Markus Harrer · Christine Koppelt
Gernot Starke · Benjamin Wolf

Software Reviews

Identifying
Risks
and Problems
in Software

INNOQ

Software Reviews

Identifying Risks and Problems in Software

Markus Harrer
Christine Koppelt
Gernot Starke
Benjamin Wolf
Martin Otten

Software Reviews – Identifying Risks and Problems in Software

Contents

Every Software has *Potential*

> Potential (from Latin *potentia* "strength, power"): a currently unrealized ability. Taken slightly differently: We have a problem that we should urgently solve.

Practically all software suffers from problems: Maybe it is too slow, crashes sometimes, doesn't look as beautiful as the competition's product, or changes take too long. Apart from `hello-world` applications, there is always room for improvement.

Symptom and Cause

Have you ever experienced a cursor that moves unspeakably slowly across the screen, and the whole application (apparently) ignores clicks or keystrokes?

This behavior represents a *symptom*—intuitively, we speak of a "slow system" or "poor performance." Instead of rushing to optimize the event handling or window management of the used GUI framework, we should explore the cause(s) of the poor performance. And this is where reviews come in.

Let us list a few possible causes for the above example of poor GUI behavior:

1. The application uses too much working memory, and the operating system has to *swap* it in the background.
2. The application is performing tedious background calculations on a sizable amount of data.
3. The runtime environment performs a so-called *garbage collection* to free up memory, leaving little CPU capacity for your application.
4. The application is blocked by input/output operations.
5. The database has no index for the search criteria you entered into the application and has to search billions of records sequentially.
6. The application needs to load information from an external system over a slow network. Usually, this system transfers a few kilobytes of data, but today it is several gigabytes.

7. The application communicates with an external system through a firewall. That firewall has been blocking some of those requests since the last update.

In this small example, you can already find a colorful mixture of possible problems regarding programming errors (1, 2), architecture/design (2, 3, 4), hardware (6) and infrastructure issues (1, 4, 7), configuration (1, 5, 6), databases (5), operations (6, 7) or external interfaces (7).

Business and IT

Change of scene: Another day, somewhere at some company. The business side complains that IT is taking far too long to deliver urgently needed *features* yet again.

We consider this situation a *classic* and have a phrase for it: bad time to market (TTM). We consider bad TTM to be a symptom triggered by one or more of the following causes:

- poor modularization, with the consequence of coupled dependencies
- poor code quality, caused by poor software craftsmanship
- lack of consistency or homogeneity, especially with cross-cutting concerns
- insufficient base technology
- poor test concept and lack of test automation
- cumbersome development processes, especially regarding requirements engineering
- too many meetings and coordination
- too many organizational units involved or teams that are too large

Bad TTM makes development more expensive, and it (systematically!) prevents *Business Agility* that is so often required, i.e., the quick adaptation an organization makes to changed market conditions or customer requirements. Therefore, you should always pay attention to TTM problems in your reviews.

Search for Unnecessary Complexity

"Complexity is anything related to the structure of a software system that makes it hard to understand and modify." [1]

We see unnecessary complexity as the cause of many subsequent problems—with the TTM mentioned above possibly being the worst of them. Many of the problems in or with software result from unnecessary complexity.

Complexity in software can manifest itself in unique ways: in source code, in architectural structures (modularization, dependencies), in cumbersome and overloaded data models, in misapplied technologies, in cumbersome development, and in operating processes—we are sure that you can continue this list.

Look for unnecessary complexity in reviews, because it almost always provides you with excellent opportunities for simplification and improvement. We will learn more about complexity in the chapters on Architecture Analysis and Code Analysis.

Before the Therapy: Diagnosis

If you want to increase your system's improvement potential, you should first develop an explicit, concrete, and specific understanding of the existing problems.

Just as in medicine, we should scrutinize patients before the responsible physicians do. Working with our therapy analogy, development teams should perform such a diagnosis ("review") before any significant improvement measures ("therapy"). Otherwise, there is the threat of making an *improvement for the worse*.

This book provides you with concrete suggestions for such reviews. We have used these approaches in reviews of systems in different industries, and they have worked well for us. We offer you a compact, practical collection of *good practices* for reviews. You can use them to search methodically for problems and risks in your systems.

How Reviews Work
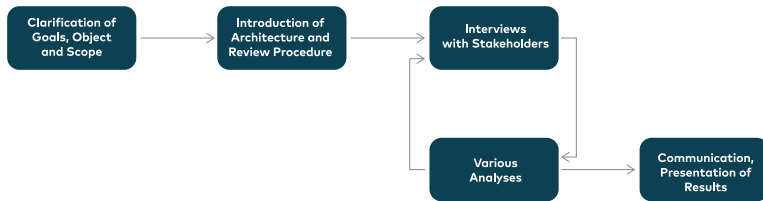
Figure 1: Schematic procedure of reviews (after P. Ghadir)

Regardless of focus and scope, reviews are always carried out in five phases:

1. Before the actual work begins, we must clarify the objectives and the exact subject of the review with the relevant stakeholders (e.g., clients, product or development managers). We should also define the scope of the review and the chosen approach. You will find more information on this in the section **A Good Start**.

2. As a first step in the substantive work, we conduct a compact kickoff workshop with a selected group of relevant people. To them, you present the goals and the procedure. In return, these stakeholders explain the system, its business significance, and summarize relevant processes. See section **Kickoff**.

3. This kickoff serves as the basis for in-depth interviews. For more information, see the section **Stakeholder Interviews**.

4. As required, these interviews are followed up by further analyses in which we look for specific categories of problems and risks. You can find methodological proposals for these analyses in **Analyses: The Heart of the Matter**. These analyses and the interviews of step 3 may alternate: The results of interviews may require individual analyses and vice versa!

5. Finally, we must process the results and conclusions for the people involved and present them to them. We explain the critical aspects of this in section **Communicating Results**.

Who Conducts Reviews?

We (Ben, Christine, Gernot, and Markus) work for INNOQ, an IT service provider with a focus on software architecture. We conduct reviews as *external* reviewers. In doing so, we look at systems in IT user organizations and bring in our neutral external perspective. We make anonymous and unbiased comparisons with similar systems or domains. Our goal is to provide our clients with new impulses or suggestions.

Internal Reviewers

External Reviewers

Pros

/ Specific knowledge
/ Networked within the company
/ Economical

/ Independent
/ Neutral
/ New impulses

Cons

/ Risk of "tunnel vision"
/ No benchmarking
/ Risk of "bias"
/ "Prophet in own country"

/ Longer familiarization
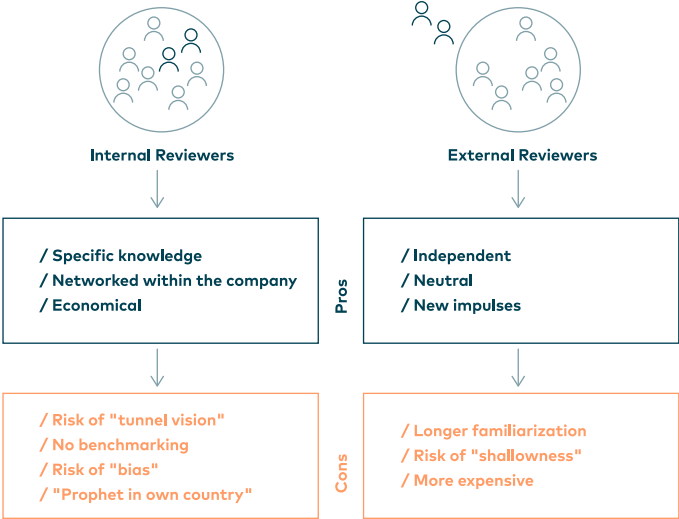/ Risk of "shallowness"
/ More expensive

Figure 2: Possible advantages and disadvantages of internal versus external reviewers

Regardless of your role in system development, you can also review your system by yourself or let it be reviewed by one of your organization's neighboring departments. Below you will find some (possible) arguments for and against internal and external reviewers.

Review? Audit? Analysis?

In the IT industry, we use various terms to identify problems or risks in existing IT systems. We have decided to use the term *review*. You can find its differentiation from other terms in the appendix.

Acknowledgement

We would like to thank our clients whose systems we have been able to examine over the past years and all those who committed to the open-source method aim42.org[1]. Also, a big thank-you to INNOQ: Here, we live freedom of expression, diversity, and open communication with great colleagues. Special thanks to Phillip Ghadir for his concrete suggestions for approaches and practices. We would also like to thank Lars Hupel and Martin Otten for their perseverance in the fine-tuning of the printing process, and Sonja Scheungrab and Robert Glaser for their suggestions for improvements in design.

Finally, thanks to our families, whose moral support has led us through the shallows of the *blank paper syndrome*[2].

---

[1]https://aim42.org
[2]Blank paper syndrome: Also called writer's block.

1 A Good Start

Before you start the actual review, you should first clarify its motivation and goals, and discuss the parts/aspects of the system to be considered with your clients.

You should clarify as concretely as possible your customers' expectations regarding the process and procedure, the communication of results, and the time frame.

Define Goals

The basis of successful reviews is a clear definition of objectives. You must ask the commissioning stakeholders for them or help them to clarify these objectives. Also, ask about the motives or drivers for the review. Knowing both of them helps to ensure your analytical methods, and your reporting and presentation of the results are aligned with the goals.

As an example, here are some goals from real reviews (anonymized):

- Get an independent opinion on a systems architecture.
- Clarify to what extent the system has a future-proof design and implementation.
- Record the actual technical state of a system for which hardly any—usually no—documentation is available.
- Challenge known problems of the system (for example, poor *TTM*, performance issues).
- Clarify the effects of new requirements on architecture, implementation, and operation of the system.
- Prove that individual decisions made in the past are no longer acceptable from today's perspective.

Record the jointly agreed goals in writing! Whether in interviews with stakeholders or to communicate the results of your review, you will need these formulations again.

Clarify the Thematic Scope

Besides the goals, you need to clarify the *subject* of the review. What exactly should you investigate? Various people could interpret the phrase "The System X" differently. So you and your clients will determine what exactly you should investigate:

1. the *architecture* of the system[1], by which we mean at least the following:

   - the structural (macroscopic) design of the system from *big* components, their dependencies, and interfaces
   - the cross-sectional concepts and technologies used for them

2. the *implementation* of the system (i.e., its source code)
3. the operation of the system in its target environment(s)
4. the development or the development process of the system, which could include the following subtasks:

   - requirement clarification and management
   - coordination and execution of implementation tasks
   - test and quality assurance
   - configuration and version management
   - release, deployment, and rollout
   - change, error correction (*change management*)

Sometimes the goals and scope of reviews contradict each other. An example: We were tasked with reviewing whether a system and its architecture were *multi-client capable*. Unfortunately, we had no access to the system's source code. We could not check whether, for example, architectural approaches to client separation in the database were correctly implemented in the code.

