples in which the approach was applied. We report on four real but anonymized architecture evaluation projects with industrial customers. We will show how critical decisions about the future of a software system were supported, how architecture evaluation was applied to identify risks, and how architecture evaluation supported the selection of a foundational technology. We will share experiences with the application of our checks and show the results as they were presented in the management presentation. We will then summarize lessons learned from more than 75 architecture evaluation projects on architecting in general, on maintainability, and on metrics, and briefly outline how our evaluation approach evolved over time.

10.1 What Are Typical Evaluation Questions and Example Cases?

Here, we will describe some of our architecture evaluation projects in greater detail and in an anonymized form. The results overview as shown in the management presentation is depicted in Fig. 10.1 as an aggregation for the four examples. Our description largely follows the recommended structure for result presentations outlined in the last section. A broader overview of a series of 50 architecture evaluation projects can be found in Knodel and Naab (2014a, b).

Q.091. What Is an Example Case of Critical Decision-Making Regarding a System's Future (Example 1)?

This example case evaluated the architecture of an existing system with special emphasis on maintainability and support of future business goals. The outcome of

J. Knodel and M. Naab, *Pragmatic Evaluation of Software Architectures*,
The Fraunhofer IESE Series on Software and Systems Engineering,
DOI 10.1007/978-3-319-34177-4_10

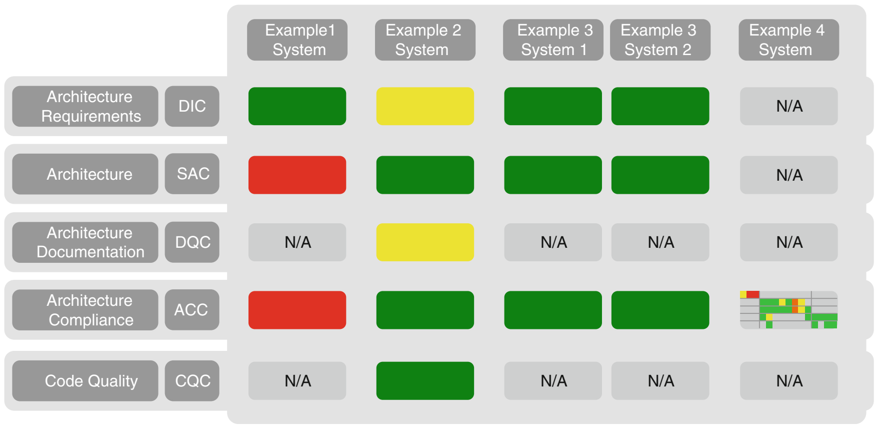


Fig. 10.1 Evaluation results overview of 4 example cases

the evaluation was to completely restart the architecting work since the intended architecture was not adequate. For an overview and aggregation of the results, see Fig. 10.1; details are given in Table 10.1.

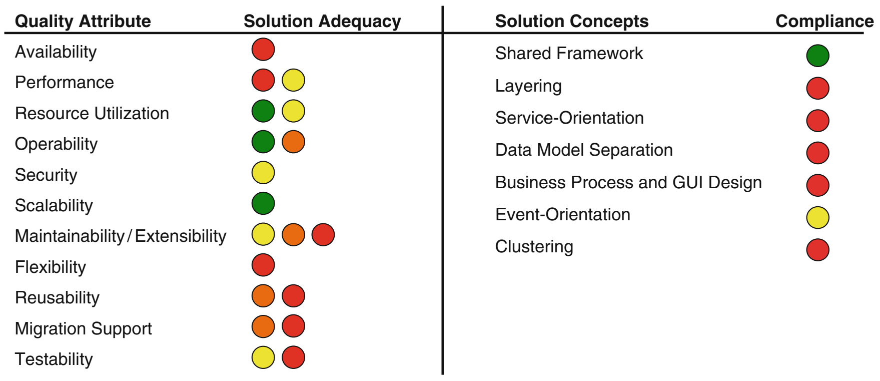
Table 10.1 Example of critical decision-making

Initial Situation • Solution provider of a large-scale information system has developed, maintained, and operated a system for one customer for several years • The system is a workflow-supporting frontend, which works on an independently developed backend • Continuous maintenance (2 releases per year) is getting more and more expensive, in particular also the backend maintenance • Frontend system should be migrated to a new backend system purchased from another vendor	**Facts about the System** • ~4 MLoC Java • ~10 years old • Distributed development incl. off-shore development
Evaluation Questions and Approach • Q1: Is the currently realized architecture of the frontend system a solid foundation for the renovated system based on the other backend (especially with respect to maintainability)? → SAC: Elicitation of maintenance scenarios and checking whether these are addressed adequately at the architecture level → ACC: Application of SAVE tool to current implementation • Q2: Is the projected architecture adequate for supporting the stakeholders' future goals? → SAC: Elicitation of stakeholder requirements as scenarios and evaluation of adequate architecture realization	Facts about the Checks *Involved stakeholders/persons* • 16 stakeholders from 2 organizations (customer and solution provider organization) were interviewed • No significant deviations in driver integrity were identified • 3 architects of the system were involved in the evaluation *Architecture drivers evaluated* • 25 architecture scenarios were elicited and ranked into 3 priorities A, B, and C • 19 architecture scenarios were evaluated in the SAC

(continued)

Table 10.1 (continued)

Evaluation Results Q1: • Original architecture provided a lot of well-designed solution concepts that were beneficial for maintainability in general • No overall maintainability concept • Maintainability concept not tailored to migration to new backend • Widely missing compliance in the realization of the solution concepts (see right side of Fig. 10.2) • ~26,800 architecture violations against the solution concepts identified in the code Q2: • Large number of scenarios for which solution adequacy is PARTIAL or even NO (see left side of Fig. 10.2; each point is an evaluated scenario) • Overall, the future architecture is not thoroughly designed yet	Confidence Level Achieved • Level of confidence is high, in particular for the scenarios with missing solution adequacy • No further means necessary to increase confidence
Recommendations • Overall, migration project should not be started yet • Thorough architecture design is needed first • A stronger architecture team is needed (missing skills) • Improvement of architecture documentation necessary as basis for reuse • Investment into refactoring and modernization of the implementation: reduce architecture violations • Migration has to be planned for inherently in the architecture design	Facts about the Evaluation Project • ~70 person-days of effort were spent by the evaluating people • ~4 person-days were spent for the interviews to elicit the architecture drivers • ~10 person-days were spent by the system's architects during the architecture evaluation • The overall architecture evaluation project lasted roughly 4 months



Quality Attribute	Solution Adequacy
Availability	🔴
Performance	🔴 🟡
Resource Utilization	🟢 🟡
Operability	🟢 🟠
Security	🟡
Scalability	🟢
Maintainability / Extensibility	🟡 🟠 🔴
Flexibility	🔴
Reusability	🟠 🔴
Migration Support	🟠 🔴
Testability	🟡 🔴

Solution Concepts	Compliance
Shared Framework	🟢
Layering	🔴
Service-Orientation	🔴
Data Model Separation	🔴
Business Process and GUI Design	🔴
Event-Orientation	🟡
Clustering	🔴

Fig. 10.2 Example of critical decision-making—SAC results (*left*) and ACC results (*right*). © Fraunhofer IESE (2009)

Q.092. What Is an Example Case for Risk Management (Example 2)?

This example case evaluated the architecture of an existing system with special emphasis on the identification of risks for the upcoming rollout in architecture and implementation. The outcome of the evaluation was largely positive, indicating some improvement potential regarding the clarification of future requirements, architecture documentation, and specific aspects of code quality. For an overview and aggregation of the results, see Fig. 10.1; details are given in Table 10.2.

Table 10.2 Example risk management

Initial Situation • Solution provider developed a solution for a market of ~15 potential customers • High criticality of the solution for the solution provider and its customers • First customer acquired the solution and now it was being finalized and prepared for bringing it into operation • Solution provider is responsible for development, maintenance, and operation • Solution provider wants to sell the solution to further customers and maintain and operate it with large synergies	**Facts about the System** • Development is partly done by solution provider directly, partly outsourced to a local company and to a near-shore company • Development in C/C++
Evaluation Questions and Approach • The main motivation behind the architecture evaluation is risk management concerning the architecture and the quality of the code • The evaluation questions cover the whole range of aspects of RATE checks • Q1: Is there consensus among the stakeholders about the requirements? → DIC • Q2: Is the architecture adequate for the requirements? → SAC • Q3: Is the code consistent with the architecture as planned? → ACC • Q4: Is the architecture documentation adequate? → DQC • Q5: Does the code have good overall quality? → CQC	Facts about the Checks *Involved stakeholders/persons* • 3 stakeholders from development management and product management were interviewed • 3 stakeholders with a development/architecture background were interviewed • No access to a broader set of stakeholders, in particular not to customers or end users *Architecture drivers evaluated* • 52 architecture scenarios elicited in total • Most of the architecture scenarios are already realized in the implementation, but some are future requirements regarding sales to more customers • Prioritization done by the available stakeholders • 24 architecture scenarios evaluated, in particular also the ones stating future requirements
Evaluation Results Q1: DIC • Extensive collection of non-functional requirements, good coverage of quality attributes in the available documentation	Confidence Level Achieved • High confidence in results of DIC, ACC, DQC, ACC • For SAC, many scenarios assessed with high confidence

(continued)

Table 10.2 (continued)

• Non-functional requirements often too abstract as documented, had to be compensated in the interviews • Future requirements are partly not agreed upon among the stakeholders Q2: SAC • Most of the architecture drivers well addressed • 94 design decisions recorded and related to architecture drivers • Architecture very well explained, excellent architect • Key trade-offs identified: demanding requirements for availability, robustness, … led to comparably high complexity of the solution concepts | High degree of configurability leads to high complexity | Resulting complexity adversely impacts maintainability • More investment into maintainability needed, in particular for selling to other customers • Figure 10.3 depicts an overview of the scenarios, the quality attributes, and the evaluation results. Gray indicates that an evaluation was not possible, mainly due to unclear requirements. In these cases, the missing clarity was not discovered before the evaluation workshop (also not in the architecture driver reviews) Q3: DQC • Overall, good and comprehensive documentation: extremely detailed, well structured, well described model, but in part missing uniformity • Architectural information: strong focus on functional decomposition, but less focus on architecture decisions and solutions for requirements • Adequacy of documentation for audience: good support for concrete development tasks, but more difficult for understanding the overall system Q4: ACC • 4 architecture concepts checked for ACC: to a large extent very high architecture compliance • In part, missing uniformity: internal and external developers created different code structures, naming conventions, placing of interfaces, and data objects. The external developers adhered to the development guidelines while the internal developers did not always do so Q5: CQC	• For SAC, for several scenarios (like performance), fulfillment strongly depends on algorithmic aspects. Due to the readiness of the implementation, further confidence was gathered from lab and field tests. The absence of architectural flaws and the measured results led to overall high confidence in the evaluation results

(continued)

Table 10.2 (continued)

• Overall, pretty good code quality: well structured, readable code; programming guidelines widely followed • Technology-specific static code analysis (tool-supported): revealed several potentially critical security and reliability warnings that should be removed • Code metrics (tool-supported measurement): overall, code metrics do not indicate major concerns regarding maintainability. A few classes exist that are very large and have high cyclomatic complexity. However, there are good reasons for the design of the classes; no major impact on maintainability • Clone detection (tool-supported measurement): a few code clones found with high level of duplication	
Recommendations • Clarification of the unclear architecture drivers • Elaborate and improve architecture concepts for a few scenarios • Increase the uniformity of the implementation • Improve the architecture documentation, in particular with respect to architecture decisions and the addressing of architecture drivers • Remove the critical warnings of static code analysis	Facts about the Evaluation Project • ~75 person-days of effort were spent by the evaluating people • ~5 person-days were spent for the interviews to elicit the architecture drivers • ~4 person-days were spent by the system's architects during the architecture evaluation • Evaluation was conducted during ~10 weeks

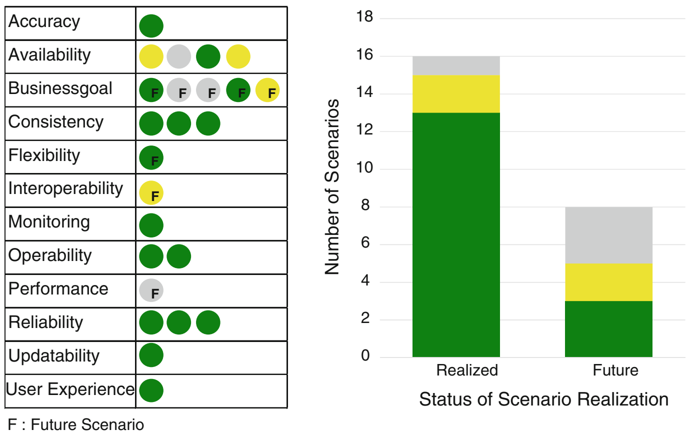


Fig. 10.3 Example risk management—SAC results. © Fraunhofer IESE (2009)

Q.093. What Is an Example Case of Making a Decision Between Two Technology Candidates (Example 3)?

