AISE – Qualitätsziele schrittweise garantieren

Modernisierung – Aber sicher!

Der Schritt zur Modernisierung eines Bestandssystems entsteht oft aus über die Jahre aufgestauten Problemen, die schließlich aktives Handeln erzwingen, um überhaupt ein Weiterbestehen zu ermöglichen. Dieses Aufschieben akkumuliert eine Menge von Risiken, die sich darin äußern, dass Aufwandsabschätzungen oft um Dimensionen daneben liegen. Im Folgenden wird ein iterativer Modernisierungsprozess beschrieben, der auf analysebasierte Entscheidungen sowie den kontinuierlichen Aufbau leichtgewichtiger und nachhaltiger QA-Maßnahmen setzt.

Eingangs soll die Frage betrachtet werden, was eigentlich ein Modernisierungsvorhaben ausmacht. Ein Refactoring, welches im Rahmen der Umsetzung eines Features durchgeführt wird, wird üblicherweise nicht dazu gezählt. Wie steht es um den Abbau technischer Schulden? Doch auch diese Kategorie passt nicht.

Modernisierung

Als typisches Beispiel für Modernisierung können Technologie-Migrationen angesehen werden, die selbst ein weites Feld aufspannen: Sie umfassen die Migration von Frameworks auf neuere Versionen ebenso wie den vollständigen Austausch technischer Komponenten. Letzteres kommt wiederum in Varianten daher. Als Beispiel sei die Migration der verwendeten Datenbank-Technologie genannt. Im einfachsten Fall handelt es sich nur um die Anhebung der (Major-)Version des verwendeten Produkts. Bereits anspruchsvoller gestaltet sich der Wechsel auf ein vergleichbares Produkt eines anderen Herstellers. Richtig herausfordernd wird es, wenn die Datenbank-Familie gewechselt wird, beispielsweise von einer relationalen Datenbank zu einem Document-Store.

Der Austausch einer Technologie ist normalerweise ein für Anwender des Systems nicht sichtbares Feature, bringt also keinen unmittelbaren Mehrwert. Damit verbundene Aufwände und Risiken müssen trotzdem gerechtfertigt sein, das heißt, es müssen andere Treiber existieren. Einen solchen kann die Migration bestehender On-premise-Systeme in Cloud-Umgebungen darstellen, der einen Technologiewechsel erzwingt. Doch auch solche Vorhaben finden nicht im luftleeren Raum statt. Dahinter stehen Geschäftsziele, welche wiederum die Ausgangsbasis für daraus abzuleitende Qualitätsziele eines Systems darstellen.

Allgemein ausgedrückt: *Die Frage nach der Modernisierung eines Systems entsteht, wenn dieses nicht (mehr) in der Lage ist, bestehende oder veränderte Qualitätsziele zu erfüllen.*

Qualitätsziele basieren auf Kriterien, wie sie zum Beispiel durch die ISO-25010 aufgelistet werden. So kann die Notwendigkeit elastischer Skalierbarkeit entstehen, die aber mit dem bestehenden Hosting-Modell nicht kosteneffizient realisierbar ist. Die Anforderung kann durch eine Cloud-Migration erfüllt werden, zieht aber weitgehende technologische Anpassungen nach sich. Das Thema Datenbank stellt dabei nur einen Teilaspekt dar, hinzu kommen Themen wie Deployment, Konfiguration, Monitoring usw.

Doch auch abseits technologischer Aspekte spielen Qualitätsziele eine Rolle als Treiber für Modernisierungen. Über die Jahre gewachsene Systeme leiden oft unter starker struktureller Erosion, sodass Änderungen mit sehr hohen Kosten oder Risiken verbunden sind. Das Hinzufügen neuer Features oder schlicht das Beheben von Fehlern ist nur noch schwer oder nahezu unmöglich durchzuführen. Eine Restrukturierung des Systems zur Erhöhung der Erweiterbarkeit beziehungsweise Wartbarkeit kann als Gegenmaßnahme angesetzt werden – als Stichwort sei Domain-Driven Design genannt.