At the least, you should point out such contradictions. If these goals are not achievable with the scope, you should reject the review in extreme cases!

---

[1]"Software architecture refers to the fundamental (*macroscopic*) organization of a system as reflected in its components, their relationship to each other and to the environment, and the principles that govern its design and evolution. This definition comes from IEEE Standard 1471.

Clarify the Time Scope

An important question to clarify with your client is how much time you have available for the review. We would like to put the classic *it depends* answer in more concrete terms by giving you some examples and rules of thumb.

We have conducted reviews of between 2 and 60 person-days (PD for short), depending on the systems' objectives and sizes. In 2 to 4 PD, you can get an overview and give a kind of "State of the Union Report" without seeing much code. The more extensive reviews (40–60 PD) covered the architecture and code of significantly larger systems, including a look at their development and operational organization.

In systems with 100 person-years of development effort, 50 PD means just a mere 0.0021%. In comparison, passenger cars' inspection costs are only under 1% of the purchase price[2] per year.

- The larger or more complex the system, the more time you should allow for the review. A halfway thorough architecture review of a 2–3 million LoC system with a 20+ people development team requires ten and more person-days.
- The more thorough your review, the more time you need.
- For example, if you only perform a tool-supported code and architecture analysis, you can get by with just a few days.
- You can do a maximum of four to six interviews in one day (our personal record is 19 interviews in two days—please don't copy that, that was a terrible idea). For each interview, you need about the same time for rework.
- A slide for a review report costs one hour of time, discussion and revision included.

Calculate at least 20% of your time budget for preparing and communicating your results. Often after a final presentation, there are still a lot of requests for changes!

---

[2] Source: https://www.fairgarage.de/inspektionskosten-vw and AutoScout24.

Clarify the Procedure for Reviews

Our recommendation: Work iteratively for reviews.

Give your clients brief preliminary feedback on possible results after about 20–30% of the total time budget. On that basis, agree on the further procedure. In particular, agree on additional focal points of the investigation.

For us, this clarification of the procedure also includes considering the individuals with whom you should conduct interviews. Your client should invite at least those people to a kickoff meeting (see the following chapter).

After the kickoff, start with interviews before you start with other analyses.

Clarify the Involvement of the Client

During the review, you will need contact with distinct people or organizational units you may not know in advance. Have the client nominate a person to help you, for example, find rooms for interviews, arrange appointments with overworked stakeholders, or get access to required documents and source code. This person should also show you the way to the coffee machine and provide a projector for the final presentation.

Obtain Preliminary Information

Have existing (technical) documentation handed over to you in preparation for the following activities. Ideal for this would be architecture documentation, if available.[3] Any overview diagrams, information about external interfaces, user groups, and roles, the operation of the system—take everything that could roughly belong to the system's scope.

---

[3] If there is no architecture documentation, you have already identified the first risk and can make the first suggestion for improvement. Ben and Gernot have written a little book [2] about this. :-)

Inform Participants (or Have Them Be Informed)

It saves you a lot of time in the subsequent interviews if your interview partners all have the same level of information.

Also, this proactive communication of the review objectives prevents that unproductive rumor mill from grinding—which you should avoid during the review.

Clarify Confidentiality

As reviewers, clients provide us with confidential information that we may use for the review only.

You should have the review team sign a nondisclosure agreement with the client, in which the review team commits to absolute secrecy towards third parties.

2 Kickoff

Put yourself in the position of the people who use, support, develop, and operate the system: suddenly, a review team (internal or external) asks a lot of questions and challenges things.

Goals of the Kickoff

Use the kickoff to establish *confidence*. Together with the client, openly communicate the objectives of the review to all parties involved. Explain your approach, including any milestones or end dates. Let the clients openly communicate their expectations in the kickoff.[1]

The participants should get to know the review team during the kickoff. Therefore, all reviewers should be present and introduce themselves and their respective work focus individually.

Who Should Attend

Together with you as the reviewer, the client of the review should define who is participating in the kickoff. Hopefully, you have worked out a proposal together in the *good start* phase (see the previous chapter).

We like to invite representatives of the following roles/tasks:

- business representative, *product owner*, product manager
- development team, implementation, and architecture
- test & QA
- build, deployment, release and operation, and, of course
- Users

---

[1] If the clients should have a *hidden agenda*, for example, to use the results of the review to justify staff or severe cost reductions, they will not welcome this open communication. In such cases: Welcome to the lions' den or the web of corporate policy. Apart from a self-evident "warning sign," we will not give you any further advice on this.

Preparation

Before the kickoff, review the documents you have received as preparation. Hopefully, you will have heard some essential technical terms and system-specific terminology. You should know the names of the key stakeholders—including their roles in the system and review. If things remain unclear, you will have some good questions for interviews.

The Agenda: What You Should Discuss

We know that every kickoff deserves and needs an individual agenda, but a few topics are essential.

As a reviewer, tell a little about your professional background and give professional or technical reasons **to show that you** are the right people to achieve the review objectives **and why**.

Ask the others involved to answer some key questions, which you can write on the flip chart for all to see:

- What is your role?
- How long have you known the system or been working on it?
- What part of the organization do you belong to? Are you internal or external?
- A personal question *outside the box* can help you get to know the *person* behind the *role*. Examples: What is your favorite book? Favorite film? Your favorite hobby? The destination of your dreams?
- Those responsible for the functional side of the system should briefly describe the **commercial or professional purpose** of the system to everyone.
- Those responsible for the technical side (e.g., architecture, development management) should show the **architecture** of the system, or at least the essential subsystems or components and the fundamental technical decisions.
- Someone should summarize the **development process**: from requirements, design, implementation, testing, build, and deployment to the release.

We have had good experiences with carrying out a short qualitative analysis of the system immediately afterward with at least some people involved in the kickoff—based on an ATAM workshop (see the chapter Qualitative Analysis).

3 Analysis: The Heart of the Matter

Before we start with the actual analysis, we urgently warn you against falling into the *microscope trap*: If you search in a narrowly defined area, you will find problems only there at best. For example, if you only examine *dependencies* in a specific software module, you may not find possible memory leaks or problems with performance or security.

As many factors can cause problems and risks in IT systems, always perform reviews as a broad search. You should consider unique aspects of the system and its development.
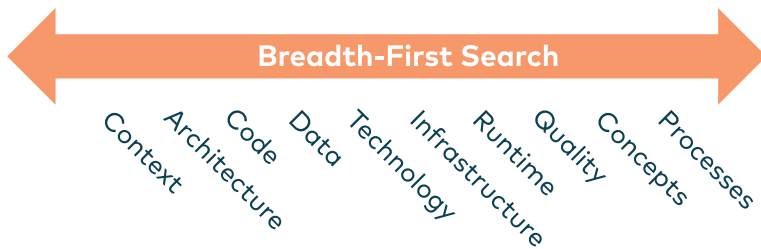
Figure 3.1: Possible topics of a breadth-first search

Which of these topics you look at more closely depends on several factors:

1. the objective and scope of the review, which you will hopefully have explicitly clarified with the relevant stakeholders in the kickoff
2. already known problems or risks that stakeholders point out

Begin your analysis phase of each review with a few interviews in which you ask critical stakeholders about the "state of affairs." We have dedicated the following chapter, Stakeholder Interviews, to these interviews.

Categories of Analysis

Because of this book's brevity, we have summarized some of the categories from the figure above. The table gives you an impression of the problems you can find using a particular category of analysis.

This analysis category	finds problems with ...
Context	external interfaces
Quality	performance, stability, changeability, time to market, security, etc.
Architecture	large-scale dependencies, technologies, concepts, distribution of knowledge in the team
Code	inconsistencies, hotspots, small-scale dependencies, poor code quality
Data	structures, distribution/replication, data structures and models
Infrastructure	release/deployment, technical infrastructure, system operation
Processes	requirements clarification and management, development, test and release processes

Structure of the Analyses

In this book, we look for problems and risks in about a dozen different *areas*. We describe the possible *analyses* of these using the following structure:

- **What is it about?** A brief description of the nature of this analysis
- **Examples of common problems** or risks that you can find with this analysis
- **Methodology**: possible procedures and methodological or technical tools
- **References** to background information, literature, and related topics

4 Stakeholder Interviews

Whom Should You Talk To?

Since we like to conduct reviews in a broad search mode, we recommend that you address a correspondingly extensive selection of people and roles involved in the interviews.

They include in particular:

- users
- persons responsible for the business side (e.g., representatives of the departments)
- persons responsible for the technical side (for example architects)
- clients and management
- representatives from the development team (for larger systems with several large subsystems or components, a mixture out of several teams)
- testing/QA, if not part of the development team
- persons from infrastructure and operations
- project and product managers
- in agile organizations: product owner, scrum master or agile coach

Prepare Interviews

Take the time to write down some questions in advance that you would like to ask your interviewees. Please take into account their role, responsibilities, experience, or involvement in the system. Share these questions with your interviewees **in advance**, for example, in the invitation e-mail. These questions will help them prepare for the interview and even prompt them to bring relevant papers or documents along.

What Questions to Ask?

Ask **open** questions that require sentences as answers rather than closed yes/no questions. Ask about **known problems or risks**, about overly complex, confusing, or other negative aspects of the system. Ask about what people **want to keep** unconditionally. Also ask about **proposals for remedies**, for example:

- "What would you change if you could control the entire development at will for one week?"
- "What would be your top 3 changes (to the system, development processes, or the organization)?"

Ask, "What else should I ask you?" to allow your interviewee to address other issues.

Conduct Interviews

It is best to interview in pairs—as a duo, you get more out of it. Alternate between asking questions and taking notes so that the person asking can concentrate on gestures, facial expressions, and behavior of the interviewee and is not continuously distracted by making notes. Note everything that requires further analysis or review. Ask for other contact persons who have more information on specific topics. Limit interviews to 60–90 minutes. You can always arrange follow-up appointments. Allow 15-minute breaks between two such interviews. You need them to reflect briefly, mark up your notes, and gather your thoughts for the following interview.

Take Notes in a Goal-Oriented Way

Use multiple colors for your notes. Mark statements on specific topics consistently in the same color. Architecture topics could be green, operations or deployment topics orange, requirements topics purple, and so on. Note the date and people involved on each note. Note on the first page the most interesting topics of the interview. You can use hashtags like #database, #requirements, or #scrum-fail.

Pre-mortem

If you notice that participants in interviews are reluctant to talk about the current problems, try a mental leap in time and ask about possible horror scenarios of the future:

"What has to happen to make the system or the development hit the wall in two years?"

With this question, you can collect risks in a brainstorming manner. Here you can also use statements from people with a negative attitude in a positive way.

