



INOQ

Software Reviews

Risiken und Probleme
zielsicher identifizieren

Markus Harrer • Christine Koppelt
Martin Otten • Gernot Starke • Benjamin Wolf

Software Reviews

**Risiken und Probleme in Software
zielsicher identifizieren**

**Markus Harrer
Christine Koppelt
Gernot Starke
Benjamin Wolf**

innoQ Deutschland GmbH
Krischerstraße 100 · 40789 Monheim am Rhein · Germany
Phone +49 2173 33660 · www.INNOQ.com

Layout: Tammo van Lessen with X₃LA^TE_X
Design: Susanne Kayser
Typesetting: André Deuerling

**Software Reviews – Risiken und Probleme in Software zielsicher
identifizieren**

Published by innoQ Deutschland GmbH
2. Auflage · 2022-09-01

Copyright © 2020-2022 Markus Harrer, Christine Koppelt, Gernot
Starke und Benjamin Wolf

Inhaltsverzeichnis

Jede Software hat Potential	1
1 Der gute Start	9
2 Kick-off	15
3 Analysen: Kern der Sache	19
4 Stakeholder-Interviews	21
5 Kontextanalyse	25
6 Qualitative Analyse	27
7 Architekturanalyse	39
8 Codeanalyse	45
9 Analyse von Anwendungsdaten	55
10 Analyse von Prozessen	59
11 Analyse von Infrastruktur	65
12 Resultate kommunizieren	73
13 Fazit	79
14 Unser Manifest für Reviews	81
15 Review? Audit? Analyse? Bewertung?	83
16 Quellen	85
Über die Autor*innen	89

Jede Software hat *Potential*

Potential (von lat. *potentia* „Stärke, Macht“): Fähigkeit zur Entwicklung, es gibt noch ungenutzte Möglichkeiten. Andere Lesart: Es gibt Probleme, die wir dringend beheben sollten.

Getreu dem Bonmot „Besser-geht-immer“ leidet praktisch alle Software an Problemen: Vielleicht läuft sie zu langsam, stürzt manchmal ab, sieht nicht so schön aus wie das Produkt der Konkurrenz oder Änderungen dauern zu lange. Mal abgesehen von hello-world-Anwendungen, lässt sich überall noch so manches besser machen.

Symptom und Ursache



Schon mal erlebt: Der Cursor bewegt sich unsäglich langsam über den Bildschirm. Die gesamte Anwendung reagiert (scheinbar) nur schleppend auf Klicks oder Tastendrucke.

Dieses Verhalten stellt ein *Symptom* dar – intuitiv sprechen wir von einem „langsamen System“ oder „schlechter Performanz“. Anstatt nun hektisch zu beginnen, das Event-Handling oder Window-Management des verwendeten GUI-Frameworks zu optimieren, sollten wir systematisch die Ursache(n) der schlechten Performanz ergründen. Und genau hier setzen Reviews ein.

Lassen Sie uns zu dem obigen Beispiel der schlechten Performanz an der GUI ein paar mögliche Ursachen aufzählen:

1. Die Anwendung verbraucht zu viel Speicherplatz, und das Betriebssystem muss im Hintergrund *swappen*.
2. Die Anwendung führt im Hintergrund langwierige Berechnungen auf einer umfangreichen Menge an Daten aus.
3. Die Laufzeitumgebung führt eine sogenannte *Garbage Collection* durch, um Speicher freizuräumen, so dass wenig CPU-Kapazität für Ihre Anwendung bleibt.
4. Die Anwendung ist durch Ein-/Ausgabeoperationen blockiert.

5. Für die Suchkriterien, die Sie in der Anwendung eingegeben haben, hat die Datenbank keinen Index und muss nun Milliarden von Datensätze sequentiell durchsuchen.
6. Die Anwendung muss Zusatzinformationen aus einem Nachbarsystem über eine langsame Netzverbindung laden. Normalerweise übergibt dieses Nachbarsystem einige Kilobyte an Daten, heute allerdings umfasst die Antwort einige Gigabyte.
7. Die Anwendung kommuniziert über eine Firewall mit einem Nachbarsystem. Die Firewall blockiert seit dem letzten Update einige dieser Anfragen.

In dieser kleinen Übersicht finden Sie schon eine bunte Mischung aus möglichen Programmierfehlern (1,2), Architektur-/Entwurfsproblemen (2,3,4), Hardwareproblemen (6) sowie Problemen mit Infrastruktur (1,4,7), Konfiguration (1,5,6), Datenbanken (5), Betrieb (6,7) oder externen Schnittstellen (7).

Business und IT

Szenenwechsel: Neulich, irgendwo in einem Unternehmen. Die Fachseite (neudeutsch: das *Business*) jammert, dass die IT wieder viel zu lange braucht, um dringend benötigte *Features* zu liefern.



Diese Situation halten wir für einen *Klassiker* und haben dafür eine passende Bezeichnung: Schlechte Time-to-Market (TTM). Schlechte TTM halten wir grundsätzlich für ein Symptom, ausgelöst durch eine oder mehrere der folgenden potentiellen Ursachen:

- Schlechte Modularisierung, mit der Konsequenz enger Abhängigkeiten
- Schlechte Codequalität, handwerklich schlecht programmiert
- Mangelnde Konsistenz oder fehlende Homogenität, insbesondere bei Querschnittsthemen
- Unzureichende Basistechnologie
- Mangelhaftes Testkonzept und fehlende Testautomatisierung

- Schwerfällige Entwicklungsprozesse, insbesondere im Bereich der Anforderungsklä rung
- Zu viele Meetings und Abstimmungen
- Zu viele beteiligte Organisationseinheiten oder zu große Teams

Schlechte TTM verteuert einerseits die Entwicklung, andererseits verhindert sie (zuverlässig!) die oftmals so notwendige *Business Agility*, also die schnelle Adaption einer Organisation an geänderte Marktbedingungen oder Kundenwünsche. Daher sollten Sie bei Reviews auf jeden Fall auf TTM-Probleme achten.

Suchen Sie überflüssige Komplexität

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

Überflüssige Komplexität gilt als Ursache für vielerlei Folgeprobleme – wobei die oben erwähnte Time-to-Market möglicherweise die schlimmste davon ist. Anders ausgedrückt: Viele der Probleme in oder mit Software resultieren aus überflüssiger Komplexität.

Komplexität in Software kann sich auf sehr unterschiedliche Arten zeigen: im Quellcode, in architektonischen Strukturen (Modularisierung, Abhängigkeiten), in schwerfälligen und überladenen Datenmodellen, in falsch angewandter Technologie, in schwerfälligen Entwicklungs- und/oder Betriebsprozessen – sicher können Sie diese Liste fortführen.

Suchen Sie in Reviews nach überflüssiger Komplexität, denn die liefert Ihnen fast immer gute Gelegenheiten für Vereinfachung und Verbesserungen. Insbesondere in der Architektur- und Codeanalyse kommen wir auf Komplexität noch näher zu sprechen.

Vor der Therapie: Diagnose

Falls Sie dieses Verbesserungspotential Ihres Systems ausschöpfen möchten, sollten Sie auf jeden Fall zuerst ein explizites, konkretes und spezifisches Verständnis der vorhandenen Probleme erarbeiten.

So wie in der Medizin stets eine gründliche Untersuchung einer Patientin oder eines Patienten erfolgen sollte, bevor die zuständigen Mediziner*innen Therapie-vorschläge ausarbeiten, sollten Entwicklungsteams vor jeglichen größeren Verbesserungsmaßnahmen („Therapie“) eine solche Diagnose („Review“) durchführen. Ansonsten droht *Verschlimmbesserung*.

Dieses Buch hilft Ihnen dabei, indem es Ihnen konkrete Anregungen für solche Reviews gibt.

Wir haben diese Ansätze in Reviews sehr unterschiedlicher Systeme in verschiedenen Branchen selbst verwendet, und bei uns haben sie gut funktioniert. Insofern halten Sie hier einen kompakten Praxisbericht über *good practices* von Reviews in der Hand, mit dem Sie in Ihren eigenen Systemen auf die systematische Suche nach Problemen und Risiken gehen können.

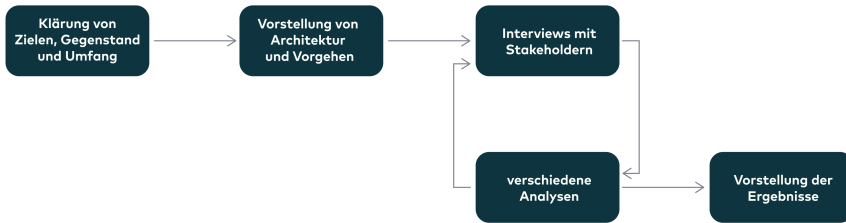
**„Often you are too close to your own work to spot errors
you've made.“**

Karl Wiegers, 2002

Wie Reviews grundsätzlich funktionieren

Unabhängig von Schwerpunkt und Umfang, finden Reviews schematisch immer in fünf Phasen statt:

1. Vor Beginn der eigentlichen Arbeit müssen wir mit den maßgeblichen Stakeholdern (z. B. Auftraggebern, Produkt- oder Entwicklungsverantwortlichen) die Zielsetzung sowie den genauen Betrachtungsgegenstand klären. Wir sollten ebenfalls den Umfang des Reviews und die gewählte Vorgehensweise klären. Dazu finden Sie mehr Informationen im Abschnitt **Der gute Start**.



Quelle: P. Ghadir

Abbildung 1: Schematischer Ablauf von Reviews (nach P. Ghadir)

2. Als ersten Schritt der inhaltlichen Arbeit führen wir mit einer ausgewählten Gruppe relevanter Personen einen kompakten Kick-off-Workshop durch, in dem wir diesen Personen die Ziele und das Vorgehen vorstellen. Im Gegenzug erklären diese Stakeholder das System, dessen geschäftliche Bedeutung sowie die beteiligten Prozesse im Überblick. Siehe Abschnitt **Kick-off**.
3. Dieses Kick-off fungiert als Grundlage vertiefender Interviews. Mehr Informationen dazu finden Sie im Abschnitt **Stakeholder-Interviews**.
4. Nach diesen Gesprächen folgen nach Bedarf weitere Analyseschritte, in denen wir gezielt auf die Suche nach Problemen und Risiken bestimmter Kategorien gehen. Methodische Vorschläge für diese Analysen finden Sie in **Analysen: Kern der Sache** ff. Diese Analysen und die Interviews von Schritt 3 können sich abwechseln: Die Ergebnisse von Interviews können bestimmte Analysen notwendig machen und umgekehrt!
5. Schließlich müssen wir die Ergebnisse und Schlussfolgerungen für die Beteiligten aufarbeiten und ihnen vorstellen. Die Kernpunkte dazu erklären wir in Abschnitt **Resultate kommunizieren**.

Wer führt Reviews durch?

Wir (Christine, Ben, Markus und Gernot) arbeiten für INNOQ, einen IT-Dienstleister mit Fokus auf Softwarearchitektur, und führen Reviews als *Externe* durch. Wir betrachten Systeme von IT-Anwenderorganisationen und bringen unsere neutrale Außensicht ein. Wir können anonymisierte und neutrale

Vergleiche mit ähnlichen Systemen oder Domänen anstellen und unseren Auftraggebern oftmals neue Impulse oder Anregungen geben.

Andererseits können Sie, egal in welcher Rolle Sie bei der Systementwicklung arbeiten, natürlich auch Ihr eigenes System oder eines der Nachbarabteilung Ihrer Organisation reviewen. Nachfolgend finden Sie einige (mögliche) Argumente für und gegen interne und externe Reviewer.

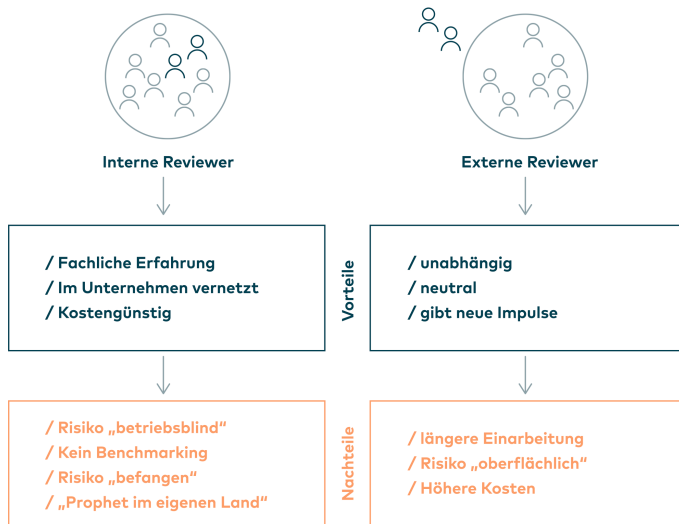


Abbildung 2: Mögliche Vor- und Nachteile interner versus externer Reviewer

Review? Bewertung? Audit? Analyse?

Es geistern verschiedene Begriffe durch die IT-Branche, die sich mit der Identifikation von Problemen oder Risiken in bestehenden IT-Systemen befassen. Wir haben uns für *Review* entschieden. Eine genauere Abgrenzung zu anderen Begriffen finden Sie im Anhang.

Danksagung

Ganz pauschal danken wir unseren Mandant*innen und Kund*innen, deren Systeme wir in den letzten Jahren untersuchen durften, sowie allen Committern der Open-Source Methode aim42.org¹. Außerdem ein großes Dankeschön an IN-NOQ: Hier leben wir Meinungsfreiheit, Diversität und offene Kommunikation mit großartigen Kolleg*innen. Besonderer Dank geht an Phillip Ghadir für seine konkreten Vorschläge für Vorgehen und Praktiken. Wir danken zudem Lars Hupel und Martin Otten für ihr Durchhaltevermögen bei den drucktechnischen Feinschliffen sowie Sonja Scheungrab und Robert Glaser für ihre gestalterischen Verbesserungsvorschläge.

Schließlich Danke an unsere Familien, deren moralischer Beistand uns durch die Untiefen des *blank paper syndroms*² geführt hat.

¹<https://aim42.org>

²Blank Paper Syndrome: Auch genannt Schreibblockade (*Writers Block*).

1 Der gute Start

Bevor Sie mit dem eigentlichen Review beginnen, sollten Sie erstmal dessen Motivation und Ziele klären sowie die zu betrachtenden Teile / Aspekte des Systems mit ihren Auftraggebern diskutieren.

Weiterhin klären Sie möglichst konkret die Erwartungshaltung bezüglich Ablauf und Vorgehen, Art und Kommunikation von Ergebnissen sowie des zeitlichen Umfangs.

Ziele definieren

Die Grundlage erfolgreicher Reviews bildet eine klare Definition der Ziele. Danach müssen Sie die Auftraggebenden Stakeholder befragen, bzw. ihnen bei der Klärung dieser Ziele helfen. Fragen Sie ebenfalls nach den Motiven oder Auslösern des Reviews. Beides trägt dazu bei, dass Sie Ihr Vorgehen, die Art und Weise Ihrer Analysen sowie Ihrer Ergebnisberichte und -präsentationen an diesen Zielen ausrichten können.

Beispielhaft zeigen wir Ihnen einige Ziele aus konkreten Reviews (anonymisiert):

- Einholen einer unabhängigen Meinung über die Architektur des Systems.
- Klären, inwieweit das System zukunftssicher entworfen und implementiert wurde.
- Erfassen des technischen Ist-Zustands eines Systems, zu dem kaum (meist: keine) Dokumentation vorliegt.
- Hinterfragen bekannter Probleme mit dem System (etwa: zu schlechte *time-to-market*, mangelnde Performance).
- Klären der Auswirkungen neuer Anforderungen auf Architektur, Implementierung und Betrieb des Systems.
- Nachweis, dass bestimmte Entscheidungen der Vergangenheit aus heutiger Sicht nicht mehr akzeptabel sind.

Halten Sie die gemeinsam abgesprochenen Ziele schriftlich fest! Bei Ihren Interviews mit diversen Stakeholdern sowie zur Kommunikation Ihrer Review-Ergebnisse am Ende werden Sie diese Formulierungen wieder benötigen.

Scope klären

Neben den Zielen müssen Sie den *Betrachtungsgegenstand* des Reviews klären. Was genau sollen Sie untersuchen? Die Formulierung „Das System X“ könnten verschiedene Personen unterschiedlich auslegen, daher legen Sie gemeinsam mit Ihren Auftraggebern fest, was genau Sie untersuchen sollen:

1. Die *Architektur* des Systems, worunter wir¹ mindestens Folgendes verstehen:
 - Den strukturellen (makroskopischen) Aufbau des Systems aus *großen* Komponenten, deren Abhängigkeiten und Schnittstellen.
 - Die querschnittlichen Konzepte und dafür eingesetzten Technologien.
2. Die *Implementierung* des Systems, also dessen Quellcode.
3. Den Betrieb des Systems in dessen Zielumgebung(en).
4. Die Entwicklung bzw. den Entwicklungsprozess des Systems, was folgende Teilaufgaben umfassen könnte:
 - Anforderungsklä rung und -management
 - Koordination und Durchführung der Implementierungsaufgaben
 - Test und Qualitätssicherung
 - Konfigurations- und Versionsmanagement
 - Freigabe/Release, Deployment und Rollout
 - Änderung, Fehlerbehebung (*change management*)

Manchmal widersprechen sich Ziele und Scope von Reviews. Ein Beispiel: Wir sollten prüfen, ob ein System und dessen Architektur grundsätzlich *mandantenfähig* sind. Leider hatten wir keinen Zugriff auf den Quellcode des Systems, das heißt wir konnten gar nicht im Detail überprüfen, ob

¹ „Softwarearchitektur bezeichnet die grundsätzliche (*makroskopische*) Organisation eines Systems, wie sie sich in dessen Komponenten, deren Beziehung zueinander und zur Umgebung widerspiegelt, sowie die Prinzipien, die für seinen Entwurf und seine Evolution gelten.“ Diese Definition stammt aus dem IEEE-Standard 1471.

z. B. architektonische Ansätze zur Mandantentrennung in der Datenbank überhaupt korrekt im Code umgesetzt waren.

