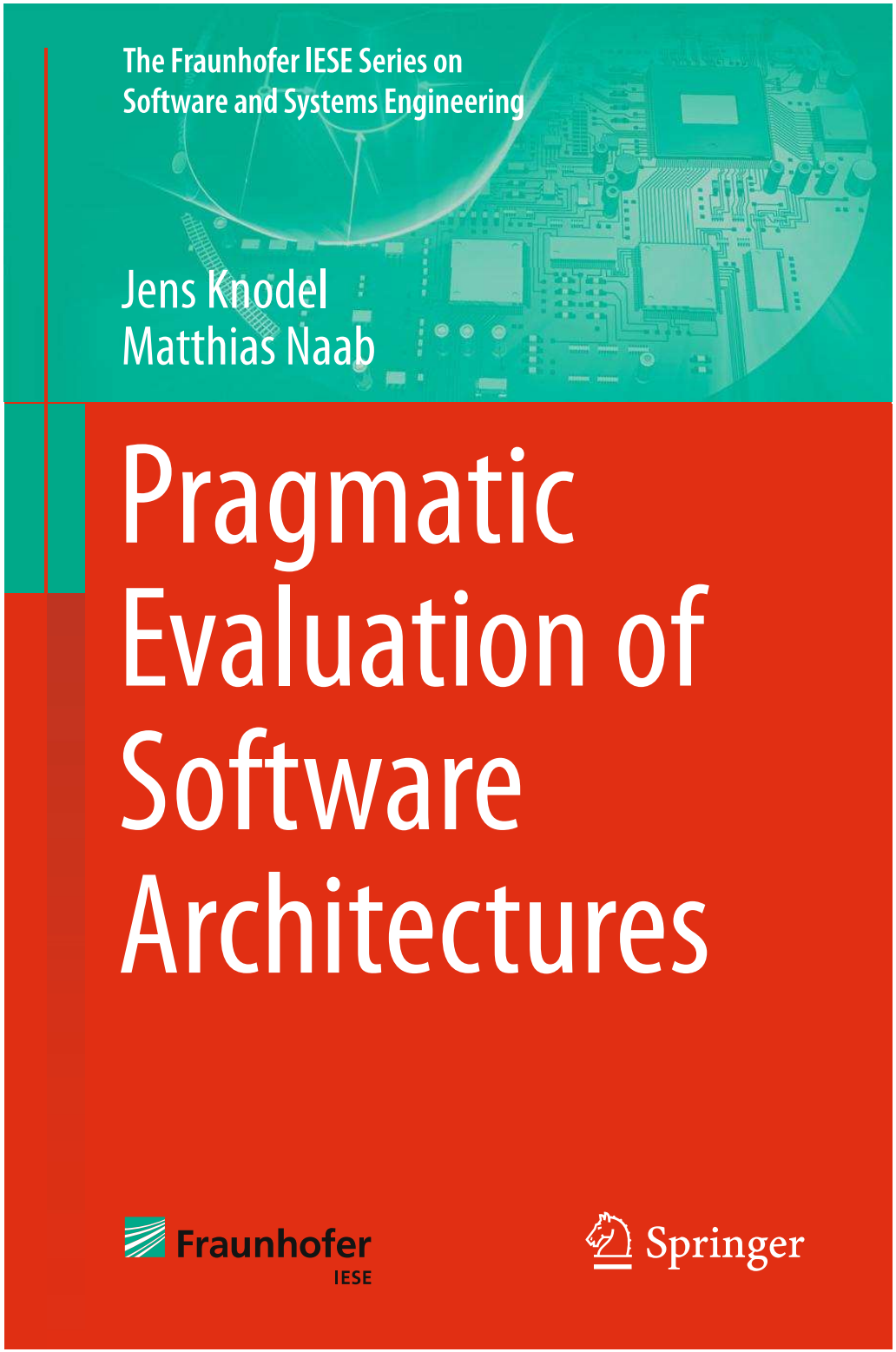

Part II

How to Evaluate Architectures Effectively and Efficiently?