Herausforderungen und Risiken

Die Erreichung von Qualitätszielen ist das zentrale Aufgabengebiet der Disziplin „Softwarearchitektur“. Damit spielt sich Modernisierung grundsätzlich auf dieser Ebene ab. Begreift man zudem den Begriff „Architektur“ als Menge „wichtiger Entscheidungen, die später nur schwer zu ändern sind“ (Martin Fowler), dann ist es wenig verwunderlich, dass Modernisierung mit Attributen wie „aufwendig“ oder „riskant“ assoziiert wird. Die Umsetzung von Architekturentscheidungen betrifft Querschnittsaspekte eines Systems und damit potenziell große Teile des Codes. Die Auswahl einer Datenbanktechnologie bestimmt nicht nur die verwendete API zum Zugriff (z. B. JDBC-Treiber, O/R-Mapper), sondern beeinflusst auch die Modellierung von Datenstrukturen oder transaktionales Verhalten.

Die Schätzung „Die Migration dauert zwei, maximal drei Sprints“ wird nach den besagten Sprints oft von der Realität widerlegt. Dieser Umstand ist primär der Größe und der Komplexität der über Jahre gewachsenen Systeme geschuldet; Umfang und Risiken notwendiger Änderungen sind durch manuelle Stichproben

im Code nur schwer zu erfassen. Dahinter verbirgt sich ein grundsätzliches Problem: Wie viel Wissen existiert überhaupt über die strukturelle Beschaffenheit des Systems? Können die Auswirkung einer Änderung erkannt beziehungsweise realistisch abgeschätzt werden? Dokumentation, soweit sie existiert, ist meist nicht aktuell und kann nicht in hinreichender Detailtiefe alle Implementierungsdetails des Systems beschreiben. Entwickler könnten gegebenenfalls Auskunft geben, ihr Blick ist aber oft auf Teilbereiche limitiert und manche Wissensträger haben das Unternehmen gegebenenfalls schon verlassen.

Die Wahrheit liegt im Code. Dieser ist gerade im Legacy-Umfeld aber häufig umfangreich und schwer zu lesen, denn er „kodiert" in erster Linie fachliches und technisches Wissen, um es für eine Maschine ausführbar zu machen. Er beschreibt damit lediglich, „wie" Informationen verarbeitet werden, selten aber den für Entwickler nötigen Gesamtkontext. Die zur Abschätzung von Auswirkungen benötigten Informationen sind oft selbst bei sehr gut dokumentierten und strukturierten monolithischen Systemen nur schwer zu erfassen. Ein gängiges Problem ist zum Beispiel Leaking von Implementierungsdetails über APIs hinweg. Dazu zählt unter anderem die unbeabsichtigte Verwendung transitiver Abhängigkeiten (Bibliotheken, Frameworks). Microservices sind zwar technologisch bedingt stärker voneinander isoliert, auch hier kann eine Änderung der Domänenlogik unbeabsichtigte Auswirkungen auf das Verhalten einer Schnittstelle haben, sofern diese nicht voneinander entkoppelt sind.

Ist Modernisierung deshalb schon von vornherein zum Scheitern verurteilt? Ist es sinnvoller, gleich auf der „Grünen Wiese" neu zu starten? Grundsätzlich kann davon ausgegangen werden, dass dieser Ansatz vergleichbare, wenn nicht sogar höhere Risiken birgt: Das zu modernisierende System hat seine Funktionsfähigkeit über Jahre unter Beweis gestellt, eine Neu-Implementierung muss einen vergleichbaren funktionalen beziehungsweise qualitativen Stand erst einmal erreichen, bevor sich überhaupt ein geschäftlicher Mehrwert ergibt. Die damit einhergehenden Herausforderungen sollten nicht unterschätzt werden.

AISE – Schritt für Schritt

Im Folgenden wird ein iterativer Prozess beschrieben, der ein bestehendes System schrittweise (wieder) an zu erfüllende Qualitätsziele heranführt. Dabei werden als Rahmenbedingungen die Minimierung von Risiken sowie die Ermöglichung paralleler Weiterentwicklung gesetzt, weiterhin wird nicht von einem vollständigen Technologie-Bruch ausgegangen (z. B. Mainframe-Migration).