Weisen Sie mindestens auf solche Widersprüche hin. Falls Ihrer Einschätzung nach diese Ziele mit dem gegebenen Scope nicht erreichbar sind, sollten Sie im Extremfall das Review ablehnen!

Umfang klären

Eine wichtige Frage zwischen Auftraggeber und Ihnen dreht sich um den zeitlichen Aufwand, der Ihnen für das Review zur Verfügung steht. Wir möchten die klassische *kommt-drauf-an*-Antwort durch einige Beispiele und Faustregeln etwas konkretisieren.

Wir haben Reviews zwischen zwei und 60 Personentagen (kurz: PT) durchgeführt, je nach Zielsetzung und Größe der Systeme. In zwei bis vier PT können Sie einen Überblick gewinnen und eine Art „Bericht zur Lage der Nation“ abgeben, ohne dabei viel Code zu sehen. Die größeren Reviews (40-60PT) bezogen sich auf Architektur und Code signifikant großer Systeme, inklusive Betrachtung der Entwicklungs- und Betriebsorganisation.

In Systemen mit 100 Personenjahren Entwicklungsaufwand bedeuten 50 PT ja gerade mal bescheidene 0.2%. Im Vergleich: Inspektionskosten für PKW liegen bei knapp 1% des Kaufpreises² pro Jahr.

- Je größer oder komplexer das System, desto mehr Zeit sollten Sie dem Review gönnen. Ein halbwegs gründliches Architekturreview eines 2-3 Millionen LoC Systems mit 20+ Personen Entwicklungsteam benötigt 10 und mehr Personentage.
- Je gründlicher Sie untersuchen sollen, desto mehr Zeit benötigen Sie.
- Wenn Sie beispielsweise nur eine werkzeuggestützte Code- und Architekturanalyse durchführen, kommen Sie mit wenigen Tagen aus.
- Sie können maximal vier bis sechs Interviews an einem Tag führen (unser persönlicher Rekord steht bei 19 Gesprächen in zwei Tagen – bitte nicht nach-

²Quelle: <https://www.fairgarage.de/inspektionskosten-vw-sowie-AutoScout24>.

machen, das war eine schlechte Idee). Sie benötigen für jedes Interview etwa nochmal soviel Zeit für Nacharbeit.

- Eine Folie für einen Review-Bericht kostet eine Stunde Zeit, Diskussion und Überarbeitung inklusive.

Kalkulieren Sie von Ihrem Zeitbudget mindestens 20% für die Vorbereitung und Kommunikation Ihrer Ergebnisse. Oftmals kommen nach einer Abschlusspräsentation noch eine Vielzahl von Änderungswünschen!

Vorgehen bei Reviews klären

Unsere Empfehlung: Arbeiten Sie bei Reviews iterativ.

Geben Sie Ihren Auftraggebern nach circa 20-30% des gesamten Zeitbudgets eine kompakte und vorläufige Rückmeldung über mögliche Resultate. Stimmen Sie auf Basis dieser ersten Rückmeldung das weitere Vorgehen ab, insbesondere die weiteren Untersuchungsschwerpunkte.

Zu dieser Klärung des Vorgehens gehört für uns auch die Überlegung, mit welchen Personen Sie Interviews führen sollten. Mindestens diese sollten Ihre Auftraggeber nämlich zum Kick-off (siehe nachfolgendes Kapitel) einladen.

Beginnen Sie nach dem Kick-off zuerst mit Interviews, bevor Sie mit anderen Analysen beginnen.

Mitwirkung der Auftraggeber klären

Sie benötigen während des Reviews öfter Kontakt zu unterschiedlichen Personen oder Organisationseinheiten, die Sie möglicherweise im Vorfeld noch gar nicht kennen. Lassen Sie vom Auftraggeber eine Person benennen, die Ihnen beispielsweise hilft, Räume für Interviews zu finden, Termine mit überarbeiteten Stakeholdern zu arrangieren, Zugriff zu benötigten Dokumenten sowie ggfs. zum Quellcode zu erhalten. Außerdem sollte dieser Mensch Ihnen den Weg in die Kaffeeküche zeigen und für die Abschlusspräsentation einen Beamer besorgen.

Vorab-Informationen beschaffen

Lassen Sie sich zur Vorbereitung der nachfolgenden Aktivitäten vorhandene (technische) Dokumentation aushändigen. Ideal dazu wäre eine Architekturdokumentation, sofern vorhanden³. Jegliche Übersichtsdiagramme, Informationen über externe Schnittstellen, Nutzergruppen und -rollen, den Betrieb des Systems – nehmen Sie erstmal alles, was grob zum Scope des Systems gehören könnte.

Beteiligte informieren (lassen)

Das spart Ihnen bei den nachfolgenden Interviews eine Menge Zeit, und Ihre Interviewpartner*innen haben alle den gleichen Informationsstand.

Außerdem beugt diese proaktive Kommunikation der Review-Ziele der unproduktiven Gerüchteküche vor (die Sie während des Reviews tunlichst vermeiden sollten).

Vertraulichkeit klären

Wir bekommen als Reviewer vertrauliche Informationen zu Gesicht, die wir exklusiv für die Zwecke des Reviews verwenden dürfen. Sie sollten zwischen Auftraggeber und Review-Team dafür eine Geheimhaltungsvereinbarung schließen (auch *Non Disclosure Agreement* genannt), in der sich das Review-Team zur absoluten Verschwiegenheit gegenüber Dritten verpflichtet.

³ Sofern es keine Architekturdokumentation gibt, haben Sie bereits das erste Risiko identifiziert und können auch gleich den ersten Verbesserungsvorschlag unterbreiten. Ben und Gernot haben dazu ein kleines Buch [2] geschrieben :-)

2 Kick-off

Versetzen Sie sich in die Lage derjenigen Menschen, die das System benutzen, betreuen, weiterentwickeln und betreiben: Plötzlich kommt ein (internes oder externes) Review-Team, stellt eine Menge Fragen und sicherlich auch Dinge-in-Frage.

Ziele des Kick-offs

Nutzen Sie das Kick-off dazu, *Vertrauen* herzustellen und gemeinsam mit den Auftraggebern die Ziele des Reviews allen Beteiligten offen zu kommunizieren. Erklären Sie Ihre Vorgehensweise, zusammen mit eventuellen Meilensteinen oder End-Terminen. Lassen Sie die Auftraggeber deren Erwartungshaltung im Kick-off offen kommunizieren¹.

Die Beteiligten sollten während des Kick-offs Gelegenheit haben, das Review-Team kennen zu lernen. Daher legen wir großen Wert darauf, dass sämtliche Reviewer hier Präsenz zeigen und sich und ihre jeweiligen Arbeitsschwerpunkte persönlich vorstellen.

Wer sollte teilnehmen

Die Auftraggeber des Reviews sollten mit Ihnen als Reviewer gemeinsam die Beteiligten des Kick-offs definieren. Einen Vorschlag haben Sie hoffentlich gemeinsam beim *guten Start* (siehe voriges Kapitel) erarbeitet.

Weiterhin laden wir gerne Vertreter*innen der folgenden Rollen/Aufgaben ein:

- Fachlich Verantwortliche, *Product-Owner*, Produktverantwortliche,
- Entwicklungsteam, Implementierung und Architektur,

¹Falls die Auftraggeber eine *hidden agenda* haben sollten, beispielsweise die Ergebnisse des Reviews zur Rechtfertigung von Personalabbau oder gravierenden Kostensenkungen zu benutzen, wird diese offene Kommunikation nicht gewünscht sein. In solchen Fällen: Willkommen in der Höhle der Löwen oder den Wirren der Unternehmenspolitik. Außer einem deutlichen „Vorsicht“ geben wir Ihnen dafür keine weiteren Ratschläge.

- Test & QS,
- Build, Deployment, Release und Betrieb sowie natürlich
- Nutzer*innen

Vorbereitung

Sie sollten vor dem Kick-off die Unterlagen durchsehen, die Sie als Vorbereitung erhalten haben. Damit haben Sie hoffentlich einige wesentliche Fachbegriffe und systemspezifische Terminologien gehört. Außerdem sollten Sie die Namen der wesentlichen Stakeholder kennen – inklusive deren Rollen in Bezug auf System und Review. Falls Ihnen Dinge unklar bleiben haben Sie direkt ein paar gute Fragen für Interviews.

Die Agenda: Was sollten Sie besprechen

Uns ist bewusst, dass jeder Kick-off eine individuelle Agenda verdient und benötigt – aber ein paar Themen gehören einfach dazu.

Erzählen Sie als Reviewer ein wenig über Ihren beruflichen Werdegang, und begründen Sie fachlich oder technisch, **dass und warum genau sie** die geeigneten Personen zur Erreichung der Review-Ziele sind.

Bitten Sie die übrigen Beteiligten um Antworten auf einige Kernfragen – die Sie gerne am Flipchart für alle sichtbar notieren können:

- Was ist Ihre Rolle?
- Wie lange kennen Sie das System bereits, oder arbeiten daran/damit?
- Zu welchem Teil der gesamten Organisation gehören Sie? Sind Sie intern/extern?
- Eine persönliche Frage *außer der Reihe* kann helfen, den *Menschen* hinter der *Rolle* zu erkennen. Beispiele: Was ist Ihr Lieblingsbuch? Lieblingsfilm? Ihr favorisiertes Hobby? Ihr Traum-Reiseziel?
- Die fachlich verantwortlichen Personen sollten für alle den **wirtschaftlichen oder fachlichen Zweck** des Systems in Kurzform darstellen.

- Technisch verantwortliche Personen (etwa: Architektur, Entwicklungsleitung) sollten die **Architektur** des Systems zeigen, zumindest die wesentlichen Subsysteme oder Komponenten und die fundamentalen technischen Entscheidungen
- Jemand sollte einen Überblick über das **Entwicklungsvorgehen** geben: von Anforderungen, Entwurf, Implementierung, Test, Build, Deployment bis zum Release

Wir haben gute Erfahrung damit gemacht, mit zumindest einigen der am Kick-off Beteiligten unmittelbar im Anschluss eine kurze qualitative Analyse des Systems durchzuführen – angelehnt an einen ATAM-Workshop (siehe Kapitel Qualitative Analyse).

3 Analysen: Kern der Sache

Bevor wir mit der eigentlichen Analyse beginnen, warnen wir eindringlich vor der *Mikroskop-Falle*: Wenn Sie in einem eng begrenzten Gebiet suchen, werden Sie höchstens dort Probleme finden. Untersuchen Sie beispielsweise nur *Abhängigkeiten* in einem bestimmten Modul einer Software, können Sie mögliche Speicherlecks, Performance- oder Sicherheitsprobleme nicht finden.

Da Probleme und Risiken in IT-Systemen durch viele unterschiedliche Faktoren verursacht werden können, sollten Sie Reviews grundsätzlich als Breitensuche durchführen, d. h. unterschiedliche Aspekte des Systems und dessen Entwicklung berücksichtigen.

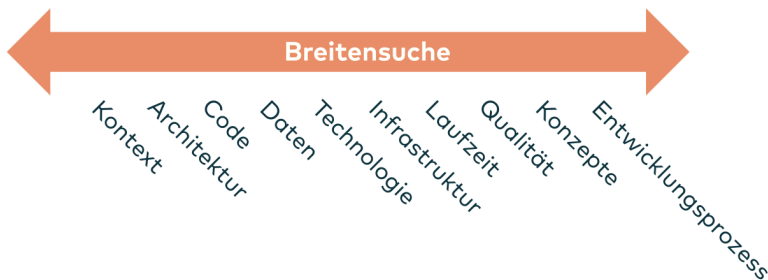


Abbildung 3.1: Mögliche Themen einer Breitensuche

Welche dieser Themen Sie genauer betrachten, hängt von mehreren Faktoren ab:

1. Der Zielsetzung und dem Scope des Reviews, die Sie hoffentlich mit den maßgeblichen Stakeholdern im Kick-off explizit geklärt haben.
2. Bereits bekannten Problemen oder Risiken, auf die uns Stakeholder hinweisen.

Beginnen Sie die Analysephase eines jeden Reviews grundsätzlich mit einigen Interviews, in denen Sie wesentliche Stakeholder zum *Stand der Dinge* befragen.

Diesen Interviews haben wir das nachfolgende Kapitel Stakeholder-Interviews gewidmet.

Kategorien von Analysen

Aufgrund der Kürze dieses Buches haben wir einige der Kategorien aus obiger Abbildung hier zusammengefasst. In der Tabelle bekommen Sie einen Vorgeschmack, welche Art von Problemen Sie mit welcher Kategorie von Analysen finden können.

Diese Analysekategorie	findet Probleme mit/bei
Kontext	Externe Schnittstellen
Qualität	Performance, Stabilität, Änderbarkeit, Time-To-Market, Sicherheit etc.
Architektur	Abhängigkeiten im Großen, Technologien, Konzepte, Wissensverteilung im Team
Code	Inkonsistenzen, Hotspots, Abhängigkeiten im Kleinen, schlechte Codequalität
Daten	Strukturen, Verteilung/Replikation, Datenstrukturen und -modellen
Infrastruktur	Release/Deployment, technische Infrastruktur, Betrieb des Systems
Prozesse	Anforderungskklärung und -management, Entwicklungs-, Test- und Release-Prozesse

Aufbau der Analysen

In diesem Buch suchen wir in rund einem Dutzend unterschiedlicher *Gebiete* nach Problemen und Risiken. Die dafür jeweils möglichen *Analysen* beschreiben wir in folgender Struktur:

- Eine **kurze Beschreibung**, worum es bei dieser Analyse geht.
- **Beispiele typischer Probleme/Risiken**, die Sie mit dieser Analyse finden können.
- **Methodik**: Mögliche Vorgehensweisen, methodische oder technische Werkzeuge.
- **Verweise** auf Hintergrundinformationen, Literatur und verwandte Themen.

4 Stakeholder-Interviews

Mit wem sollen Sie sprechen?

Da wir Reviews ja gerne im Modus der Breitensuche durchführen, empfehlen wir Ihnen, eine entsprechend breite Auswahl an beteiligten Personen und Rollen in Interviews anzusprechen.

Dazu gehören insbesondere:

- Nutzer*innen
- Fachlich Verantwortliche (etwa: Vertreter der Fachbereiche)
- Technisch Verantwortliche (etwa: Architekt*innen)
- Auftraggeber*innen, Management
- Beteiligte aus dem Entwicklungsteam. Bei größeren Systemen mit mehreren großen Subsystemen oder Komponenten eine Mischung aus mehreren Teams.
- Test/QS, falls nicht Teil des Entwicklungsteams
- Personen aus Infrastruktur und/oder Betrieb
- Projekt- und Produkt-Manager*innen
- In agilen Organisationen: Product-Owner, Scrum-Master oder Agile-Coach

Interviews vorbereiten

Nehmen Sie sich die Zeit, um vorab einige Fragen zu notieren, die Sie Ihren Interview-Partner*innen stellen möchten – abhängig von deren Rolle, Aufgaben, Erfahrung oder Art der Beteiligung am System. Teilen Sie diese Fragen Ihren Interview-Partner*innen **vorab** mit, etwa in der Einladungs-Email. So können sich diese Personen besser auf das Interview einstellen, bei Bedarf sogar passende Unterlagen oder Dokumente mit zum Gespräch bringen.

Welche Fragen stellen?

Stellen Sie **offene** Fragen, die Sätze als Antworten erfordern, und lieber keine geschlossenen Ja/Nein-Fragen. Fragen Sie nach **bekannten Problemen oder Risiken**, nach Schwierigkeiten oder übermäßig aufwändigen, schwierigen oder sonstigen negativen Aspekten des Systems. Fragen Sie, was Personen **unbedingt beibehalten** möchten. Fragen Sie auch nach **Vorschlägen für Abhilfen**, beispielsweise:

- „Was würden Sie ändern, wenn Sie eine Woche lang die gesamte Entwicklung beliebig steuern dürften?“
- „Was wären Ihre Top-3 Änderungen (am System, an Entwicklungsprozessen oder der Organisation)?“

Fragen Sie „Was sollte ich Sie sonst noch fragen?“, um Ihrem Gegenüber die Gelegenheit zu geben, noch weitere Themen anzusprechen.

Interviews führen

Führen Sie Interviews am besten zu zweit, als Duo nehmen Sie mehr mit. Wechseln Sie fragen und mitschreiben ab, dann kann sich die fragende Person besser auf Gestik, Mimik und Verhalten der anderen konzentrieren und ist nicht ständig durch mitschreiben abgelenkt. Markieren Sie Bemerkungen, die weitere Analysen oder Nachprüfungen erfordern. Lassen Sie sich bei Bedarf weitere Ansprechpersonen nennen, die mehr Informationen zu bestimmten Themen haben. Begrenzen Sie Interviews auf 60-90 Minuten und vereinbaren bei Bedarf lieber Folgetermine. Zwischen zwei solcher Interviews sollten Sie für sich selbst 15 Minuten Pause einplanen, damit Sie ganz kurz reflektieren können, Ihre Notizen kennzeichnen und sich für das nachfolgende Interview mental sammeln können.