How to Perform the Driver Integrity Check (DIC)?

5

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

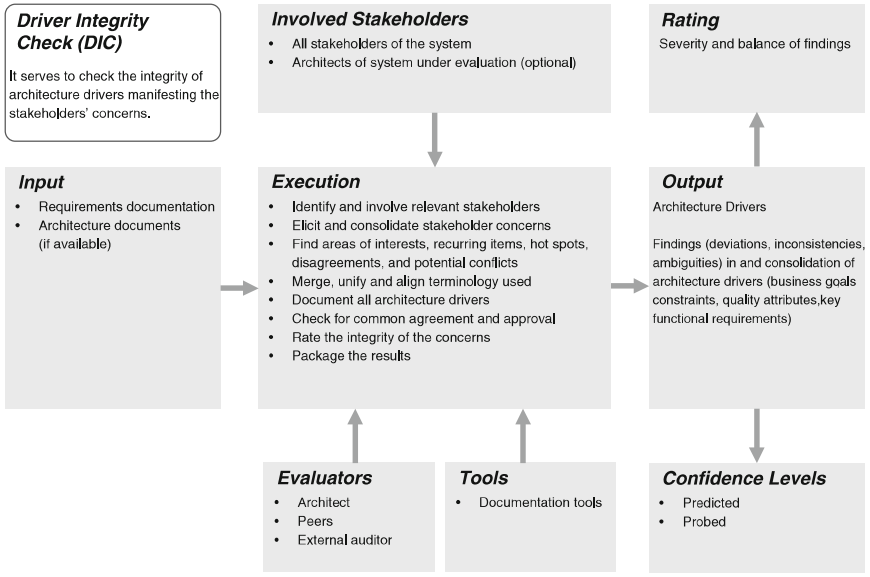

Fig. 5.1 DIC overview

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	
Driver Name	Application startup time
Driver ID	AD.01.PERFORMANCE
Status	Realized
Priority	High

Responsibilities	
Supporter	
Sponsor	
Author	
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.

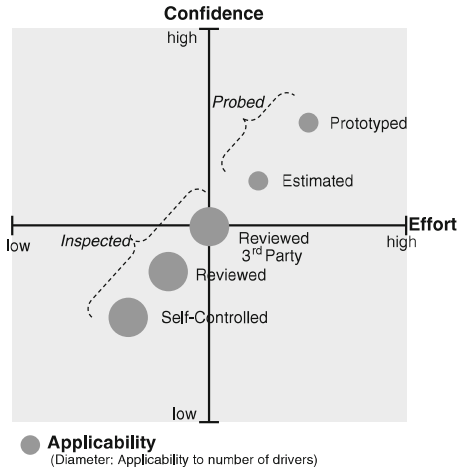

Fig. 5.3 DIC confidence levels. © Fraunhofer IESE (2015)