Am Anfang steht die Formulierung einer Modernisierungsstrategie:

- Welche Qualitätsziele sollen erreicht werden (elastische Skalierbarkeit)?
- Worin besteht die grundlegende Lösungsstrategie (z. B. Cloud-Deployment) und wie trägt sie zu den Qualitätszielen bei (nutzungsabhängige horizontale Skalierung)?
- Was sind notwendige Modernisierungsetappen, die umgesetzt werden müssen (Datenbank-Technologie, Deployment, Konfiguration, Sicherheit, Monitoring, Logging) und welche Abhängigkeiten existieren zwischen diesen?
- Welche bekannten Risiken (z. B. geringe Testabdeckung, fehlendes technologisches Wissen) müssen betrachtet und wie können sie erkannt werden? Welche Möglichkeiten zur Eingrenzung bestehen (Erhöhung der Testabdeckung, Prototyping)?
- Welche grundsätzlichen Rahmenbedingungen (z. B. parallele Weiterentwicklung) müssen beachtet werden und wie beeinflussen sie den Modernisierungsprozess (z. B. Durchführung in kleinen Entwicklungsschritten, regelmäßige Integration bei Vermeidung langlebiger Feature-Branches)?

Die Strategie definiert das Vorgehen und die Leitplanken für die Durchführung des Modernisierungsvorhabens und soll insbesondere den Fokus der Beteiligten auf das Wesentliche schärfen: Das Bearbeiten von Querschnittsfunktionen (Stichwort: Architektur) bedeutet für Entwickler, dass sie mit einem großen Teil bestehenden Codes in Berührung kommen. Dabei werden sie mit einer hohen Anzahl sogenannter „Dreckecken" konfrontiert und die Versuchung kommt auf, diese sofort zu beheben. Damit wird aber einer unkontrollierten Explosion von Aufwänden beziehungsweise Risiken Tür und Tor geöffnet. So unschön es im Einzelfall erscheinen mag: Sofern eine solche Stelle kein Problem im Sinne von Zielerreichung oder Risiken darstellt, sollte diese nicht bearbeitet werden. Einer Erfassung als technische Schulden steht hingegen nichts im Wege.

Der folgende Modernisierungsprozess wurde in Kundenprojekten erfolgreich angewandt und besteht aus der iterativen Wiederholung von vier Schritten zur Umsetzung einzelner Modernisierungsetappen: Analyse -> Implementierung -> (Ab) Sicherung -> Evaluierung (**siehe Abbildung 1**). Auf die einzelnen Schritte soll näher eingegangen werden, zuvor seien aber folgende Hinweise gegeben:

