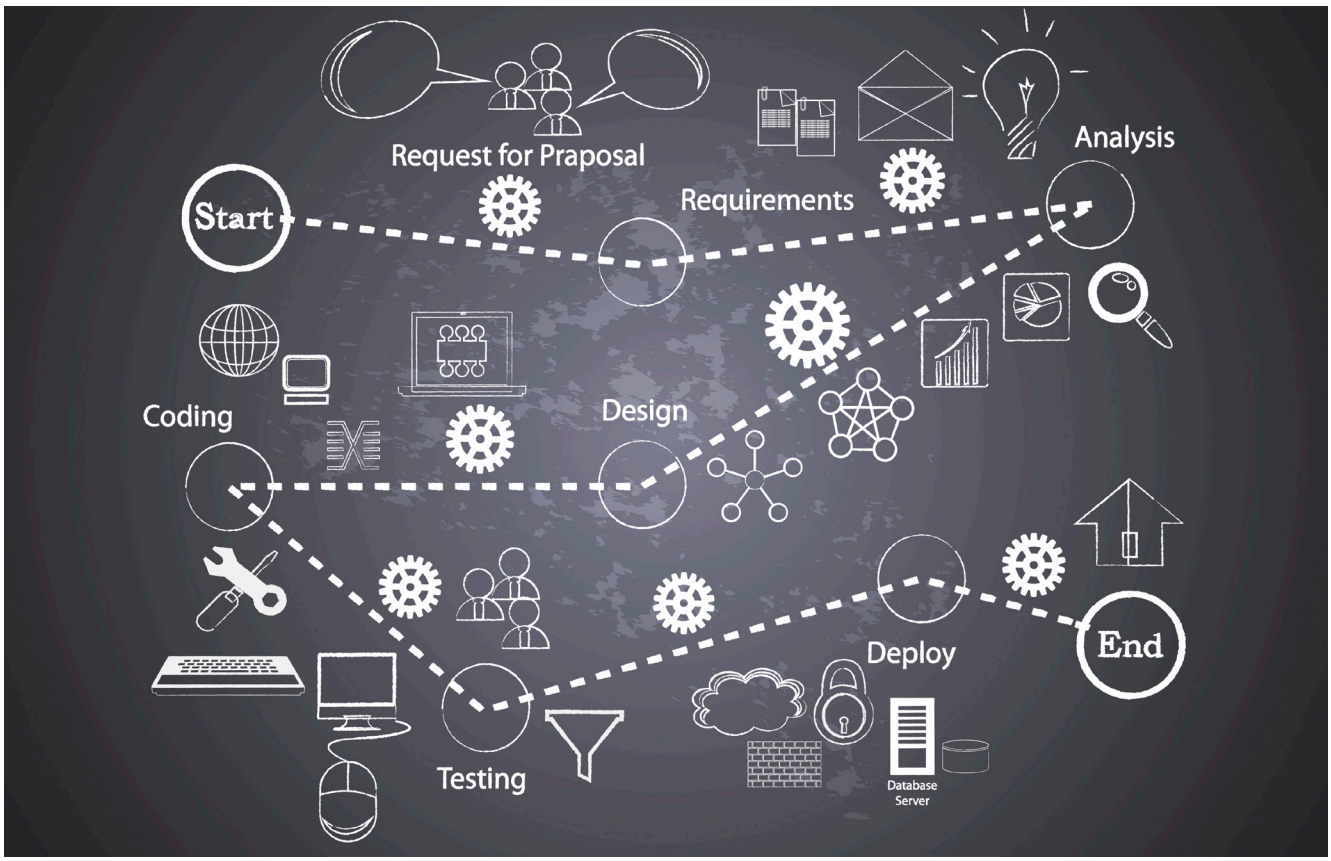


DIE ARCHITECTURE IMPROVEMENT METHOD

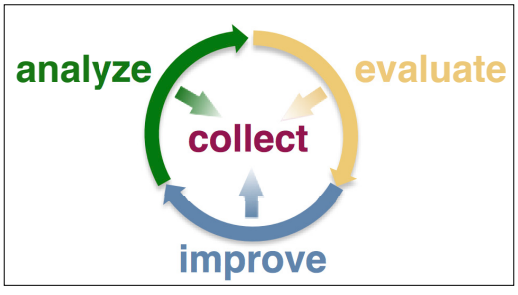
Software systematisch verbessern

Mit etablierten, iterativ angewandten Praktiken zeigt aim42 Wege aus der Legacy-Hölle.