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

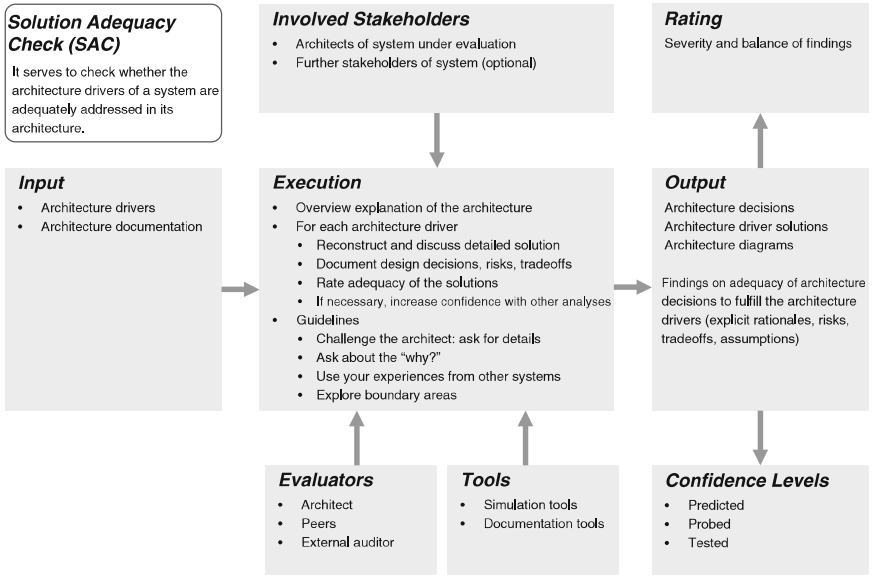

Fig. 6.1 SAC overview

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

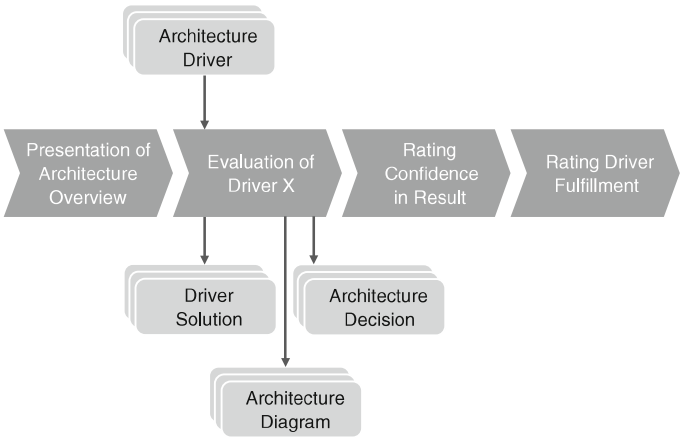

Fig. 6.2 SAC procedure. © Fraunhofer IESE (2014)

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.

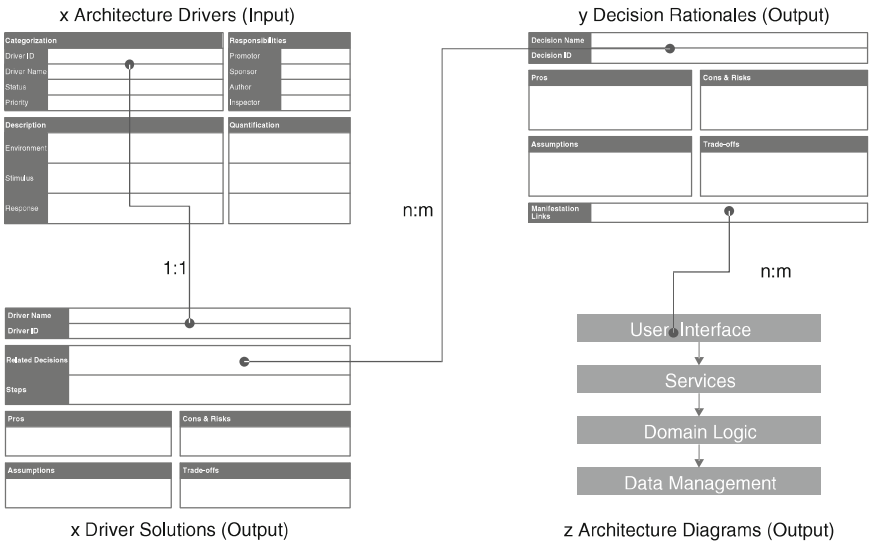

Fig. 6.3 Relationships—drivers, solutions, decisions, and diagrams. © Fraunhofer IESE (2012)

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.

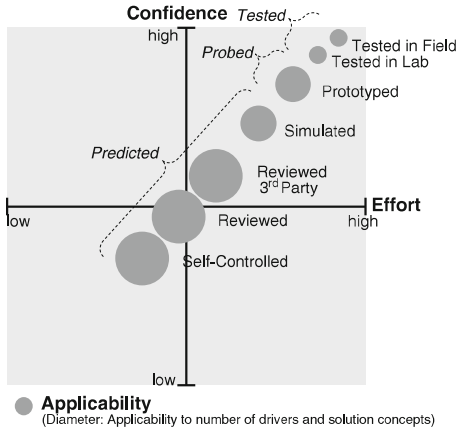

Fig. 6.6 SAC confidence levels. © Fraunhofer IESE (2015)

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.