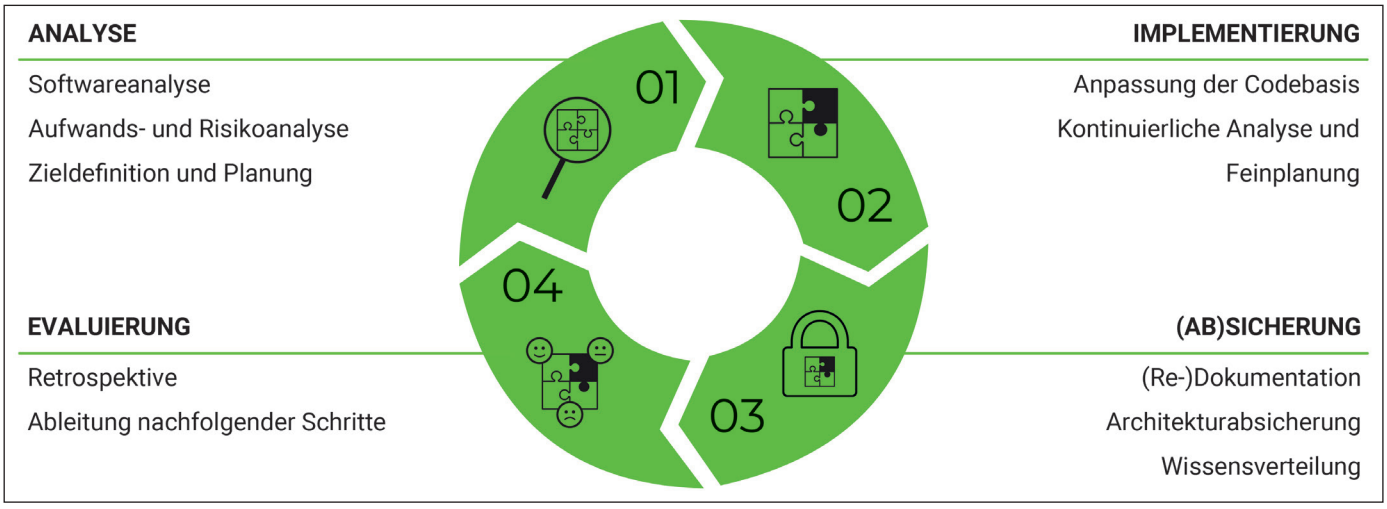

Abb. 1: AISE-Zyklus

- Es gibt erprobte Methoden zum Umgang mit Bestandssystemen, zum Beispiel [LegacyCode]. Wie gut diese im Einzelfall funktionieren, hängt sehr von den konkreten Umständen ab. Aus diesem Grunde ist es sinnvoll, sich an das System „heranzutasten“ und verlorenes Wissen „On-the-Job“ wiederzugewinnen. Daraus folgt, dass die ersten Modernisierungsetappen eher nicht solche mit höchsten Aufwänden oder Risiken sein sollten.
- In verschiedenen Projekten hat eine Exploration wertvolle Erkenntnisse geliefert: Eine prototypische Implementierung innerhalb weniger Tage beziehungsweise eines Sprints, ohne auf funktionale Korrektheit oder Codequalität zu achten, die anschließend wieder verworfen wird. Dies führt zu einem schnellen Kennenlernen betroffener Bestandteile des Systems und darin enthaltener Muster. Hypothesen über das angestrebte Vorgehen können validiert werden, das gewonnene Wissen fließt in die Modernisierungsstrategie ein.
- Zwar führen alle Wege nach Rom, bei einer Modernisierung aber nicht immer direkt. Bei der Planung von Etappen kann es sinnvoll sein, Umwege einzuplanen, um den Umfang einzelner Schritte zu reduzieren und eine schnelle Integration für parallele Weiterentwicklung zu ermöglichen. Ein Beispiel sind Abstraktionen, die für die Zeitdauer des Modernisierungsvorhabens die Koexistenz alter und neuer technischer Ansätze ermöglichen, im finalen Zustand aber wieder entfernt werden.