This example case evaluated the architecture of two competing technologies that were the final candidates in a selection process for the future core technology of a system to be built. The overall results of the evaluation for both candidates were extremely positive. Only very rarely has such a positive evaluation result been achieved in our experience of evaluation projects. The outcome was the selection of one of the candidates, taking into account factors beyond the architecture evaluation since both architectures very equally adequate. For an overview and aggregation of the results, see Fig. 10.1; details are given in Table 10.3.

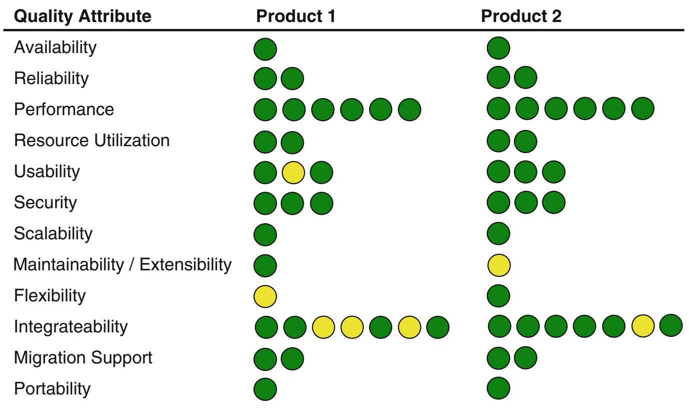
Table 10.3 Example selection between technology candidates

Initial Situation • A solution provider of an information system in the media domain wanted to renew their product • The strategy was to acquire a product from the market as a stable basis and to extend it with value-added features • A pre-selection was made based on required features and resulted in two candidate systems (called Product 1 and Product 2 in the following) • The final selection was to be made between the two candidates • An architecture evaluation conducted by Fraunhofer IESE was to provide the necessary in-depth analysis, complementing the comparison on the feature level	**Facts about the System** • Both candidate systems were written in Java and had roughly 1 MLoC, but a completely different overall architecture (regarding the functional decomposition, deployment, etc.)
Evaluation Questions and Approach • Evaluation goal: take a deep look at the architecture of the two candidate products and determine which product is suited better • Q1: How do the two candidate products fulfill the key requirements for the new solution to be built? Which strengths and weaknesses do exist? → SAC 2 times independently • Q2: Do the implementations of the candidate products really conform to the intended architectures, in particular with respect to support for extension and evolution? → ACC 2 times independently	Facts about the Checks *Involved stakeholders/persons* • 10 stakeholders of the customer provided architecture drivers • No significant deviations in driver integrity were identified • 1–3 architects of the providers of Product 1 and Product 2 were involved *Architecture drivers evaluated* • 30 architecture scenarios were elicited and also evaluated • Different quality attributes were covered and with varying numbers of scenarios (see Fig. 10.4)

(continued)

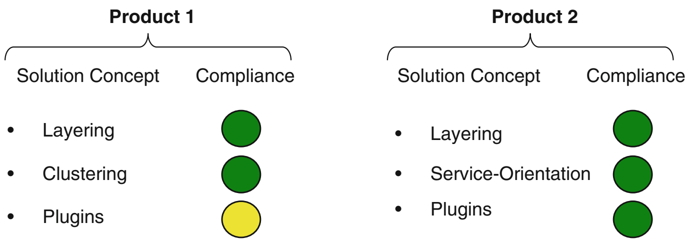
Table 10.3 (continued)

Evaluation Results Q1: • Figure 10.4 gives an overview of the evaluation results for Product 1 and Product 2 for the 30 architecture scenarios (all of them could be evaluated) • The result shows a very positive and homogeneous picture for both products. Also, the yellow results only originate from minor risks and are not severe problems • Architecture documentation was not assessed for both products as no architecture documentation was available • Both products had 3 dominating solution concepts. The concepts were partially the same across the products, but with different instantiations. For the concepts, their adequacy in the context of the respective system and the compliance in the source code were rated Q2: • The compliance checking results of the dominating solution concepts were also extremely positive (see Fig. 10.5) • Only very few architecture violations (in total not more than a few dozens for each of the products, all with minor severity) could be counted	Confidence Level Achieved • The confidence in the results is pretty high: due to the very detailed information provided in the architecture evaluation workshops and due to the fact that both products are already implemented and certain critical requirements such as response time can be measured on the running products, we have high confidence in the evaluation results
Recommendations • No clear recommendation to select one of the candidate products could be given as a result of the architecture evaluation • As a further result of the architecture evaluation, we outlined technical aspects of the products and their potential implications (e.g., that the user interfaces are based on different technologies) • The customer could make their selection based on other factors, such as the features provided, without worrying about the internal quality of the software, which was not visible to the customer	Facts about the Evaluation Project • ~35 person-days of effort were spent by the evaluating people • ~2 person-days of the customer were spent for the interviews to elicit the architecture drivers • ~8 person-days were spent by each of the providers of Product 1 and Product 2 during the architecture evaluation • Overall, the evaluation project lasted roughly 1.5 months • At the point of acquisition, only the Fraunhofer evaluation team was allowed to look at the source code of Product 1 and Product 2. The providers did not allow the customer to look at the source code before buying the product



Quality Attribute	Product 1	Product 2
Availability	🟢	🟢
Reliability	🟢🟢	🟢🟢
Performance	🟢🟢🟢🟢🟢🟢	🟢🟢🟢🟢🟢🟢
Resource Utilization	🟢🟢	🟢🟢
Usability	🟢🟡🟢	🟢🟢🟢
Security	🟢🟢🟢	🟢🟢🟢
Scalability	🟢	🟢
Maintainability / Extensibility	🟢	🟡
Flexibility	🟡	🟢
Integrateability	🟢🟢🟡🟡🟢🟡🟢	🟢🟢🟢🟢🟢🟡🟢
Migration Support	🟢🟢	🟢🟢
Portability	🟢	🟢

Fig. 10.4 Example case technology decision—SAC results. © Fraunhofer IESE (2008)



Product 1		Product 2	
Solution Concept	Compliance	Solution Concept	Compliance
• Layering	🟢	• Layering	🟢
• Clustering	🟢	• Service-Orientation	🟢
• Plugins	🟡	• Plugins	🟢

Fig. 10.5 Example case technology decision—ACC results. © Fraunhofer IESE (2008)

Q.094. What Is an Example Case of Institutionalizing Architecture Compliance Checking at an Organization (Example 4)?

This example case evaluated the architecture of an existing system with special emphasis on maintainability and support of future business goals. The outcome of the evaluation was to completely restart the architecting work since the intended architecture was not adequate. For an overview and aggregation of the results, see Fig. 10.1; details are given in Table 10.4. A detailed description of the case has been published in (Knodel et al. 2008).

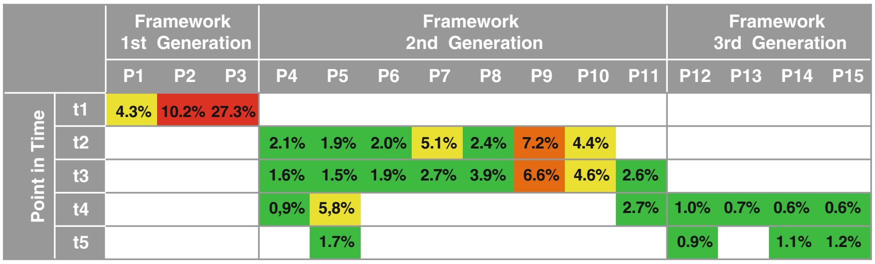
Table 10.4 Example of institutionalizing architecture compliance checking at an organization

Initial Situation • Software product line of embedded systems—flue gas and climate measurement devices • Regular architecture compliance over a period of 30 months • Reference architecture for all product line instances: family-specific implementation with generic components and application-specific implementation • Three generations of the framework (i.e., an internal reuse infrastructure)	**Facts about the System** • The size of the measurement devices ranged from 10 KLoC to 600 KLoC • Reuse degree of about 40 % • (i.e., the framework comprises approximately 40 % of each product line instance). Values were measured with various size metrics such as lines of code (LoC), number of framework files used, number of framework functions used • All products were implemented in the C programming language
Evaluation Questions and Approach • Q1: Do the investments into reuse (framework, generic components) pay off? Is the currently realized implementation of each product line member compliant with the reference architecture of the product line? → ACC: Application of SAVE tool to 15 product implementations based on different framework generations, 5 times within a period of 30 months, with significant framework extensions and refactorings in between → t1: checking of P1–P3 based on 1st generation framework → t2: checking of P4–P10 based on 2nd generation framework → t3: checking of P4–P11 based on 2nd generation framework → t4: checking of P4, P5, P11 based on 2nd generation framework, P12–P15 based on 3rd generation framework → t5: checking of P5 based on 2nd generation framework, P12, P14, P15 based on 3rd generation framework	Facts about the Checks *Involved stakeholders/persons* • About 35 software engineers involved in workshops • 5 architects and core developer interviewed for defining compliance checking scope *Solution concepts evaluated* • Several central solution concepts such as reuse dependencies or layering were checked across the products • 15 products evaluated in total • Checks applied after new products had been implemented based on new generation of the internal reuse framework (before release of products)
Evaluation Results Q1: • See Fig. 10.6 for the degree of diverging code dependencies in the total of all dependencies • The results of the compliance checking activities indicate that architectural knowledge has been established successfully in the minds of the developers. For instance, when comparing products P3, P4, P11 and P13 (all are mid-sized systems), a significant and sustainable decline in the number of divergences could be observed	Confidence Level Achieved • Level of confidence is high • ACC results enabled controversial discussions among architects and developers about the reuse infrastructure

(continued)

Table 10.4 (continued)

Recommendations	Facts about the Evaluation Project
• For all evaluation dates, it can be stated that the results enabled improvement of the reuse infrastructure • For all evaluation dates, it can be stated that compliance checking results and the rationales of the stakeholders involved both served as input to the decision-making process for the further evolution of the product line architecture • Adaptations for both the architecture and the product implementations were decided on and realized • Customizations made to tooling to adapt to the evaluation context → The analyses have to be automated to a large extent, for instance, by processing make-files and build scripts for parsing and generating the source code model required for compliance checking • The visualization of the compliance checking results was extended to export detailed lists of violations to support developers	• ~50 person-days of effort were spent by the evaluating people • ~2 days were spent on average for the workshops to discuss violations at each checkpoint in time • The overall architecture evaluation project lasted roughly 30 months • The architects perceived a great benefit in avoiding and counteracting architectural degeneration with the help of compliance checking



		Framework 1st Generation			Framework 2nd Generation								Framework 3rd Generation			
		P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
Point in Time	t1	4.3%	10.2%	27.3%												
	t2				2.1%	1.9%	2.0%	5.1%	2.4%	7.2%	4.4%					
	t3				1.6%	1.5%	1.9%	2.7%	3.9%	6.6%	4.6%	2.6%				
	t4				0,9%	5,8%						2.7%	1.0%	0.7%	0.6%	0.6%
	t5					1.7%							0.9%		1.1%	1.2%

Fig. 10.6 Example case of institutionalizing architecture compliance checking—ACC results. © Fraunhofer IESE (2008)

10.2 What Lessons Can Be Learned from Architecture Evaluations?

This whole book is built on and supported by our lessons learned. However, this section aims at stating some lessons learned very explicitly again. All of our lessons learned have been derived from the practical experiences made in the projects. At least one of the authors was involved either directly or indirectly in each of the projects. We retrospectively analyzed our projects first in 2013 and published the results in Knodel and Naab (2014a, b). In part, the following lessons learned were already published there. We are aware that our lessons learned might not be valid in projects settings with other context factors. Of course, we do not claim generalizability, but nevertheless believe each single lesson learned to be a valuable piece of experience that might help other practitioners and researchers to avoid pitfalls and facilitate their own evaluation.