How to Perform the Documentation Quality Check (DQC)?

7

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.

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_7

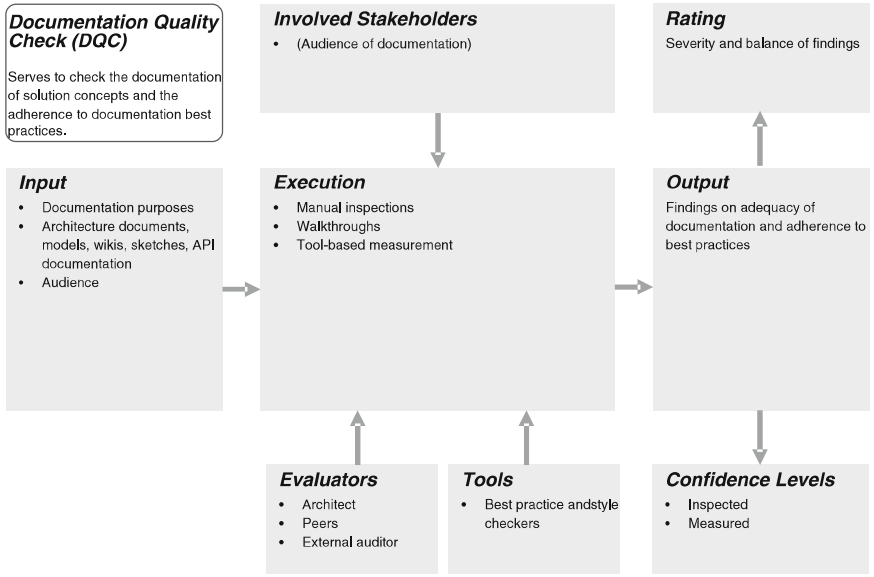

Fig. 7.1 DQC overview

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

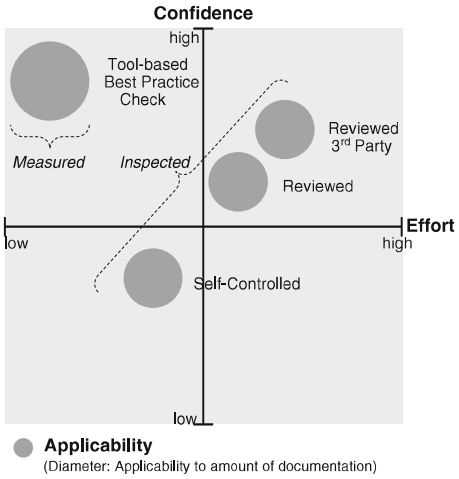

Fig. 7.2 DQC confidence levels. © Fraunhofer IESE (2015)

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

How to Perform the Architecture Compliance Check (ACC)?