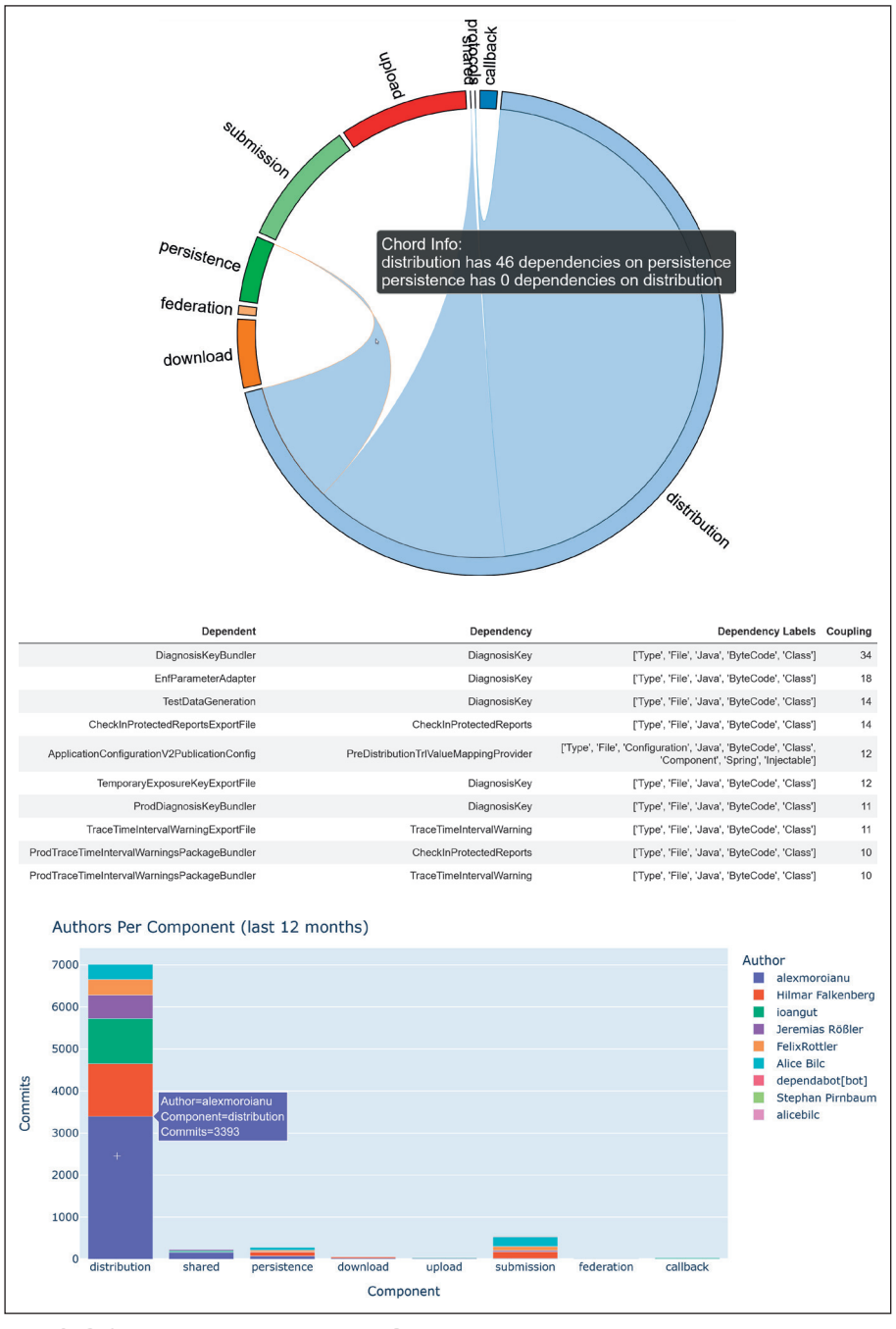

Abb. 2: Softwareanalyse – Kopplung und Ownership

A – Analyse

Im einleitenden Abschnitt wurde ein grundlegendes Problem beschrieben: Es werden umfangreiche Eingriffe an einem System vorgenommen, deren Auswirkungen nur schwer bestimmbar sind, sprich Umfang und Risiken von Änderungen. Es existieren jedoch Werkzeuge zur Softwareanalyse (z. B. für Java jQAssistant [AnalyticsjQA]), welche insbesondere bei in die Jahre gekommenen Systemen wertvolle Informationen zugänglich machen, unter anderem (**siehe Abbildung 2**):

- Umfang (Kopplungsgrad) und Art (z. B. Interfaces, Entitäten, Modellklassen) von Abhängigkeiten sowie potenzielle Auswirkungen (Impact) auf andere Bereiche,
- Testabdeckung und Code-Metriken (z. B. Komplexität) kritischer Bereiche,
- Ownership und Knowledge-Loss (noch oder nicht mehr verfügbare Experten für Code-Bereiche).

Die Umsetzung einer Etappe (z. B. Migration eines Frameworks) gleicht damit nicht mehr einem Blindflug, sondern orientiert sich an gesammelten Fakten („Informed Decisions“). Das Aufsetzen beziehungsweise Kennenlernen eines entsprechenden Werkzeugs ist zwar mit Aufwand verbunden, gerade bei größeren Vorhaben wird eine am Anfang vergleichsweise geringe Zahl investierter Tage schnell durch frühzeitig erkannte Risiken kompensiert. Wenig überraschend können derartige Analysen nicht nur für einzelne Etappen durchgeführt werden, sondern schon bei der initialen Erarbeitung der Modernisierungsstrategie äußerst hilfreich sein.

I – Implementierung

Die Implementierung stellt die Umsetzung einer einzelnen, möglichst klar abgegrenzten Modernisierungsetappe dar und basiert unter anderem auf den Erkenntnissen der Analyse. Es ist allerdings davon auszugehen, dass parallele Weiterentwicklungen stattfinden, auch wenn es sich „nur“ um Fehlerbehebungen handelt (ansonsten wäre das System als „tot“ einzustufen). Parallele Entwicklungsstränge stellen bereits im normalen Entwicklungsalltag eine Herausforderung dar – am Ende müssen alle Änderungen fehlerfrei integrierbar sein. Das Bearbeiten von