Q.095. What Did We Learn About Architecture During System Construction and Software Change?

Early and essential architecture design decisions are indeed fundamental.

No matter how long the system has evolved, the initial description of the architecture is still valid (and used) for communicating the basic ideas and key functions of the systems. This means we can confirm the common belief that architectures stick to their initial ideas for a long time in the system lifecycle, at least for the 13 systems aged ten years or more.

Development becomes agile, but architecting in sprints only is not enough.

Reviewing the past decade of architecture evaluation projects, we can see that more and more of our customers have adopted agile development processes. Architecting has to be "re-defined" or "re-invented" in the context of more and more companies "going agile". The main point we have observed is that if architecting is performed within the scope of the current sprint only, it does not help to solve problems that arise across individual sprints, across teams, and in distributed development. This is especially true for quality requirements that cannot be solved in one sprint only. See Toth (2015), Rost et al. (2015) for more insights on the combination of agile development and architecting.

Architecting plays a first-class role during development, but not during maintenance.

Over the years, architecting has been established as playing a first-class role during initial development. In many cases, experienced developers are promoted to being responsible for architecting. However, during maintenance the situation is different: no architects are available to review change requests or design solutions. This leads to a gradual drift between architecture and implementation over time and confirms that architecture erosion is a fact.

Some architecture problems can be fixed easily.

In our experience, problems such as missing documentation or missing support for several new scenarios can be fixed as long as the basic knowledge about the system is up-to-date and implicitly known by the architects. The same is true for minor non-compliance in the code, which typically can be fixed in the next iteration.

Some architecture problems can't be fixed (easily).

Problems such as a high degree of non-compliance in the code or a strong degree of degeneration of the architecture over time reveal a systemic misunderstanding of the architectural concepts among architects and developers and would require enormous effort to resolve them. In our evaluations, we had just one case where such effort was actually spent without any other action (such as improvement and coaching). In this case, there was an urgent need to reduce side-effects in the implementation as a change in one place in most cases resulted in a problem in another place. Another problem that is difficult to fix afterwards is missing thoroughness in the definition of the initial architecture.

Daily workload wins over architecture evaluations.

We often encountered willingness to do an architecture evaluation project, but time and effort could not be spent (by the way, this is a fact that corresponds to typical architecture practice in these companies as well). Instead of analyzing the root cause, firefighting work was performed on the symptoms.

Patient dies on the way to the hospital.

In more than ten cases, the project was even stopped by management before the architecture evaluation (already under discussion) could take place and potential improvements could have been identified.

Rather refactor in the small than challenge your own decisions made in the past.

In some cases, the architects did not dare to question their own decisions made in the past. They ignored the possibility that decisions that once were correct in the past might not be correct anymore in the present, as the context and the system had evolved.

New features kill architecture work.

As the architecture does not deliver any direct end customer value, it is at risk of being put off or getting dropped. However, the insight that architecting delivers business value to the developing company (if it is going to maintain the software) by preparing the future or making development more efficient is often neglected. Even when we made customers aware of critical issues, we had eight cases where nothing was done afterwards.

Tool-based reverse engineering often leads to impressive but useless visualizations.

Reverse engineering of implementation artifacts is often used in architecture evaluations and partly also in the development process of our customers. We experienced that whenever such reverse engineering activities were not driven by clear evaluation questions, complex and threatening visualizations resulted. Such visualizations serve to increase awareness, but provide no guidance for improvements.

Q.096. What Did We Learn About Maintainability as a Quality Attribute?

Maintainability is a versatile quality attribute.

The quality attribute Maintainability covers many different aspects: In solution adequacy checks, it can be checked how adequate an architecture is in terms of supporting certain anticipated changes by assessing how large the impact of a change and thus the effort to execute it would be. With architecture level metrics, it can be checked to which extent rules of good design are adhered to, which allows making at least some general statements about maintainability. While these two aspects are mainly aimed at predicting the actual effort required to execute a change, code level metrics can cover another aspect of maintainability: readability and understandability of the source code. Maintainability of a software system is crucial for the success of a system in the future.

The definition of maintainability in ISO 25010 (ISO 25010 2011) shows that the aspects described above match very well with sub-qualities in the model:

- *Readability/analyzability* are aspects that are mainly aimed at the understanding of the developers and strongly depend on the code quality (of course not exclusively; the overall architecture also has some impact here). That is, when developers have to change a certain part of the code, they first have to understand it. Good code quality obviously supports this. There has been a lot of research and numerous approaches exist regarding how to measure what good readability of code means. What is interesting now is that this type of code quality does not depend on the concrete product under development. Rather, it depends on the mental capabilities of the available developers (see Fig. 3.3). For example, if methods are too long or the degree of nesting is too high, they are hard to read, or cyclic dependencies are hard to understand.
- *Modifiability/changeability* are aspects that mainly determine the overall change effort by the degree to which a concrete change is distributed over the whole system (from local to nearly global). This is mainly determined by major architecture decisions. Whether a change can be performed with low effort (meaning the system is maintainable) can be strongly influenced by architecture decisions, which cannot be measured locally in the code. For example, the requirement to replace the UI framework (e.g., because it is no longer supported by the vendor) might spread out over large parts of a system if there is no clear separation of the UI part and maybe even the concrete technology.

The measurement of maintainability plays an important role in software evaluations. It contributes to answering questions like "Is our system a solid basis for the future?" or "Can we make the upcoming changes within a reasonable amount of time and budget?"

Maintainability is a quality attribute with many indirections.

Most of the time, maintainability is not directly visible, in particular not for the end user, often not for the customer, and often not even for the developer. It is very common that there is not so much focus on maintainability during initial system development, which has to deliver fast. Then the lack of maintainability and its consequences are perceived during later system development and maintenance phases. Even then, the perception is mainly indirect, visible only in the high cost of changes.

Another reason that makes maintainability harder to handle is that it is mostly difficult to precisely formulate maintainability requirements. Anticipated changes can be stated, but often it is pretty open how a system will evolve. In terms of readability, requirements are rather stated in terms of coding conventions than as real requirements.

Measuring maintainability requires a mix of checks.

When it comes down to measuring maintainability, this is not an easy task. In practice, the simple solution is often to buy a tool that measures code metrics and also outputs results on maintainability (CQC). These quantitative results can, of course, be valuable. However, they are only part of the answer. They are that part of the answer that deals with the readability/analyzability of the source code. The good thing is that it is easy to codify such rules and quantitative metrics and measure them with standard tools. This is often done in practice. However, as stated above, this is only part of the truth.

What is missing are considerations of major architectural decisions and concrete change scenarios of the software system. However, measuring this part of maintainability is not so easy for several reasons:

- Measurement needs concrete (anticipated) change requests as a baseline and often change requests that may occur further along in the future are not known yet.
- Measurement is not possible in absolute terms, but rather requires the usage of architecture evaluation techniques, which produce only qualitative results.
- Measurement is only possible manually with the help of experts; tool support is quite limited as the evaluation is individual for each specific product. Thus, this type of measurement is often neglected in practice and, as a consequence, maintainability is not measured sufficiently.

Concrete measurement can indicate positive values for both aspects independent of each other. To reliably measure maintainability, both aspects are essential. This shows that performing a measurement requires a good understanding of the quality attributes and the concrete goals behind the measurement. It may also require the combination of different measurement techniques (such as quantitative code quality measurement and qualitative architecture evaluation) performed on different artifacts.

Measuring quality attributes in general requires a mix of checks.

The lessons learned for measuring maintainability are partly transferrable to other quality attributes. Other attributes such as performance, security, or reliability exhibit similar properties, meaning that code quality and architectural solution adequacy are both crucial for fulfilling the requirements.

Q.097. What Did We Learn About the Interpretation of the Results from Architecture Evaluation?

No standard interpretation of evaluation results is possible.

Interpretation has to consider evaluation questions and context factors. Even when there are quantitative results, the interpretation of the results remains a difficult but crucial step in architecture evaluations. Due to the nature of architecture evaluation, the results often cannot be fully objective and quantifiable. It is very important for the evaluators to manage the expectations of the evaluation owners and the stakeholders and to clearly communicate this. For instance, it is not possible to establish standard thresholds for the number of acceptable architecture violations. Rather, it is always necessary to keep the goals and the context of the customer in mind. Merely providing facts and findings is not enough. Over time, we have learned that interpretation also includes the preparation of potential follow-up steps. Today, we consider an architecture evaluation to be successful if at least one of the action items or recommendations proposed has been implemented.

Representation of evaluation results for management people and non-technical decision makers is challenging.

Often, the sponsors of an architecture evaluation are people from senior management. Despite abstracting from the system under evaluation, architectures are still technical constructs. Presenting the evaluation results to such stakeholders (who often do not have much of a technological background) is very challenging. On the one hand, the results have to be very condensed and understandable intuitively. Recommendations and alternative ways should be shown and supported with quantitative data. On the other hand, evaluators have to be careful to present subtle differences in an understandable way as these can have a substantial impact on far-reaching decisions. This holds especially true for qualities at development time. Our traffic-light-colored rating was motivated by the need to give clear presentations.

Plain numbers are preferred over statements and interpretations provided by architecture evaluations.

Many companies decide to buy a code metric tool instead of performing an architecture evaluation. We value the capabilities of metrics and continuous measurement, but we doubt their use for deriving answers to typical architecture evaluation questions (as discussed in Chap. 1).

"It depends" is not a good final answer.

Although true in many cases, it is important to clearly distinguish and delineate alternatives among follow-up actions. Companies want to know what they can do next. Ideally, findings and recommendations should be translated into business terms (gain or loss of money, meeting or failing deadlines, etc.).

Stakeholders sometimes try to influence the interpretation to achieve their own goals.

Such attempts are not very frequent but they do occur (as the results of architecture evaluations may have a significant impact on stakeholders). Being neutral is a key prerequisite for evaluators. External evaluators are often hired exactly for this reason.

Q.098. What Did We Learn About Risk Mitigation in General?

Evaluation results are expected to be delivered immediately.

Despite feeling and communicating the pressing need for having an architectural evaluation (or rather having important questions or doubts in decision-making), ordering an architecture evaluation project for some reason can take up to several months. Once ordered, expectations regarding the delivery of results do not accommodate the long waiting time for being allowed to start the evaluation. Industry customers expect evaluation results to be delivered promptly, which is contrary to other architecture-related projects we did in the past (e.g., supporting the design or coaching of architecture capabilities).

Architecting lacks a clear mission in software projects.