Zielorientiert mitschreiben

Verwenden Sie für Mitschriften mehrere Farben. Markieren Sie Äußerungen zu bestimmten Themen konsistent in derselben Farbe. Beispiel: Architekturthemen

markieren Sie grün, Betriebs- oder Deployment-Themen orange, Anforderungsthemen lila usw.) Vermerken Sie auf jedem Notizzettel Datum und beteiligte Personen. Schreiben Sie auf die erste Seite (in Form von Hash-Tags), zu welchen Themen dieses Interview besonders interessante Auskünfte enthielt (beispielsweise: #database, #anforderungen, #scrum-fail).

Pre-Mortem

Falls Sie merken, dass Teilnehmende an Interviews nur zögerlich über die aktuellen Probleme reden möchten, versuchen Sie einen gedanklichen Zeitsprung und stellen die Frage nach möglichen Horrorszenarien der Zukunft:

„Was muss passieren, um das System und/oder die Entwicklung in zwei Jahren so richtig gegen die Wand fahren zu lassen?“

Mit dieser Frage können Sie Risiken im Brainstorming-Manier sammeln. Hier können Sie auch Aussagen von negativ eingestellten Personen positiv nutzen.

Risk-Storming

Wenn Sie merken, dass Interview-Teilnehmende das System und dessen Risiken nicht wirklich kennen, lassen Sie von der Gruppe zuerst ein Systemübersichtsdiagramm auf ein Flipchart oder Whiteboard zeichnen. Fragen Sie danach nach möglichen Risiken in den Systemkomponenten und deren Schnittstellen. Lassen Sie die Risiken von allen Teilnehmenden in Einzelarbeit auf Klebezetteln sammeln, auf dem Systemübersichtsdiagramm platzieren und in der Gruppe vorstellen.

5 Kontextanalyse

Bei der Kontextanalyse suchen wir Risiken und Probleme in der Umgebung (*im Kontext*) des Systems, d. h. insbesondere bei externen Schnittstellen und/oder Nachbarsystemen.

Beispiele typischer Probleme

- Externe Nachbarsysteme liefern Daten fehlerhaft, unvollständig, zu spät oder zu unerwarteten Zeitpunkten.
- Externe Nachbarsysteme ändern häufig ihre Schnittstelle.
- Die Spezifikation einer externen Schnittstelle ist unvollständig, fehlerhaft oder fehlt ganz.
- Im laufenden Betrieb reagiert ein Nachbarsystem zu langsam.
- Das System erhält über eine externe Schnittstelle zu viele Anfragen (eine Vorstufe zu einem *Denial-of-Service*-Angriff).
- Das System (oder ein Nachbarsystem) kann *Service Level Agreements* zur Laufzeit nicht einhalten.
- Die Betriebskosten einer externen Schnittstelle liegen deutlich über den erwarteten Werten.

Methodik

Stellen Sie sicher, dass Sie sämtliche externen Schnittstellen kennen. Eine Tabelle aller externen Schnittstellen mit Verweise auf deren fachlichen Inhalt sowie Spezifikation hilft schon sehr. Eine grafische Darstellung in Form eines Kontextdiagramms finden wir zur Übersicht nützlich.

1. Befragen Sie die Verantwortlichen für diese Schnittstellen zu möglichen Problemen und Risiken. Beziehen Sie hier bei Bedarf auch die Verantwortlichen der Nachbarsysteme ein, die ja möglicherweise außerhalb der zu betrachteten Organisation positioniert sind.
2. Hinterfragen Sie die Qualitätsanforderungen und -zusicherungen (*Service Level Agreements*) dieser Schnittstellen (*Soll-Zustand*).

3. Ermitteln Sie die Einhaltung dieser Qualitätsanforderungen, etwa durch Befragung verantwortlicher Personen oder durch eine Laufzeitanalyse.
4. Noch spezieller: Beobachten Sie diese Schnittstellen im laufenden Betrieb, etwa durch Analyse von Logfiles oder spezifisches Monitoring (siehe Laufzeitanalyse).
5. Überprüfen Sie die Implementierung externer Schnittstellen durch gezielte Codereviews. Voraussetzung: Es gibt eine angemessene Spezifikation dieser Schnittstellen.

Widmen Sie folgenden Kategorien externer Schnittstellen besondere Aufmerksamkeit:

- Die Nutzung dieser Schnittstelle verursacht direkte Kosten, wenn Schnittstellenpartner beispielsweise pro Aufruf oder pro übertragenem Datensatz eine Gebühr verlangen.
- Die Definition oder Implementierung der Schnittstelle unterliegt außergewöhnlich hoher Volatilität, d. h. ändert sich häufig.
- Der laufende Betrieb dieser Schnittstelle erfordert manuelle Eingriffe.

Verweise

- **Betriebliche Probleme** an externen Schnittstellen können Sie mit Laufzeitanalyse identifizieren.
- Untersuchung von Performance-/Stabilitätsproblemen an externen Schnittstellen gehört zur qualitativen Analyse.

6 Qualitative Analyse

Qualität (von lat. *qualitas*, Beschaffenheit, Eigenschaft): Summe aller Eigenschaften eines Systems. Qualität gibt an, inwieweit ein System den bestehenden (impliziten und expliziten) Anforderungen entspricht.

Softwarequalität

Diese abstrakte Definition des Begriffes *Qualität* hilft weder in der praktischen Softwareentwicklung noch bei Reviews. Uns gefällt die pragmatische Erklärung besser:

Die Qualität eines Systems gibt an, inwieweit es bestehenden Anforderungen genügt.

Zum Glück definiert [3] ein praktikables Qualitätsmodell für Software, mit dessen Hilfe Sie die konkreten Qualitätseigenschaften eines Systems analysieren können. Dieser Ansatz basiert auf einer Zerlegung des abstrakten Begriffes in acht Untergruppen, welche in der folgenden Abbildung dargestellt sind.



Abbildung 6.1: Qualitätsmodell ISO 25010

Diese Untergruppen können Sie sich nun systematisch *vorknöpfen* und jeweils analysieren, ob Ihr System die Anforderungen der jeweiligen Untergruppe erfüllt

oder nicht. Diese Art der Qualitätsanalyse ist unter dem Namen ATAM¹ bekannt und im methodischen Software-Engineering unserer Erfahrung nach relativ verbreitet.

In diesem Buch möchten wir aus dem Zoo der ISO-Qualitätsmerkmale einige beispielhaft herausgreifen, die uns bei Reviews konkreter Systeme vielfach negativ aufgefallen sind:

- Performance
- Erweiterbarkeit bzw. Änderbarkeit
- Sicherheit

Diese drei behandeln wir in den nachfolgenden Abschnitten. Zum Schluss dieses Kapitels geben wir Ihnen noch ein paar Hinweise zu ATAM.

Laufzeitanalyse

In Laufzeitanalysen untersuchen Sie das System während es abläuft, Sie versuchen Probleme („Übeltäter“) sozusagen *in flagranti* zu ertappen. Damit finden Sie beispielsweise Performance-Engpässe, versteckte Abhängigkeiten, Speicherlecks, Deadlocks und Mängel. Laufzeitanalyse untersucht ganz allgemein die Ressourcennutzung des Systems, neben Prozessoren auch Speicher- oder Ein-/Ausgaberesourcen.

Beispiele typischer Probleme

- Eine bestimmte Operation im System benötigt ungebührlich lange, ungebührlich viel CPU-Zeit oder auch übermäßig viel Speicher für ihre Ausführung („Performance- oder Speicher-Bottleneck“).
- Eine Komponente des Systems benötigt zur Laufzeit Dienste oder Daten einer anderen Komponente, die sie laut statischer Codeanalyse aber gar nicht verwenden sollte („unbekannte Abhängigkeiten“).
- Das System stürzt bei bestimmten Eingaben ab („Instabilität“).

¹ *Architecture Tradeoff Analysis Method*, entwickelt vom Software-Engineering Institute. Siehe [ATAM].

- Das System führt Operationen mehrfach aus.
- Das System erzeugt oder konsumiert mehr Daten als vorgesehen.
- Das System führt zur Erfüllung bestimmter Aufgaben unnötige Operationen aus.
- Operationen blockieren sich gegenseitig („Deadlocks“).
- Operationen laufen sequenziell, obwohl sie parallel ablaufen sollten.
- Bestimmte Features werden von Nutzer*innen zu häufig oder gar nicht verwendet.

Methodik

Für alle Laufzeitanalysen geben wir Ihnen einen grundlegenden Rat: Vor jeder dynamischen Analyse klären Sie Ihre konkrete Erwartungshaltung. Reflektieren Sie, was gemäß Ihrer Erwartung *geschehen sollte*, wie beispielsweise die relativen Zeit- oder Speicherbedarfe bestimmter Systemteile aussehen sollten. Am besten klären Sie diese Erwartungshaltung mit dem Entwicklungsteam der betroffenen Komponenten.

- Die einfachste und jedem Entwicklungsteam geläufige Möglichkeit heißt **Debugger**, mit dessen Hilfe wir einem Stück Software zur Laufzeit fast auf atomarer Ebene bei der Arbeit zusehen können. Mit einem Debugger überprüfen Sie sehr feingranular Ihre Annahmen über spezifische Abläufe. Ja, theoretisch könnten Sie das auch, indem Sie den Quellcode lesen, aber manche Abhängigkeiten werden nun einmal erst zur Laufzeit aufgebaut. Oftmals ist es schlicht bequemer, den Debugger zu benutzen: Und im Gegensatz zu den Vermutungen beim Code-Lesen sagt der Debugger die Wahrheit. :-)
- Eine ebenfalls sehr verbreitete Art der Laufzeitanalyse ist **Profiling**: Dabei misst ein Werkzeug (der *Profiler*), welche Teile des Systems wieviel Zeit zur Ausführung benötigen. Je nach Werkzeug und Einstellung wird das bis zur Ebene einzelner Codezeilen gemessen. Profiling findet üblicherweise in Entwicklungs- oder Testumgebungen statt. Vorsicht: Solche detaillierten Vermessungen benötigen ihrerseits (teilweise signifikante) Zeit.
- Das Pendant zum Profiling im echten Betrieb heißt **Monitoring**.
- Untersuchen Sie die **Logfiles** des Systems und optimieren Sie das Logging des Systems daraufhin, Ihnen Antworten auf konkrete Fragen zu liefern. Letzteres

ist eine invasive Operation, d. h. Sie müssen das System dafür ändern. Für eine Logfile-Analyse sollten Sie (siehe oben) vorab klären, welche Einträge oder Nachrichten Sie erwarten.

- Sie können manche Details der Nutzung des Systems mit **Usage-Analytics** beobachten. Hierfür benötigen Sie die Einwilligung der Betroffenen.

Verweise

- Manche Probleme mit externen Schnittstellen (siehe Kontextanalyse) finden Sie nur durch Laufzeitanalysen.
- Gerade in hochgradig verteilten Systemen mit vielen Microservices stellt Monitoring eine große technische Herausforderung dar, für die viele Werkzeughersteller Abhilfe versprechen. Herstellerneutrale Vergleiche sind selten, einige methodische Hinweise finden Sie in unserem Blog².
- Usage-Analytics kann möglicherweise sensible oder personenbezogene Informationen offenlegen. Sie müssen dabei auf die notwendige Vertraulichkeit und Einhaltung von DSGVO/GDPR³-Regeln achten.

Analyse von Änderbarkeit und Erweiterbarkeit

Änderbarkeit und Erweiterbarkeit gehören zu den *inneren Qualitäten*⁴, für deren Analyse Sie den inneren Aufbau wie Komponenten, Code, technische Konzepte oder auch Laufzeitkonfiguration betrachten müssen.

Beispiele typischer Probleme

- Mehrfach lag der Fokus unserer Reviews auf schlechter *Time-to-Market*⁵ von Systemen, also möglichem Zeitverzug bei der Umsetzung neuer Features.

²<https://www.innoq.com/de/articles/2016/10/microservices-deployment-und-monitoring/>

³<https://dsgvo-gesetz.de/>

⁴Im Gegensatz zu den *externen Qualitäten* wie Performance, Benutzbarkeit, Robustheit etc.

⁵Aus Wikipedia: Unter dem Begriff „time to market“ (TTM, englisch für etwa Vorlaufzeit, Produkt einführungszeit) versteht man die Dauer von der Produktentwicklung bis zur Platzierung

- Ähnlich sieht es mit übermäßig hohen Aufwänden selbst für kleine Änderungen am System aus, also hohen Personalkosten.

Methodik

Änderbarkeit und Erweiterbarkeit hängen von verschiedenen Faktoren ab:

- Komplexität des Systems, u. a. durch folgende Aspekte mitbestimmt:
 - Kopplungsgrad im System auf verschiedenen Abstraktionsebenen. Hohe Kopplung bedeutet oftmals, dass für Änderungen viele einzelne Bestandteile modifiziert werden müssen. Dadurch steigen sowohl Aufwand wie auch Änderungsrisiken.
 - Komplexität der Implementierung, d. h. Verständlichkeit und Konsistenz des Quellcodes.
 - Kohäsion innerhalb von Implementierungsteilen (Subsystemen, Bausteinen).
 - Konsistenz, d. h. sind wiederkehrende Aufgaben im System gleichartig (*konsistent*) gelöst?
 - Größe des Systems oder der jeweils für eine Änderung relevanten Einheiten.
- Testbarkeit und das Vorhandensein automatisierter Tests
- Wissen über Architektur und Implementierung innerhalb des Entwicklungsteams, u. a. bestimmt durch *vergemeinschaftetes* Wissen oder Dokumentation
- Prozesskomplexität, beispielsweise leichtgewichtige versus schwergewichtige Entwicklungsprozesse, mehr oder weniger notwendige Formalismen

Kopplung prüfen: Aufgrund der unterschiedlichen *Arten von Kopplung* müssen Sie zu deren Messung etwas tiefer in die Werkzeugkiste greifen:

- Die einfachen Arten von Kopplung finden Sie durch statische Analyse von Quellcode. Im Abschnitt über statische Analyse zeigen wir dazu einige Möglichkeiten und Werkzeugkategorien auf.
- Kopplung über gemeinsame Datenbanken finden Sie in der Analyse der Anwendungsdaten.

des Produkts am Markt. In dieser Zeit entstehen für das Produkt Kosten, es erwirtschaftet aber keinen Umsatz.

- Andere Arten von Kopplung (temporale Kopplung, Kopplung über Hardware, implizite Kopplung über gemeinsame Schnittstellen) finden Sie durch eine Architekturanalyse.

Komplexität von Code prüfen: Komplexität von Code eignet sich beispielsweise zur Prognose von Fehlerwahrscheinlichkeiten: In komplexerem Code stecken wahrscheinlich mehr Fehler als in einfacherem, und Entwicklungsteams werden bei Änderungen in komplexem Code (versehentlich) mehr Fehler neu *einbauen* als in einfachem Code. Diese Zusammenhänge sind über Studien intensiv nachgewiesen, siehe etwa [4]. Die Komplexität von Quellcode können Sie mit unterschiedlichen Metriken bewerten:

- Zyklomatische Komplexität (auch bekannt als McCabe-Metrik), d.h. die Anzahl linear unabhängiger Pfade durch ein gegebenes Stück Quellcode. Intuitiv steigt dieses Maß der Komplexität mit jeder Verzweigungsanweisung um zwei Punkte.
- Größe in Lines-of-Code. Kann sich auf einzelne Funktionen oder auf größere Einheiten beziehen.
- Sonstige Programmierkonstrukte, etwa: Anzahl Parameter von Funktions-/Methodensignaturen, Typkomplexität dieser Parameter, Rekursion, parallele Verarbeitung mit Synchronisation, Mischung verschiedener Programmierparadigmen etc.

Einsatz geeigneter Technologien prüfen: Die Wahl von Basistechnologien und -frameworks kann Änderbarkeit von Systemen deutlich beeinflussen. Beispiel: In frühen Architektur- und Programmierparadigmen von Enterprise-Java mussten Entwicklungsteams neben komplizierten Programmiervorgaben auch noch eine manuelle Konfiguration ihrer Systeme in XML-basierten Deployment-Deskriptoren vornehmen. Das war sowohl aufwändig als auch fehleranfällig. Vereinfachte Modelle wie Spring-Boot reduzieren den *cognitive load* für Entwicklungsteams und führen daher in vielen Fällen zu leichter änderbaren Systemen.

Überprüfen Sie daher die eingesetzten Technologien und Frameworks, ob notwendige oder häufig wiederkehrende Änderungen dadurch erleichtert oder erschwert werden.

Verweise

- Komplexität von Quellcode finden Sie durch Codeanalyse, beispielsweise durch Codemetriken.
- Organisatorische Maßnahmen im Kontext von DevOps (etwa: *Continuous Integration*, Build- und Testautomatisierung, *trunk-based Development*) führen erfahrungsgemäß mittelfristig zu deutlich verbesserter Änderbarkeit von Systemen.

Security-Analyse

Hier identifizieren Sie Probleme oder Risiken im Umfeld von Datensicherheit, Datenschutz, Vertraulichkeit und Betriebsstabilität/-zuverlässigkeit.

Beispiele typischer Probleme

Unbefugte

- lesen oder manipulieren personenbezogene oder anderweitig sensible Daten.
- ersetzen die reguläre Anwendung (z. B. Website) durch eine gefälschte.
- verhindern den regulären Zugang zum System, z. B. indem sie das System abschalten.
- stehlen oder vertauschen Passwörter, so dass sie auf die Daten fremder Person zugreifen können.