Es ist ein bekanntes Phänomen: Obwohl zu Beginn beim Entwickeln des Projekts alles sauber und ordentlich zuging, degeneriert das System mit der Zeit – das Phänomen der „verfaulenden Software" schlägt zu. Änderungen werden dann immer riskanter, schwieriger und langwieriger. In der Entwicklung und im Betrieb mehren sich die Probleme, die zu beheben immer mehr Zeit in Anspruch nimmt. Gleichzeitig steigen aber auch Änderungs- und Betriebskosten, während die Zufriedenheit von Entwicklern, (fachlichen) Auftraggebern, Testern, Administratoren und anderen Beteiligen ständig abnimmt. Vermutlich kennt jeder Leser diese Situation: Willkommen in der Legacy-Hölle.

Kein technisches Problem

Viele Praktiken von Softwareentwicklung und -architektur versprechen Hilfe in dieser Situation – allerdings helfen sie in der Regel nicht gegen knappe Budgets und drängelnde Manager.



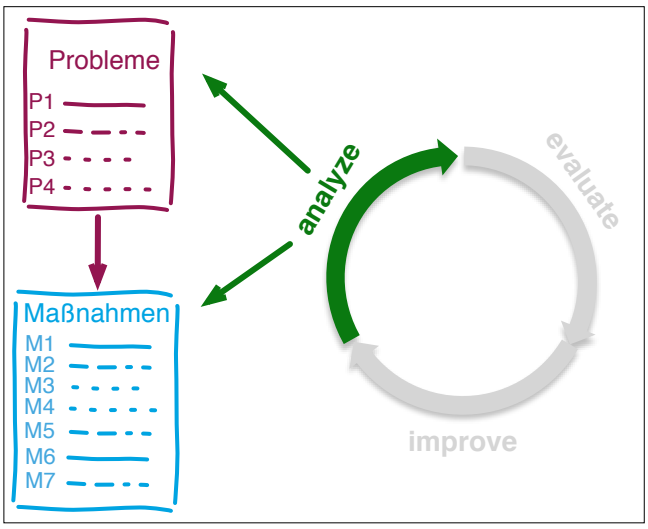
Die drei Phasen (Kreis) beim aim42-Prozess (Bild 1)

Die ursprüngliche Ordnung im System geht immer mehr verloren. Ausnahmen werden zur Regel, Code wird immer schwerer verständlich. Das Entwicklungsteam beginnt von „historisch gewachsen" zu sprechen – weil kaum jemand nachvollziehen kann, wie es zu dieser kontinuierlichen Verschlechterung kommen konnte.

Aber wo liegen die Gründe? Verlernen Teams über die Zeit das sau-

bere Arbeiten? Keinesfalls: Aller Erfahrung nach liegt die Legacy-Hölle sehr häufig in der Kombination verschiedener Umstände begründet: dies sind Zeitdruck, schlechte Kommunikation im Team, Wechsel von Teammitgliedern, unterschiedliches Know-how, Altlasten, inkonsistenter Code, mangelnder Fokus auf langfristige Ziele wie Wartbarkeit oder Verständlichkeit.

Um dieser Hölle zu entkommen, müssen Teams sowohl technische wie auch organisatorische Herausforderungen in Angriff nehmen. Der Autor und Kollegen haben hierzu etablierte Praktiken gesammelt, die sie iterativ einsetzen. Dieses Verfahren nennen sie „Architecture Improvement Method", kurz „aim42" [1][2], und veröffentlichen es als Open Source. Da Entwicklungsteams die Vielzahl von Problemen kaum alleine bewältigen können, richtet sich diese Initiative sowohl an technische wie auch an betriebswirtschaftliche Stakeholder. aim42 schlägt dazu ein iteratives Vorgehen in drei Phasen vor, die sich nahtlos in bestehende Entwicklungsprozesse integrieren lassen.