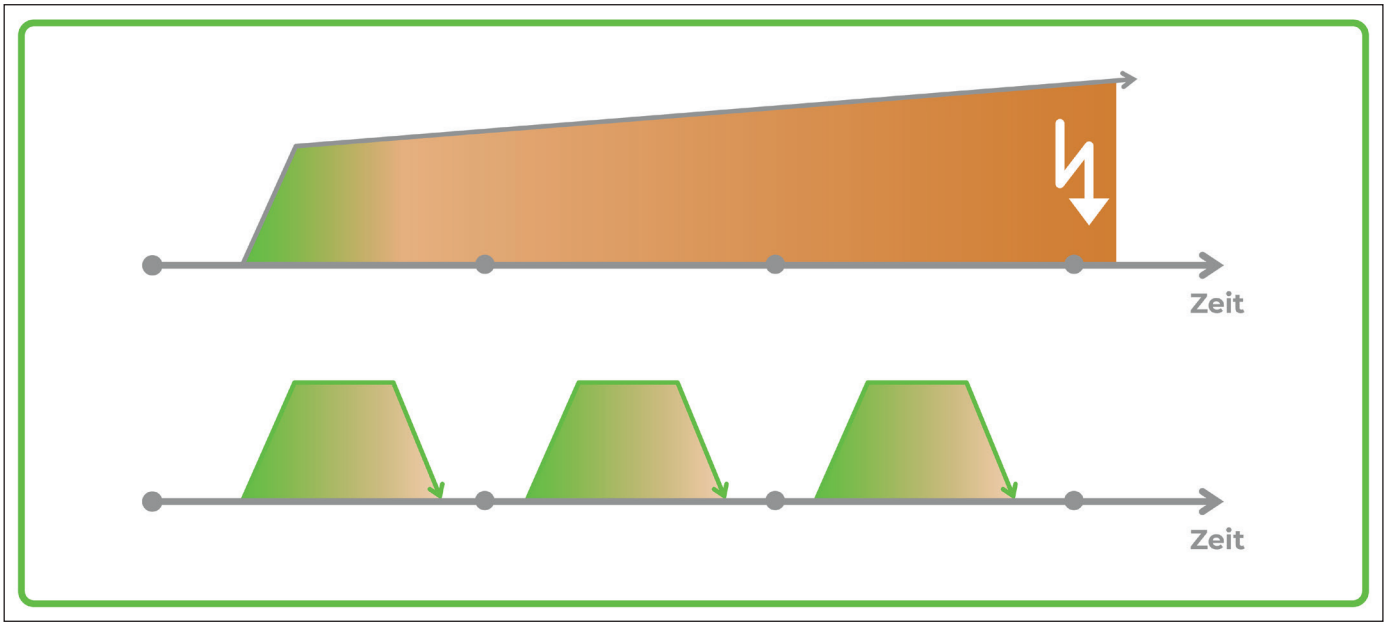

Abb. 3: Steigende Integrationsrisiken bei langläufiger Entwicklung in parallelen Zweigen

Querschnittsaspekten (z. B. Austausch eines Frameworks) erhöht jedoch massiv die Wahrscheinlichkeit möglicher Konflikte und damit einhergehender Risiken. Daher wird an dieser Stelle noch einmal die Empfehlung ausgesprochen, die Etappen möglichst klein zu schneiden und gegebenenfalls durch Umwege über temporäre Abstraktionen zu gehen. Die damit zwischenzeitlich erhöhte Komplexität des Codes stellt einen Tradeoff zur Abfederung von Integrationsrisiken dar (**siehe Abbildung 3**).

S – (Ab)Sicherung

So aufwendig und riskant Modernisierung auch erscheinen mag – mit ihr ergibt sich die Chance, Fehler der Vergangenheit zu korrigieren oder zumindest nicht zu wiederholen. Es geht dabei um die Dokumentation gefällter Architekturentscheidungen. Dies richtet sich nicht nur an die Zukunft, sondern insbesondere an Entwickler, die zum gleichen Zeitpunkt in parallelen Entwicklungszweigen an Features oder Fehlerbehebungen arbeiten. Diese sind an der Umsetzung nicht aktiv beteiligt, müssen aber Veränderungen verstehen und umsetzen. Auch die Vergangenheit spielt eine Rolle: Durch die Bearbeitung von Querschnittsaspekten wird verloren gegangenes, aber wertvolles Wissen über die Struktur des Systems sowie technische beziehungsweise fachliche Aspekte wiedergewonnen und kann festgehalten werden.