Our experiences show that architecting typically goes slowly because it lacks a goal-oriented focus. Architects in industry often spend a lot of time on designing templates, evaluating several technologies, modeling and pimping diagrams, but forget the point of architecting: delivering solutions for current and future design problems. As there are always a lot of problems, we think it is important to explicitly focus, plan, and track architecture work. For this reason, we proposed the construct of architecture engagement purposes (Keuler et al. 2012), an auxiliary construct to align architecting with other engineering activities in the product lifecycle.

Q.099. How Did Our Evaluation Approach Evolve Over Time?

Architecture evaluations have to evaluate implicit decisions made in people's minds, explicit decisions found in documentation, and decisions manifested in system implementations.

The big picture and integration of evaluation techniques (as depicted in Fig. 3.1) emerged over time. At Fraunhofer IESE, architecture evaluation research has been driven from three directions: reconstruction (Knodel and Muthig 2008), tool development (Knodel et al. 2009), and literature about architecture evaluation in general (e.g. Clements et al. 2001; Dobrica and Niemela 2002; Babar and Gorton 2009; Bellomo et al. 2015, to name but a few). In our first evaluation projects, we started with coarse-grained recovery and reconstruction. Over the course of time, we learned that customers rather require one concrete answer to one current, urgent, and pressing question (where an architecture evaluation may or may not be the means to answer their question). They do not care about exploiting the power of reconstruction for other parts. This resulted in our request-driven reverse engineering approach, where we always have concrete stakeholder scenarios guiding all subsequent analysis. Hence, this manifests the need to evaluate implicit decisions in architects' minds, explicit decisions described in documentation, and the source code.

All architecture evaluation is not the same.

We are covering more and more initial situations in which architecture evaluation can provide benefits. Furthermore, in every new project, we learn about something that is different than before. Every software system is unique in its characteristics, lifecycle, and context, including the people and organization behind the system, and the same is true for the evaluation of the system. In this way, architecture evaluations are always interesting, as the evaluators learn about something new within a short period of time.

11 How to Engage Management in Architecture Evaluation?

Management support is crucial for starting and conducting architecture evaluations, and even more important for exploiting the results to improve the software system. This chapter presents data, numbers, and insights gained from our project retrospective on points that are typically important to management: effort, duration, benefits, scaling factors, and improvement actions.

11.1 What Is the Point?

Q.100. Why Does Management Need to Be Engaged in Architecture Evaluation?

Architecture evaluation is an activity whose objective is to identify (potential) risks. However, architecture evaluation does not come for free:

- It requires investments to hire the auditor party (external or internal) to perform the architecture evaluation (see Chap. 4).
- It requires time and availability of relevant stakeholders of the software system under evaluation. Typically, these people are busy, under high workload, and have tight time constraints.
- It requires access to confidential and business-critical information about software systems under evaluation, and to (parts of) the overall business strategy of the development organization owning the software system.
- It may require budget for acquiring tools (and training regarding their usage) for architecture compliance checking and code quality checking.

J. Knodel and M. Naab, *Pragmatic Evaluation of Software Architectures*,
The Fraunhofer IESE Series on Software and Systems Engineering,
DOI 10.1007/978-3-319-34177-4_11

- It requires attention to the revealed findings. The results of an architecture evaluation support informed decision-making and tracking of the realization of decisions made. Findings may include design flaws, open issues, or unpleasant surprises for the development organization.
- It requires making decisions on follow-up activities (see also Question Q.106). Some findings may reveal an urgent need for action, while other improvement items may be deferred.

For all these points above, it is imperative to have the support of management, with the ability and the power to promote and push first the architecture evaluation project and then the improvement actions.

Q.101. How to Convince Management of the Need for Architecture Evaluation?

Among a lot of other aspects, making management decisions means reasoning about how to spend the available budget on software development projects or migration projects or maintenance. To convince management, architects have to play the numbers game. For managers with a strong software engineering background, we can expect the game can be easily won because these people usually have enough awareness and experience to know when to call for an architecture evaluation, or more generally speaking, when risk management is expedient. For managers with a non-technical background, the numbers game will be more difficult: The investments and their potential return (i.e., the benefits) have to be sketched and estimated to provide a sound basis for informed decision-making.

In our projects, we experienced different situations where architecture evaluation became an instrument to support decision-making (see Fig. 11.1). We distinguish

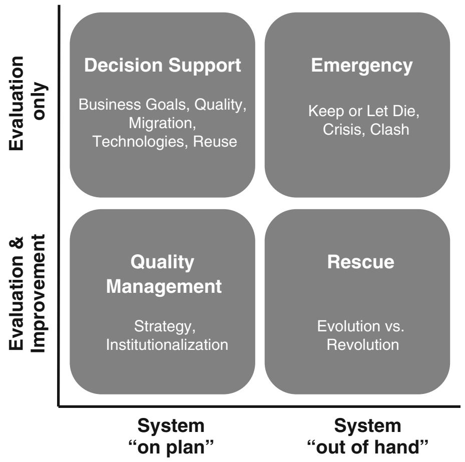


Fig. 11.1 Situations calling for architecture evaluation. © Fraunhofer IESE (2014)

between different types of criticality of a situation: Either the software system is in good shape overall (known requirements, sound architecture, compliant implementation, good documentation), which we call system on plan, or the software system already lacks quality (e.g., architecture is eroded/requirements not up-to-date/not documented, implementation lacks compliance), which is what we call a software system out of hand. Furthermore, we classified the evaluation projects according to whether the situation was a short-term assessment (called evaluation only), or an assessment embedded within a longer-term architecture improvement project. In each situation, different aspects come into play when the aim is to convince management to go for an architecture evaluation.

- **Decision Support** means performing an architecture evaluation for a central and important design decision with respect to the achievement of business goals, key functional or quality requirements, technology candidates, migration paths, or reuse of existing solutions. Management has to consider the investment for architecture evaluation on the one hand, and the risk and cost of making a wrong decision (and reversing it afterwards) on the other hand. We are confident that a single risk avoided or a wrong decision prevented can pay for all architecture evaluation activity within the budget. Architects have to make management aware of the criticality of the impending decision and confront management with the cost for realizing the decision afterwards. In our experience, the investments for an architecture evaluation will be a marginal percentage of the overall investment. Architects need to get heard, either on the merits of their credibility, by having a feeling for the situation and good timing, or as a result of their persuasive and diplomatic skills. Management will not accept if the architects call for an architecture evaluation too often nor often enough, so they have to prepare a good case for it and come up with a strong line of arguments for why it is needed at a particular time.
- **Quality Management** means establishing architecture evaluation as a recurring quality assurance instrument in a broader context (e.g., across projects, or across quality gates of long-running projects, or as a service of a central method support group). For this strategic goal of institutionalization (see also Chap. 13), it might be difficult to get the necessary budget, as no single project would pay for the basic infrastructure and for building up competencies. Nevertheless, architecture evaluations pay off in these cases through their high number of applications. Evaluators gain a lot of experience over time and become more efficient and effective in conducting architecture evaluations. Management will likely be in favor of institutionalization as the individual project only has to provide a share of the whole investment or almost no investment at all if the central group is financed otherwise.
- **Emergency** characterizes situations where there is a pressing need for decisions with high impact, which often have to be made under tight time constraints. Examples of this category are crises where one or many clients react unhappily or angrily, where a sponsor or subcontractor does not deliver as promised, where a project gets stopped, or when a decision must be made as to whether to keep a

software system or let it die. Why should management invest into an architecture evaluation in such cases and delay its decision? Because such situations tend to be hot, are often overloaded with politics, and because sometimes there is a need to blame someone (not oneself). However, what is missing in such cases is a neutral, objective analysis of the situation: the properties of the software systems and its underlying architecture and whether it offers a path to escaping the situation. Because of their high impact and often high budget, decisions in emergency cases mostly cannot be reversed. Architecture evaluation (in these cases better performed with an external auditor) provides fact-based data and information. We believe that in such overheated situations, it is important to keep a clear head. Architecture evaluation serves as an instrument to make decisions with confidence and to avoid hasty reactions.
- **Rescue** means that a software system out of hand should be brought on track again. There are two natural options: evolving the existing system at hand or revolving it by starting over from scratch. In practice, the solution is often a mix of both of these options (for more details, see aim42 2014). However, the question remains which of the two is the better option for which situation. Improvement initiatives benefit from the insights delivered by architecture evaluations to make such a decision. If the need for improvement is obvious to management, it should also be obvious for them to use architecture evaluations as an instrument for decision-making to avoid ending up again with a software system that is out of hand.

Q.102. What Are the Scaling Factors for Effort Spent on Architecture Evaluations?

Architecture evaluation is a strongly risk-driven activity. It is mostly motivated by the identification and mitigation of risks. Furthermore, the management of architecture evaluations steers the evaluation in such a way that most effort is spent on the riskiest parts. Consequently, the effort to be spent on an architecture evaluation is driven by several aspects regarding risks:

- Number and type of evaluation questions.
- Criticality of the situation.
- Need for fast results.
- Required confidence and details of results.

Besides these risk-related aspects, further aspects are relevant. These aspects mainly express the size of the system and the organizational constellation.

- System size and complexity.
- Organizational complexity.
- Number of stakeholders to be involved.

The overall effort spent on an architecture evaluation can range from a few hours up to hundreds of person-days. Both examples are absolute extremes but illustrate the spectrum of effort that can be spent. If an architecture evaluation tends to be critical and requires more effort, it is often better to involve dedicated external evaluators in order to get the evaluation done with great dedication and power. Most such evaluation projects might require between 20 and 80 person-days of effort. Example efforts from real projects are presented alongside the examples in Chap. 10.

The most relevant architecture drivers can be collected within a reasonable amount of time—even those of very large systems. By talking to all relevant stakeholder groups within a limited amount of time [we recommend 1–2 h per stakeholder (group)], the overall amount of time needed is limited. It has been our experience that the resulting coverage of architecture drivers is always good enough.

Solution adequacy checks can be controlled with respect to the invested effort by evaluating a representative and prioritized set of architecture scenarios. Of course, very complex systems require more effort; however, our experience has shown that one to three days of solution adequacy checking are typically enough to get a good understanding of the key problems in an architecture.

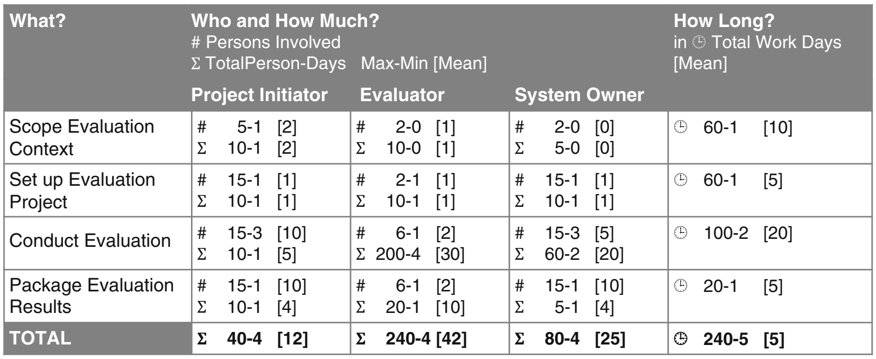
Finally, we want to offer some experiences regarding the effort of architecture evaluation projects in industry:

- Our architecture evaluation projects are typically conducted by two evaluating persons.
- In our past evaluation projects, the companies developing the software systems were involved with up to 30 persons, with a typical number of 8.
- Most architecture evaluation projects were conducted with 20–80 person-days of the evaluating persons. Additional effort was spent by the stakeholders and architects of the customer companies. The extremes are evaluations as small as 4 person-days and as large as 200 person-days.

Architecture evaluation is one of the disciplines in architecting that allows very good scaling even for very large systems and organizations. In contrast to architecture design, architecture evaluation can often leave out details or focus on a subset of architecture drivers and still provide reliable results.

Q.103. What Are Typical Figures for Architecture Evaluation Activities?