Risk Storming

Your interview participants might not know the system and its risks inside out. In this case, have the group draw a system overview diagram on a flip chart or whiteboard first. Then ask about risks in the system components and their interfaces. Have all participants individually collect the risks on stickers, place them on the system overview diagram, and present them to the group.

5 Context Analysis

In context analysis, we look for risks and problems in the environment—in context—of the system, especially in external interfaces and neighboring systems.

Examples of Common Problems

- External neighboring systems supply data incorrectly, incompletely, too late, or at unexpected times.
- External neighboring systems often change their interfaces.
- The specification of an external interface is incomplete, incorrect, or missing entirely.
- During operation, a neighboring system reacts too slowly.
- The system receives too many requests via an external interface—a preliminary stage to a *Denial of Service* (DoS) attack.
- The system or one of its neighbors cannot meet *Service Level Agreements* at runtime.
- The operating costs of an external interface are significantly higher than expected.

Methodology

Make sure you know all external interfaces. A table of all external interfaces regarding their technical content and specification is beneficial. We find a graphical representation in the form of a context diagram useful for an overview.

1. Ask those responsible for these interfaces about problems and risks. Also, involve the people responsible for the neighboring systems. They may even be located outside the organization under inspection.
2. Question the quality requirements and assurances (*Service Level Agreements*) of these interfaces (*target* state).
3. Determine the compliance with these quality requirements, e.g., by interviewing responsible persons or by conducting a runtime analysis.

4. More specific: Observe these interfaces during operation, for example, by analyzing log files or monitoring (see Runtime Analysis).
5. Check the implementation of external interfaces by conducting specific code reviews. Prerequisite: There must be an appropriate specification of these interfaces.

Pay particular attention to the following categories of external interfaces:

- interfaces that cause direct usage costs, for example, if interface partners charge a fee per call or per transferred data record
- interfaces whose definition or implementation are subject to exceptionally high volatility (e.g., they change frequently)
- interfaces that require manual intervention to ensure ongoing operation

References

- You can identify **operational problems** at external interfaces by performing a runtime analysis.
- Investigation of performance or stability problems at external interfaces is part of qualitative analysis.

6 Qualitative Analysis

Quality (from lat. *qualitas*, condition, property): The sum of all properties of a system. Quality indicates the extent to which a system meets existing—implicit and explicit—requirements.

Software Quality

This abstract definition of the term *quality* helps in neither practical software development nor reviews. We like the pragmatic explanation better:

The quality of a system indicates to what extent it meets existing requirements.

Fortunately, [3] defines a workable quality model for software that allows you to analyze the system's particular quality properties. This approach decomposes the abstract term into eight subgroups, which we show in the following figure.

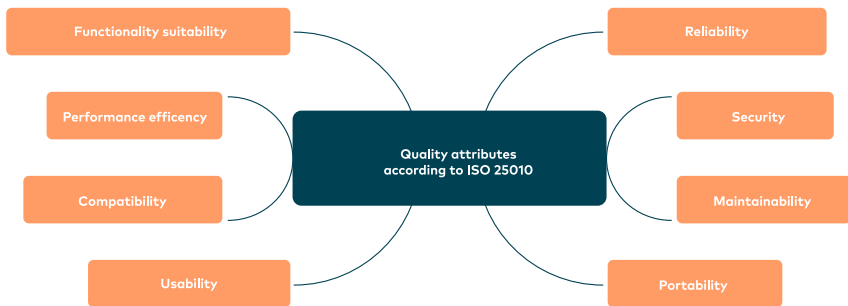| Functionality suitability		Reliability
Performance efficency		Security
Compatibility	Quality attributes according to ISO 25010	Maintainability
Usability		Portability

Figure 6.1: Quality model ISO 25010

You can now systematically sort out these subgroups. Analyze in each case whether your system meets the respective requirements. We know this quality analysis as *Architecture Tradeoff Analysis Method* (short ATAM). It is relatively common in methodical software engineering.

In this book, we would like to pick out some examples from the pool of ISO quality characteristics that we have often noticed in reviews of concrete systems:

- performance
- expandability or changeability
- security

We discuss these three in the following sections. At the end of this chapter, we will give you a few notes on ATAM.

Runtime Analysis

In runtime analyses, you examine the system while it is running, trying to catch problems ("culprits") in flagrante delicto. Among other things, this helps you find performance bottlenecks, hidden dependencies, memory leaks, deadlocks, and defects. Runtime analysis examines the system's resource usage, including processors, memory, and input/output resources.

Examples of Common Problems

- A particular operation in the system takes an unreasonable amount of CPU time or memory to perform ("performance or memory bottleneck").
- A component of the system needs services or data from another component at runtime, but it should not be used according to static code analysis ("unknown dependencies").
- The system crashes at specific inputs ("instability").
- The system executes operations multiple times.
- The system generates or consumes more data than expected.
- The system performs unnecessary operations to fulfill specific tasks.
- Operations block each other ("deadlocks").
- Operations run sequentially, although they should run in parallel.
- Certain features are used too often or not at all by users.

Methodology

For all runtime analyses, we'll provide you with some basic advice. Before each dynamic analysis, clarify your concrete expectations. Reflect on what should have happened according to your expectations. For example, what are the relative time or memory requirements of certain system parts like? It is best to clarify these expectations with the them that developed the affected components.

- The most straightforward tool for testing, and one that every development team is familiar with, is called **debugger**. A debugger allows us to track the operations of a piece of software at runtime, almost at the atomic level. With a debugger, you can check your assumptions about specific processes in a finely detailed way. Yes, theoretically, you could do that by reading the source code, but some dependencies are only built at runtime. Often it is more convenient to use the debugger: And unlike assumptions made when reading code, the debugger tells the truth. :-)
- Another popular form of runtime analysis is **profiling**. The profiler measures how much execution time parts of the system need. Depending on the tool and settings, it measures this down to the level of individual lines of code. Profiling usually takes place in development or test environments. Caution: Such detailed measurements require time—sometimes significantly.
- We call the counterpart to profiling in actual operation **monitoring**.
- Examine the **log files** and optimize the system's logging to provide you with answers to specific questions. The latter is an invasive operation, so you must change the system for this. For a log file analysis, you should clarify in advance (see above) which entries or messages you expect.
- You can observe some details of system usage with **usage analytics**. You need the consent of every person affected by this.

References

- Some problems with external interfaces (see Context Analysis) can only be found by using runtime analysis.
- Usage analytics may disclose sensitive or personal information. You must ensure confidentiality and compliance with GDPR[1] rules.

---

[1] https://ec.europa.eu/info/law/law-topic/data-protection/eu-data-protection-rules_en

Analysis of Changeability and Expandability

Changeability and expandability count among the *inner qualities*[2]. Analyzing them requires considering the inner structure, like components, code, technical concepts, and runtime configuration.

Examples of Common Problems

- Several times, the focus of our reviews was on the poor *time to market*[3] of systems (e.g., delays in implementing additional features).
- This is also true for minor changes to the system causing excessive effort (e.g., high personnel costs).

Methodology

Changeability and expandability depend on various factors:

- complexity of the system, which is determined by the following aspects, among others:

  - degree of coupling in the system at various levels of abstraction: High coupling often means that we must modify many individual components for changes, increasing both effort and change risk
  - complexity of the implementation, such as comprehensibility and consistency of the source code
  - cohesion within parts of the implementation (subsystems, building blocks)
  - consistency (e.g., are recurring tasks in the system solved in the same way (*consistent*)?)
  - size of the system or the units relevant for a change

---

[2] In contrast to the *external qualities* such as performance, usability, robustness, etc.

[3] From Wikipedia: In commerce, time to market (TTM) is the length of time it takes from a product being conceived until its being available for sale. TTM is important in industries where products are outmoded quickly (see https://en.wikipedia.org/w/index.php?title=Time_to_market&oldid=926119380).

- testability and the existence of automated tests
- knowledge about architecture and implementation within the development team, which is determined by *common knowledge* or documentation, among other things
- process complexity, for example, lightweight versus heavyweight development processes, more or less necessary formalisms

Check coupling: Due to the different *types of coupling*, you will have to dig a little deeper into the toolbox to measure it:

- Static code analysis can find simpler types of coupling. In the section on static analysis, we will show some options and tool categories.
- Coupling via shared databases can be found in the chapter Application Data Analysis.
- Other types of coupling can be found in an architecture analysis. Examples are temporal coupling, coupling via hardware, and implicit coupling via common interfaces.

Check the complexity of code: Measured complexity of code is useful for predicting the probability of errors: Complex code is likely to contain more errors than simpler code. With each change, development teams will inadvertently add more errors to complex code than to simple code. These correlations have been proven by studies, see for example [4].

You can evaluate the complexity of source code using different metrics:

- Cyclomatic complexity (i.e., the number of linearly independent paths through a program's source code): Intuitively, this measure of complexity increases by two points with each branching statement.
- Size in lines of code: It can refer to individual functions or larger units.
- Other programming constructs: the number of function parameters or method signatures, type complexity of these parameters, recursion, parallel processing with synchronization, mixing of different programming paradigms, and others

Check use of appropriate technology: The choice of underlying technologies and frameworks can significantly influence the changeability of a system. Example: In early architecture and programming paradigms of Enterprise Java, develop-

ment teams had to configure their systems manually in XML-based deployment descriptors besides fulfilling complicated programming requirements. This was both time-consuming and error-prone. Simplified models such as Spring Boot reduce the *cognitive load* for development teams and, therefore, most times lead to systems that are easier to change.

Therefore, you should check the technologies and frameworks used to determine whether necessary or frequently recurring changes are made easier or more difficult as a result.

References

- Complexity of source code can be found by code analysis, for example, by code metrics.
- Organizational measures in DevOps (for example, *continuous integration*, build and test automation, *trunk-based development*) lead to significantly improved changeability of systems in the medium term.

Security Analysis

Here you identify problems or risks in the field of data security, data protection, confidentiality, and operational stability/reliability.

Examples of Common Problems

An unauthorized person

- reads or manipulates personal or otherwise confidential data.
- replaces the regular application (e.g., website) with a fake one.
- prevents regular access to the system, e.g., by turning off the system.
- steals or swaps passwords so that they can access someone else's data.

The (security-critical) system

- transmits confidential data via easily accessible but *unencrypted* channels.
- uses a custom implementation of security features instead of using established (open-source) libraries.
- leaves backups unencrypted and stored in easily accessible rooms.

Methodology

As a reviewer in IT security, always assume that potential attackers have more experience, knowledge, time, money, and motivation, and, above all, better ideas than you or the team that developed the system. In the area of *security*, you are fighting with a (potentially) much stronger enemy, who also likes foul play.

We strongly advise you to consult experts for security-critical systems or requirements!

- Is there explicit documentation of possible attack vectors against the system?
- Do the system and the associated organization implement the requirements of ISO-27001[4]?