Das (sicherheitskritische) System

- überträgt vertrauliche Daten über einfach zugängliche (*unverschlüsselte*) Kanäle.
- implementiert wichtige Security-Funktionen selbst, anstatt etablierte (Open-Source) Bibliotheken zu verwenden.
- wird unverschlüsselt gesichert, und die Backups lagern in leicht zugänglichen Räumen.

Methodik

Gehen Sie als Reviewer im Bereich IT-Sicherheit immer davon aus, dass mögliche Angreifer mehr Erfahrung, mehr Wissen, mehr Zeit, mehr Geld, mehr Motivation und vor allem bessere Ideen haben als Sie selbst bzw. das Entwicklungsteam des Systems. Im Bereich *Security* kämpfen Sie mit (potentiell) wesentlich stärkeren Gegnern, die sich dazu noch unfair verhalten.

Wir raten Ihnen dringend, bei sicherheitskritischen Systemen oder Anforderungen, entsprechende Expert*innen zu Rate zu ziehen!

- Gibt es eine explizite Dokumentation der Angriffsvektoren gegen das System?
- Setzt das System und die zugehörige Organisation die Anforderungen des IT-Grundschutz⁶/ISO-27001 um?
- Gibt es neben technischen auch organisatorische Sicherheitsmaßnahmen, z. B. Zugangskontrolle, Einbruchs- und Gebäudesicherheit?
- Basieren sämtliche (!) Verfahren zur Authentisierung, Verschlüsselung, Hashing oder Ähnlichem auf etablierten und geprüften Standardimplementierungen von zeitgemäßen Lösungen? Anders gesagt: Kryptographische Algorithmen (Verschlüsselung, Hashing) selbst zu implementieren birgt in jedem Fall ein hohes Risiko!
- Falls das System öffentlich (z. B. per Web oder Mobile-App) zugänglich ist: Finden Penetrationstests in regelmäßigen Abständen statt?
- Enthalten Logfiles vertrauliche Daten?

Verweise

[5] enthält eine umfangreiche Sammlung von *good practices* im Bereich IT-Sicherheit.

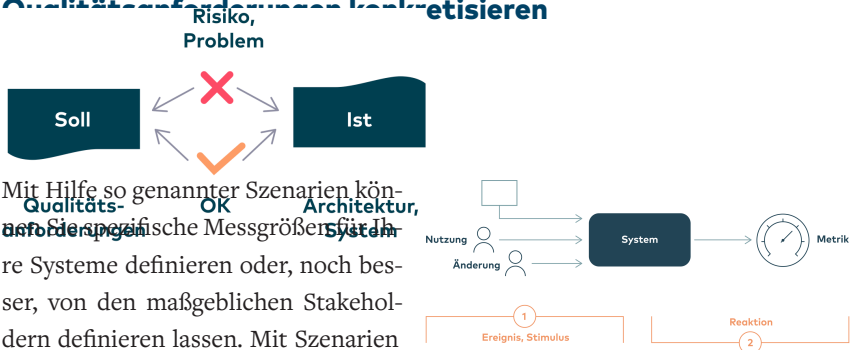
⁶https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzKompodium/itgrundschutzKompodium_node.htm

Qualitative Architekturanalyse und -bewertung (ATAM)

Abbildung 6.2: Qualitative Analyse - schematisch

Qualitative Architekturanalyse vergleicht den Soll-Zustand mit dem Ist-Zustand und leitet daraus Probleme oder Risiken ab. Als wesentliche Grundlage dafür benötigen Sie einen detaillierten und möglichst konkreten Katalog an Qualitätsanforderungen. Als Vorbild könnten Sie die bekannten Produktbewertungen der „Stiftung Warentest“ heranziehen, die detaillierte Kriterienkataloge (=Anforderungen) als Basis ihrer Untersuchungen verwenden. Beispielsweise treffen Sie Aussagen darüber, in welchem Maß das jeweilige System den Anforderungen an Wartbarkeit, Flexibilität, Performance oder Sicherheit genügen muss. Dabei liefert eine qualitative Analyse kein absolutes Maß, d. h. sie analysiert nicht die „Güte-über-alles“, sondern lediglich die Zielerreichung im Hinblick auf spezifische Kriterien („Qualitätsanforderungen“). Qualitative Analyse hilft, Risiken zu identifizieren, die sich möglicherweise aus problematischen Entwurfsentscheidungen ergeben.

Qualitätsanforderungen konkretisieren



Metrik. In der Abbildung erkennen Sie diesen schematischen Aufbau. Formulieren Sie Szenarien als Sätze, die immer den „Auslöser“ sowie die „messbare Reaktion des Systems“ enthalten.

Das klingt zu abstrakt? Einige Beispiele helfen, diesen Nebel zu lichten:

Auslöser (Ereignis)	Metrik (Reaktion)
Nutzer fordert den Monatsreport der Kostenstelle xyz an ...	das System erzeugt den Report innerhalb von 3 Sekunden.
Produktsuche mit Standardkriterien (Artikelbezeichnung/-nummer) ...	dauert maximal 500 ms bis zur Anzeige der ersten 5 Treffer.
Falls das externe System xyz mehr als 3 Sekunden lang nicht antwortet ...	meldet das System diesen Fehler innerhalb von 30 Sekunden an den Administrator-vom-Dienst.
Der Fachbereich fordert eine Änderung des xyz-Tarifs im System.	Diese Änderung kann innerhalb von 4h im System umgesetzt werden.

Mit diesen Szenarien für die Qualitätsanforderungen besitzen Sie nun einen konkreten Maßstab für die qualitative Analyse.

Architektur gegen Qualitätsanforderungen prüfen

Gemeinsam mit Architekten oder dem Entwicklungsteam untersuchen Sie nun, ob das System oder dessen Architektur die definierten Szenarien (= Qualitätsanforderungen) erfüllen kann. Lassen Sie sich dazu die zugehörigen Architektur-entscheidungen und -ansätze erläutern. Spielen Sie gemeinsam mit Beteiligten aus dem Entwicklungsteam diese Szenarien in Form von *Walkthroughs* durch, möglichst detailliert und feingranular. Finden Sie heraus, wie die Bausteine des Systems zur Erreichung dieses Szenarios zusammenspielen und welche Entwurfs-entscheidungen das jeweilige Szenario unterstützen.

Dabei können Ihnen auch die folgenden Fragen helfen:

- Welche Architekturentscheidungen wurden zur Erreichung dieses Szenarios getroffen?
- Welcher Architekturansatz unterstützt die Erreichung des Szenarios?
- Welche Kompromisse wurden durch diese Entscheidung eingegangen?
- Welche anderen Qualitätsmerkmale oder Architekturziele werden von dieser Entscheidung noch beeinflusst?
- Welche Analysen, Untersuchungen oder Prototypen stützen diese?
- Welche Risiken entstehen aus dieser Entscheidung oder aus diesem Ansatz?
- Welche Risiken gibt es für die Erreichung des jeweiligen Szenarios und den damit verbundenen Qualitätsanforderungen?

Damit können Sie unserer Erfahrung nach recht zuverlässig herausfinden, ob das betroffene Szenario mit der gegebenen Architektur *funktioniert*, oder ob es in dieser Hinsicht gravierende Risiken oder Defizite gibt.

7 Architekturanalyse

Bevor wir auf Details zu diesem wichtigen Aspekt von Reviews eingehen, lassen Sie uns kurz zusammenfassen, was wir unter dem Begriff „Softwarearchitektur“ verstehen und was wir demzufolge unter der Rubrik *Architektur* untersuchen möchten. Wenn wir von Architektur sprechen, meinen wir damit mehrere Ebenen:

- **Domänenarchitektur:** Die fachliche Modularisierung der Anwendungslandschaft und die Informationsflüsse zwischen den Anwendungen. Dies ist unabhängig von konkreten Technologien.
- **Makroarchitektur:** Vorgaben zu Themen, die weitestgehend unabhängig von der internen Struktur einzelner Systeme sind, sondern sicherstellen, dass eine Anwendungslandschaft entsteht, die sinnvollen Regeln folgt. Beispiele sind Kommunikationsprotokolle, UI-Integration, Mechanismen zur Datenreplikation, Monitoring/Logging, Betriebsschnittstellen, etc.
- **Mikroarchitektur:** (Team-) Lokale Entscheidungen über die Interna eines Systems wie interne Modularisierung, Schichtenmodelle und Architekturmuster, konkrete Lösungskonzepte, Frameworks und Bibliotheken, Ablaufumgebungen und weitere, technische Aspekte.

Im Folgenden haben wir einige bedenkenswerte Punkte für die unterschiedlichen Architekturebenen zusammengetragen.

Domänenarchitektur

Beispiele typischer Probleme

Die Erfahrung zeigt, dass eine Architektur, die sich nicht an den fachlichen Strukturen orientiert, zu komplexen Datenmodellen und Flaschenhälsen in mehreren Dimensionen (Verzögerung, inhaltliche Einigung, Koordinierungsaufwand etc.) führt. Systeme werden dadurch unnötig komplex, fehleranfällig, schlecht wart- und erweiterbar und können so weder Entwickler noch Fachbereiche zufriedenstellen.

Methodik

Um zu verstehen ob die Ursache für unsere Probleme in der Domänenarchitektur liegt, müssen wir die fachlichen Problembereiche verstehen und prüfen, wie diese durch jeweilige Anwendungen gelöst werden. Eine gute Zusammenarbeit zwischen Entwicklung und dem Fachbereich ist hier entscheidend, daher sollte dieser Teil der Architektur gemeinsam mit dem Fachbereich entwickelt werden, um sicherzustellen, dass alle Beteiligten ein gemeinsames Verständnis des Problems und der Lösung besitzen. Wir betrachten daher auf dieser Ebene ausschließlich die Fachlichkeit, Technische Details spielen hier keine Rolle. Am Ende sollten wir entscheiden können welche Änderungen notwendig sind und wo sich Investitionen in eine bessere Domänenarchitektur lohnen.

Eine gute Möglichkeit, diese Probleme zu analysieren, bieten Methodiken aus dem Bereich **Domain Driven Design**. Beispielsweise mittels Event Storming, einem Workshopformat, um Domänen zu analysieren, zu strukturieren und zu visualisieren. Dies kann sowohl bei der Modellierung neuer Systeme eingesetzt werden, als auch als Reverse-Engineering-Werkzeug, um einen Überblick über die vorhandenen Abläufe in der Fachdomäne zu gewinnen.

Eine weitere Möglichkeit sind **Wardley Maps**. Diese ermöglichen es, die vorhandene IT-Landschaft mit ständigem Blick auf den Kundennutzen zu kartografieren. So entsteht ein Überblick etwa über mögliche Fehlinvestitionen, suboptimale Entwicklungspraktiken oder Know-How-Engpässe auf neutralem Boden. Daraus lassen sich fundierte Erkenntnisse für Make-or-Buy-Entscheidungen, Outsourcing oder der Reorganisationen von Teams ableiten.

Verweise

- Michael Plöd: Hands-on Domain-driven Design - by example[6]
- Eric Evans: Domain-Driven Design Referenz[7]
- Markus Harrer: Software-Modernisierung mit Wardley Maps[8]

Makroarchitektur

Beispiele typischer Probleme

Probleme können sich auf zwei Ebenen ergeben:

- Auf der Ebene der Vorgaben: Diese können zu detailliert oder zu lückenhaft sein oder auch einfach nicht zu den geforderten Qualitätseigenschaften passen.
- Auf der Ebene der konkreten Umsetzung in den Systemen: Hier kann es Mängel bei der korrekten Umsetzung oder auch bei der Art und Weise der Umsetzung geben, beispielsweise wenn sich Funktionalität immer wieder *kreuz und quer* in einem Softwaresystem wiederfinden lässt (die sog. *Cross-cutting Concerns*). Typische Vertreter dieser Gattung sind Persistenz, Authentifizierung, Replikation, Monitoring, Logging, Mandantenfähigkeit, Auditierung, Internationalisierung und so weiter. Hier stehen Sie vor der Aufgabe, zu bewerten, ob die dafür eingesetzten Technologien im Softwaresystem auch richtig umgesetzt wurden.

Methodik

Mit qualitativen Bewertungsverfahren, wie mit der zuvor vorgestellten Architecture Tradeoff Analysis Method (ATAM), können Sie bereits bewerten, ob die passenden Querschnittskonzepte zu den geforderten Qualitätseigenschaften vorhanden sind.

Führen Sie einen *Code-Walkthrough* durch, um die Umsetzung direkt im Code zu sehen. Suchen Sie sich dafür einen fachlich besonders interessanten und/oder relevanten Use Case aus. Gehen Sie dann schrittweise die Abarbeitung im Code durch (statisch oder grob per Debugger). Sie werden dadurch an diversen Stellen über Cross-cutting Concerns stolpern, deren Umsetzung Sie dann genauer unter die Lupe nehmen können. Für die Bewertung können Sie als Input einerseits Ihre Vorgaben und Programmierrichtlinien im Unternehmen verwenden, um die umgesetzten querschnittlichen Konzepte Stück für Stück zu prüfen. Andererseits

könnten Sie auch mit Mini-Checklisten bzw. den derzeit vorhandenen Best Practices der Software-Community prüfen.

Verweise

- Till Schulte-Coerne: Optionen der Frontend-Integration¹
- Beispiel für Logvorgaben: Elastic Common Schema²
- Konkrete Verbesserungsvorschläge: aim42, Abschnitt „Improve“³
- Checklisten für Microservices in: Susan J. Fowler: Production-Ready Microservices[9]

Mikroarchitektur

Beispiele typischer Probleme

Eines der typischen Probleme auf der Mikroarchitekturebene sind schlechte Modularisierung und undurchsichtige Abhängigkeiten. Das aktive Management dieser Art von Komplexität ist aber sehr wichtig in größeren Systemen (100.000+ Codezeilen). Es ermöglicht Entwickler*innen, die richtigen Stellen für Codeänderungen schnell zu finden⁴ und bei vorgenommenen Änderungen keine bösen Überraschungen zu erleben⁵.

Methodik

Einen Ansatz für das Review dieser beiden Eigenschaften beschreibt Carola Lienthal[10] in ihrem Buch „Langlebige Software-Architekturen“. Hier wird die Struktur der Software auf ihre konzeptionelle Integrität mit Hilfe von Architekturmanagementwerkzeugen nach Folgendem (hier vereinfacht dargestellten) Verfahren geprüft:

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/>

⁴hier helfen hierarchisch sauber strukturierte Module

⁵hier helfen klare Abhängigkeiten zwischen Modulen

1. Finde Gruppierungseigenschaften (z. B. aus der Fachlichkeit, technischen Schichtung oder einer Mustersprache), zu denen sich die im Quellcode enthaltenen Softwareelemente (z. B. Klassen) zuordnen lassen.
2. Ordne die Gruppen nach ihren vorgesehenen Funktionen im Softwaresystem hierarchisch an.
3. Bewerte die Abhängigkeitsbeziehungen zwischen den entstandenen Gruppierungsstrukturen.

Im Idealfall gibt es in der Softwarearchitektur bereits passende Strukturen, die sich sehr stark mit der Gruppierung aus der Analyse decken. Wenn hier möglichst viel Quellcode den Gruppierungen zugeordnet werden kann, so kann dem System eine hohe konzeptionelle Integrität zugesprochen werden.

Rein Code-zentrierte Reviews sind keine Reviews!

Mit der Analyse der Softwarearchitektur rein auf Grundlage des vorhandenen Quellcodes ist es leider nicht getan. Es ist ein Irrglaube, dass alle Informationen im Code stehen – und unserer Erfahrungen nach gilt sogar: Oft sind dies noch nicht einmal die wichtigsten.

Das klingt zunächst alles andere als intuitiv. Wie sollte für die Architektur eines Softwaresystems etwas wichtiger sein als der eigentliche Quellcode der Software? Dafür haben wir im Folgenden einige Gründe aufgelistet:

- Möglicherweise haben Sie eine hervorragend strukturierte, nach allen Regeln der Kunst durchgeführte Implementierung einer äußerst schlechten Idee vor sich, z. B. weil es für den gleichen Einsatzzweck bessere, fertige Komponenten gibt.
- Vielleicht würde der ein oder andere problematische oder vielleicht sogar besonders elegante Teil der Software gar nicht benötigt, wenn man eine technische Anforderung hinterfragen würde.
- Manchmal befindet sich ein großer Teil der wichtigen Information nicht in „klassischem“ Quellcode, sondern in Konfigurationsdateien oder „Glue-Code“, der auf den ersten Blick gar nicht ins Auge fällt.

- Vielleicht ist die Laufzeitarchitektur des Systems äußerst elegant, führt aber zu enormen Herausforderungen im Entwicklungsprozess – oder andersherum.
- Unter Umständen spielt die Aufteilung in verschiedene Teilkomponenten in einem verteilten System eine sehr viel größere Rolle als die Implementierung der einzelnen Systeme.
- Vielleicht entstehen Probleme durch eine suboptimale Aufteilung zwischen Hardware- und Softwarekomponenten (besonders bei Embedded-Systemen) oder durch eine verbesserungswürdige Verwendung von Standardsoftware.

Die genannten Fälle sind alle sehr unterschiedlich, und darin liegt der vielleicht spannendste Aspekt von Reviews: Sie müssen sich in das System hineindenken und es aus sehr vielen unterschiedlichen Perspektiven hinterfragen. Es gibt daher kein Standardrezept und auch nicht das eine, ultimative Werkzeug, das alle Probleme sichtbar macht.