8

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

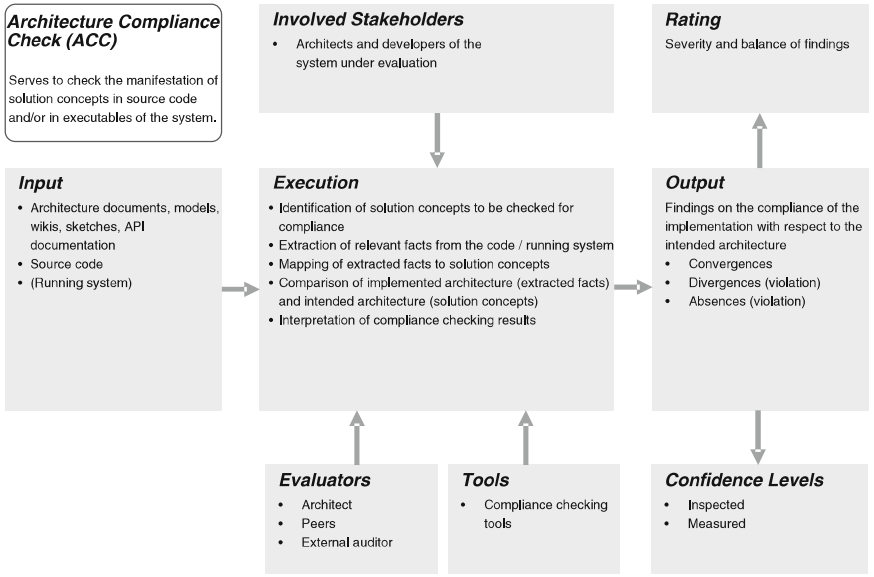

Fig. 8.1 ACC overview

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.

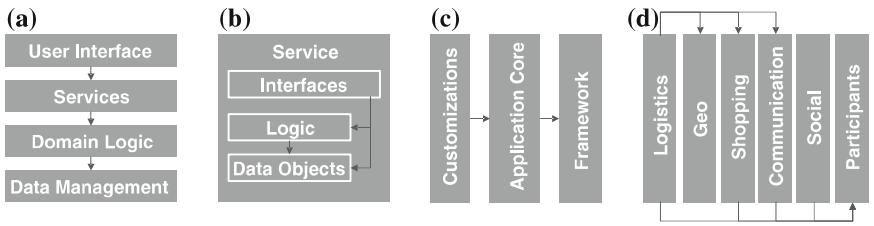

Fig. 8.2 ACC solution concept input examples. © Fraunhofer IESE (2013)

- 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.

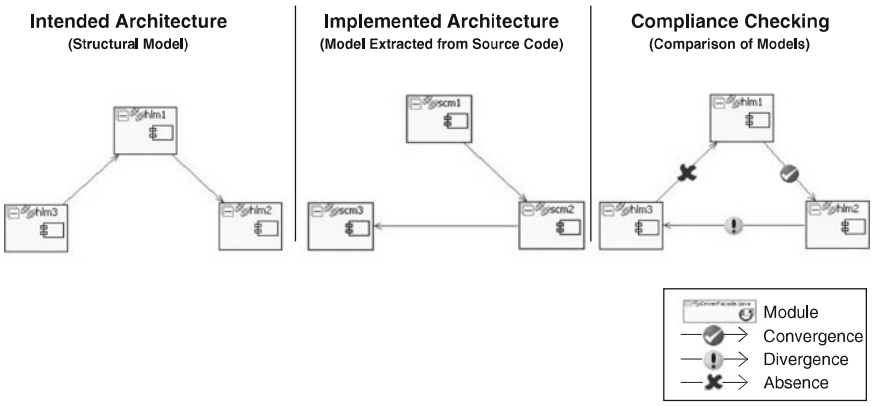

Fig. 8.3 ACC Output of structural checking. © Fraunhofer IESE (2006)

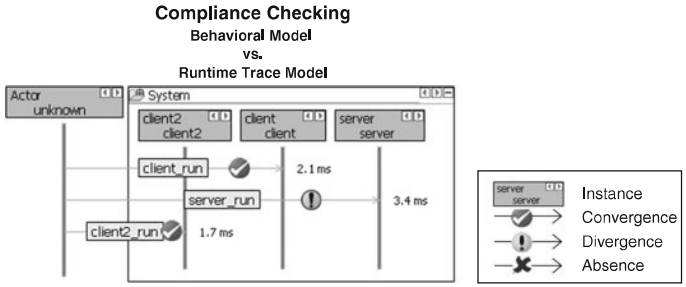

Fig. 8.4 ACC output of behavioral checking. © Fraunhofer IESE (2009)

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).