Architecture evaluation is an investment requiring time and effort by the evaluating party, the initiating party, and the party owning the software system under evaluation. As all architecture evaluation is not the same, the figures differ from case to case. However, our retrospective gives some indicators for the range and the mean across the projects we have conducted so far. We do not claim that this empirical data can be generalized for all cases, but it can still be considered as a rough ballpark figure.



What?	Who and How Much? # Persons Involved Σ TotalPerson-Days Max-Min [Mean]			How Long? in ◷ Total Work Days [Mean]
	Project Initiator	Evaluator	System Owner	
Scope Evaluation Context	# 5-1 [2] Σ 10-1 [2]	# 2-0 [1] Σ 10-0 [1]	# 2-0 [0] Σ 5-0 [0]	◷ 60-1 [10]
Set up Evaluation Project	# 15-1 [1] Σ 10-1 [1]	# 2-1 [1] Σ 10-1 [1]	# 15-1 [1] Σ 10-1 [1]	◷ 60-1 [5]
Conduct Evaluation	# 15-3 [10] Σ 10-1 [5]	# 6-1 [2] Σ 200-4 [30]	# 15-3 [5] Σ 60-2 [20]	◷ 100-2 [20]
Package Evaluation Results	# 15-1 [10] Σ 10-1 [4]	# 6-1 [2] Σ 20-1 [10]	# 15-1 [10] Σ 5-1 [4]	◷ 20-1 [5]
TOTAL	**Σ 40-4 [12]**	**Σ 240-4 [42]**	**Σ 80-4 [25]**	**◷ 240-5 [5]**

Fig. 11.2 Typical effort numbers for architecture evaluation. © Fraunhofer IESE (2014)

Figure 11.2 shows the data collected. It is organized along the steps to follow when conducting an architecture evaluation as introduced in Question Q.024. We distinguish three different parties involved in an architecture evaluation: The project initiator is the party asking and sponsoring the evaluation; the evaluator conducts the evaluation; and the system owner is the party responsible for the development of the software system under evaluation. Note that all three roles may be performed by the same development organization. These figures may serve as input (together with the concrete examples in Chap. 10 and the scaling factors given in Question Q.102). To convince management at the end of the day, some figures have to be given to obtain the necessary budget. The figures given in Fig. 11.2 may help, but use them with care and common sense.

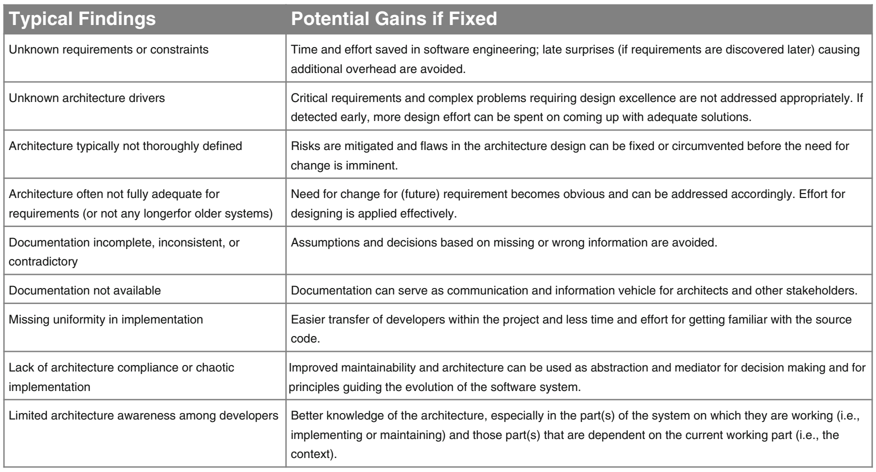
Most effort is spent on conducting the evaluation, i.e., on executing the selected checks (DIC, SAC, DQC, ACC, and CQC). As a risk-driven activity, the initiator has the authority over the priorities and the desired confidence level to be achieved. In doing so, the effort for the main drivers and scaling factors is controlled by the sponsor of the architecture evaluation.

Q.104. What Are Typical Findings and Potential Gains of Architecture Evaluation?

Findings in architecture evaluations, if addressed properly after being revealed, may lead to certain improvements. Figure 11.3 lists a set of sample findings recurring across architecture evaluation projects and discusses potential gains for a development organization.

Q.105. Why Turn Architecture Evaluation Results into Actions?

Architecture evaluation is an investment made to detect and mitigate technical risks and inadequate design decisions for a software system under evaluation. It only



Typical Findings	Potential Gains if Fixed
Unknown requirements or constraints	Time and effort saved in software engineering; late surprises (if requirements are discovered later) causing additional overhead are avoided.
Unknown architecture drivers	Critical requirements and complex problems requiring design excellence are not addressed appropriately. If detected early, more design effort can be spent on coming up with adequate solutions.
Architecture typically not thoroughly defined	Risks are mitigated and flaws in the architecture design can be fixed or circumvented before the need for change is imminent.
Architecture often not fully adequate for requirements (or not any longerfor older systems)	Need for change for (future) requirement becomes obvious and can be addressed accordingly. Effort for designing is applied effectively.
Documentation incomplete, inconsistent, or contradictory	Assumptions and decisions based on missing or wrong information are avoided.
Documentation not available	Documentation can serve as communication and information vehicle for architects and other stakeholders.
Missing uniformity in implementation	Easier transfer of developers within the project and less time and effort for getting familiar with the source code.
Lack of architecture compliance or chaotic implementation	Improved maintainability and architecture can be used as abstraction and mediator for decision making and for principles guiding the evolution of the software system.
Limited architecture awareness among developers	Better knowledge of the architecture, especially in the part(s) of the system on which they are working (i.e., implementing or maintaining) and those part(s) that are dependent on the current working part (i.e., the context).

Fig. 11.3 Findings and potential gains of architecture evaluations

pays off if the recommended improvements are actually turned into action. Otherwise the investments were made in vain (as the situation remains unchanged). Consequently, architects must always strive to exploit the results. Typically, it is not possible to cover every recommendation made, but at least some have to be realized in order to exploit the results.

Q.106. What Are Possible Improvement Actions?

Architecture evaluations are performed to increase confidence regarding decision-making about the system under evaluation. After we presented the outcome in a final meeting (see Question Q.033), our customers had to decide what to do next. We did a post-mortem analysis to find out what actions were taken afterwards. We found the following categories of action items performed by the customers (please note that combinations of multiple actions were also applied):

- **Everything OK**: In these cases nothing was done because the architecture evaluation basically did not reveal any severe findings. There were only minor points (if at all), which could be addressed as part of regular development.
- **Selection of one out of many options**: One of several candidate systems/ technologies being evaluated was actually selected (for instance, see Question Q.091). In these cases, the architecture evaluation provided valuable input to decision-making and management found itself confident to select one winner from the alternatives offered.

- **Improvement of existing architecture**: Dedicated effort was spent on improving the existing architecture. This was particularly true for projects at an early stage (without any implementation or at the beginning of development). At this point in time, corrective decisions regarding the design could be integrated into the development process. In one case, the improvement took place on both levels, in the architecture and in the implementation. On the one hand, the architecture was changed to accommodate the findings of the architecture evaluation, and on the other hand, significant numbers of architectural violations in the implementation were removed at the same time.
- **Design next-generation architecture**: The need for a complete redesign of the architecture has been accepted and decided. In these cases, instead of improving the existing architecture, the findings and insights regarding the existing systems served as input for the design of a new architecture (followed by a re-implementation from scratch). Although conceptual reuse and very limited code reuse took place, fundamental design decisions were made in the light of the findings from the architecture evaluation.
- **Removal of architecture violations**: We observed the definition of an explicit project for removing architecture violations, which were the results of the ACC. This happened in several cases and required spending significant amounts of time and effort (e.g., one team working for six months) each time only on the removal of architecture violations (i.e., changing the code to remove violations by refactoring or re-implementing).
- **Coaching architecture capabilities**: An initiative for training and improvement of architecture capabilities in the organization was started. Coaching was never done in isolation; in one case, it was performed together with an improvement of the existing architecture, and in two cases, it was performed together with the design of the new next-generation architecture and a dedicated project for removing architecture violations.
- **Project stopped**: We also had cases where all engineering activities were canceled and the product development (or the next release) was stopped completely. Management was convinced that achieving adequate quality was no longer possible with reasonable effort within a reasonable amount of time. In these cases, the implementation exhibited such severe flaws in the architecture and a high number of violations in the implementation that it was decided that it would be better to stop altogether.
- **None (although actions would be necessary)**: We also experienced a number of cases where nothing happened after the evaluation—although the results revealed many action items, either in the architecture or in the implementation. We have explicit confirmation by the customers that—in fact—nothing did happen afterwards (although it might be that we were not informed on purpose). The need was recognized and acknowledged, but neither budget nor time was allocated to actually doing something about it. The reasons are diverse, e.g., strategic shifts in the company setting a different focus for the next actions. In these cases, the investments for the architecture evaluation were made in vain because no benefits were exploited.

Q.107. Why Is Architecture Evaluation Worthwhile?

To summarize the points mentioned above, any wrong or inadequate decision prevented or any risk avoided pays off. The costs for reversing such fundamental or business-critical design decisions outweigh the investments required for performing an architecture evaluation. Avoiding just a single risk does already save more time and effort than what is required for conducting the architecture evaluation.

When contacting customers in our retrospective of past projects, there was not a single case where regret was voiced over the investments made for the architecture evaluation. Other publications on the field confirm our impression. Unfortunately, only very limited data (besides our paper Knodel and Naab 2014a) on architecture evaluation is publicly available to support our claim.

11.2 What Mistakes Are Frequently Made in Practice?

Starting architecture evaluation without management support.

Management support is crucial for getting investments, for convincing stakeholders to participate in the evaluation, for turning recommendations into actions items, and for benefiting from the results. Without management support, an architecture evaluation is likely to be an unsuccessful endeavor.

→ Questions Q.100 and Q.101.

Having no patience to wait for the evaluation results.

It takes time and effort to conduct an architecture evaluation. Architects and developers often do not want to spend that time as they feel the pressure to push on with their development tasks. Management often asks for an architecture evaluation very late, when the decision to be made based on the results is almost due. Then their patience to wait for the results is rather low.

→ Questions Q.097, Q.102 and Q.103.

Neglecting follow-up activities despite a pressing need.

An architecture evaluation often points out needs for improvement, which typically require significant effort. As architectural changes are typically effort-intensive and cannot be estimated and planned exactly, decision makers often tend to defer investments into necessary improvements, which typically leaves the project members in a state of disappointment. The great time pressure in development often leads to neglecting the required actions and to collecting more technical debt.
→ Questions Q.097, Q.098, Q.105, Q.106 and Q.113.

Architecture evaluation used for politics instead of improvement.

In some cases, we experienced that architecture evaluation was used for company-internal politics instead of for technical improvements of the underlying software system. This resulted in stakeholders trying to influence the findings, the recommendations, or the follow-up action items. We think it is crucial to have one shared goal among all stakeholders: the improvement of the software system under evaluation.
→ Questions Q.097 and Q.113.

How to Acquire Architecture Evaluation Skills? 12

Acquiring architecture evaluation skills is crucial for successful architecture evaluation projects. While this book provides answers to many questions in the area of architecture evaluation and introduces a comprehensive methodology, very good architecture evaluation skills will only come with practical experience. We will point out in the following how skills for architecture evaluation complement general architecting skills. A great source of skills is to explore a large number of software systems and their architectures in great detail and to try to follow these software systems and their maintenance over their entire lifetime. Accompanying experienced evaluators is a great source of learning: observing how they work with stakeholders, thoroughly investigating the technical solutions, and presenting the results to management at the right level of abstraction.

12.1 What Is the Point?

Q.108. What Is the Relationship to Architecture Skills in General?