8 Codeanalyse

Quellcode gilt mit Berechtigung als ein (oder sogar *das*) zentrale Artefakt Ihres Systems. Codeanalyse fördert also mit ziemlicher Sicherheit einige Probleme oder Risiken Ihres Systems zutage.

In diesem Kapitel stellen wir Ihnen methodisches Handwerkszeug vor, das Ihnen solche Analysen ermöglicht. Auch wenn Sie heute schon Code-Reviews oder statische Analysen durchführen – lesen Sie trotzdem weiter: Wir halten ein paar Überraschungen bereit.

Eine Anmerkung am Rande: Falls Ihr Entwicklungsteam Unit-Tests einsetzt, dann betreiben Sie bereits aktiv Codeanalyse, denn diese Unit-Tests *analysieren* das Verhalten Ihres Systems auf detaillierter Ebene.

Beispiele typischer Probleme

Mittels Codeanalyse finden Sie beispielsweise folgende Kategorien von Problemen:

- Schwer verständliche Teile, verursacht durch übermäßige Größe, Komplexität oder Abhängigkeiten sowie Abweichung von vereinbarten Stilvorgaben oder Code-Konventionen
- Riskante Stellen im Code, etwa aufgrund von übermäßig hoher Änderungsrate, Fehlerrate, oder Abweichung von Architekturvorgaben
- Wenig performante oder zu ressourcenintensive Implementierung
- Teile, für die keine automatisierten Tests existieren

Methodik

Von den vielen möglichen Herangehensweisen der Codeanalyse stellen wir Ihnen drei unserer Favoriten vor:

1. Statische Codeanalysen auf Basis von Werkzeugen wie SonarQube, TeamScale, ReSharper oder Checkstyle.
2. Hotspot-Analyse, bei der Sie auf Basis Ihrer Versionsverwaltung die am häufigsten geänderten Stellen im Code identifizieren können. Unserer Einschätzung nach eine sehr hilfreiche Methode, in der Praxis aktuell erschreckend wenig verbreitet.
3. „Manuelle“ Code-Reviews, ebenfalls mit Unterstützung passender Werkzeuge. Hierbei betrachtet eine zweite oder auch dritte Person den Code hinsichtlich Best-Practices sowie möglicher Fehler oder Probleme.

Statische Analyse

Welche Metriken?

Von den vielen Dutzenden *theoretisch* möglicher Kennzahlen von Quellcode möchten wir Ihnen eine kleine, aber praxistaugliche Auswahl empfehlen:

- **Komplexität:** Wir betrachten Komplexität (genauer: unnötige Kompliziertheit) als ein zentrales Problem in der Informatik. Schauen Sie unbedingt in den Abschnitt „Hotspot-Analyse“, dort kombinieren wir Komplexität mit Änderungsrate, um besonders riskante oder gefährdete Stellen im Code zu identifizieren.
- **Kopplung** (Abhängigkeiten, *dependencies*): Zu viele Abhängigkeiten wirken sich stark negativ auf Verständlichkeit und Änderbarkeit aus, daneben erhöhen sie das Fehlerrisiko.
- Einhaltung von **Kodier-Richtlinien** (*Coding Guidelines* oder *Style Guides*): Das klingt vielleicht wie Formalismus, aber einheitlich geschriebener Code erhöht die Verständlichkeit signifikant – und er lässt sich wesentlich leichter reviewen.

In Quellcode zeigt sich Komplexität in verschiedenen Ausprägungen:

- Zyklomatische Komplexität (McCabe-Metrik), die Anzahl möglicher Pfade durch ein gegebenes Stück Code. Diese Metrik hängt von der gewählten Programmiersprache ab.
- Kognitive Komplexität, beispielsweise Einrückungs- oder Verschachtelungstiefe oder Anzahl Funktions- oder Methodenparameter
- Größe von Elementen (beispielsweise Länge von Methoden oder Funktionen, Anzahl von Methoden pro Klasse, Anzahl Klassen pro Package)

Vorsicht: Zahlen

Mit Hilfe von statischer Codeanalyse und passenden Werkzeugen können Sie viele Arten von Kennzahlen erheben, die bestimmte Eigenschaften Ihres Codes quantifizieren. Berücksichtigen Sie, unabhängig von Ihren gewählten Metriken, folgende wichtige Aspekte:

- **Relevanz:** Entscheiden Sie sich für maximal 3-5 Arten von Kennzahlen, die für Ihr System Relevanz besitzen. Wenn Sie viel mehr Zahlen ermitteln, sehen Sie früher oder später *den Wald vor lauter Bäumen* nicht mehr.
- **Trend:** Beobachten Sie den Trend, also die Veränderung von Kennzahlen über die Zeit, ansonsten haben Sie immer nur eine (aus unserer Sicht weniger aussagekräftige) Momentaufnahme.
- **Relativität:** Ihr Management verwendet (andere) Zahlen zur Steuerung des Unternehmens und besitzt auf Zahlen oft eine eher *absolute* Sichtweise. Für Software-Reviews nutzen wir Zahlen lediglich als Indikator, aber praktisch niemals als absolute Grenzwerte. Geben Sie Ihrem Management deswegen nur Ihre Schlussfolgerungen aus den Metriken weiter, nicht jedoch die Zahlen selbst.
- **Bewerten Sie Code, nicht Menschen:** Statische Codeanalyse bewertet Ihren Code objektiv, aber nicht die Menschen, die den Code geschrieben haben. Sehen Sie davon ab, Menschen an Hand dieser Zahlen zu beurteilen.

Probleme mit Hilfe von Metriken finden

- Niemand wünscht sich eine verlangsamte Time-to-Market für neue Features. Ursache davon sind oft enge Kopplungen zwischen Bausteinen, die Sie mit statischen Analysen aufdecken.
- Zeigen Metriken an verschiedenen Stellen hohe Komplexität auf, weisen diese direkt auf ein erhöhtes Fehlerrisiko hin, sowohl in der Implementierung selbst als auch bei zukünftigen Änderungen.
- Das Entwicklungsteam verwendet keine Metriken zur aktiven Steuerung der Entwicklung: Dann haben Sie ein großes Risiko gefunden, zu dem es methodisch relativ einfache Abhilfen gibt, beispielsweise unsere Null-Warnungen-Richtlinie (siehe unten).

Null-Warnungen-Richtlinie und die Pfadfinder-Regel

Null-Warnungen-Richtlinie: Wir empfehlen eine Null-Toleranz-Konvention für (neu geschriebenen) Code: Neuer Code muss **frei** von Warnungen statischer Analysewerkzeuge sein und darf gegen **keine** der für das System etablierten Coding-Guidelines verstoßen. Lassen Sie bei Verletzung der Regeln den Build Ihres Systems fehlschlagen!

Dadurch halten Sie die handwerkliche Qualität Ihres Codes hoch. Unsere Erfahrungen aus vielen Entwicklungsprojekten zeigen, dass diese Konvention peu à peu zu schnellerer Entwicklung (!) und geringerer Fehlerrate führt.

Führen Sie diese Richtlinie erstmal nur für neuen Code ein. Schließen Sie bestehenden Code explizit von den passenden statischen Analysen aus. Wenn Sie dann eine Änderung am bestehenden Code durchführen müssen, nehmen Sie zuerst nur dieses (hoffentlich kleine) Stück in Ihre Analyse auf und arbeiten hier nach der Pfadfinder-Regel: Hinterlassen Sie das (alte) Stück Code sauberer, als Sie es vorgefunden haben.

Hotspot-Analyse

Hotspot: Baustein im System, der häufig geändert wird und gleichzeitig hohe innere Komplexität besitzt. (nach: [4])

Die innere Komplexität von Bausteinen eignet sich hervorragend als Instrument, um die Wahrscheinlichkeit von Fehlern vorherzusagen. Viele Fehler treten statistisch dort auf, wo Code besonders komplex oder tief verschachtelt ist. Daneben dauern Änderungen an solchem Code auch besonders lange.

Denken Sie diesen Gedanken etwas weiter: Wenn ein Entwicklungsteam an *komplexem* Code arbeiten muss, steigt die Wahrscheinlichkeit von Fehlern gegenüber der Arbeit an *einfachem* Code deutlich an. Sie kennen mit Sicherheit das KISS-Prinzip¹, die goldene Regel der Softwareentwicklung. Einfache Lösungen gewinnen praktisch immer gegen übermäßig komplizierten.

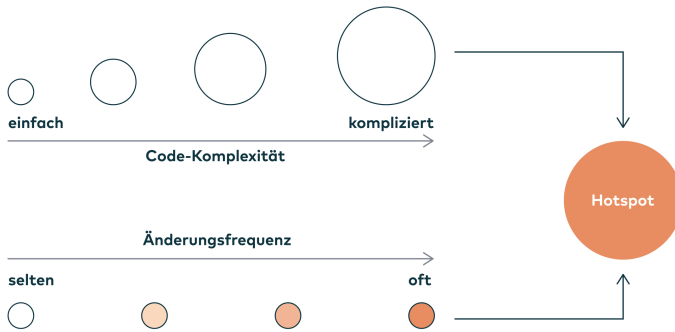


Abbildung 8.1: Legende für die Hotspot Analyse

Sortieren Sie die Bausteine (d.h. den Quellcode) Ihres Systems gemäß der Abbildung ein: Finden Sie die Stellen, die häufig geändert werden und gleichzeitig sehr komplex sind. Zum Glück können wir uns bei dieser Methodik und passenden Werkzeugen dazu aus den großartigen Büchern von Adam Tornhill „Software Design X-Rays“[11] sowie „Your Code as a Crime Scene“[4] bedienen.

¹Keep It Stupid Simple, halte es so einfach wie möglich

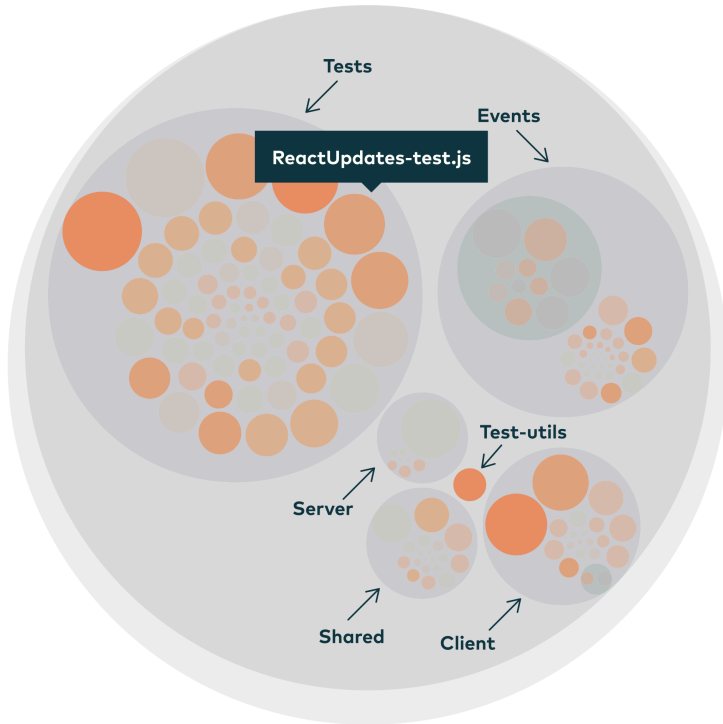


Abbildung 8.2: Hotspots in React.js (von Adam Tornhill aus [4])

Das oben stehende Diagramm zeigt diese Hotspot-Analyse anhand des Open-Source-Systems React². In den interaktiven Original-Diagrammen können Sie in die Hotspots hineinzoomen und sich damit einen guten Einblick verschaffen, welche Bausteine des Systems sowohl komplex als auch volatil sind.

²<https://reactjs.org/>

Manuelles Code-Review

Mit Hilfe von automatischen Analysen können Sie zwar einiges an Verbesserungspotential in Ihrer Software aufdecken. Dennoch können rein automatische Prozesse das „manuelle Code-Review“ nicht ersetzen, wohl aber unterstützen. Manuell bedeutet in diesem Fall, dass mindestens eine weitere Person den Quellcode liest (4-Augen-Prinzip).

Im Rahmen eines gesamthaften Reviews sollten Sie aus wesentlichen Gründen auch *manuell* in Quellcode schauen:

- **Schutz vor bösem Code:** Finden Sie bösartigen Code, der das System bewusst zum Absturz bringt, Nutzer*innen ausspioniert oder sonstige Sicherheitslücken enthält
- **Konformität zu Architektur und Stil:** Prüfen Sie, ob Code zu den geltenden Konventionen, Prinzipien oder Konzepten der Architektur passt und der Code in dem von Ihnen gewünschten Stil geschrieben wurde. Hierbei geht es **nicht** darum, die Einhaltung von Coding Guidelines zu überprüfen – das übernimmt in großen Teilen (hoffentlich) ihr Werkzeug zur statischen Codeanalyse.
- **Verständlichkeit:** Sauber benannte Variablen, sprechende Methoden und Testfälle sowie die korrekte Anwendung grundlegender Prinzipien – all das sind Dinge, die Sie bezüglich Lesbarkeit und guter Verständlichkeit im Code-Review prüfen sollten.

Code-Reviews richtig machen – aber wie?

Sicherlich sind Sie längst überzeugt, dass Sie Code-Reviews durchführen sollten. Und nun stellen Sie sich die Frage, wie genau das eigentlich funktionieren soll.

Werkzeuge

Natürlich gibt es auch für Code-Reviews Werkzeuge (wie etwa GitLab oder Up-source), welche die Arbeit erleichtern. Sogenannte Merge-Requests (oder Pull-Requests) ermöglichen es, in den Werkzeugen Codestellen zu kommentieren. Auf diese Weise bleiben die Anmerkungen eines Reviews dokumentiert und später einsehbar. Wenn alle Beteiligten den Änderungen zustimmen, wandern diese in den „master“.

Unsere Empfehlung für den Fall eines einmaligen oder initialen Code-Reviews: Legen Sie einen eigenen Review-Branch an und erstellen Sie einen zugehörigen Merge-Request. Alle Dinge, die Sie beim Durchforsten der Code-Basis finden, merken Sie in Kommentaren im Review-Werkzeug an den jeweiligen Dateien an. Das Entwicklungsteam hat so die Gelegenheit, die schlimmsten Dinge gleich in diesem Branch zu korrigieren. Der Vorteil daran: Sie sehen direkt die Änderungen und können diese ebenfalls reviewen bzw. diese akzeptieren – oder eben ablehnen.

Checklisten

Für einen kontrollierten Ablauf des Reviews verwenden Sie am besten eine Checkliste, die Sie vorab mit den wichtigsten Punkten befüllen. Auf diese Weise vergessen Sie keinen der Punkte, selbst wenn Sie tiefer in den Code tauchen oder intensive Diskussionen mit dem Entwicklungsteam führen. Wir haben für Sie eine Reihe an wichtigen Fragen zusammengefasst, welche Sie gerne als Grundlage für Ihr nächstes Review verwenden dürfen.

- Hält sich die Implementierung an die Architekturvorgaben?
- Gibt es mögliche Flaschenhälse hinsichtlich Performance?
- Existieren logische Fehler im Code?
- Gibt es Logging und Fehlerbehandlung?
- Drücken Methoden- und Variablennamen ihre Funktion entsprechend aus?
- Weisen Dateien eine immense Anzahl an Warnungen der statischen Analyse auf? Es rentiert sich meistens, diese Klassen einer tiefergehenden Untersuchung zu unterziehen.
- Gibt es einen geregelten Review-Prozess? Wenn ja, existieren Checklisten für Reviewer und Reviewee?

Verweise

- 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>
- Kognitive Komplexität: <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>

Exkurs: Datenanalysen in der Softwareentwicklung

Wussten Sie das schon? Sie leben in aufregenden Zeiten, denn Sie können die Schmerzpunkte in Ihrer Software für das Management oft mit Hilfe von Datenanalysen aufzeigen! Keine mulmigen Bauchgefühle mehr, sondern Zahlen, Daten, Fakten über die Probleme, die Sie wirklich behindern. Wenn wir die Hotspot-Analyse etwas abstrakter betrachten, sprechen wir davon, mittels Datenanalysen beliebige Daten aus Softwaresystemen auszuwerten:

- Statische Daten wie Quellcode, Spezifikationen oder Dokumentation
- Laufzeitdaten wie Performance-Daten, Nutzungsdaten oder Testergebnisberichte
- Chronologische Daten wie Log-Dateien oder Daten aus Versionskontrollsystemen
- Community-Daten wie Open-Source-Projekte, zugehörige Ticketsysteme oder Diskussionsforen

Jede Datenquelle ist für sich bereits wertvoll, um situationsspezifischen Analysen für ein stark abgegrenztes Problem vorzunehmen. Beispielsweise, um ungewollte Codeänderungsabhängigkeiten zwischen Teams zu identifizieren, Auswirkungen unperformanter Drittbibliotheken zu lokalisieren oder schwächelnde Open-Source-Communities frühzeitig zu erkennen.

Wirklich interessant wird es, wenn Sie verschiedene Datenquellen zusammenführen, um Erkenntnisse mittels unterschiedlicher Blickwinkel zu erarbeiten, wie etwa „in welchem fachlichen Bereich wurden in letzter Zeit überproportional viele Änderungen an großen Methoden durchgeführt, die aber in Produktion gar nicht mehr ausgeführt werden“?