[4] https://www.iso.org/isoiec-27001-information-security.html

- Are there organizational security measures in addition to technical ones (e.g., access control, intrusion detection, and building security)?
- Are all (!) procedures for authentication, encryption, hashing, and the like, based on established and tested standard implementations? In other words, implementing cryptographic algorithms (encryption and hashing) yourself is a high risk in any case!
- If the system is publicly accessible (e.g., via a web or mobile app): Do penetration tests take place frequently?
- Do log files contain confidential data?

References

[5] contains an extensive collection of *good practices* in the field of IT security.

Qualitative Architecture Analysis and Assessment (ATAM)
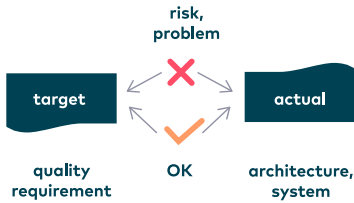
Qualitative architecture analysis compares the target state with the actual state and identifies potential problems or risks. As an essential basis for this, you need a detailed catalog of quality requirements, that is as concrete as possible. As a model, you could use thorough product evaluations, such as those performed by the well-known Consumer Reports in the US, that use detailed criteria catalogs (= requirements) as a basis for their investigations. For example, they objectify the extent to which the system in question must meet the requirements for maintainability, flexibility, performance, and security. A qualitative analysis does not provide an absolute measure. It should not be an end in itself, but should achieve objectives concerning specific relevant criteria ("quality requirements"). Qualitative analysis helps to identify risks that may arise from questionable design decisions.

Concretize Quality Requirements

Using so-called scenarios, you can define specific metrics for your systems or, even better, have them defined by the relevant stakeholders. You use scenarios to describe which specific criteria the system and its architecture

must meet. It is vital to keep the formulation of the scenarios as concrete and operational as possible, preferably by specifying tangible metrics. You can see this schematic structure in the figure. Formulate scenarios as sentences that always contain the "cause" and the "measurable reaction of the system."
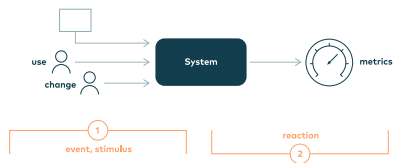
Does that sound too abstract? A few examples will help to make this clear:

Trigger (event)	Metric (reaction)
If the user requests the monthly report of the cost center XYZ …	the system will generate the report within 3 seconds.
If a user conducts a product search with default criteria (article description) …	it will take a maximum of 500 ms until the first five hits are displayed.
If the external system XYZ does not respond for more than 3 seconds …	the system will report this error to the administrator on duty within 30 seconds.
If the department requests a change in the XYZ tariff in the system …	this change can be implemented in the system within 4 hours.

With these scenarios for the quality requirements, you now have a concrete benchmark for the qualitative analysis.

Check Architecture Against Quality Requirements

With architects or the development team, you now examine whether the system or its architecture can meet the defined scenarios (= quality requirements). Have the corresponding architecture decisions and approaches explained to you. Play through these scenarios in the form of walkthroughs, in as detailed and fine-grained a way as possible, together with members of the development team. Find out how the building blocks of the system interact to achieve this scenario and which design decisions support the respective scenario.

The following questions may also help you in this process:

- What architectural decisions were made to achieve this scenario?
- Which architectural approach supports the achievement of the scenario?
- What compromises were made by this decision?
- What other quality features or architectural goals are affected by this decision?
- What analysis, research, or prototypes support it?
- What are the risks associated with this decision or approach?

- What are the risks to achieving the scenario and the associated quality requirements?

In our experience, this allows you to find out quite reliably whether the affected scenario *works* with the given architecture, or whether there are severe risks in this respect.

7 Architecture Analysis

Before we go into the details of this aspect of reviews, let's briefly summarize what we understand by the term "software architecture." When we speak of architecture, we mean several levels:

- **Domain Architecture**: The domain-oriented modularization of the application landscape and the information flows between applications. This is independent of specific technologies.
- **Macroarchitecture**: Specifications on topics that are largely independent of the internal structure of individual systems, but ensure that an application landscape is created that follows meaningful rules. Examples are communication protocols, UI integration, mechanisms for data replication, monitoring/logging, operating interfaces, etc.
- **Microarchitecture**: (Team-)local decisions about the internals of a system like internal modularization, layer models and architecture patterns, concrete solution concepts, frameworks and libraries, runtime environments, and other technical aspects.

In the following, we have compiled some points worth considering for the different levels of architecture.

Domain Architecture

Examples of Common Problems

Experience shows that an architecture that is not oriented towards the domain and business structures leads to complex data models and bottlenecks (time delay, agreement on content, coordination effort, etc.). This makes systems unnecessarily complex, error-prone, difficult to maintain, difficult to expand, and thus unable to satisfy either developers or business departments.

Methodology

In order to understand whether the cause of our problems lies in the domain architecture, we need to understand the domain-specific problem areas and examine how they are solved by respective applications. Good collaboration between the development them and the domain experts is crucial here, so this part of the architecture should be developed together with the domain experts to ensure that all parties involved have a common understanding of the problem and the solution. We therefore only consider the business level at this point: technical details do not play a role here. In the end, we should be able to decide which changes are necessary and where investments in a better domain architecture are worthwhile.

A good way to analyze these problems is to use methods from **domain-driven design**. For example, event storming, a workshop format to analyze, structure, and visualize domains. This be used not only for modeling new systems, but also as a reverse engineering tool to get an overview of the existing processes in the domain.

Another possibility is to use **Wardley maps**. They enable the existing IT landscape to be mapped with a constant eye on customer benefits. This provides an overview of possible misinvestments, suboptimal development practices, or know-how bottlenecks on neutral ground. From this, well-founded insights can be derived for make-or-buy decisions, outsourcing, or the reorganization of teams.

References

- Michael Plöd: *Hands-on Domain-driven Design – By Example* [6]
- Eric Evans: *Domain-Driven Design Reference* [7]
- Simon Wardley: *Wardley Maps* [8]

Macroarchitecture

Examples of Common Problems

Problems can arise on two levels:

- The specification level: Specifications may be too detailed or incomplete, or they may simply not match the required quality characteristics.

- The implementation level: Here there can be deficiencies in the correct implementation or also in the path of implementation, for example, if functionality can be found *cross-cutting* a software system again and again (so-called *cross-cutting concerns*). Typical representatives of this genre are persistence, authentication, replication, monitoring, logging, multi-client capability, auditing, and internationalization, and so on. Here you are faced with the task of evaluating whether the technologies used for this purpose have been implemented correctly in the software system.

Methodology

Qualitative assessment procedures, such as the previously presented Architecture Tradeoff Analysis Method (ATAM), can already be used to evaluate whether the appropriate cross-sectional concepts are available for the required quality characteristics.

Perform a *code walkthrough* to see the implementation directly in the code. Select a particularly interesting and/or relevant use case. Then go through the processing in the code step by step (statically or roughly by debugging). You will stumble across cross-cutting concerns at various points, whose implementation you can then examine more closely. As input for the evaluation, you can use the specifications and programming guidelines in your company to check the implemented cross-cutting concepts step by step. You can also use mini-checklists or the currently available best practices of the software community to check.

References

- Till Schulte-Coerne: *Options for Frontend Integration.*[1]
- Example for log specifications: Elastic Common Scheme.[2]
- Concrete suggestions for improvement: aim42, section "Improve."[3]
- Checklists for microservices in Susan J. Fowler: *Production-Ready Microservices* [9]

Microarchitecture

Examples of Common Problems

Among the common problems at the microarchitecture level are poor modularization and opaque dependencies. But active management of this kind of complexity is very important in larger systems (100,000+ lines of code). It enables developers to find the right places for code changes quickly[4] and to avoid unpleasant surprises when changes are made.[5]

Methodology

An approach for the review of these two properties is described by Carola Lilienthal [10] in her book *Sustainable Software Architecture*. Here the structure of the software is checked for its conceptual integrity with the help of architecture management tools according to the following (here simplified) procedure:

1. Find grouping properties (e.g., from the domain, technical layering, or a pattern language) to which the software elements (e.g., classes) contained in the source code can be assigned.
2. Arrange the groups hierarchically according to their intended functions in the software system.

---

[1] https://www.innoq.com/de/articles/2019/08/frontend-integration
[2] https://www.elastic.co/guide/en/ecs/current/ecs-reference.html
[3] https://aim42.github.io/
[4] Here, hierarchically cleanly structured modules help.
[5] Here, clear dependencies between modules help

3. Evaluate the dependency relationships between the resulting grouping structures.

Ideally, there will already be suitable structures in the software architecture that are very similar to the grouping from the analysis. If as much source code as possible can be assigned to the groupings here, the system can be assigned a high level of conceptual integrity.

Purely Code-Centric Reviews Are Not Reviews!

Unfortunately, it is not enough to analyze the software architecture purely on the basis of the existing source code. It is a misconception that all information is in the code—and in our experience, it is even true that this is often not even the most important information.

This sounds anything but intuitive at first. How could something be more important for the architecture of a software system than the actual source code of the software? We have listed some reasons for this in the following:

- You may have an excellently structured implementation of an awful idea, executed by every trick in the book (e.g., because there are better, ready-made components for the same purpose)
- Perhaps a problematic part of the software, or even a particularly elegant part of it, would not be needed at all if one were to scrutinize a technical requirement
- Sometimes a large part of the important information is not in "classic" source code, but in configuration files or glue code that is not obvious at first glance
- Maybe the runtime architecture of the system is extremely elegant but leads to enormous challenges in the development process or the other way around.
- Under certain circumstances, the division into different subcomponents in a distributed system plays a much greater role than the implementation of the individual systems

- Problems may be caused by a suboptimal division between hardware and software components (especially in embedded systems) or by the use of standard software that needs improvement

The cases mentioned are all very different, and this is perhaps the most exciting aspect of reviews: You have to think your way into the system and question it from very many different perspectives. Therefore, there is no standard recipe and no ultimate tool that makes all problems visible.

8 Code Analysis

Source code is one (or even *the*) central artifact of your system. So, code analysis will almost certainly reveal some problems or risks to your system.

In this chapter, we introduce you to methodological tools that allow you to perform such analyses. Even if you already perform code reviews or static analysis. Read on anyway. We have a few surprises in store for you.

A side note: If your development team uses unit testing, you are already actively engaged in code analysis. These unit tests *analyze* the behavior of your system on a detailed level.