The typical skills of an architect are the basis of the skills needed by a good evaluator of architectures. This comprises all areas of technical, engineering, organizational, economic, and interpersonal skills. Additionally, some specifics should be pointed out here that complement the skill set of an experienced evaluator:

J. Knodel and M. Naab, *Pragmatic Evaluation of Software Architectures*,
The Fraunhofer IESE Series on Software and Systems Engineering,
DOI 10.1007/978-3-319-34177-4_12

- **Evaluation methods**: Evaluators should know how to apply typical evaluation methods such as ATAM. The framework of different checks provided in this book provides a consistent toolbox that can be enhanced with further evaluation methods if necessary.
- **Levels of confidence**: Evaluators should know which evaluation methods can lead to which level of confidence and should be able to estimate their own level of confidence in certain situations, which might strongly depend on their experiences in the past. Only if the evaluator understands which level of confidence is required to answer an evaluation question can she determine when the evaluation can be finished or if further analyses, probably with extra effort, are necessary.
- **Manifold experiences—systems, domains, architectures, architecture patterns, technologies, companies, cultures**: All the knowledge about evaluation methods is worth nothing if the evaluator does not know about and understand software systems very well. Having gained experience with different systems from different domains with different architectures creates a background for evaluations. Experiences made in different companies and their respective cultures add further background and credibility to the evaluator. Often, evaluators are asked to compare the current findings to the situation in other companies because development teams want to compare themselves to others. In such cases, the evaluator should be able to report on comparisons.
  In general, comparing against experiences from the past is very helpful in architecture evaluations because this helps to increase the level of confidence: This solution has worked in a similar context, so there is confidence that it might work here as well.
- **Fast learning of new domains and technologies**: Evaluators of architectures often face the situation that they have to evaluate an architecture in a domain that is completely new to them. Learning the key aspects of the domain and relating them to the architecture of the software system is crucial for getting a quick and productive start into an evaluation project.
- **Accepting partial knowledge and getting credibility**: External evaluators can never have as much knowledge about a system as the architects and developers have accumulated over the years. Nevertheless, they have to work productively in the evaluation and have to accept this partial knowledge. They have to give the owners of the system the feeling that they are learning fast and asking the right questions, and that they are adding value for them.
- **Specific moderation and communication skills**: Architecture evaluations entail a lot of interview and workshop work. Evaluators have to organize and moderate this work. In situations of high criticality and when problems are encountered, sensitive moderation is necessary in order to maintain an open and cooperative climate among all parties involved in the evaluation.
- **Specific presentation skills**: In architecture evaluations, special attention does not only need to be paid to the gathering of information from stakeholders, but

also to the presentation of the (intermediate) results. An evaluator needs to have good intuition regarding how to present the results corresponding to the evaluation questions to a certain audience. This might range from a very sensitive and careful presentation to very powerful and hard formulation of the results.

Q.109. What Sources of Learning Are There?

The following sources have been proven in practice to build up skills of successful evaluators:

- **Learn from experienced evaluators**: This is the best source of learning. Accompanying evaluation projects provides many learning opportunities. In addition, reports and presentations about evaluation projects can offer many insights.
- **Read about evaluation methods**: Evaluation methods are mostly described in books and reports. Evaluators should read about these and know how to use them, if necessary.
- **Read about architectures, architectural trends, technologies**: Many information sources are available about other systems and their architectures. Evaluators can never know everything, but they should constantly try to keep up-to-date and to at least know about current trends and how they influence architectures. If possible, they should also at least try something out and make themselves comfortable with what the new technologies look like at the code level and what the architectural consequences are. Building this mental model consumes some effort as the reports and elaborations are mostly not provided in a way that points out the architectural implications. Any knowledge and experience regarding architectures and technologies is helpful for understanding a concrete system under evaluation.
- **Visit seminars for soft skills**: Soft skills seminars provide the opportunity to improve moderation, communication, and presentation skills; all three being tremendously useful for evaluators.

Q.110. How Does the Approach of This Book Relate to Other Methods?

Architecture evaluation is neither a new topic nor is this the first book on this topic. We strongly recommend reading (Rozanski and Woods 2005; Bosch 2000; Clements et al. 2001), with the latter putting particular emphasis on architecture evaluation. We do not aim at reinventing the wheel; rather, we picked up useful concepts and ideas and, where necessary, we filled in gaps in order to make them applicable efficiently and effectively in industry.

The most prominent architecture evaluation methods are: SAAM (Software Architecture Analysis Method, see Kazman et al. 1994), ATAM (Architecture Tradeoff and Analysis Method, see Clements et al. 2001, which is the de facto standard published by the Software Engineering Institute, SEI), ALMA (Architecture Level Modifiability Analysis, see Bengtsson et al. 2004), Palladio (see Becker et al. 2009). For a more comprehensive overview of architecture evaluation methods, refer to (Babar and Gorton 2009; Dobrica and Niemela 2002). The most prominent publications on compliance checking are Reflexion Models (see Murphy et al. 2001; Koschke and Simon 2003, or Knodel and Popescu 2007). For examples of compliance checking tools, refer to Question Q.078.

12.2 What Mistakes Are Frequently Made in Practice?

Focusing solely on methodical skills for the evaluation.

See next mistake.

Focusing solely on technological and system-related skills for the evaluation.

Architecture evaluation requires both methodical skills and experience with software systems as described above. Focusing on either of these skills alone is not sufficient. Evaluators have to acquire and continuously improve both types of skills.
→ Question Q.109.

Reading a book and thinking that this is all that is necessary to be able to evaluate software architectures.

In order to be able to consciously apply any method, there is no way around making practical experiences by doing it, doing it, and doing it again. The same holds true for architecture evaluation.
→ Questions Q.108, Q.109, Q.113 and Q.114.

How to Start and Institutionalize Architecture Evaluation? 13

Here, we will offer guidance on how to get started with architecture evaluation and how to institutionalize it in one's own company. There are many good opportunities for doing a first and beneficial architecture evaluation—new development, modernization of a system, or selection of new technologies. We will share best practices regarding how to have a successful start and how this success can be sustained. Finally, we will look at more sophisticated architecture evaluation methods and how the discipline can be further improved with even more dedicated guidance or a higher degree of automation, for example by integrating repeated compliance checks into the build process.

13.1 What Is the Point?

There are many good reasons for conducting architecture evaluations. Many companies already do some kind of architecture evaluation. This chapter is intended to offer support during the introduction of architecture evaluation in software-related organizations and industrial software projects in order to allow effective application of the methodology presented in this book.

Q.111. Why Is It Often Difficult to Start Architecture Evaluation?

- Architecture evaluation as a means of risk management and quality assurance is new for many companies. It is an additional investment and implies making

J. Knodel and M. Naab, *Pragmatic Evaluation of Software Architectures*,
The Fraunhofer IESE Series on Software and Systems Engineering,
DOI 10.1007/978-3-319-34177-4_13

changes in an organization. In many organizations, this obstacle has to be removed first prior to getting started.
- Architecture evaluation needs a climate of openness and cooperation. This is true for the technical level as well as for the connection to the managerial level. Presenting a system's architecture with the goal of checking and improving it requires an organizational culture that is open to accept mistakes and eager to learn and improve. However, many organizations still do not have this type of culture.
- Architecture evaluation is an investment without immediate return. It requires spending the time of people who are typically very busy. Spending the necessary time and budget requires trusting that the architecture evaluation will be worthwhile. Often the necessity of architecture evaluation is recognized too late, when the risk has materialized already and is a real problem. In such an event, an architecture evaluation can still help to assess the situation and make decisions, but the resulting cost is typically much higher.
- Architecture evaluation requires additional investments into evaluators: This may mean qualifying internal architects with the required skills or hiring external evaluators as consultants.
- Most of the time, architecture evaluation is not a one-time activity. Rather, it accompanies software projects over their lifecycle with an emphasis on different checks and with different degrees of intensity. Viewing it as a continuous risk management activity is even harder than treating it as a one-time investment and requires strong discipline.

Q.112. What Opportunities Exist for Starting with Architecture Evaluation?

Software projects lead to many situations that are good starting points for architecture evaluation. Chapter 1 introduced numerous motivations for architecture evaluation. Below, we will briefly mention several situations that occur quite often and that are well suited for this purpose.

- **Architecture for a new system is just under design**: Checking the foundation of architecture drivers and their initial adequacy early can help to avoid expensive rework. Most of the time, an internal review is adequate and can deliver enough confidence. DIC, SAC, and DQC can be conducted.
- **Initial development of a new system is under way**: During the initial development, teams are ramped up and the system is developed under high time pressure. Questions pop up continually that are not yet answered by the architecture design so far: In such cases, a quick SAC is advisable from time to time to check whether the architecture should be changed. There is the risk that architectural ideas and the implementation might diverge and that uniformity may be missing. Thus, ACC can help to avoid a very early drift, which will be very expensive to correct later on.

- **Legacy system has to be renovated or replaced**: Decisions about the future of legacy systems are strongly dependent on the architecture of the system (as implemented) and the overall quality of artifacts such as documentation and code quality. To allow making informed decisions about legacy systems, architecture evaluation provides a full set of means to reveal the facts needed for making a decision. CEC and SAC are needed to check the current set of architecture drivers and the adequacy of the implemented architecture, which often has to be reconstructed first. DQC helps to check whether the documentation is still useful or rather outdated and useless. ACC helps to analyze the current implementation and how far it has drifted away from the assumed and formerly intended architecture. Additionally, ACC can be used to monitor the progress in migration projects towards a new target architecture. CQC helps to assess the code quality of the legacy system in general.
- **Software system or technology to be acquired; several candidates have to be compared**: Every system or technology acquired comes with its own architecture, which has implications on its usage and integration into other systems or system landscapes. Thus, checking whether a system to be acquired is adequate for one's own drivers is very advisable (SAC). Not so obvious is the fact that it may also be advisable to conduct an ACC and a CQC of such systems since these report on the inner quality of the system or technology. This inner quality has impact on maintenance cost in the later lifecycle.

Q.113. What Are Best Practices for Starting and Institutionalizing Architecture Evaluation?

- **Qualify people**: Architecture evaluation is only meaningful if skilled people do it. In the beginning, learning from experienced external people can help. Qualification should aim in particular at methodical skills; architectural skills have to be acquired through experience over many years of work.
- **Define evaluation goals**: Clear goals help to shape clear expectations, to select the right methodology, and to estimate the level of confidence needed as well as the resulting effort.
- **Evaluate early and regularly**: Every software project should have an early quality gate where an architecture evaluation is conducted.
- **Start small**: Do not overwhelm people with heavyweight evaluation activities. Rather start with lightweight activities and systematically produce fast results. To do so, a selective evaluation focusing on the evaluation goals and architecture drivers with the highest priority is beneficial.
- **Emphasize importance and allocate effort**: Make clear to all involved stakeholders that architecture evaluation is an important means of risk management and quality assurance. Make adequate provision for it in the project plan.

- **Turn results into visible actions**: Evaluation results that are simply ignored have a very demotivating impact. Thus, make it clear early on how the evaluation results are treated and maintain a visible backlog of derived actions.
- **Create success stories**: There will always be people who have to be convinced of the benefits of architecture evaluation. In the early phase, make sure to spend time on doing it according to the methods and to create transparency with regard to the architecture evaluation activities. Invest rather a bit more effort into the presentation of the results and keep all stakeholders involved through intense communication.
- **Proliferate culture of cooperation**: This point is not restricted to the topic of architecture evaluation, but architecture evaluation strongly depends on an open and cooperative climate. Changing the overall climate in an organization is a broad topic and requires time, and is beyond the scope of this book. However, we often recognize its impact and thus want to emphasize its importance.
- **Institutionalize quality gates**: Every organization can find situations that are perfectly suited as quality gates that require an architecture evaluation. For example, the procurement of new software products could be such a situation ("architecture-centric procurement").

Q.114. How to Make Architecture Evaluation Sustainable Within a Development Organization?