Diagnostizieren Sie mit diesem Ansatz richtige Schlamassel, indem Sie deren Problemursachen datengestützt sachlich ergründen (und damit zwischenmenschliche Konflikte vermeiden). Bereiten Sie Ihre Ergebnisse mit passenden Visualisierungen für das Management verständlich auf – komplett nachvollziehbar von den Rohdaten bis hin zu den Erkenntnissen. Diese Art von Analysen ist heutzutage keine Herausforderung mehr: Dank Data Science, Big Data oder Deep Learning, lernen immer mehr Softwareentwickler*innen das notwendige Handwerkszeug im Umgang mit der Analyse von Daten. Sie wenden das gleiche Vorgehen und dieselben Werkzeuge an, nur eben auf Daten, die während der Erstellung oder des Betriebs der Software anfallen. Wenn Sie in dieses Thema einsteigen wollen, schnappen Sie sich das Buch von Tornhill[11] oder sehen Sie sich einen Talk von Markus Harrer³ an.

³<https://www.feststelltaste.de/dsmsd/>

9 Analyse von Anwendungsdaten

Worum geht es?

Die zuverlässige Speicherung und Bereitstellung von Daten gehören für die meisten Anwendungen zu den Kernaufgaben. Die **Daten einer Anwendung leben oft länger als der Quellcode**. Quellcode lässt sich vergleichsweise einfach erweitern, refaktorisieren oder auf eine neue Frameworkversion anpassen, Bugfixes lassen sich meist relativ einfach deployen. Fehlerhafte oder verlorene Daten lassen sich dagegen nur schwer und aufwendig reparieren. Änderungen an den Strukturen eines produktiven Datenbestands sind oftmals zeitintensiv und erfordern sorgfältige Planung. **Fehler in Datenmodellierung werden Sie entsprechend länger verfolgen**. Auch der Wechsel einer Datenbank ist teuer und zeitintensiv. Ein Wechsel von MySQL zu PostgreSQL ist aufwendig, weil alle datenbankspezifischen Funktionen gefunden und ersetzt werden müssen. Ein Wechsel von MongoDB auf MySQL noch aufwendiger, weil es einen kompletten Wechsel des Datenmodells und des Schemamanagements erfordert. Bei der Auswahl eines Datenmodells und eines Datenbanksystems sollte also entsprechend sorgfältig vorgegangen werden, um unliebsame Überraschungen zu vermeiden.

Im Rahmen Ihres Reviews ist es nun ihre Aufgabe, herauszufinden, ob die Auswahl der datenspeichernden Systeme, die Modellierung der Daten und Ihre Übermittlung, Transformation und Qualität den Erfordernissen der jeweiligen Anwendung entspricht.

Beispiele typischer Probleme

- Der fachliche Anwendungsfall benötigt stärkere Konsistenzgarantien als die gewählte NoSQL Datenbank bieten kann.
- Mangelhafte Performance mancher Datenzugriffe oder datenintensiver Operationen.

- Veraltete Datenmodelle, die den aktuellen fachlichen Bedürfnissen nicht mehr genügen.
- Fehlerhafte Dateninhalte aufgrund mangelnder Validierung oder unzureichender Modellierung.
- Übermäßig große oder komplexe Datenmodelle erschweren Weiterentwicklung des Systems.
- Veraltete Technologie für Datenspeicherung.
- Datenänderungen brauchen mehrere Tage bis sie in allen nachgelagerten Systemen ankommen.

Methodik

Verschaffen Sie sich einen Überblick über alle **datenspeichernden Systeme** sowie der dazu eingesetzten Software:

- Welche Arten von datenspeichernden Systemen werden eingesetzt (etwa: lokale oder Cloud-Datenbanken, Dateisysteme, Objektspeicher)
- Auf welchen Grundlagen basieren die Entscheidungen für diese Systeme?
- Sind die Versionen aktuell?
- Wieviel Speicherplatz benötigen diese Systeme?

Ermitteln sie deren **architekturelle Aufgabe** (z. B. Anwendungsdatenbank, Cache, Data Warehouse, Integration zwischen verschiedenen Teilen der Anwendung, Suche):

Gibt es

- ungewöhnlich viele Caching Systeme?
- Daten, die über verschiedene Systeme synchron gehalten werden müssen?
- Datenspeicher, die mehrere verschiedene Systeme zum Datenaustausch verwenden (*Integrationsdatenbanken*)? Ein Anti-Pattern.

Suchen Sie nach Problemen mit **Antwortzeit**, **Durchsatz** oder **Stabilität**:

Gibt es

- wiederkehrende Abfragen (*Queries*), die übermäßig lange laufen?
- Funktionen innerhalb des gesamten Systems, die besonders hohe Last in den Datenspeichern verursachen?
- reproduzierbare Abstürze oder Fehlverhalten aufgrund von Operationen in den Datenspeichern?

Verschaffen Sie sich einen Überblick über die **fachliche Struktur der Daten und deren Abbildung auf Datenbanken**:

- Welches grundlegende Modell wird verwendet: Relational? Dokumentenorientiert? Zeitreihe?
- Passen Fachlichkeit und Modell zusammen?
- Passen Modell und gewählter Datenspeicher zusammen?
- Gewährt die Datenbank die fachlich notwendigen Konsistenzbedingungen?
- Werden Transaktionen verwendet, wo fachlich geboten? Falls nicht, welche Alternativen sind implementiert?
- Gibt es ein festes Schema für die Daten? Wird dies durch die Datenbank oder die Anwendung umgesetzt? Wie erfolgen Schemaänderungen?
- Gibt es übermäßig große Einheiten, z. B. Tabellen mit mehr als hundert Spalten?
- Wirkt das Datenmodell übermäßig komplex?
- Gibt es eine Vielzahl von Anfragen, die größere Teile des Datenmodells traversieren?
- Gibt es durch das Datenmodell bedingte Performance-Engpässe?
- Gibt es realitätsnahe Testdaten?

Identifizieren Sie **Schnittstellen, Quelle und Ziel eingehender und ausgehender Datenflüsse**:

- Wie zugänglich sind Daten für nachgelagerte Systeme?
- Gibt es dokumentierte Schnittstellen?
- Gibt es Zugriffsbeschränkungen aufgrund von Performance-Engpässen?
- Liegen Daten in einem effizient abfragbaren bzw. durchsuchbaren Datenspeicher?
- Wie erfolgt die Übertragung dieser Daten?
- Liefern die Datenquellen eine hinreichende Qualität oder müssen empfangene Daten (manuell) nachbearbeitet werden?
- Wie lange dauert die Übermittlung? Nächtliche oder wöchentliche Batch Jobs?

Verweise

- Probleme mit Bezug zu Migrationen oder Backup/Restore können Sie mit Prozessanalyse oder Infrastrukturanalyse herausfinden
- Analysen von Datenstrukturen und Technologien sind auch Bestandteil einer Architekturanalyse
- [13] beinhaltet Hinweise für den Umgang mit evolutionärer Entwicklung und Refactorings im Hinblick auf relationale Datenbanken

10 Analyse von Prozessen

Viele Probleme mit oder innerhalb von Systemen resultieren aus Defiziten in den beteiligten Prozessen, in Kommunikationsproblemen zwischen Beteiligten, in unangemessenen Regulierungen oder sogar politisch-organisatorischen Differenzen zwischen Menschen, Abteilungen oder Unternehmen. Statische Codeanalysen finden diese Art von Problemen nicht.

Dieser Themenkomplex rechtfertigt sicherlich ein eigenes Buch, daher halten wir unsere Ausführungen dazu kurz.

Manche technischen Probleme in Systemen stellen sich bei näherer Betrachtung als *Symptome* organisatorischer Defizite, d.h. von Prozessproblemen, heraus. In dutzenden von Reviews und Audits haben wir praktisch immer solche organisatorischen Defizite gefunden und konnten strukturelle oder konzeptionelle Probleme im Code auf diese organisatorischen Sachverhalte zurückführen.

Adressieren Sie in Reviews daher auch Prozessthemen! Solche Probleme können Sie niemals durch bessere Architektur oder Programmierung beheben, sondern müssen dazu die Ablauf- oder sogar Aufbauorganisation verändern! Als IT'ler*in fehlt Ihnen in der Regel das Mandat, aber: Nur wenn Sie die resultierenden Probleme explizit adressieren, wird Ihr Management auch die Notwendigkeit solcher organisatorischen Änderungen einsehen!

Welche Prozesse können Probleme bereiten?

Wir haben in zusammen mehr als 50-jähriger Praxiserfahrung schwerpunktmäßig Prozessprobleme in folgenden Bereichen gefunden (besser gesagt: *erlitten...*):

- Anforderungsprozesse: Aus mangelhaften Anforderungen kann auch ein hervorragendes Entwicklungsteam unmöglich gute Software produzieren. *Garbage-in, garbage-out.*
- Entwicklungsprozesse: Agilität falsch umgesetzt, *Scrum-but*, Teamkonflikte, übermäßige Formalismen – auch hier drohen Probleme.

- Testprozesse: keine oder schlechte automatisierte Tests, Differenzen zwischen Entwicklungs- und Testteams.
- Build-, Deployment- und Release-Prozesse: Zu viele manuelle Schritte für Build/Deploy/Release notwendig, fehleranfällige Deployments.
- Betriebliche Prozesse: Laufzeitprobleme (Performance, Speicher, Threads etc.) nicht antizipierbar, Fehlerdiagnose schwerfällig und unzuverlässig.
- Supportprozesse: Zu wenig Rückmeldung von Support in die Entwicklung, Support optimiert auf kurze *closing-time* von Tickets, anstatt auf gründliche Lösung von Problemen.

Budgetierung, HR und sonstige Prozesse lassen wir hier mal außen vor.

Anforderungsprozesse

Einige Beispiele für Probleme in/mit Anforderungen:

- *Sie wissen nicht was sie wollen*: Anforderungen bleiben vage, unpräzise, grob. Das Resultat: Entwicklungsteams implementieren stark *konfigurierbare* Systeme, und verschieben damit die Entscheidungen über die konkrete Bedeutung von Anforderungen auf die Laufzeit.
- *Sie wollen dauernd etwas anderes*: Hohe Volatilität in Anforderungen, so dass Entwicklungsteams häufig *umbauen* müssen.
- Verschiedene Stakeholder*innen fordern widersprüchliche Dinge: Das können verschiedene Anforderungen an Daten, Validierungs- oder Geschäftsregeln bis hin zu unterschiedlichen Algorithmen sein.
- Anforderungen benötigen innerhalb der gesamten Organisation zu lange, bis sie beim Entwicklungsteam landen. Es gibt zu viele beteiligte Personen, zu viele Gremien oder zu viele Abstimmungsprozesse. Das sind typische Probleme von Entwicklungsprozessen, siehe nachfolgender Abschnitt.

Entwicklungsprozesse

Auch hierzu einige aus unserer Sicht typische Beispiele:

- Abstimmungsprobleme im Team oder zwischen verschiedenen Teams.
- Divergierende Ziele verschiedener beteiligter Personen (*hidden agenda*), insbesondere im Zusammenhang mit verschiedenen Organisationseinheiten oder Entwicklungsverantwortlichkeiten. In solchen Fällen sind technische Probleme häufig Symptome dieser organisatorischen Probleme.
- *Knowhow-Flaschenhalse*, wesentliche Aufgaben können nur von sehr wenigen (einzelnen) Personen übernommen werden, die als Folge davon unter chronischer Überlast arbeiten.
- Entscheidungskompetenz („Wer darf entscheiden?“) bei Personen, die nicht über die fachliche/technische Kompetenz verfügen („sich nicht auskennen“).
- Organisation des Entwicklungsteams widerspricht der internen Struktur des Systems, d.h. Verletzung von *Conways Law*.
- Zu wenig Feedback zwischen Beteiligten, entweder durch organisatorische Schranken, Medienbrüche oder sonstige Kommunikationsbarrieren.
- Zu wenig Freiraum des Entwicklungsteams für technische oder architektonische Grundlagen (etwa: Update auf neue Sprach- oder Bibliotheksversionen oder grundsätzlich sinnvolle Technologieupdates).

Suchen Sie innerhalb von Entwicklungsprozessen nach Aufgaben, an denen ungebührlich viele Personen beteiligt sind, die ungebührlich lange dauern oder mit denen Stakeholder sonstige Probleme berichten.

Wir alle arbeiten unter Zeit- und/oder Budgetdruck – das allein lassen wir erstmal nicht als „Ursache aller Probleme“ gelten. Allerdings können Organisationen bzw. Entwicklungsteams diesen Druck leicht übertreiben.

Testprozesse

Haben „Tests“ im Entwicklungsprozess einen angemessenen Stellenwert?

- Gibt es eine angemessene Menge automatisierter Tests?
- Laufen automatisierte Tests effizient (d.h. schnell genug)?
- Arbeiten automatisierte Tests effektiv, d.h. finden diese Tests überhaupt die richtigen oder wichtigen Probleme? Ineffektive Tests führen dazu, dass auch im produktiven Betrieb des Systems noch Fehler auftreten, die Tests (theoretisch) hätten früher finden können.
- Unterstützt die interne Modularisierung (*Komponentenschnitt*) des Systems die Testbarkeit? Bietet die innere Struktur des Systems genügend explizit definierte Schnittstellen, die Sie automatisiert testen können?

Deployment, Releases und betriebliche Prozesse

Im Bereich Release-Erstellung, Deployment und betriebliche Prozesse gibt es nach unserer Erfahrung einige typische Schwachstellen:

- **Kopfmonopole:** Dokumentation von Abläufen erfährt nur geringe Wertschätzung, das Wissen rund um betriebliche Themen befindet sich in einzelnen Köpfen.
- **Mangelnde Automatisierung:** Zu viele Aufgaben erfordern manuelle Eingriffe.
- **Fehlender Self Service:** Regelmäßig auftretende Routineaufgaben, z. B. das Anlegen einer neuen Entwicklungsdatenbank, erfordern umfangreiche Abstimmungs- oder Genehmigungsprozesse zwischen mehreren Betriebsteams oder Entwicklungsteams.
- **Fehlendes Feedback in Entwicklung:** Das Entwicklungsteam bekommt keine Rückmeldungen über betriebliche Probleme (etwa an externen Schnittstellen) - und kann demzufolge diese Probleme auch nicht beheben.

- **Fehlende Abstimmung:** Grundlegende Änderungen an der Software- oder Infrastrukturarchitektur werden nicht zwischen Entwicklungs- und Betriebsteams abgesprochen. Releases werden ohne hinreichende Kommunikation von Änderungen über die metaphorische Mauer geworfen.

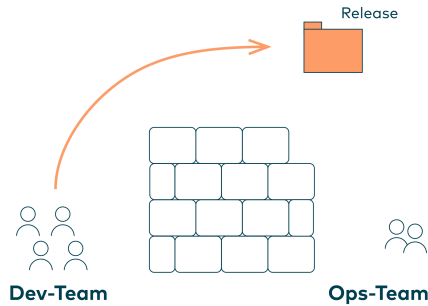


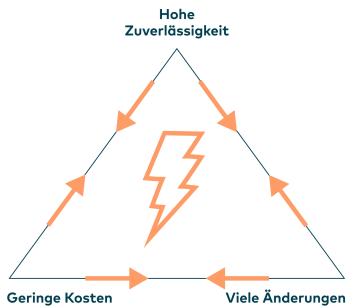
Abbildung 10.1: Fehlende Kommunikation zwischen Entwicklungsteams und Betrieb

11 Analyse von Infrastruktur

Worum geht es?

In diesem Abschnitt schauen wir genauer auf Infrastruktur und Betriebsabläufe: Passt die Infrastruktur zum System und den Architekturzielen? Unterstützt das vorhandene Monitoring und Logging eine schnelle Problemerkennung und Fehleranalyse? Berücksichtigt die Konzeption von Deployment und Betrieb auch mögliche Ausfallrisiken? Können passende Mechanismen bei Systemausfällen die schnelle Wiederherstellung gewährleisten? Wie hoch ist der Grad an Automatisierung?

Häufig bewegen sich Betriebsinfrastruktur und die zugehörigen Betriebsabläufe in einem Spannungsfeld konfligierender Ziele: Hohe Ausfallsicherheit mit geringen Kosten bei gleichzeitig hoher Änderungsgeschwindigkeit.



- Hohe Ausfallsicherheit bedingt höhere Kosten und Zeitaufwand, wegen der notwendigen (aufwändigen) Verfügbarkeits- und Testszenarien
- Viele Änderungen am System führen neben hohen Kosten durch umfangreiches Testsetup gleichzeitig zu höherem Risiko von Systemausfällen

Sie sollten die Gewichtung dieser drei Ziele für Ihr System individuell abstimmen und bei allen Beteiligten ein gemeinsames Verständnis der eingegangenen Kompromisse herbeiführen. Hilfreich kann dabei beispielsweise die Site Reliability Engineering Methodik¹ sein.

¹„Site Reliability Engineering“ (SRE) ist eine Reihe von Praktiken, um die Zuverlässigkeit des Betriebs von verteilten Anwendungen zu messen und stetig zu verbessern. Das Hauptziel von SRE ist die Bereitstellung skalierbarer und hochverfügbarer Softwaresysteme. Neben der Bearbeitung von Problemen und Bereitschaftsdiensten verbringen Site Reliability Engineers einen Großteil ihrer Zeit mit Entwicklungsaufgaben wie neue Plattformfunktionen, Skalierung oder

Beispiele typischer Probleme

- Es gibt zwar regelmäßige Backups, aber Wiederherstellen wurde nicht getestet.
- Änderungen finden direkt auf den Produktivsystemen statt.
- Monitoring produziert viele irrelevante Alerts, die niemand mehr ernst nimmt.
- Zu hoher Anteil an manuellen Tasks oder Feuerwehreinsätzen.
- Manuelle Konfiguration von Servern, statt automatisiertes Aufsetzen.