Die Voraussetzung für erfolgreiche Dokumentation ist, den Umfang so groß wie nötig, den entstehenden Aufwand aber so klein wie möglich zu halten. Eine klare Empfehlung soll hier zugunsten leichtgewichtiger Ansätze wie arc42 [arc42] in Kombination mit Architecture Decision Records (ADRs) [ADRNygard] ausgesprochen werden. Letztere bestehen aus kompakten Dokumenten, die jeweils eine konkrete Entscheidung beschreiben und stets der gleichen Struktur folgen: Titel, Status, Kontext und Konsequenzen. **Abbildung 4** zeigt ein Minimal-Beispiel eines ADR, es darf gern umfangreicher sein. Insbesondere sollten in den Konsequenzen getroffene Abwägungen erkennbar sein, zum Beispiel wenn es um Technologie-Auswahl geht.

Es ist weiterhin empfehlenswert, einen Docs-As-Code-Ansatz [DocsAsCode] zu wählen: Die Dokumente werden in einer Markup-Sprache gepflegt (Asciidoc, Markdown), im Versionskontrollsystem direkt beim Anwendungscode abgelegt und im Rahmen des Build-Prozesses kontinuierlich publiziert. Die Vorteile liegen auf der Hand: Die Dokumentation ist für Entwickler gut zugänglich, kann leicht aktualisiert werden, Änderungen werden nachvollziehbar und können Bestandteil einer „Definition of Done“ beziehungsweise von Reviews bei einer Branch-Integration gemacht werden.

Wer konsequent ist, geht einen Schritt weiter: Neben der funktionalen Absicherung durch Tests können im Rahmen des Build-Prozesses auch Werkzeuge zur kontinuierlichen Validierung der dokumentierten Architekturentscheidungen eingesetzt werden. Einerseits geht es darum, langfristiger Erosion vorzubeugen. Jedoch ist ein zweiter Aspekt zum Zeitpunkt der Modernisierung wichtiger: Es muss sichergestellt werden, dass zum Beispiel bereits umgesetzte Technologiemigrationen beziehungsweise Restrukturierungen nicht versehentlich durch parallele Weiterentwicklungen wieder rückgängig gemacht werden.

Die Existenz von Dokumentation, die sich stetig verändert, ist als Kommunikationsinstrument dafür unzureichend. Im Java-Umfeld stehen hierfür Werkzeuge wie ArchUnit [ArchUnit] oder jQAssistant [ADRjQA] zur Verfügung. Letzteres erlaubt die Einbettung von Architekturregeln in die Dokumentation, die daraus vom Build-Prozess publizierten ADR-Dokumente enthalten Reports über Strukturen sowie unerwünschte Abweichungen als Tabellen, Diagramme, CSV-Dateien usw. Der aktuelle Zustand des Systems und seine Abweichungen vom gewünschten Zielzustand werden damit kontinuierlich transparent gemacht.

E – Evaluierung

Der Abschluss einer Etappe sollte mit einer Retrospektive abgeschlossen werden und folgende Punkte umfassen:

- Hat die Etappe den erwarteten Beitrag geleistet, welche Lücken konnten wider Erwarten nicht geschlossen werden, sind neu erkannt worden oder möglicherweise hinzugekommen?
- Welche Erkenntnisse haben sich ergeben, die eine Anpassung der Modernisierungsstrategie erfordern?
- Konnten Probleme erkannt werden, die den nächsten Etappen im Wege stehen, zum Beispiel fehlendes technisches Wissen, methodische Defizite im Umgang mit Bestandscode, unzureichende Infrastruktur beziehungsweise Testdaten für Integration und Test?