We recommend applying a goal-oriented measurement approach (for instance, the GQM paradigm, see Basili and Weiss 1984) to facilitate successful institutionalization of architecture evaluation. Architecture evaluation as an instrument for ensuring the quality of the software system needs to be aligned with the overall business and development goals when developing software. We further believe that incremental introduction leads to increased adoption over time. Tailoring architecture evaluation to the characteristics of the work environment is another crucial aspect.

The main task of a development organization is to deliver products to the market; hence, they have only limited resources available for learning new approaches and applying tools to produce architecture evaluation results. Consequently, architecture evaluation must be customized to allow smooth integrated into an organization.

We recommend raising architectural awareness within the organization. We think that one key enabler for technology transfer in general is to build up knowledge in the minds of the affected stakeholders. Thus, to raise awareness for architecting and for the need for architecture evaluation in the minds of stakeholders, it is not sufficient to provide them with documentations of the approach or presentations about it. The support of a team of champions (a special team of skilled evaluators spreading the word, communicating success stories, and being available for support) is required. These people need to possess the skills that are necessary for architecture evaluation.

Q.115. What Are Promising Improvements of the Evaluation Methods?

Architecture evaluation has a sound foundation in research and industry. Although successfully applied for many years and across thousands of systems, there is still a lot of methodical and technical improvement potential. In this section, we will sketch promising areas that we think would bring further benefits to practitioners. In part, they are already available in research prototypes, but they are not widely adopted in practice.

- **Automated and high-frequency checks**: Mostly, architecture evaluations are triggered manually and conducted in rather longer intervals. However, the longer the time from one check to the next, the larger the drift can get. Thus, high-frequency checks are desirable. Checks such as compliance checking, which, once initialized, can be run automatically can also be integrated into continuous integration and build environments (Knodel 2011).
- **Integration with runtime operation monitoring**: Architecture evaluation is mainly a design time/development time activity. Observing the real behavior of a system while it is being operated can deliver further insights and can be used for calibrating prediction and simulation models.
- **Better tool support of architecture reconstruction**: Architecture reconstruction from code is still a cumbersome and highly manual effort. Many research activities are ongoing to provide further tool support, but they are mostly not ready yet for practical application.
- **More specific quality models**: In their most general form, architecture evaluation methods are applicable for all types of requirements and quality attributes. This means, on the other hand, that there is not much guidance with respect to quality attributes. The impact of the evaluator's experience is then pretty high. There are already specializations for several quality attributes such as modifiability (Bengtsson et al. 2004; Naab 2012; Stammel 2015), security (Ryoo et al. 2015), performance (Becker et al. 2009), and others. Such specializations always come with trade-offs such as learning effort and limited generality.
- **Integration with effort estimation**: Evaluating an architecture is often done with respect to upcoming changes and evolution. The respective changes to a system result in effort and time being needed and strongly depend on the current situation of the architecture and on the code quality (including also the effects of accumulated technical debt). Thus, combining the evaluation of the architecture and the estimation of efforts seems to be a promising synergy (Stammel 2015).
- **Better tool support for documenting workshops and evaluation projects**: Today, conducting architecture evaluation workshops and preparing results and presentations is still tedious manual work, which is mostly done using standard

office tools. A better supported workflow with more automation would be beneficial. This could include:

- Taking pictures of whiteboard sketches
- Generating scenario lists and prioritization tables
- Managing the traces between scenarios and solution concepts and generating readable tables
- Generating expressive charts about the evaluation results for management presentations
- Automatically updating charts in all presentations following parameter adjustment and recalculation of evaluations
- And a lot more …

Despite all potential improvements of architecture evaluation techniques and tools, architecture evaluation will remain an area that requires experienced people and manual effort. However, recurring and tedious tasks could be gradually eliminated and tools could guide architects with more and more meaningful suggestions.

13.2 What Mistakes Are Frequently Made in Practice?

Postponing the start of architecture evaluation activities.

Waiting for the point in time in a software project where the architecture is stable enough or when enough time is available often leads to continuous postponing of architecture evaluation activities.

→ Questions Q.111 and Q.112

Always giving the architecture evaluation reduced priority.

Short-term fixes, firefighting, or continuous release pressure often have higher priority than architecture evaluation, as these are directly visible, constructive activities. As time is always scarce, we have often observed that future-oriented and risk-mitigating activities such as architecture evaluation get neglected.

→ Questions Q.028, Q.029, Q.031 and Q.113

14 What Are the Key Takeaways of Architecture Evaluation?

Q.116. What Is the Key Message of This Book?

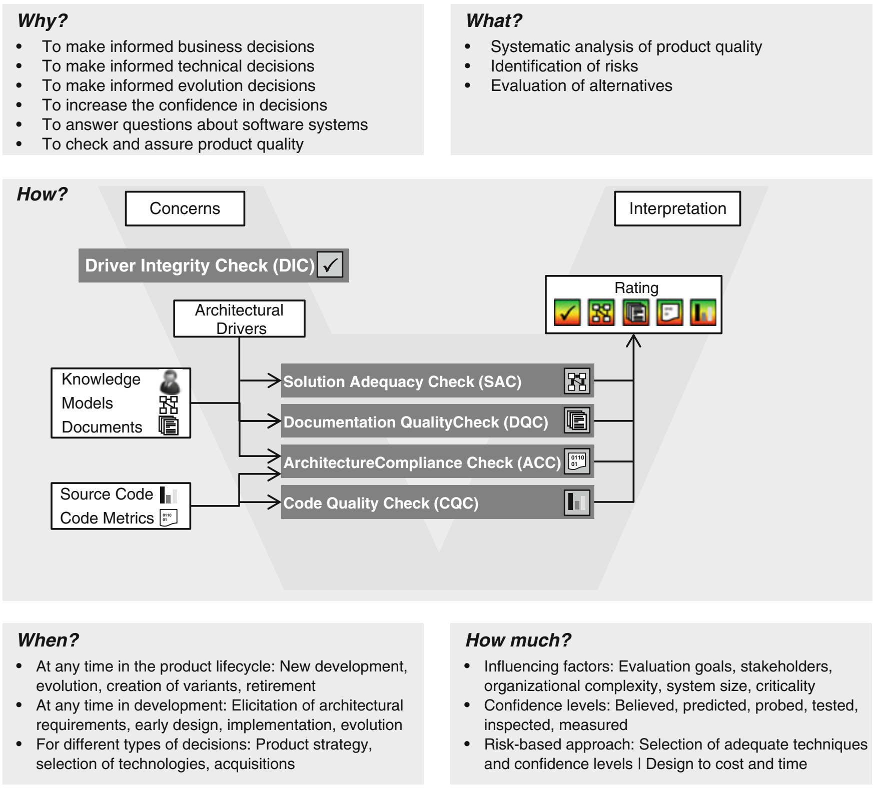
Thorough and continuous architecting is the key to overall success in software engineering, and architecture evaluation is one crucial part of architecting.

- Evaluate your architecture—early and regularly, and exploit the results!
  - Be aware of goals and evaluation questions when selecting checks!
  - Tailor and adapt to the software system, its context, and the situation at hand!
  - Be careful in the interpretation of results as this is the most challenging step!
- Increase the value of the software system!
  - Turn results into improvement actions—if not, all effort was invested in vain!
  - Start small and convince with success stories!

J. Knodel and M. Naab, *Pragmatic Evaluation of Software Architectures*,
The Fraunhofer IESE Series on Software and Systems Engineering,
DOI 10.1007/978-3-319-34177-4_14

Q.117. What Is the Essence of Architecture Evaluation?

The essence of architecture evaluation is summarized in the following figure (Fig. 14.1).



Why?
- To make informed business decisions
- To make informed technical decisions
- To make informed evolution decisions
- To increase the confidence in decisions
- To answer questions about software systems
- To check and assure product quality

What?
- Systematic analysis of product quality
- Identification of risks
- Evaluation of alternatives

When?
- At any time in the product lifecycle: New development, evolution, creation of variants, retirement
- At any time in development: Elicitation of architectural requirements, early design, implementation, evolution
- For different types of decisions: Product strategy, selection of technologies, acquisitions

How much?
- Influencing factors: Evaluation goals, stakeholders, organizational complexity, system size, criticality
- Confidence levels: Believed, predicted, probed, tested, inspected, measured
- Risk-based approach: Selection of adequate techniques and confidence levels | Design to cost and time

Fig. 14.1 The essence of architecture evaluation

About Fraunhofer IESE

Fraunhofer IESE in Kaiserslautern is one of the worldwide leading research institutes in the area of software and systems engineering. A major portion of the products offered by its customers is defined by software. These products range from automotive and transportation systems via automation and plant engineering, information systems, healthcare and medical systems to software systems for the public sector. The institute's software and systems engineering approaches are scalable, which makes Fraunhofer IESE a competent technology partner for organizations of any size—from small companies to major corporations.

Under the leadership of Prof. Peter Liggesmeyer and Prof. Dieter Rombach, the contributions of Fraunhofer IESE have been a major boost to the emerging IT hub Kaiserslautern for more than 20 years. In the Fraunhofer Information and Communication Technology Group, the institute is cooperating with other Fraunhofer institutes to develop trendsetting key technologies for the future.

Fraunhofer IESE is one of the 67 institutes of the Fraunhofer-Gesellschaft. Together they have a major impact on shaping applied research in Europe and contribute to Germany's competitiveness in international markets.

J. Knodel and M. Naab, *Pragmatic Evaluation of Software Architectures*,
The Fraunhofer IESE Series on Software and Systems Engineering,
DOI 10.1007/978-3-319-34177-4

Bibliography

(aim42 2014) aim42, Gernot Starke, “Architecture Improvement Method”, http://aim42.org/, 2014.

(Ayewah et al. 2007) N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, Y. Zhou, “Evaluating static analysis defect warnings on production software”. ACM SIGPLAN-SIGSOFT Workshop on Program analysis for software tools and engineering, 2007.

(Babar & Gorton 2009) M.A. Babar, I. Gorton, “Software Architecture Reviews: The State of the Practice”, IEEE Computer, 42(7): pp. 26-32, 2009.

(Baggen et al. 2012) R. Baggen, J.P. Correia, K. Schill, J. Visser. “Standardized Code Quality Benchmarking for Improving Software Maintainability”, Software Quality Journal, Volume 20 - 2, 287-307, 2012.

(Basili & Weiss 1984) V.R. Basili, D.M. Weiss, “A Methodology for Collecting Valid Software Engineering Data,” IEEE Transactions on Software Engineering, vol. SE-10, no. 6, pp. 728–738, 1984.

(Basili et al. 1996) V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørumgård, M. V. Zelkowitz. “The empirical investigation of Perspective-Based Reading“. Empirical Software Engineering, January 1996, Volume 1, Issue 2, pp 133-164, 1996.

(Basili et al. 2014) V. Basili, A. Trendowicz, M. Kowalczyk, J. Heidrich, C. Seaman, J. Münch, D. Rombach, ``Aligning Organizations Through Measurement', Berlin: Springer-Verlag, 2014.

(Baxter et al. 1998) I. Baxter, A. Yahin, L. Moura, M. Sant’Anna, L. Bier. “Clone detection using abstract syntax trees”. IEEE International Conference on Software Maintenance (ICSM), 1998.

(Becker et al. 2009) S. Becker, H. Koziolek, R. Reussner, “The Palladio component model for model-driven performance prediction“, Journal of Systems and Software, vol. 82(1), pp. 3-22, January 2009.

(Bellomo et al. 2015) S. Bellomo, I. Gorton, R. Kazman, “Toward Agile Architecture – Insights from 15 Years of ATAM Data”. IEEE Software, vol 32, no 5, pp. 38-45, 2015.

(Bengtsson et al. 2004) P. Bengtsson, N. Lassing, J. Bosch, H. van Vliet, “Architecture-level modifiability analysis (ALMA)”, Journal of Systems and Software, vol 69(1-2), pp. 129-147, January 2004.