Methodik

Finden Sie zunächst heraus, ob bei allen beteiligten Personen ein gemeinsames Verständnis der Betriebsziele und einzugehender Kompromisse im Hinblick auf die oben genannten potentiellen Konflikte existiert. Passen diese Betriebsziele mit den Architekturzielen des Systems zusammen? Abhängig davon führen Sie dann detailliertere Analysen einzelner Teilgebiete durch. Dazu können Sie beispielsweise folgende Tools und Artefakte heranziehen:

- **Bebauungspläne, Architekturdokumentation, Serverlisten:** Vermitteln einen Eindruck von der Komplexität der Infrastruktur, dem Reifegrad der Dokumentationsprozesse und der Strukturiertheit beim Aufbau der Infrastruktur.
- **Unterlagen oder Post Mortems zu Systemausfällen** der letzten Monate: Diese können wertvolle Hinweise auf Betriebsstabilität, den Reifegrad der Notfallprozesse und die Fehlerkultur liefern.
- **Source Code zum Infrastruktur Setup:** Ermöglicht eine Einschätzung des Grads an Automatisierung, der Softwarequalität und der Deployment/Update Prozesse.
- **Betriebshandbücher, Playbooks, Incident Management Dokumente:** Bieten eine Gelegenheit, Risikomanagement und Fehlerkultur einzuschätzen.
- **Monitoringsysteme & zentrale Logverwaltungen:** Struktur, Detailgrad und Abdeckung können Hinweise auf Durchschaubarkeit und Nachvollziehbarkeit in Krisensituationen liefern.
- **Interviews mit Stakeholder*innen** wie Betriebsteam, Supportteam und Entwicklungsteam: Kennen viele Probleme aber auch Verbesserungschancen.

Automatisierung. Von den betreuten Softwaresystemen wird erwartet, dass diese hochgradig automatisiert und selbstheilend sind. Für weitere Details siehe [14], [15].

Folgen die Betriebsabläufe standardisierten Verfahren wie ITIL oder SRE, sollten Sie auch deren spezifische Artefakte betrachten, beispielsweise die Definitionen von *Service Level Objectives* und *Service Level Agreements*.

Grundlegende Infrastruktur und Systemarchitektur

Lassen Sie sich die grundlegende Infrastruktur- und Systemarchitektur anhand von Diagrammen erklären. Bringen Sie dabei die folgenden Punkte in Erfahrung:

- Werden Cloud Provider genutzt?
- Auf welcher physischen Hardware laufen die Services?
- Wo (in der Welt) steht die physische Hardware?
- Welche Betriebssysteme und Laufzeitumgebungen kommen zum Einsatz?
- Gibt es Spezialhardware wie etwa: USV, Storage-Array, Grafik- oder Krypto-Hardware o. Ä.?
- Welche Software-Komponenten laufen auf welcher Hardware?
- Welche Umgebungen (Staging, Produktion) existieren?
- Welche *Lock-Ins* (d.h. Abhängigkeiten von einzelnen Anbietern) gibt es?
- Welche Abhängigkeiten existieren zwischen den einzelnen Komponenten?

Betriebsabläufe

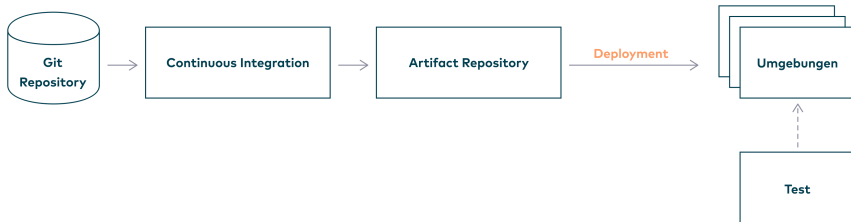
Finden Sie etwas über die Kultur und Arbeitsorganisation des Betriebsteams heraus, und über deren Zusammenarbeit mit anderen Teams:

- Wie arbeiten Entwicklung und Betrieb zusammen?
- Welche Feedbackschleifen gibt es (vom Betrieb an Entwicklung, vom Support an Betrieb)? Wie lange dauert die Umsetzung dieses Feedbacks?
- Wie viel Zeit verbringt das Betriebsteam mit manuellen Aufgaben?
- Wie hoch ist die Anzahl an „Feuerwehreinsätzen“?
- Wer entscheidet über den Einsatz neuer Infrastruktur-Technologien?

- Wie werden die für das System notwendigen *Secrets* (etwa administrative Accounts, Betriebssystem-Zugänge, DB-Admin-Zugänge o. Ä.) verwaltet?
- Welche Zugriffsregelungen gibt es für administrative Aufgaben (Wer besitzt *root* Rechte)? Gibt es dafür Notfall-/Vertretungspläne?
- Finden regelmäßige Security- und Datenschutzaudits statt?
- Gibt es 24/7 Administration oder Support?
- Welche Weiterbildungsmöglichkeiten gibt es für das Betriebsteam?

Deployment

Wir erachten zuverlässige und unkomplizierte Deployments als notwendig, um neue Funktionalität planbar und effizient in Produktion bringen zu können. Gleichzeitig stellen sie ein potentiell Risiko für Systemausfälle und neue Fehler dar. Prüfen Sie, ob das Vorgehen hinsichtlich seiner Dauer und seines Risikoprofils zum System passt.



- Wie werden Deployments durchgeführt (d.h. wie funktionieren Übergänge zwischen den *Stages*)?
- Wie oft erfolgen Deployments neuer Versionen und Updates der Infrastruktur?
- Gibt es eine Deployment-Strategie wie Blue/Green oder Canary Releases?
- Wie lange dauert ein Deployment und das Starten der einzelnen Komponenten?
- Wie werden Deployments überwacht? Gibt es ein Rollback-Verfahren?
- Wie verläuft der Test neuer Deployments?
- Können vorhandene Instanzen gestoppt werden, ohne Nutzer*innen zu stören?
- Wie werden Schemaupdates der Datenbank ausgerollt?

- Wird das System zur Installations-/Deployment-/Laufzeit konfiguriert? Gibt es Feature Toggles? Wie werden deren Einstellungen verwaltet?

Monitoring, Logging & Analytics

Monitoring, Logging und Analytics haben im Wesentlichen zwei Aufgaben:

1. Eine Datenbasis für Entscheidungen liefern
2. Auf Probleme hinweisen und Kontextinformationen dazu bereitstellen

Achten Sie darauf, ob das Monitoring die Fragen der Stakeholder*innen hinreichend beantworten kann. In Krisensituationen soll Monitoring für Transparenz und Reaktionsfähigkeit sorgen, indem es die notwendigen Daten aller Komponenten bereitstellt, zentral auswertet, visualisiert und wenn notwendig alarmiert. Dies sollte nach einem einheitlichen und systematisch strukturierten Schema erfolgen. Informationen sollten angemessen konsolidiert und unnötiger *Noise* vermieden werden.

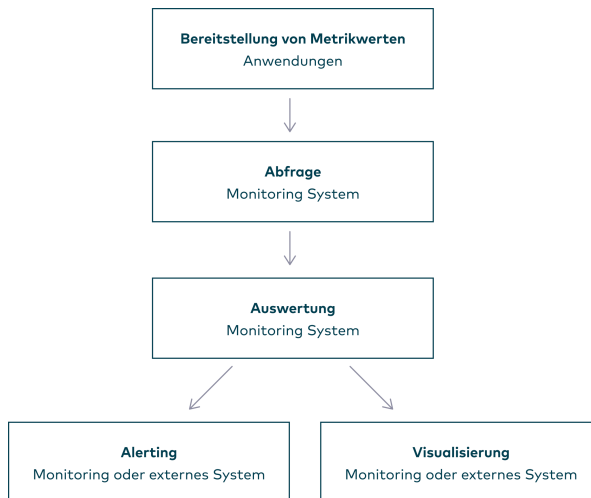


Abbildung 11.1: Möglichkeiten mit Monitoring

- **Metriken:** Gibt es vorgegebene Standards? Wer definiert die überwachten Metriken? Werden nur technische oder auch geschäftliche Metriken überwacht? Lassen sich mit den Monitoringdaten Schlüsse auf die echte Nutzung des Systems ziehen?
- **Alerting:** Werden Alerts nur versendet, wenn auch tatsächlich ein menschliches Eingreifen notwendig ist? Wie hoch ist die Anzahl der „Fehlalarme“?
- **Dashboards:** Dashboards sollten einen *High Level Überblick* über den Gesamtzustand geben und wilde Sammlungen von Visualisierungen vermeiden.
- **Logausgaben:** Gibt es ein standardisiertes Logformat über Anwendungen hinweg? Gibt es die Möglichkeit Logausgaben zentral zu durchsuchen?
- **Analytics Dienste:** Werden solche eingesetzt? Welche Erkenntnisse liefern sie?

Kosteneffizienz

Systeme sollten über Erfolgsmetriken verfügen, nach denen Sie deren Nutzen (mehr oder weniger) objektiv bewerten können, beispielsweise Umsatz in Geldeinheit oder Anzahl der Seitenaufrufe. Systeme verursachen jedoch auch Kosten, beispielsweise durch die Infrastruktur, Datentransfer und Betriebs-/Wartungsaufwände. Beide Aspekte sollten Sie im Auge behalten und deren Verlauf über die Zeit beobachten.

- Wie sieht das Kostenmodell der genutzten Ressourcen aus?
- Was kostet der Betrieb der Ressourcen? Was ist hier das Kostenlimit?
- Lassen sich Kosteneinsparungen erzielen, etwa durch Vereinfachung redundanter Setups, Automatisierung manueller Tasks oder Kostenanalysetools des Cloud Providers?

Infrastructure as Code

Infrastructure as Code (IaC) ist ein elementarer Baustein bei der Automatisierung manueller Tasks. Automatisierung wiederum hilft, Fehler bei manuellen Eingriffen zu vermeiden. Auch leistet IaC einen wesentlichen Beitrag, Infrastruktur schneller bereitzustellen, Setups nachvollziehbar zu machen und Deployments

zu beschleunigen. Dem gegenüber steht möglicherweise hoher Aufwand zur Automatisierung.

- Wie werden Updates der Infrastrukturkomponenten durchgeführt?
- Gibt es IaC zum Einrichten der Umgebungen?
- Welche Tools werden eingesetzt?
- Für welche Aufgaben gibt es (noch) keine Skripte und warum nicht?
- Liegen die Skripte komplett unter Versionsverwaltung? Muss Infrastruktur-Code ähnlichen Maßstäbe genügen wie in der Softwareentwicklung (z. B. Code Reviews)?

Risikoanalyse Ausfallsicherheit

Unter Zuhilfenahme der bisherigen Erkenntnisse können sie abschließend eine Risikoanalyse des Systems im Hinblick auf Ausfallsicherheit vornehmen:



- Wie funktioniert der Failover bei Abstürzen von Komponenten oder Ausfällen des Cloud Providers oder Rechenzentrums?
- Wie hoch ist die Wahrscheinlichkeit eines Ausfalls oder einer Überlastung einer spezifischen Komponente? Wie lange dauert es, bis dies bemerkt wird?
- Wie lange dauert es, bis das System wiederhergestellt ist bzw. ein anderes System übernommen hat?
- Wie groß ist der maximale Datenverlust beim Ausfall eines Datenspeichers?
- Wird regelmäßig überprüft und geübt, wie ein ausgefallenes System wiederhergestellt werden kann?
- Existieren konkrete und detaillierte Handlungsanweisungen (*Playbooks*) für Standardfehlerszenarien?
- Werden Post Mortems zu Ausfällen erstellt und ggf. die entsprechenden Playbooks angepasst?

12 Resultate kommunizieren

Auftraggeber wollen Lösungsvorschläge

Wir haben es häufig erlebt, dass unsere Auftraggeber*innen im Grunde die bestehenden Probleme bereits kannten oder sie zumindest befürchtet hatten. Wir als Reviewer*innen sollten dann primär als Quelle für pragmatische und realistische Lösungsansätze dienen.

In eine Zusammenfassung Ihrer Review-Ergebnisse gehören daher unbedingt auch Vorschläge zur Verbesserung, Abhilfe oder Risikominimierung.

Die effektive Management-Summary

Die kompakte Zusammenfassung Ihrer Befunde, Schlussfolgerungen und Vorschläge für das Management ist möglicherweise das wichtigste Resultat des gesamten Reviews. Das Management soll darin maximal drei **Handlungsempfehlungen** mit deren Herleitungen vorfinden.

- Schreiben Sie dieses Management- oder Executive-Summary erst zum Schluss, wenn Ihre Resultate feststehen.
- Diese Zusammenfassung steht grundsätzlich am Anfang – sowohl einer Präsentation als auch eines Dokumentes.
- Verweisen Sie darin auf die detaillierten Erläuterungen, die weiter hinten folgen.

Es geht bei einem Management-Summary **nicht** darum zu erklären, warum das Review so schwierig und aufwändig war, bzw. welche Strapazen wir dabei erleiden mussten.

Reden und/oder schreiben?

Ob Sie Resultate in einer Abschlusspräsentation einem Publikum vorstellen, oder ob Sie einen schriftlichen Report verfassen, entscheiden letztlich Ihre maßgeblichen Stakeholder*innen. Wir finden eine Kombination hilfreich, in folgenden Schritten:

1. Sie diskutieren in kleinem Kreis mit den Auftraggeber*innen des Reviews die Ergebnisse und deren Priorisierung.
2. Sie stellen danach die (eventuell überarbeiteten) Resultate in Form einer Abschlusspräsentation einem etwas größeren Kreis vor. Auch hier wünschen wir uns aktives Feedback und Diskussion.
3. Erst danach fassen Sie Ihre Ergebnisse in Schriftform zusammen.

Drei Schritte bedeuten mehr Aufwand – daher haben sich in unserer persönlichen Vergangenheit viele Reviews mit einer Abschlusspräsentation begnügt. Dabei wandern häufig vielerlei Erläuterungen und Details in den Anhang, so dass eine solche Sammlung von Folien auch ohne die Tonspur der Live-Präsentation noch verständlich bleibt.

Probleme und Maßnahmen priorisieren

Probleme und auch Maßnahmen sollten Sie priorisieren und kategorisieren, beispielsweise nach:

- geschäftlichem Schaden oder Risiko
- technischer Schwierigkeit bzw. geschätzte Höhe des Lösungsaufwandes

Wir bevorzugen sehr einfache Schemata zur Priorisierung, mit drei Kategorien.

- 1** Hohe Mehrkosten oder starke Verzögerungen, unverzügliche Behebung empfohlen
- 2** Verursacht Kosten oder Verzögerungen, Behebung empfohlen
- 3** Störend, leicht verzögernd, leichte Mehraufwände

Abbildung 12.1: Die drei Kategorien der Kategorisierung von Problemen

Was gehört in eine Abschlusspräsentation oder einen Abschlussbericht

Leser*innen sollten eine ganzheitliche Darstellung des Reviews vorfinden, angefangen von Zielen und Scope des Reviews, über die konkret gewählte Vorgehensweise bis hin zu Schlussfolgerungen und Handlungsempfehlungen. Folgende Themen halten wir für wichtig:

1. Ein kurzes, sehr prägnantes Management-Summary (siehe oben).
2. Zusammenfassung *organisatorischer* Aspekte des Reviews:
 - Was genau waren die Ziele und Scope des Reviews?
 - Welche Beschränkungen gibt es?
 - Wie sind Sie vorgegangen? Wieviel Zeit haben Sie (circa) worin investiert?
 - Mit wem haben Sie worüber gesprochen, inklusive Daten und Dauer?
 - Welche Review-Aktivitäten haben Sie neben Gesprächen und Interviews durchgeführt.
3. Welche Aspekte am System und dessen Entwicklung fanden Sie positiv (*keep, don't change*)?
4. Eine Zusammenfassung der Probleme und Risiken, Top-down beschrieben
 - In welchen Kategorien haben Sie Probleme und Risiken gefunden?
 - Welche Probleme haben Sie gefunden, beginnend mit den höchsten Prioritäten (also die schlimmsten, gravierendsten Probleme zuerst)

- Wo drohen Risiken?
 - Welche Auswirkungen sind zu erwarten oder sind bereits akut?
5. Zusammenfassung der vorgeschlagenen Maßnahmen (optional)
- In welchen strategischen, technischen oder organisatorischen Bereichen schlagen Sie Maßnahmen vor?
 - Welche unterschiedlichen Optionen gibt es?

Die Abschlusspräsentation ist nicht das Ende

Gehen Sie davon aus, dass während einer Abschluss- oder Ergebnispräsentation diverse Stakeholder*innen mit teilweise grundlegenden Änderungswünschen zu Ihnen kommen.

Formulierungen kommen unter das sprachliche Mikroskop aller Beteiligten – und möglicherweise werden Sie Ergebnisse in verkürzter oder modifizierter Form vor unterschiedlichen Gremien präsentieren müssen (oder dürfen).

Rechnen Sie mit Gegenwind

Beteiligte werden Ihre Resultate anzweifeln, abstreiten oder schlichtweg leugnen. Viele Aspekte in komplexer Software erlauben unterschiedliche Interpretation – etwa die Wichtigkeit einer hohen prozentualen Testabdeckung oder der strikten Einhaltung von Programmierkonventionen.