Examples of Common Problems

Through code analysis, you can find the following categories of problems:

- parts that are difficult to understand due to their excessive size, complexity, or dependencies, and deviation from agreed style guides or code conventions
- risky parts of the code, for example, due to excessive change rate, error rate, or deviation from architectural specifications
- low-performance or an excessively resource-hungry implementation
- parts for which no automated tests exist

Methodology

Of the many possible approaches to code analysis, we present three of our favorites:

1. Static code analysis based on tools like SonarQube, TeamScale, ReSharper, or Checkstyle
2. Hotspot analysis, which can identify the most frequently changed places in the code based on your source control history (in our opinion, this is a beneficial method that is frighteningly little used in practice)

3. *"Manual"* code reviews, which can be conducted with the support of suitable tools (with a second or third person reviewing the code for best practices and possible errors or problems)

Static Analysis

What Metrics?

From the dozens of *theoretically* possible metrics for source code, we would like to recommend a small but practical selection:

- **Complexity**: We regard complexity (more precisely: unnecessary complexity) as a central problem in computer science. Please look at the section "Hotspot Analysis," where we combine complexity with change rate to identify especially risky or vulnerable parts of the code.
- **Coupling** (dependencies): Too many dependencies have a severe negative effect on comprehensibility and changeability. Tight coupling increases the risk of errors.
- Compliance with **coding guidelines or style guides**: This may sound like formalism, but uniform code significantly increases comprehensibility, and it is much easier to review.

In source code, complexity shows itself in various forms:

- Cyclomatic complexity, or the number of possible paths through a given piece of code (this metric depends on the chosen programming language)
- Cognitive complexity (for example, caused by indentation, levels of nesting, number of function or method parameters)
- Size of elements (for example, length of methods or functions, number of methods per class, number of classes per package)

Caution: Metrics

Using static code analysis and appropriate tools, you can collect many types of metrics that quantify your code's characteristics. Regardless of your chosen metrics, consider the following essential aspects:

- **Relevance**: Decide on a maximum of 3–5 key metrics relevant to your system. If you determine too many metrics, sooner or later, it will be impossible to see *the forest for the trees*.
- **Trend**: Observe the trend (e.g., the change in key metrics over time). Otherwise, you will have to work with a less meaningful snapshot.
- **Relativity**: Your management uses other metrics to operate the company and often has its own view on metrics themselves. For software reviews, we use metrics only as indicators, but practically never as absolute limits. Therefore, only pass on to your management the conclusions you draw from the metrics, but not the measurements themselves.
- Evaluate code, not people: Static code analysis evaluates your code objectively, not the people who wrote it. Do not judge people by these numbers.

Find Problems Using Metrics

- No one wants a slower time to market for new features. The cause for this is often close coupling between building blocks, which you uncover with static analyses.
- If metrics show high complexity at various points, they directly indicate an increased risk of error, both in the implementation itself and in future changes.
- If the development team does not use metrics to actively manage development, then you have found a significant risk for which methodically there are relatively simple remedies, such as our Zero-Warning Policy (see below).

Zero-Warning Policy and the Boy Scout Rule

Zero-Warning Policy: We recommend a zero-tolerance convention for (rewritten) code: New code must be **free** of warnings from static analysis tools and must violate **none** of the coding guidelines established for the system. If code violates the rules, let your system's build fail!

In this way, you keep the level of craftsmanship of your code high. Our experience from many development projects shows that this convention gradually leads to faster development (!) and lower error rates.

For the time being, introduce this policy only for new code. Exclude existing code explicitly from the relevant static analysis. If you need to make a change to the existing code, first include only this (hopefully small) piece in your analysis and work here according to the boy scout rule: Always leave the (old) source code cleaner than you found it.

Hotspot Analysis

Hotspot: Building block in a system that is changed frequently and at the same time has high internal complexity (after: [4]).

The inner complexity of building blocks is an excellent tool for predicting the probability of errors. Many errors occur where code is particularly complicated or deeply nested. Changes to such code also take a long time.

Think this a little further: If a development team has to work on *complex* code, the probability of errors increases significantly compared to working on *simple* code. You are undoubtedly familiar with the KISS principle[1], the golden rule of software development. Simple solutions practically always defeat overly complicated ones.
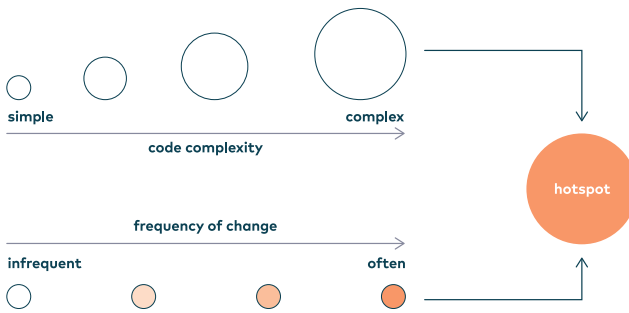

Figure 8.1: Model of hotspot analysis

Sort the building blocks (i.e., the source code) of your system according to the illustration: Find places that change frequently and are very complicated. Fortunately, we can take this methodology and appropriate tools for this purpose from Adam Tornhill's great books *Software Design X-Rays* [11] and *Your Code as a Crime Scene* [4].

---

[1]*Keep It Simple, Stupid*, keep it as simple as possible.

The following diagram (by Adam Tornhill from [4]) shows this hotspot analysis using the open-source system React[2]. In the original interactive diagrams, you can zoom into the hotspots and get a good insight into which building blocks of the system are both complex and volatile.
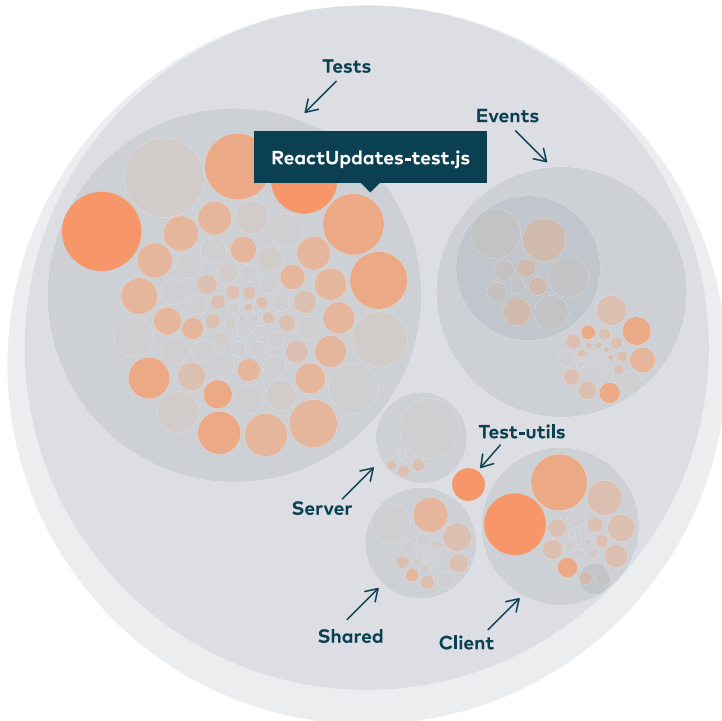

Figure 8.2: Hotspots in ReactJS

---

[2]https://reactjs.org/

Manual Code Reviews

With automatic analyses, you can discover potentials for improvement in your software. However, while purely automated processes cannot replace the "manual code review," they can support it. In the context of a review, *manual* means that at least one other person reads the source code (four seyes principle).

In the context of an overall review, you should also look *manually* at source code for some essential reasons:

- **Protection against malicious code**: Find malicious code that deliberately crashes the system, spies on users, or contains other security holes.
- **Conformity to architecture and style**: Check whether code conforms to applicable architectural conventions, principles, and concepts, and whether the code is written in the style you want. This is **not** about checking for compliance with coding guidelines.
- This check is (hopefully) mostly done using static code analysis.
- **Comprehensibility**: Cleanly named variables, self-explanatory method names, well-structured test cases, and the correct application of basic principles.
- All these are things you should check for in the code review to achieve readability and good comprehensibility.

Do Code Reviews Correctly—But How?

By now you should already be convinced that you should do code reviews. And now, you ask yourself the question: "How exactly is this supposed to work?"

Tools

Of course, some tools make code reviews easier, like GitLab or Upsource. So-called merge requests (or pull requests) make it possible to comment on code passages. This way, the comments of a review remain documented and can be viewed later. If all participants agree to these changes, they get merged into the main branch.

Our recommendation is to create your own review branch and a corresponding merge request in case of a one-time or initial code review. All things you find while

browsing the code base you can mark in comments in the review tool attached to the individual files. This allows the development team to correct the worst things right away directly in this branch. The advantage of this is that you can see the changes immediately and review, accept, or reject them.

Checklists

For a controlled review process, it is best to use a checklist, which you fill out with the most critical points in advance. In this way, you will not forget any of the points, even if you dive deeper into the code or have intense discussions with the development team. We have summarized some critical questions for you, which you are welcome to use as a basis for your next review.

- Does the implementation adhere to the architecture specifications?
- Are there possible bottlenecks concerning performance?
- Are there logical errors in the code?
- Is there logging and error handling?
- Do method and variable names express their function?
- Do files have an immense number of warnings from the static analysis? It usually pays off to subject these classes to a more in-depth examination.
- Is there a regulated review process? If so, do checklists exist for reviewers and reviewees?

References

- SonarQube: https://www.sonarqube.org
- TeamScale: https://www.cqse.eu/en/products/teamscale/landing/
- ReSharper: https://www.jetbrains.com/resharper/
- Checkstyle: https://checkstyle.sourceforge.io
- Cognitive complexity: https://www.sonarsource.com/docs/CognitiveComplexity.pdf
- Upsource: https://www.jetbrains.com/upsource
- GitLab: https://www.gitlab.com
- Michaela Greiler: Code Reviews [12]
- Awesome Code Review: https://github.com/joho/awesome-code-review

Digression: Data Analysis in Software Development

Did you know that you live in exciting times because you can often identify the pain points in your management software with data analysis? No more queasy gut feelings, but numbers, data, and facts about the problems that get in your way! If we look at hotspot analysis more abstractly, we use data analysis techniques to analyze any data from software systems:

- static data like source code, specifications, or documentation
- runtime data such as performance data, usage data, or test result reports
- chronological data like log files or data from version control systems
- community data such as open-source projects, including their related ticket systems, or discussion forums

Each data source by itself is valuable for carrying out situation-specific analyses in a bounded problem space, for example, to identify unwanted source code dependencies between teams, localize the effects of under-performing third-party libraries or identify weakening open-source communities at an early stage.

It becomes even more exciting when you combine different data sources to gain insights from various perspectives, such as "Which area in the code base has many methods that were often changed but aren't executed in production?".

With this approach, you can diagnose the cause of real messes while avoiding interpersonal conflicts. Prepare your results with suitable visualizations for the management in an understandable way—traceable from the raw data to the findings in case there should be any doubts about your analysis. These days, this type of analysis is no longer a challenge: thanks to data science, big data, or deep learning, more and more software developers are learning the necessary tools to handle the analysis of data. They can apply the same approach and the same tools, but on data generated during the creation or operation of the software. If you want to get into this topic, grab the book by Tornhill [11] or take a look at Markus's blog[3].