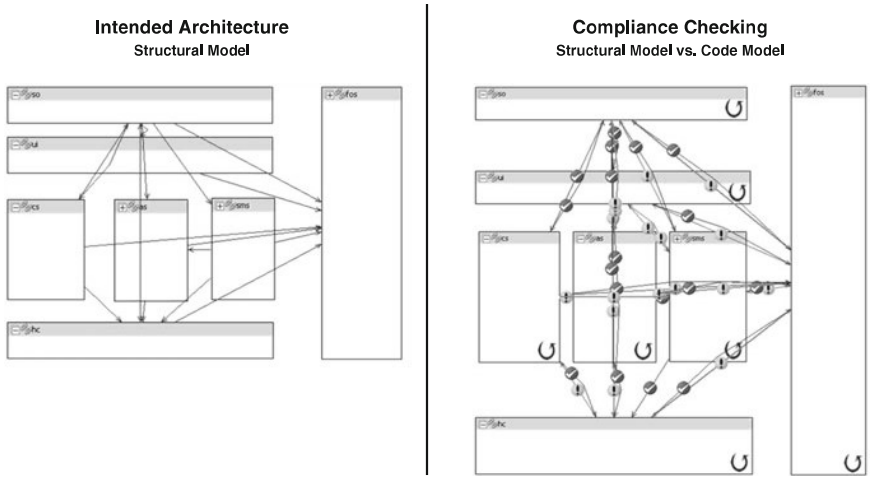

Fig. 8.5 ACC example results: compliance check for layered architecture. © Fraunhofer IESE (2008)

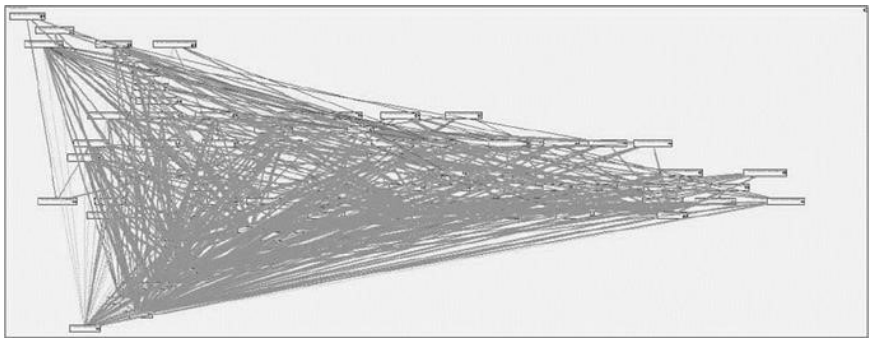

Fig. 8.6 ACC example results: visualization of critical gap in layered architecture. © Fraunhofer IESE (2008)

- **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

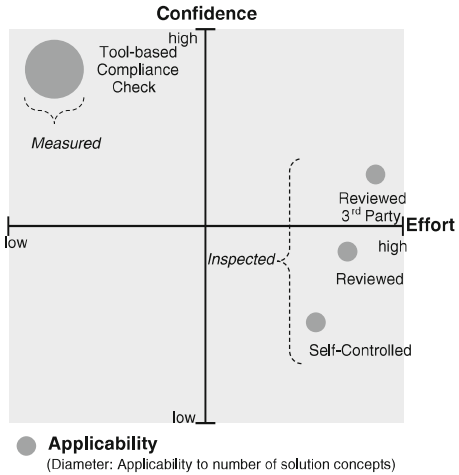

Fig. 8.7 ACC confidence levels. © Fraunhofer IESE (2015)

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.

How to Perform the Code Quality Check (CQC)?

9

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).