Gehen Sie davon aus, dass einige Ihrer Resultate der Meinung anderer Beteiligter krass widersprechen – und diese Menschen deswegen möglicherweise generalisierend den gesamten Review ablehnen.

Von enthusiastischer Zustimmung, Verwunderung, Zweifel, Verniedlichung bis zu offenem Widerstand durften und mussten wir ein Spektrum von Reaktionen auf Review-Ergebnisse erfahren. Ähnliches könnte Ihnen auch widerfahren.

Hierzu ein paar Ratschläge:

- Bei politischem oder organisatorischem Gegenwind benötigen Sie Unterstützung auf höherer Ebene: Versuchen Sie, das Top-Management von Ihren Resultaten zu überzeugen, damit könnten Sie Widersachern elegant den Wind aus den Segeln nehmen.
- Sie können Widerstand und Opposition fast nie mit rein sachlichen Argumenten auflösen, vielmehr benötigen Sie Hilfe aus Politik, Wirtschaft und Psychologie.
- Erledigen Sie Ihre (formalen) Hausaufgaben: Ihre Resultate müssen auf Fakten basieren, die Sie möglichst gründlich recherchieren und belegen müssen. Insbesondere kritische Resultate sollten Sie doppelt nachprüfen – um die *Angriffsfläche* für mögliche Gegner zu minimieren.
- Etwas mehr dazu finden Sie im Artikel „Die Gegner“¹.

¹<https://jaxenter.de/knigge-fur-softwarearchitekten-die-gegner-386>

13 Fazit

Durch die systematische *Breitensuche* haben Sie an einigen Stellen Ihres Systems Probleme und Risiken gefunden. Dank fokussierter Interviews mit wichtigen Beteiligten verstehen Sie deren Sichtweise auf das System jetzt besser. Sie haben eine Menge über die Architektur, den Code und sonstige *Innereien* Ihres Systems gelernt und die Aktivitäten rund um Anforderungen, Entwicklung, Rollout und Betrieb hinterfragt.

Diese Probleme haben Sie nach wirtschaftlichen und technischen Kriterien priorisiert und für einige davon gleich ein paar angemessene Abhilfemaßnahmen vorgeschlagen. Ihre wesentlichen Stakeholder konnten im Rahmen der Abschlusspräsentation Stellung beziehen, und vielleicht hat Ihr Management bereits grünes Licht für ein paar vielversprechende Maßnahmen signalisiert. Klopfen Sie sich und Ihrem Review-Team mal anerkennend auf die Schulter – Sie haben eine Menge geleistet.

Reviews als Grundlage von Verbesserung und Evolution

Dennoch: Ihnen und dem Entwicklungsteam steht noch ein großes Stück Arbeit bevor, nämlich die Erkenntnisse des Reviews in wirtschaftlichen Mehrwert zu überführen. Dazu gilt es, die erkannten Probleme in einer der Situation angemessenen Reihenfolge durch passende Abhilfemaßnahmen aus der Welt zu schaffen. Ob Sie dabei die berühmten *low hanging fruits* pflücken (also die Dinge beseitigen, die sich quasi *nebenbei* erledigen lassen, oder sich auf die aus betriebswirtschaftlicher Sicht *teuersten* Probleme stürzen (also diejenigen Probleme, die hohe Kosten oder Aufwände verursachen)), bleibt in der Entscheidungshoheit Ihres Managements.

Ihre Ergebnisse stellen die wesentliche Grundlage für zukünftige Verbesserungen Ihres Systems dar. Statt dem leider weit verbreiteten *hektischen Aktionismus* können Sie mit Ihrem Entwicklungsteam nun auf Basis des systematischen Reviews die Verbesserung und Evolution Ihres Systems angehen. Für diese spannende Aufgabe wünschen wir Ihnen viel Erfolg und gutes Gelingen.

Unser Angebot

Ein Software-Review hilft Ihnen, frühzeitig konkretes Optimierungspotentiale Ihrer Systeme zu identifizieren. Wir verfügen über viele Jahre Erfahrung in der Durchführung solcher Reviews sowie der nachfolgenden Evolution und Weiterentwicklung bestehender Systeme. Gerne führen wir Reviews Ihrer Systeme auch mit Ihnen **gemeinsam** durch.

Unsere Schulung zum **iSAQB Modul „IMPROVE – Reduzierte Wartungskosten durch evolutionäre Verbesserung“** zeigt praxisorientiert auf, wie Sie Probleme und Risiken ihrer Systeme identifizieren und einordnen können sowie Verbesserungen zielgerichtet schrittweise umsetzen können. Weitere Details finden Sie unter: <https://www.innoq.com/de/trainings/improve>. Bei Bedarf konzipieren wir gerne individuelle Schulungen entsprechend ihrer Anforderungen.

Weitere Infos zu Schulungen und Reviews erhalten Sie unter <https://www.innoq.com>.



Wir beraten ehrlich, denken innovativ und entwickeln leidenschaftlich gern. Das Ergebnis: Erfolgreiche Softwarelösungen, Infrastrukturen und Geschäftsmodelle.

Als Technologieunternehmen fokussieren wir uns auf Strategie- und Technologieberatung, Softwarearchitektur und -entwicklung, Methoden- und Technologietraining sowie Plattform-Infrastrukturen.

Wir unterstützen mit über 150 Mitarbeiter*innen an Standorten in Deutschland und der Schweiz Unternehmen und Organisationen bei Konzeption und Umsetzung komplexer Vorhaben und der Verbesserung bestehender Softwaresysteme.

Wir engagieren uns in Open-Source-Projekten sowie dem iSAQB e.V., und geben Wissen und Erfahrungen auf Konferenzen und Meetups sowie in zahlreichen Büchern und Fachartikeln weiter.

14 Unser Manifest für Reviews

- Wir arbeiten mit der erforderlichen **Sorgfalt** und nach bestem **Wissen**.
- Wir arbeiten **unbestechlich** und mit einer **selbstreflektierenden Grundhaltung**.
- Wir behandeln Ergebnisse und Informationen selbstverständlich **vertraulich**.
- Wir arbeiten **ergebnisoffen** und starten **ohne vorgefasste Meinung**.
- Wir **legen** den **Kontext** und die **Grenzen** unserer Reviews **offen**, um unsere Ergebnisse und Schlussfolgerungen für andere Personen nachvollziehbar zu machen.
- Wir lassen unser **Ego zuhause**, um offen für Anregungen, Beobachtungen, Ideen und Probleme anderer zu sein.
- Wir trennen **Probleme** von der **Lösungsdiskussion**.
- Wir versuchen, die **Beweggründe** der Entscheider*innen **nachzuvollziehen**, bevor wir bewerten.
- Wir bewerten nur im Rahmen unserer **Kompetenz**.
- Wir ziehen bei **Spezialthemen** punktuell **weitere Expertise** hinzu.
- Wir halten das **Review-Team klein** (2-5 Reviewer*innen).
- Wir **blicken nach vorne**, anstatt die Vergangenheit vollständig aufzuarbeiten.
- Wir **entwickeln uns** und unsere **Review-Vorgehen** ständig weiter.

Einige dieser Tipps haben wir im schon fast historischen [16] gefunden (dessen Ansätze zwar in Teilen recht formal wirken, der jedoch viele zeitlose Vorschläge für Reviews enthält). Andere Anregungen stammen vom Bundesverband deutscher Unternehmensberater [17].

15 Review? Audit? Analyse? Bewertung?

Für die Aktivitäten, ein System auf Risiken, Probleme oder auch Potentiale hin zu untersuchen, finden wir in unserer Branche sehr unterschiedliche Bezeichnungen. Dem Titel dieses Primers entnehmen Sie bereits, dass wir den Begriff *Review* favorisieren. Allerdings möchten wir hier noch einige ähnliche Begriffe vorstellen. Einige dieser Definitionen haben wir auf Basis der entsprechenden Wikipedia-Einträge formuliert.



Abbildung 15.1: Eine Auswahl an Synonymen für den Begriff *Review*

Analyse bezeichnet eine systematische Untersuchung der Bestandteile eines Systems, wobei insbesondere Beziehungen und Wechselwirkungen zwischen diesen Bestandteilen betrachtet werden.

Assessment identifiziert offene Punkte bzgl. der erfolgreichen Weiterentwicklung eines Systems und liefert erste Handlungsbedarfe für das System bzw. dessen umgebende Organisation.

Audit untersucht, ob Systeme oder Prozesse die jeweils geforderten Standards erfüllen. Audits erfolgen häufig im Rahmen eines formalen Qualitätsmanagements.

Bewertung (engl: *Evaluation*) ist eine oftmals *quantitative* Begutachtung von Systemen oder Prozessen sowie Organisationseinheiten.

Inspektion bezeichnet die tiefergehende, standardisierte Analyse eines Systems mittels Leitfadens/Checkliste.

Post Mortem bezeichnet eine Analyse, die nach dem Ende eines zu analysierenden Ereignisses stattfindet – meist nach gravierenden Fehlern, Ausfällen oder sonstigen wirtschaftlich schädigenden Vorkommnissen.

Prüfungen „sind arrangierte Situationen, in denen bestimmte Leistungsvollzüge provoziert werden“. Prüfungen stellen grundlegende Fähigkeiten und Grenzen des betrachteten Systems möglichst neutral fest. Prüfungszeugnisse bescheinigen die Eignung bzw. Nichteignung.

Retrospektive (lat. *retrospectare* „zurückblicken“) bezeichnet einen kurzen Rückblick auf bereits stattgefundene Ereignisse. Der Begriff findet insbesondere in iterativ-agilen Prozessen Anwendung, wo im Team gemeinsam analysiert wird, um aus Fehlern sowie positiven Aspekten zu lernen.

Review prüft Arbeitsergebnisse mit einem mehr oder weniger formal geplanten und strukturierten Analyse- und Bewertungsprozess. Projektergebnisse werden einem Team von Gutachter*innen präsentiert und von diesem kommentiert oder akzeptiert.

Root-Cause-Analyse versucht die Ursachen aufgetretener Probleme zu ergründen, insbesondere durch die Differenzierung von *Symptomen* und *Ursachen*.

Software Due Diligence (sinngemäß übersetzt etwa *Bewertung mit besonders gebotener Sorgfalt*) analysiert Stärken und Schwächen eines Systems sowie die entsprechenden Risiken und spielt daher eine wichtige Rolle in Fällen, wo es um Käufe/Verkäufe von Systemen (bzw. auch Unternehmen) geht.

Software Risk Evaluation identifiziert und analysiert Risiken im Softwaresystem und versucht, Abhilfen oder Minderungsstrategien für diese Risiken zu finden.

Untersuchung ist ein Oberbegriff für *Analyse* (vertiefende, systematische Untersuchung), *Inspektion* (technische Untersuchung), *Studie* (wissenschaftliche Untersuchung) oder *Test* (meist funktionelle Untersuchung) eines Systems.

16 Quellen

Bücker, Artikel, Videos und Webseiten.

- [1] J. Ousterhout, *A Philosophy of Software Design*. Yaknyam Press, 2018, Ein praxisorientierter Ratgeber für besseren Code. Gar nicht philosophisch, sondern voller handfester Tipps gegen überschüssige Komplexität im Code.
- [2] B. Wolf und G. Starke, *Softwarearchitekturen pragmatisch dokumentieren. Eine kompakte Einführung in arc42*. 2019 [Online]. Verfügbar unter: <https://leanpub.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, Nur kostenpflichtig verfügbar (<https://www.iso.org/standard/35733.html>), jedoch auf [Wikipedia](https://en.wikipedia.org/wiki/ISO/IEC_9126) passabel erläutert. [Online]. Verfügbar unter: <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, Pragmatischer Ansatz zur Identifizierung von Hotspots im Code – zum Beispiel durch Analyse der Codekomplexität und Änderungshäufigkeit. Für uns war dieses Buch ein echter Meilenstein, da es die Codehistorie berücksichtigt, wie sie in Ihrem Versionskontrollsystem gespeichert ist. Adam hat mehrere Open-Source-Bibliotheken geschrieben, mit denen Sie Ihren eigenen Code analysieren können. Sehr zu empfehlen für alle, die mit Reviews oder Software-Engineering zu tun haben.
- [5] M. Schumacher, E. Fernandez, D. Hybertson, und 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]. Verfügbar unter: <https://leanpub.com/ddd-by-example>
- [7] E. Evans, *Domain-Driven Design Referenz*. 2019 [Online]. Verfügbar unter: <https://leanpub.com/ddd-referenz>

- [8] M. Harrer, *Software-Modernisierung mit Wardley Maps*. 2020, Vortragsmitschnitt INNOQ Technology Lunch. [Online]. Verfügbar unter: <https://www.youtube.com/watch?v=ARp3EIUCKM4>
- [9] S. J. Fowler, *Production-Ready Microservices*. O'Reilly Media, 2016 [Online]. Verfügbar unter: <http://shop.oreilly.com/product/0636920053675.do>
- [10] C. Lilienthal, *Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen*, 3. Auflage. dpunkt.verlag, 2019.
- [11] A. Tornhill, *Software Design X-Rays: Fix Technical Debt with Behavioral Code Analysis*. Pragmatic Bookshelf, 2018, Analyse basierend auf Commits und Diffs. Sehr empfehlenswert für alle, die sich mit größeren Codebasen beschäftigen.
- [12] Dr. M. Greiler, *Code Reviews*. 2019, eBook auf Anfrage [Online]. Verfügbar unter: www.michaelagreiler.com¹
- [13] S. Ambler und P. Sadalage, *Refactoring Databases – Evolutionary Database Design*. Addison-Wesley Professional, 2006.
- [14] B. Beyer, C. Jones, J. Petoff, und N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*, 1. Auflage. O'Reilly Media, Inc., 2016 [Online]. Verfügbar unter: <https://landing.google.com/sre/books/>
- [15] B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, und S. Thorne, *The Site Reliability Workbook: Practical Ways to Implement SRE*, 1. Auflage. O'Reilly Media, Inc., 2018 [Online]. Verfügbar unter: <https://landing.google.com/sre/books/>
- [16] K. Wiegers, *Peer Reviews in Software – A Practical Guide*. Addison-Wesley, 2002, Schon alt, enthält aber viele zeitlose Vorschläge für Reviews.
- [17] „Qualität in der Beratung“. Bundesverband deutscher Unternehmensberater [Online]. Verfügbar unter: https://www.bdu.de/media/296535/qualitaet_in_der_unternehmensberatung.pdf
- [18] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, und M. Stal, *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley, 1996.

- [19] M. Carr, S. Konda, I. Monarch, C. Walker, und F. Ulrich, „Taxonomy-Based Risk Identification“, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, CMU/SEI-93-TR-006, 1993 [Online]. Verfügbar unter: <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11847>
- [20] P. Kruchten, R. Nord, und I. Ozkaya, *Managing Technical Debt: Reducing Friction in Software Development*, 1st Aufl. Addison-Wesley Professional, 2019.

Über die Autor*innen



Markus Harrer

Markus Harrer arbeitet als Softwareentwickler und Berater sowohl in konservativen Branchen als auch in Start-Ups. Seine Spezialgebiete sind Softwarelandschaftmodernisierung und Software Analytics. Er hilft, Software nachhaltig und strategisch sinnvoll zu verbessern. In seinem Blog feststelltaste.de schreibt er unter anderem über die Analyse von Softwaresystemen mit Hilfe von Data-Science-Methoden und Wardley Maps. [🐦 @feststelltaste](https://twitter.com/feststelltaste)



Christine Koppelt

Christine Koppelt arbeitet als Senior Consultant bei INNOQ. Ihr Schwerpunkt liegt auf der Realisierung von Digitalisierungsprojekten für mittelständische Unternehmen. Dabei beschäftigt sie sich vor allem mit den Themen Machine Learning, DevOps und Data Engineering. Nebenbei organisiert Sie in München das Data Engineering Meetup. Sie verfügt über Erfahrung mit Code- und Architektur-Reviews in mittleren und großen Systemen. [🐦 @ckoppelt](https://twitter.com/ckoppelt)



Gernot Starke

Dr. Gernot Starke (INNOQ Fellow) arbeitet seit über 25 Jahren in der Softwareentwicklung als Softwarearchitekt, Coach, Consultant und Trainer. Für Mandanten unterschiedlicher Branchen durfte er an der Realisierung mittlerer und großer IT-Systemen mitwirken, ebenso an vielen Reviews. Nebenbei hat er arc42 und aim42 (mit-)gegründet, betreibt aktiv diverse Open-Source Initiativen und schreibt Bücher und Fachartikel. 🐦 @gernotstarke



Benjamin Wolf

Ben Wolf ist Softwarearchitekt und -entwickler bei INNOQ. Er mag sauberen Code und gibt seine Vorstellung von Softwarequalität als Sprecher bei Konferenzen und Meetups sowie in Trainings weiter. Dabei ist ihm wichtig, dass vor allem die Einstellung von Teams gute Softwarequalität prägen. Ben beschäftigt sich aus verschiedenen Perspektiven mit Code-Reviews und Möglichkeiten, Code systematisch zu verbessern. 🐦 @ichaos1985

**Getreu dem Bonmot »Besser geht immer«
leidet praktisch jede Software an Problemen:**

Vielleicht läuft sie zu langsam, stürzt manchmal ab, sieht nicht so schön aus wie das Produkt der Konkurrenz oder Änderungen dauern zu lange. Mal abgesehen von Hello-World-Anwendungen, lässt sich überall noch so manches besser machen.

Dieser praktische Leitfaden zeigt, wie langjährig gewachsene Softwaresysteme effektiv analysiert und bewertet werden können.