001 - Verwendung von Slf4j für Logging

Status

Akzeptiert

Kontext

Notwendigkeit eines Logging-Frameworks, über welches Informationen aus dem Produktiv-Code bzw. Fremdbibliotheken auf konfigurierbare Kanäle und Level geloggt werden können

Entscheidung

- Verwendung der Logging-Facade Slf4j im Produktiv-Code
- Als Backend kommt log4j 2.x zum Einsatz

Konsequenzen

- Slf4j wurde gewählt, um über vorhandene Adapter die Ausgaben von Bibliotheken (z.B. via JCL/JUL) verfügbar zu machen
- Die Verwendung anderer Logging-Frameworks sowie direkte Ausgaben auf Konsolen sind nicht zulässig

⊘ *Only the usage of Slf4 is allowed* ☐

Type	Logger	Lines
app.coronawarn.server.services.distribution.Application	org.apache.logging.log4j.LogManager	51
app.coronawarn.server.services.distribution.assembly.component.BusinessRulesArchiveBuilder	org.apache.logging.log4j.util.Strings	65
app.coronawarn.server.services.submission.ServerApplication	org.apache.logging.log4j.LogManager	55
app.coronawarn.server.services.callback.ServerApplication	org.apache.logging.log4j.LogManager	49
app.coronawarn.server.services.download.Application	org.apache.logging.log4j.LogManager	43
app.coronawarn.server.services.federation.upload.Application	org.apache.logging.log4j.LogManager	56

Abb. 4: Architecture Decision Record mit eingebetteten Validierungen

- Haben sich zwischenzeitlich Änderungen in der Zieldefinition ergeben, auf die eingegangen werden muss? Sind gegebenenfalls aus parallelen Weiterentwicklungen Aspekte hinzugekommen, die sich auf das Modernisierungsvorhaben auswirken könnten?

Mit diesem Feedback und daraus abgeleiteten Konsequenzen geht es in die nächste Runde, bis das gesamte Vorhaben abgeschlossen ist – die Bewertung dessen erfolgt anhand der definierten Qualitätsziele.

Zum Schluss

Lohnt sich Modernisierung? Ja, sie schützt Investitionen in Bestandssysteme, macht diese fit für neue Anwendungsfelder und bewahrt dabei wertvolles fachliches Wissen beziehungsweise macht es wieder zugänglich.

Ist Modernisierung ein aufwendiges und riskantes Unterfangen? Ja, aber man kann damit umgehen: Klare Zieldefinitionen sowie Abgrenzungen als Leitplanken, ein iterativer Prozess für Feedback und Nachjustierung, Entscheidungen basierend auf gezielten Analysen anstatt eines Bauchgefühls sowie die schrittweise (Re-)Etablierung geeigneter QA-Maßnahmen sind die notwendigen Werkzeuge. An dieser Stelle sei auch ausdrücklich auf das Projekt aim42 [aim42] hingewiesen, in welchem Community-gestützt eine Fülle von Methoden und Prozesse zusammengetragen werden.

Abschließend sei noch ein Denkanstoß gegeben: Ähnlich dem Management technischer Schulden kann Modernisierung als ein kontinuierliches Unterfangen aufgefasst werden. Im Grunde geht es darum zu überwachen, ob die Architektur eines Systems bestehenden qualitativen Anforderungen gerecht wird. Erkannte Abweichungen müssen bewertet und adressiert werden, je zeitiger dies passiert, desto geringer gestalten sich Aufwände und Risiken. ||

Literatur & Links

[ADRjQA] St. Pirnbaum, Einführung in Dokumentation und Absicherung von Architekturentscheidungen mit ADRs und jQAssistant, siehe: **https://vimeo.com/638976964**

[ADRNygard] Documenting Architecture Decisions, siehe: **https://www.cognitect.com/blog/2011/11/15/documenting-architecture-decisions**

[aim42] Architecture Improvement Method, siehe: **https://www.aim42.org/**

[AnalyticsjQA] St. Pirnbaum, Einführung Software Analytics mit jQAssistant and Jupyter Notebooks, siehe: **https://vimeo.com/612435725**

[arc42] Erfolgreiche Softwarearchitektur – arc42, siehe: **https://www.arc42.de/**

[ArchUnit] ArchUnit (Projekt-Homepage), siehe: **https://archunit.org**

[DocsAsCode] Documentation As Code – Docs-as-Code, siehe: **https://docs-as-co.de/**

[LegacyCode] M. Feathers, Working Effectively With Legacy Code, Pearson, 2004

Der Autor

Dirk Mahler
(dirk.mahler@buschmais.com)
ist Senior-Consultant der BUSCHMAIS GbR, einem Dresdner Beratungsunternehmen. Er beschäftigt sich mit Softwarearchitektur sowie Qualitätssicherung und engagiert sich in diesem Rahmen in der Entwicklung des Open-Source-Werkzeugs jQAssistant.