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

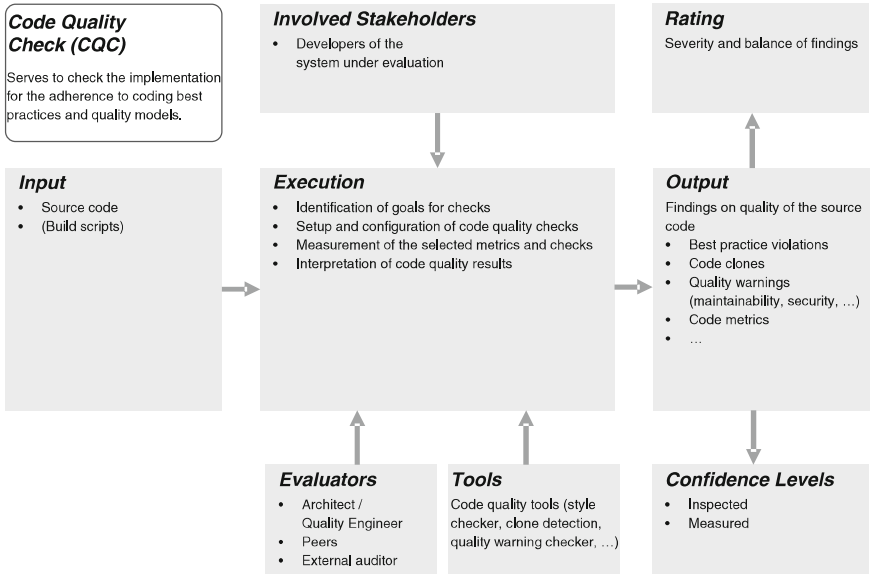

Fig. 9.1 CQC overview

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.

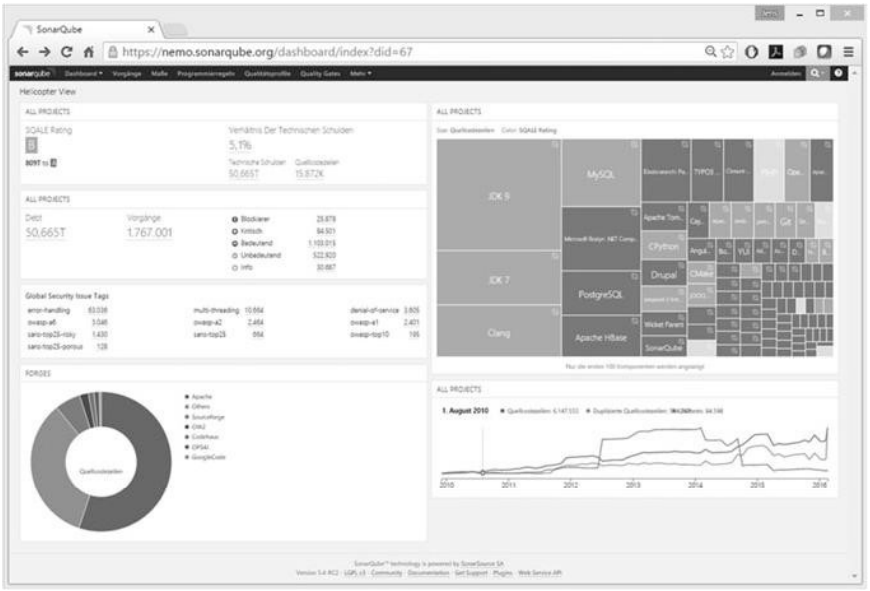

Fig. 9.2 CQC output (screenshot with SonarQube) (screenshot taken from Nemo, the online instance of SonarQube dedicated to open source projects, see https://nemo.sonarqube.org/)

- **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

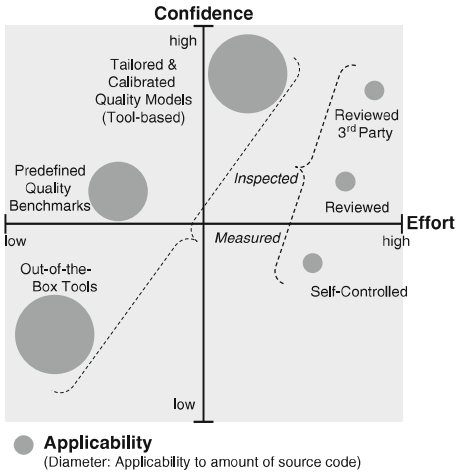

Fig. 9.3 CQC confidence levels. © Fraunhofer IESE (2015)

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.