---

[3] https://www.feststelltaste.de/category/software-analytics/

9 Application Data Analysis

What Is It About?

The reliable storage and provision of data is one of the core tasks for most applications. The **data of an application often lives longer than the source code**. Usually, you can extend, refactor, or adapt source code with relative ease. The same goes for adopting a new framework version or deploying bug fixes. Incorrect or lost data, on the other hand, is difficult and costly to repair. Changes to the structure of a production database are often time-consuming and require careful planning. **Data modeling errors will take longer to solve**. Changing a database system is expensive and time-consuming (e.g., switching from MySQL to PostgreSQL is expensive because all database-specific functions have to be found and replaced). Moving from MongoDB to MySQL is even more expensive because it requires a complete change of the data model and schema management. So when choosing a data model and a database system, you should be careful to avoid unpleasant surprises. Within the scope of your review, it is now your task to find out whether the selection of the data storage system, the modeling of the data and data transmission, transformation, and quality meet the requirements of the application.

Examples of Common Problems

- The technical use case requires stronger consistency guarantees than the selected NoSQL database can offer.
- Poor performance of some data accesses or data-intensive operations.
- Outdated data models that no longer meet current business needs.
- Incorrect data content due to lack of validation or inadequate modeling.
- Excessively large or complex data models make further development of the system difficult.
- Outdated technology for data storage.
- Data changes take several days to arrive in all downstream systems.

Methodology

Get an overview of all **data storage systems** and their corresponding software:

- What types of data storage systems are used (e.g., local or cloud databases, file systems, object storage)?
- On what basis are the decisions for these systems based?
- Are the versions up to date?
- How much storage space do these systems require?

Determine their **architectural responsibility** (e.g., application database, cache, data warehouse, integration between different parts of the application, search):

- Is there an unusual number of caching systems?
- Is there data that has to be kept synchronized across different systems?
- Are there data stores that use several different systems to exchange data (*integration databases*)? This is an anti-pattern.

Look for problems with **response time**, **throughput** or **stability**:

Are there

- recurring queries that run for an excessive amount of time?
- functions within the entire system that cause particularly high loads in the data stores?
- reproducible crashes or malfunctions due to operations in the data stores?

Get an overview of the **domain structure of the data and how it is mapped to databases**:

- Which basic model is used: Relational? Document oriented? Time series?
- Do the subject matter and model fit together?
- Do the data model and the selected data store match?
- Does the database provide the necessary consistency required by the business?
- Are transactions used where necessary? If not, which alternatives have been implemented?
- Is there a fixed schema for the data? Is this implemented by the database or the application? How are schema changes made?

- Are there excessively large units (e.g., tables with more than a hundred columns)?
- Does the data model seem overly complex?
- Are there a large number of queries that traverse large parts of the data model?
- Are there performance bottlenecks caused by the data model?
- Is there realistic test data?

Identify **interfaces** and **source and destination of incoming and outgoing data flows**:

- How accessible is data to downstream systems?
- Are there documented interfaces?
- Are there access restrictions due to performance bottlenecks?
- Is data stored in an efficiently queryable or searchable data storage?
- How is this data transferred?
- Do the data sources provide sufficient quality, or does received data have to be (manually) post-processed?
- How long does the transmission take? Nightly or weekly batch jobs?

References

- You can find problems related to migrations or backup/restore with process analysis or infrastructure analysis.
- Analyses of data structures and technologies are also part of an architecture analysis.
- [13] contains hints for dealing with evolutionary development and refactorings about relational databases.

10 Process Analysis

Many problems with or within systems result from deficits in the processes involved. This might be communication problems between participants, inappropriate regulations, or even policy and organizational differences between people, departments, or companies. Static code analyses cannot find these kinds of problems.

This complex of topics certainly justifies a book of its own, so we will keep it brief.

On closer examination, some technical problems in systems turn out to be *symptoms* of organizational deficits (i.e., process problems). In dozens of reviews and audits, we found organizational deficits to be very common, and have been able to trace structural or conceptual problems in the code back to these issues.

Therefore you should also address process issues in reviews! We can never solve such issues by better architecture or programming. Instead, we have to change the process or even organizational structure. As an IT person, you usually lack the mandate, but management will only see the necessity of organizational changes if you explicitly address the problems!

Which Processes Can Cause Problems?

In more than 50 years of practical experience we have found (or rather *endured*) process problems in the following areas:

- Requirement processes: Even an excellent development team cannot produce good software from poor requirements. Garbage in, garbage out.
- Development processes: Agility incorrectly implemented, *ScrumBut*, team conflicts, excessive formalisms. Here too, problems are imminent.
- Test processes: Missing or inadequate automated tests, differences between development and test teams.
- Build, deployment, and release processes: Too many manual steps required for build/deploy/release, error-prone deployments.

- Operational processes: Runtime problems (performance, memory, threads, etc.) cannot be anticipated, error diagnosis can be cumbersome and unreliable.
- Support processes: Too little feedback from support to development, support optimized for short *closing time* of tickets instead of thorough problem resolution.

We leave budgeting, HR, and other processes out of this scope.

Request Processes

Some examples of problems with requirements:

- *They don't know what they want*: Requirements remain vague, imprecise, crude. The result: development teams implement highly *configurable* systems, thus postponing decisions about the concrete meaning of requirements until runtime.
- *They always want something different*: High volatility in requirements leads to too frequent changes to the code base.
- Different stakeholders demand contradictory things: These can be different requirements for data, validation or business rules, or even different algorithms.
- Requirements take too long to reach the development team throughout the organization. There are too many people involved, too many committees, or too many coordination processes. These are common problems of development processes, discussed in the following section.

Development Processes

Here too are some typical examples from our point of view:

- Coordination problems within the team or between different teams.
- Diverging goals of different people involved (*hidden agenda*), especially the connection between organizational units or development responsibilities. In such cases, technical problems are often symptoms of organizational issues.
- *Know-how bottlenecks*, essential tasks can only be taken over by very few (individual) people, who, as a consequence, work under chronic overload.

- Decision-making authority ("Who can decide?") for people who do not have the professional/technical competence ("do not know their way around").
- Organization of the development team contradicts the system's internal structure (i.e., violation of *Conway's Law*).
- Too little feedback between participants, either through organizational barriers, competing communication channels, or other communication barriers.
- Too little freedom of the development team for technical or architectural basics (e.g., update to a new language or library version or other sensible technology).

Search within development processes for tasks that involve an unreasonable number of people, take an undue amount of time, or about which stakeholders report other problems.

We all work under time or budget constraints—but that is not the "cause of all problems". However, organizations can easily exaggerate this pressure on development teams.

Test Process

Do "tests" have an appropriate status in the development process?

- Is there a reasonable amount of automated testing?
- Do automated tests run efficiently (i.e., fast enough)?
- Do automated tests work effectively? Do these tests find the right problems or significant ones? Ineffective tests lead to errors occurring even during the system's productive operation, which tests (theoretically) could have found earlier.
- Does the internal modularization (*component section*) of the system support testability? Does the internal structure of the system provide enough explicitly defined interfaces that you can test automatically?

Deployment, Releases, and Operational Processes

In our experience, there are some typical weaknesses in release creation, deployment, and operational processes:

- **Head monopolies**: Documentation of processes is not valued, individual heads hold the knowledge about operational topics.
- **Lack of automation**: Too many tasks require manual intervention.
- **Lack of self-service**: Routine tasks (e.g., creating a new development database) require extensive coordination or approval processes between several operational or development teams.
- **Lack of feedback in development**: The development team does not receive any feedback about operational problems (for example, at external interfaces) and, therefore, cannot solve these problems.
- **Lack of coordination**: The operational and development teams do not talk about fundamental changes to the software or infrastructure architecture with each other. Releases are thrown over the metaphorical wall without sufficient communication of changes.
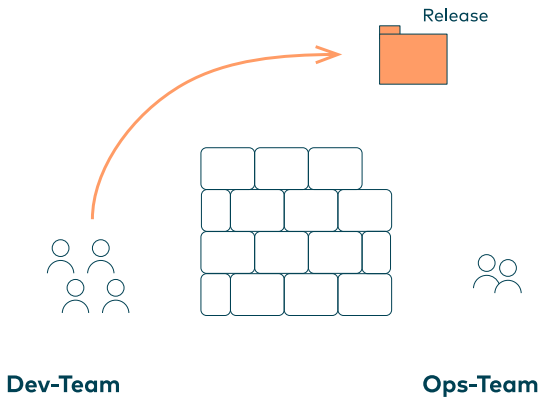

Figure 10.1: Lack of communication between the development and operational teams

11 Analysis of Infrastructure

What Is It About?

In this section, we take a closer look at infrastructure and operational procedures: Does the infrastructure fit the system and architectural goals? Does the existing monitoring and logging setup support fast problem detection and error analysis? Does the design of deployment and operation also take into account possible failure risks? Can suitable mechanisms ensure rapid recovery from system failures? How high is the degree of automation?

Infrastructure and the associated operating processes are often in a field of tension between conflicting goals: They should guarantee high reliability while keeping costs low and enable a high change rate.
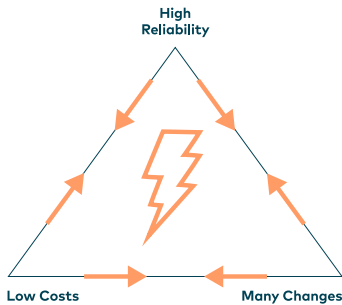
- High failure safety requires higher costs and time investment because of the necessary (complex) availability and test scenarios.
- Frequent changes to the system lead to high costs due to extensive test setup and at the same time to a higher risk of system failures.

You should individually decide on the weighting of these three priorities for your system and develop a shared understanding of the compromises made by all parties involved. Site reliability engineering methodology[1], for example, can be helpful here.

---

[1] Site reliability engineering (SRE) is a set of practices to measure and continuously improve the reliability of the operation of distributed applications. The main goal of SRE is to provide scalable and highly available software systems. In addition to troubleshooting problems and providing on-call services, site reliability engineers spend much of their time on development tasks such as developing new platform features, scaling, or automation. The software systems supported are expected to be highly automated and self-healing. See [14] and [15] for more details.

Examples of Common Problems

- There are regular backups, but restoration has not been tested.
- Changes take place directly on production systems.
- Monitoring produces many irrelevant alerts that nobody takes seriously anymore.
- Too many manual tasks or firefighting.
- Manual configuration of servers instead of automated setup.