(Bessey et al. 2010) A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S Hallem, D. Engler. “A few billion lines of code later: using static analysis to find bugs in the real world”. Communication of ACM, 53(2), 66-75, 2010.

(Boehm 1981) B.W. Boehm, “Software Engineering Economics”, Englewood Cliffs, NJ : Prentice-Hall, 1981.

(Bosch 2000) J. Bosch, “Design and use of software architectures. Adopting and evolving a product-line approach”, ACM Press, 2000.

(Bourquin & Keller 2007) F. Bourquin, R.K. Keller, „High-impact Refactoring Based on Architecture Violations”. European Conference on Software Maintenance and Reengineering (CSMR), 2007.

(Chidamber & Kemerer 1994) S.R. Chidamber, C.F. Kemerer, “A metrics suite for object oriented design”, IEEE Transactions on Software Engineering, 20:476–493, 1994

J. Knodel and M. Naab, *Pragmatic Evaluation of Software Architectures*,
The Fraunhofer IESE Series on Software and Systems Engineering,
DOI 10.1007/978-3-319-34177-4

(Clements et al. 2001) P. Clements, R. Kazman, L. Bass, “Evaluating Software Architectures”, Addison Wesley, 2001.

(Clements et al. 2010) P. Clements, D. Garlan et al., “Documenting Software Architectures: Views and Beyond”. 2nd Edition. Pearson Education, 2010.

(Deissenboeck et al. 2009) F. Deissenboeck, E. Juergens, K. Lochmann, S. Wagner. ”Software quality models: Purposes, usage scenarios and requirements” ICSE Workshop on Software Quality, 2009.

(Dobrica & Niemela. 2002) L. Dobrica, E. Niemala, “A survey on software architecture analysis methods”, IEEE Transactions on Software Engineering, vol 28(7), pp. 638-653, July 2002.

(Ferenc et al. 2014) R. Ferenc, P. Hegedűs, T. Gyimóthy. ”Software Product Quality Models” Evolving Software Systems. Springer, pages 65-100, 2014.

(Fjelstad & Hamlen 1983) R. K. Fjelstad, W. T. Hamlen, “Application program maintenance study: report to our respondents”, G. Parikh and N. Zvegintzov, eds. “Tutorial on Software Maintenance”. Los Angeles, CA: IEEE Computer Society Press, 11 27, 1983.

(Forster et al. 2013) T. Forster, T. Keuler, J. Knodel, M. Becker, “Recovering Component Dependencies Hidden by Frameworks – Experiences from Analyzing OSGi and Qt” European Conference on Software Maintenance and Reengineering (CSMR), 2013.

(Garlan & Ockerbloom 1995) D. A. Garlan, J. R. Ockerbloom, „Architectural mismatch: Why reuse is so hard” IEEE Software, 12(6), 17 26, 1995.

(Godfrey & Lee 2000) M. W. Godfrey, E. H. S. Lee, “Secrets from the monster: Extracting Mozilla's software architecture”, International Symposium on Constructing software engineering tools (CoSET), 2000.

(Halstead 1977) M. H. Halstead, “Elements of Software Science”, Ser. Operating, and Programming Systems. New York: Elsevier, vol. 7, 1977.

(Heitlager et al. 2007) I. Heitlager, T. Kuipers, J. Visser, “A practical model for measuring maintainability”. International Conference on Quality of Information and Communications Technology (QUATIC), pages 30–39, 2007.

(ISO 25010, 2011) ISO/IEC 25010, “Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models”, 2011.

(ISO/IEC 42010, 2011) ISO/IEC 42010, “Systems and software engineering — Architecture description”, 2011

(Kazman et al. 1994) R. Kazman, L. Bass, M. Webb, G. Abowd, “SAAM: a method for analyzing the properties of software architectures”, International Conference on Software Engineering (ICSE), 1994.

(Keuler et al. 2011) T. Keuler, J. Knodel, M. Naab. Fraunhofer ACES: Architecture-Centric Engineering Solutions. Fraunhofer IESE, IESE-Report, 079.11/E, 2011.

(Keuler et al. 2012) T. Keuler, J. Knodel, M. Naab, D. Rost, “Architecture Engagement Purposes: Towards a Framework for Planning `Just Enough'-Architecting in Software Engineering”, WICSA/ECSA, 2012.

(Knodel 2011) J. Knodel, “Sustainable Structures in Software Implementations by Live Compliance Checking”. Dissertation, Fraunhofer Verlag, 2011.

(Knodel et al. 2009) J. Knodel, S. Duszynski, M. Lindvall, “SAVE: Software Architecture Visualization and Evaluation” European Conference on Software Maintenance and Reengineering (CSMR), 2009.

(Knodel et al. 2006) J. Knodel, D. Muthig, M. Naab, M. Lindvall, “Static Evaluation of Software Architectures. Conference on Software Maintenance and Reengineering (CSMR), 2006.

(Knodel et al. 2008) J. Knodel, D. Muthig, U. Haury, G. Meier, “Architecture Compliance Checking - Experiences from Successful Technology Transfer to Industry”, European Conference on Software Maintenance and Reengineering (CSMR), 2008.

(Knodel & Popescu 2007) J. Knodel, D. Popescu, “A Comparison of Static Architecture Compliance Checking Approaches”, Working IEEE/IFIP Conference on Software Architecture (WICSA), 2007

(Knodel & Muthig 2008) J. Knodel, D. Muthig, “A Decade of Reverse Engineering at Fraunhofer IESE - The Changing Role of Reverse Engineering in Applied Research”, 10th Workshop Software Reengineering (WSR), 2008.

(Knodel & Naab 2014a) J. Knodel, M. Naab, “Mitigating the Risk of Software Change in Practice - Retrospective on More Than 50 Architecture Evaluations in Industry (Keynote Paper)”, IEEE CSMR-18/WCRE-21 Software Evolution Week, 2014.

(Knodel & Naab 2014b) J. Knodel, M. Naab, “Software Architecture Evaluation in Practice: Retrospective on more than 50 Architecture Evaluations in Industry”, International Working Conference on Software Architecture (WICSA), 2014.

(Kolb et al. 2006) R. Kolb, I. John, J. Knodel, D. Muthig, U. Haury, G. Meier, “Experiences with Product Line Development of Embedded Systems at Testo AG”. International Software Product Line Conference (SPLC), 2006.

(Koschke & Simon 2003) R. Koschke, D. Simon, “Hierarchical Reflexion Models”. Working Conference on Reverse Engineering (WCRE), 2003.

(Koziolek 2011) H. Koziolek, “Sustainability evaluation of software architectures: A systematic review”. QoSA, 2011.

(Kruchten 1995) P. Kruchten, “The 4+1 View Model of Architecture,” IEEE Software, vol. 12, no. 6, pp. 42–50, 1995.

(Kuhn et al. 2013) T. Kuhn, T. Forster, T. Braun, R. Gotzhein, “FERAL - Framework for simulator coupling on requirements and architecture level”. MEMOCODE, 2013.

(Lehman & Belady 1985) M. M. Lehman, L. A. Belady, “Program evolution: processes of software change”. Academic Press Professional, Inc, 1985.

(Lilienthal 2015) C. Lilienthal, “Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen“. dpunkt Verlag, 2015.

(Lindvall et al. 2005) M. Lindvall, I. Rus, F. Shull, M. V. Zelkowitz, P. Donzelli, A. Memon et al., “An Evolutionary Testbed for Software Technology Evaluation”, Innovations in Systems and Software Engineering - a NASA Journal,, 1(1), 3-11, 2005.

(McCabe 1976) T. McCabe, ``A Complexity Measure'. IEEE Transaction on Software Engineering, 308–320, 1976.

(MISRA 2004) MISRA, “Guidelines for the use of the C language in critical systems”, MIRA Ltd., 2004.

(Mordal-Manet et al. 2009) K. Mordal-Manet, F. Balmas, S. Denier, S. Ducasse, H. Wertz, J. Laval, F. Bellingard, P. Vaillergues, “The Squale Model – A Practice-based Industrial Quality Model”, IEEE International Conference on Software Maintenance (ICSM), 2009.

(Murphy et al. 2001) G. C. Murphy, D. Notkin, K. J. Sullivan, “Software reflexion models: bridging the gap between design and implementation”, IEEE Transactions on Software Engineering, 27(4), 364-380, 2001.

(Naab 2012) M. Naab, “Enhancing Architecture Design Methods for Improved Flexibility in Long-Living Information Systems”. Dissertation, Fraunhofer Verlag, 2012.

(Pollet et al. 2007) D. Pollet, S. Ducasse, L. Poyet, I. Alloui, S. Cimpan, H. Verjus, “Towards A Process-Oriented Software Architecture Reconstruction Taxonomy”, European Conference on Software Maintenance and Reengineering (CSMR), 2007.

(Rosik et al. 2008) J. Rosik, A. L. Gear, J. Buckley, M.A. Babar, “An industrial case study of architecture conformance”, ACM-IEEE international symposium on Empirical Software Engineering and Measurement (ESEM), 2008.

(Rost et al. 2015) D. Rost, B. Weitzel, M. Naab, T. Lenhart, H. Schmitt, “Distilling Best Practices for Agile Development from Architecture Methodology – Experiences from Industrial Application”. ECSA, 2015.

(Roy et al. 2014) C. K. Roy, M. F. Zibran, R. Koschke, “The vision of software clone management: Past, present, and future (Keynote paper)”, IEEE CSMR-18/WCRE-21 Software Evolution Week, 2014.

(Rozanski & Woods 2005) N. Rozanski, E. Woods, “Software Systems Architecture : Working With Stakeholders Using Viewpoints and Perspectives”, Addison-Wesley, 2005

(Ryoo et al. 2015) J. Ryoo, R. Kazman, P. Anand, "Architectural Analysis for Security". IEEE Security & Privacy 13(6): 52-59, 2015.

(Schulenklopper et al. 2015) J. Schulenklopper, E. Rommes, "Why They Just Don't Get It: Communicating Architecture to Business Stakeholders". SATURN, https://youtu.be/_PY2ZRgesEc, 2015.

(Stammel 2015) J. Stammel, "Architektur-Basierte Bewertung und Planung von Änderungsanfragen", Dissertation, KIT Scientific Publishing, 2016.

(Starke & Hruschka 2015) G. Starke, P. Hruschka, "arc42: Pragmatische Hilfe für Software-architekten", Hanser Verlag, 2015.

(Smit et al. 2011) M. Smit, B. Gergel, H.J. Hoover, E. Stroulia, "Code convention adherence in evolving software". IEEE International Conference on Software Maintenance (ICSM), 2011.

(Toth 2015) S. Toth, "Vorgehensmuster für Softwarearchitektur: Kombinierbare Praktiken in Zeiten von Agile und Lean", 2. Auflage, Hanser Verlag, 2015.

(Visser et al 2016) J. Visser, P. van Eck, R. van der Leek, S. Rigal, G. Wijnholds, "Building Maintainable Software – Ten Guidelines for Future-Proof Code". O'Reilly Media, 2016.

(Wagner et al. 2015) S. Wagner, A. Goeb, L. Heinemann, M. Kläs, C. Lampasona, K. Lochmann, A. Mayr, R. Plösch, A. Seidl, J. Streit, A. Trendowicz, "Operationalised product quality models and assessment: The Quamoco approach." Information & Software Technology 62: 101-123, 2015.

(Zakrzewski 2015) S. Zakrzewski, "An Overview of Mechanisms for Improving Software Architecture Understandability". Bachelor Thesis, Department of Computer Science, Technical University of Kaiserslautern, http://publica.fraunhofer.de/dokumente/N-382014.html, 2015.

(Zörner 2015) S. Zörner, "Softwarearchitekturen dokumentieren und kommunizieren: Entwürfe, Entscheidungen und Lösungen nachvollziehbar und wirkungsvoll festhalten". 2. Auflage, Hanser Verlag, 2015.