Methodology

First, find out whether there is a common understanding of the operational objectives and compromises to be made by all those involved concerning the potential conflicts mentioned above. Do these operational goals match the architectural goals of the system? Based on this, you conduct more detailed analyses of individual areas. For example, you can use the following tools and artifacts:

- **Building plans, architectural documentation, server lists**: Get an impression of the infrastructure's complexity, the maturity of the documentation processes, and the composition of the infrastructure.
- **Documents or postmortems on system failures** in recent months: These can provide valuable information on operational stability, the maturity of the emergency processes, and the failure culture.
- **Source code for the infrastructure setup**: Enables an assessment of the degree of automation, software quality, and deployment/update processes.
- **Operation manuals, playbooks, incident management documents**: Provide an opportunity to assess risk management and error culture.
- **Monitoring systems and central log management**: Structure, level of detail, and coverage can deliver transparency and traceability in crises.
- **Interviews with stakeholders** such as the operational team, support team, and development team: They know many problems and opportunities for improvement.

If the operational processes follow standardized procedures such as ITIL or SRE, you should also look at their specific artifacts, such as the definitions of *Service Level Objectives* and *Service Level Agreements*.

Basic Infrastructure and System Architecture

When reviewing, have the underlying infrastructure and system architecture explained to you with the help of diagrams. Find answers about the following points:

- Are cloud providers used?
- On which physical hardware do the services run?
- Where (in the world) is the physical hardware located?
- What operating systems and runtime environments are used?
- Is there specialized hardware such as USV, storage array, graphics, or crypto hardware?
- Which software components run on which hardware?
- Which environments (staging, production) exist?
- Which *lock-ins* (i.e., dependencies on individual providers) exist?
- What dependencies exist between the individual components?

Operating Procedures

Find out about the company team's culture and work organization and how they work with other teams:

- How do the development and operational teams work together?
- What are the feedback loops (from business to development, from support to business)? How long does it take to implement this feedback?
- How much time does the operational team spend on manual tasks?
- How many "firefighting missions" are there?
- Who decides on the use of new infrastructure technologies?

- How are the *secrets* necessary for the system (such as administrative accounts, operating system access, DB admin access, etc.) managed?
- What are the access rules for administrative tasks (who has *root* rights)? Is there an emergency or replacement plan for this?
- Are there regular security and data protection audits?
- Is there 24/7 administration or support?
- What training opportunities are there for the operational team?

Deployment

We consider reliable and uncomplicated deployments necessary to be able to plan and efficiently bring new functionality into production. At the same time, they represent a potential risk for system failures and new errors. Check whether the procedure fits the system in terms of its duration and risk profile.
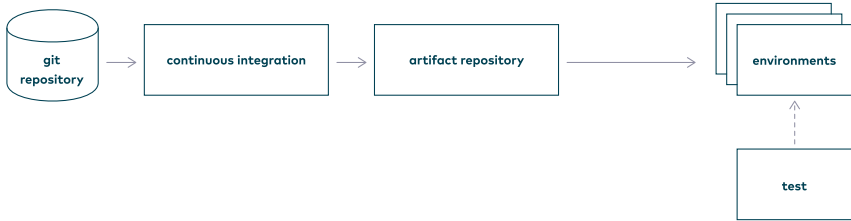
- How are deployments performed (i.e., how do transitions between the *stages* work)?
- How often are new versions of the infrastructure deployed and updated?
- Is there a deployment strategy like blue/green or canary releases?
- How long does it take to deploy and launch the individual components?
- How are deployments monitored? Is there a rollback procedure?
- How does the testing of new deployments proceed?
- Can existing instances be stopped without disturbing users?
- How are schema updates of the database rolled out?
- Is the system configured for installation, deployment, and runtime? Are there feature toggles? How are their settings managed?

Monitoring, Logging, and Analytics

Monitoring, logging, and analytics essentially have two tasks:

1. to provide a data basis for decisions
2. to point out problems and provide contextual information about them

Pay attention to whether the monitoring can adequately answer the questions of the stakeholders. In crises, monitoring should ensure transparency and responsiveness by providing the necessary data from all components, evaluating it centrally, visualizing it, and alerting if necessary. This should be done according to a uniform and systematically structured scheme. Information should be consolidated appropriately, and unnecessary *noise* should be avoided.
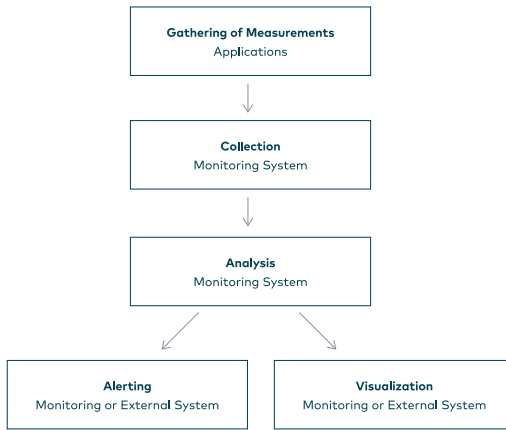

Figure 11.1: Possibilities of monitoring

- **Metrics**: Are there preset standards? Who defines the monitored metrics? Are only technical metrics monitored or also business metrics? Can conclusions be drawn from the monitoring data about the real use of the system?

- **Alerting**: Are alerts only sent when human intervention is actually required? How many "false alerts" are there?
- **Dashboards**: Dashboards should give a *high-level overview* of the overall status and avoid wild collections of visualizations.
- **Log outputs**: Is there a standardized log format across applications? Is it possible to search for log outputs centrally?
- **Analytics services**: Are such services used? What insights do they provide?

Cost Efficiency

Systems should have success metrics that allow you to evaluate their usefulness (more or less) objectively, such as turnover in monetary units or number of page views. However, systems also generate costs, such as for infrastructure, data transfer, operation, and maintenance. You should keep both aspects in mind and monitor their development over time.

- What does the cost model of the resources used look like?
- What does the operation of the resources cost? What is the cost limit here?
- Can cost savings be achieved by simplifying redundant setups, automating manual tasks, or using the cloud provider's cost analysis tools?

Infrastructure as Code

Infrastructure as code (IaC) is an elementary building block in the automation of manual tasks. Automation, in turn, helps to avoid errors during manual interventions. IaC also makes a significant contribution to providing infrastructure faster, making setups easier to understand, and speeding up deployments. On the other hand, automation may require a lot of effort.

- How are updates of infrastructure components performed?
- Is there IaC for setting up the environments?
- Which tools are used?
- For which tasks are there no scripts (yet), and why not?

- Are the scripts completely under version control? Does infrastructure code have to meet similar standards as in software development (e.g., code reviews)?

Risk Analysis Fail-safety

With the help of the previous findings, you can finally carry out a risk analysis of the system concerning fail-safety:

| collect risks | → | evaluate risks | → | prioritize risks | → | choose strategy |

- How does failover work in case of component crashes or failures of the cloud provider or data center?
- What is the probability of a specific component failing or being overloaded? How long does it take to notice this?
- How long does it take for the system to recover or for another system to take over?
- What is the maximum data loss if a data storage device fails?
- Is recovering a failed system regularly checked and drilled?
- Are there concrete and detailed instructions (*playbooks*) for standard error scenarios?
- Are postmortems for failures created, and, if necessary, are the corresponding playbooks adapted?

12 Communicating Results

Clients Want Solutions

We have often experienced that our clients already knew the existing problems, or at least anticipated them. Then we, as reviewers, should primarily serve as a source for pragmatic and realistic solutions.

Therefore, a summary of your review results must include suggestions for improvement, corrective measures, or risk minimization.

The Effective Management Summary

The compact summary of your findings, conclusions, and suggestions for the management is possibly the most important result of the entire review. Management should find a maximum of three **recommendations for action** with their derivations.

- Only write this management or executive summary at the end when your results are clear.
- This summary should always be at the beginning—of either a presentation or a document.
- Please refer to the detailed explanations that follow later.

The point of a management summary **is not** to explain why the review was so difficult and time-consuming, or what kind of stress we had to endure.

Speaking or Writing?

Whether you present results to an audience in a final presentation or write a written report is ultimately decided by your key stakeholders. We find a combination helpful in the following steps:

1. You discuss the results and their prioritization with the client, such as a review in a small group.
2. You then present the (possibly revised) results to a slightly larger group in the form of a final presentation. Here too, we would like to see active feedback and open discussion.
3. Only then do you summarize your results in writing.

Three steps mean more effort—that is why many of our reviews have been satisfactorily completed with a final presentation. We add explanations and details to the appendix. It keeps the remaining slides understandable, even without the audio track of the live presentation.

Prioritize Problems and Measures

You should prioritize and categorize problems and measures. Possibly in terms of:

- potential business damage or risk
- technical difficulty or estimated amount of effort required to solve the problem

We prefer straightforward schemes for prioritization, with three categories.

1 **High extra costs or heavy delays,** immediate adjustment recommended

2 Causes extra **costs** or **delays,** adjustment recommended

3 **Annoying,** slight delay, some additional cost

Figure 12.1: Three categories of prioritization for problems

What Belongs in the Final Presentation or Final Report

Readers should find a holistic presentation of the review, starting with the review's goals and scope, the chosen approach, conclusions, and recommendations for action. We consider the following topics to be relevant:

1. a short, very concise management summary (see above)
2. an overview of *organizational* aspects of the review:

   - What exactly were the objectives and scope of the review?
   - What are the limitations?
   - How did you proceed? How much time did you invest (approximately) in what?
   - Who did you talk to about what, including dates and duration?
   - What review activities did you carry out in addition to discussions and interviews?

3. the aspects of the system and its development which you found positive (*keep, don't change*)?
4. a summary of problems and risks, described top-down:

   - In which categories did you find problems and risks?
   - Which problems have you found, starting with the highest priorities (i.e., the worst, most serious problems first)?
   - Where are the risks?
   - Which effects are imminent or already acute?

5. an optional summary of the proposed measures:

   - In which strategic, technical, or organizational areas do you propose measures?
   - What are the different options?

The Final Presentation Is Not the End

Assume that during a final or result presentation, various stakeholders will come to you with requests for changes, some of them fundamental.

Formulations will come under the linguistic microscope of all participants—and you may have to (or be allowed to) present results to various committees in a shortened or modified form.

Expect Headwind

Those involved will doubt, question, or simply deny your results. Many aspects of complex software can be interpreted differently—for example, the importance of high percentage test coverage or strict adherence to programming conventions.

Be aware that some of your results may contradict the opinions of others involved and that these people may, therefore, reject the review entirely.

From enthusiastic approval, amazement, doubt, and belittlement to open resistance, we have had to endure a spectrum of reactions to review results. Something similar could happen to you.

Here are a few suggestions:

- In the case of policy-related or organizational headwind, you will need support at a higher level. Try to convince the top management of your results, so you can elegantly steal the wind from your opponents' sails.
- You can rarely resolve resistance and opposition with purely factual arguments. Instead, you need help from policy, economics, and psychology.
- Do your (formal) homework. You must base your results on facts you have researched and proved as thoroughly as possible. You should double-check critical results to minimize the "attack surface" for possible opponents.

13 Conclusion

Through the systematic *breadth-first search*, you have found problems and risks at some points in your system. Thanks to focus interviews with key stakeholders, you now understand their view of the system better. You have learned a lot about the architecture, code, and other *interiors* of your system, and questioned the activities around requirements, development, rollout, and operation.

You prioritized these problems according to economic and technical criteria, and for some of them, you suggested some appropriate corrective actions. Your key stakeholders were able to give their views during the final presentation, and perhaps your management has already given the green light for some promising measures.

Give yourself and your review team a pat on the back—you have done a lot.

Reviews as a Basis for Improvement and Evolution

Nevertheless, you and the development team still have a lot of work ahead of you, namely, to translate the review's findings into economic value. To do this, it is necessary to eliminate the problems identified in an order appropriate to the situation using appropriate corrective measures. Whether you pick the famous *low hanging fruit* or the *expensive* solution for a complex problem that causes high cost from a business point of view first remains your management's decision.

Your results are the essential basis for future improvements to your system. Instead of the unfortunately widespread *frenetic, unfocused activity many people succumb to*, you and your development team can now tackle the improvement and evolution of your system based on the systematic review.

We wish you much success and good luck with this exciting task.

Our Offer

A software review helps you to identify the real optimization potential of your systems at an early stage. We have many years of experience conducting such reviews, as well as in the subsequent evolution and further development of existing systems. We would also be happy to conduct reviews of your systems **together with you**.

Our training for the **iSAQB module "IMPROVE – Evolution and Improvement of Software Architectures"** shows in a practice-oriented way how you can identify and classify problems and risks of your systems and how you can implement improvements step by step.

We are happy to design individual training courses according to your requirements.

Further information on training courses and reviews is available at **https://innoq.com**.

INNOQ

We advise honestly, think innovatively, and are passionate about development—the result: successful software solutions, infrastructures, and business models.

As a technology company, we focus on strategy and technology consulting, software architecture and development, methodology and technology training, and platform infrastructures.

With over 150 employees at locations in Germany and Switzerland, we support companies and organizations in designing and implementing complex projects and improving existing software systems.

We are involved in open-source projects and the iSAQB e.V., and pass on our knowledge and experience at conferences and meetings as well as in numerous books and professional articles.

14 Our Manifesto for Reviews

- We work with the necessary care and to the best of our knowledge.
- We work **incorruptibly** and with a **self-reflecting basic attitude**.
- We treat results and information **confidentially** as a matter of course.
- We work **openly** and start **without preconceived opinions**.
- We **openly communicate** the **context** and **limits** of our reviews to make our results and conclusions comprehensible to others.
- We leave our **egos at home** to be open to suggestions, observations, ideas and problems of others.
- We separate the **problems** from the **discussions of the solutions**.
- We try to understand the **motivations of the decision makers** before we evaluate.
- We only evaluate within the scope of our competence.
- We selectively call in **further expertise** on **special topics**.
- We keep the review team small (2–5 reviewers).
- We **look ahead** instead of completely reappraising the past.
- We are **constantly developing ourselves** and our **review procedures**.

We found some of these tips in the almost historical [16] (whose approaches may seem quite formal in parts, but which contains many timeless suggestions for reviews).

15 Review? Audit? Analysis? Evaluation?

In our industry, we find very different names for the activities of examining a system for risks, problems, or even potentials. You can already see from the title of this primer that we favor the term *review*. However, we would like to introduce some similar terms here. We have formulated some of these definitions based on the corresponding Wikipedia entries.

Figure 15.1: A set of terms that can be used for review

Analysis refers to a systematic study of the components of a system, with particular emphasis on the relationships and interactions between these components.

Assessment identifies open points regarding the successful further development of a system and outlines the initial actions required for the system or its surrounding organization.

Audits examine whether systems or processes meet the required standards. Audits often take place within the framework of formal quality management.

Evaluations are often a quantitative assessment of systems, processes, and organizational units.

Inspection refers to the more in-depth, standardized analysis of a system by means of a guideline/checklist.

Postmortem refers to an analysis that takes place after the end of an event to be analyzed—usually after serious errors, failures, or other economically damaging incidents.

Retrospective (lat. *retrospectare* "to look back") refers to a brief review of events that have already taken place. The term is used in particular in iterative-agile processes, where a team collaboratively conducts an analysis in order to learn from mistakes and positive features.

Reviews check work results with a more or less formally planned and structured analysis and evaluation process. Project results are presented to a team of reviewers and commented on or accepted by them.

Root cause analysis attempts to fathom the causes of problems that have arisen, in particular, by differentiating between *symptoms* and *causes*.

Software due diligence analyzes the strengths and weaknesses of a system as well as the corresponding risks. It therefore plays an important role in cases where the purchase/sale of systems (or even companies) is involved.

Software risk evaluation identifies and analyzes risks in the software system and attempts to find remedies or mitigation strategies for these risks.

Testing is a generic term for the *analysis* (in-depth, systematic investigation), *inspection* (technical investigation), *study* (scientific investigation), or *test* (mostly functional examination) of a system.

Tests are scenarios set up to elicit certain performances in the system. The tests are designed to be as neutral as possible to determine the basic capabilities and limits of the system under examination. Test certificates validate the suitability or non-suitability of the system.

16 Sources

Books, articles, videos, and websites.

[1]   J. Ousterhout, *A philosophy of software design*. Yaknyam Press, 2018, A practical guide for better code. Not philosophical at all, but full of tangible tips for avoiding excessive complexity in the code.

[2]   B. Wolf and G. Starke, *Softwarearchitekturen pragmatisch dokumentieren. Eine kompakte einführung in arc42*. 2019 [Online]. Available: https://lean pub.com/arc42-primer

[3]   ISO/IEC 25010, *ISO/IEC 25010:2011, systems and software engineering — systems and software quality requirements and evaluation (SQuaRE) — system and software quality models*. 2011, Only available for a fee (https://www.iso.org/standard/35733.html), but on [Wikipedia](https://en.wikipedia.org/wiki/ISO/IEC_9126) well explained. [Online]. Available: https://www.iso.org/standard/35733.html

[4]   A. Tornhill, *Your code as a crime scene: Use forensic techniques to arrest defects, bottlenecks, and bad design in your programs*. Pragmatic Bookshelf, 2015, A pragmatic approach to identifying hotspots in the code—for example, by analyzing code complexity and change frequency. For us, this book was a real milestone because it takes into account the code history as it is stored in your version control system. Adam has written several open-source libraries that allow you to analyze your own code. Highly recommended for anyone involved in reviews or software engineering.

[5]   M. Schumacher, E. Fernandez, D. Hybertson, and F. Buschmann, *Security patterns: Integrating security and systems engineering*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2005.

[6]   M. Plöd, *Hands-on domain-driven design — by example*. 2019 [Online]. Available: https://leanpub.com/ddd-by-example

[7]   E. Evans, *Domain-driven design reference*. 2015 [Online]. Available: https://www.domainlanguage.com/ddd/reference/

[8]   S. Wardley, *Wardley maps — topographical intelligence in business*. 2018 [Online]. Available: https://medium.com/wardleymaps

[9]   S. J. Fowler, *Production-ready microservices*. O'Reilly Media, 2016 [Online]. Available: http://shop.oreilly.com/product/0636920053675.do

[10]   C. Lilienthal, *Sustainable software architecture: Analyze and reduce technical debt*. Rocky Nook, 2019.

[11]   A. Tornhill, *Software design x-rays: Fix technical debt with behavioral code analysis*. Pragmatic Bookshelf, 2018, Analysis based on commits and diffs. Highly recommended for all who deal with larger code bases.

[12]   Dr. M. Greiler, *Code reviews*. 2019, eBook available upon request [Online]. Available: https://www.michaelagreiler.com

[13]   S. Ambler and P. Sadalage, *Refactoring databases — evolutionary database design*. Addison-Wesley Professional, 2006.

[14]   B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site reliability engineering: How google runs production systems*, 1st ed. O'Reilly Media, Inc., 2016 [Online]. Available: https://landing.google.com/sre/books/

[15]   B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, and S. Thorne, *The site reliability workbook: Practical ways to implement SRE*, 1st ed. O'Reilly Media, Inc., 2018 [Online]. Available: https://landing.google.com/sre/books/

[16]   K. Wiegers, *Peer reviews in software — a practical guide*. Addison-Wesley, 2002, Quite old, but contains many timeless suggestions for reviews.

[17]   "Qualität in der beratung." Bundesverband deutscher Unternehmensberater [Online]. Available: https://www.bdu.de/media/296535/qualitaet_in _der_unternehmensberatung.pdf

[18]   F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-oriented software architecture – a system of patterns*. Wiley, 1996.

[19]   M. Carr, S. Konda, I. Monarch, C. Walker, and F. Ulrich, "Taxonomy-based risk identification," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, CMU/SEI-93-TR-006, 1993 [Online]. Available: ht tp://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11847

[20]   P. Kruchten, R. Nord, and I. Ozkaya, *Managing technical debt: Reducing friction in software development*, 1st ed. Addison-Wesley Professional, 2019.

About the Authors

Markus Harrer

Markus Harrer works as a software developer and consultant both in conservative industries and in start-ups. He specializes in software landscape modernization and software analytics. He helps to improve software sustainably and strategically. In his blog feststelltaste.de, he writes about the analysis of software systems using data science methods and Wardley maps. 🐦 @feststelltaste

Christine Koppelt

Christine Koppelt works as a senior consultant at INNOQ. Her focus is on the realization of digitalization projects for SMEs. In doing so, she mainly deals with the topics machine learning, DevOps and data engineering. Besides, she organizes the Data Engineering Meetup in Munich. She has experience with code and architecture reviews in medium and large systems. 🐦 @ckoppelt

Gernot Starke

Dr. Gernot Starke (INNOQ Fellow) has been working in software development for over 25 years as a software architect, coach, consultant and trainer. For clients from different industries, he has taken part in the implementation of medium and large IT systems, and in many reviews. Besides that he (co-)founded arc42 and aim42, actively runs various open-source initiatives, and writes books and professional articles. 🐦 @gernotstarke

Benjamin Wolf

Ben Wolf is a software architect and developer at INNOQ. He likes clean code and passes on his idea of software quality as a speaker at conferences and meetups as well as in training sessions. It is important to him that in particular team attitude shapes good software quality. Ben deals with code reviews and ways to systematically improve code from different perspectives. 🐦 @ichaos1985

Martin Otten

Martin Otten works as a senior consultant at IN-NOQ. His focus lies on Domain-driven design, and agile and lean methodology. In his organizational and architectural reviews, he emphasizes a holistic view of all parts of the system. 🐦 @martinotten