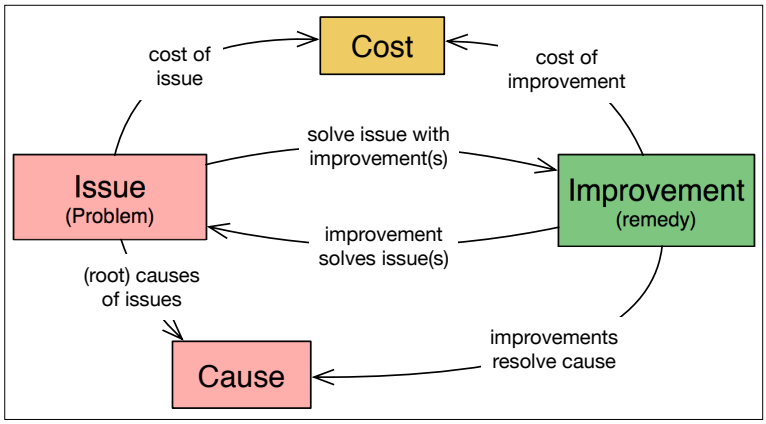


aim42 sammelt Probleme (Issues) und leitet folgerichtig daraus Maßnahmen (Improvements) ab (Bild 3)

Ein erster Überblick

aim42 erfindet keine neuen Ideen, sondern bindet bestehende Praktiken in ein definiertes Vorgehen ein. Dabei bewähren sich die drei Phasen die Bild 1 zeigt: analyze, evaluate, improve. Dabei gehen Sie so vor:

- Analysieren Sie das System (analyze) und sammeln Sie die Probleme (collect), die sich im und um das System und dessen Organisation finden. Daraus ergibt sich eine „Issue List".
- Jedes Problem wird hinsichtlich seiner einmaligen und/oder wiederholten Kosten bewertet. Das ist die wesentliche Aufgabe der Evaluate-Phase.
- Noch beim Sammeln der Probleme suchen Sie nach Maßnahmen und Ansätzen, welche die Probleme oder deren Ursachen lösen könnten (improve). Diese Sammlung bildet dann das „Improvement Backlog".
- Auch Maßnahmen verursachen Kosten. Auch diese sind systematisch zu ermitteln oder zu schätzen.
- Das Gegenüberstellen der Kosten von Maßnahmen sowie der Kosten des Problems ist eine wertvolle Hilfe, um beim Priorisieren und Planen der Verbesserungen die richtigen Entscheidungen zu treffen.



Die Terminologie von aim42 (Bild 2)

Kosten schätzen hilft!

Bis auf das vermeintlich lästige Schätzen von Kosten klingt dies sicher logisch. Warum aber sollen sich Entwickler beim Verbessern von Systemen ausgiebig mit Geld und Kosten beschäftigen? Der Grund ist das Unverständnis fachlicher oder betriebswirtschaftlicher Stakeholder für eher technische oder methodische Argumentation: Für „lose Kopplung", „hohe Kohäsion" oder „verbesserte Modularisierung" mag kein Fachbereich und kaum ein Projektleiter Geld bereitstellen.

Werden allerdings die tatsächlichen Kosten der vorhandenen Probleme offengelegt, wie aim42 es ausdrücklich fordert, erleichtert dies die Kommunikation mit Budget- oder fachlich Verantwortlichen. Dann müssen Softwareentwickler und -architekten endlich nicht mehr über die schwer vermittelbaren inneren Qualitäten, Kopplung, Kohäsion oder Implementierungsdetails streiten, sondern können in der universellen Sprache von Business-Experten argumentieren: Geld.

Problem, Ursache, Maßnahme

Nach Erfahrung des Autors stürzen sich Entwicklungsteams zu oft auf scheinbar offensichtliche Lösungsansätze, ohne zuvor angemessen über Alternativen und Konsequenzen nachzudenken.

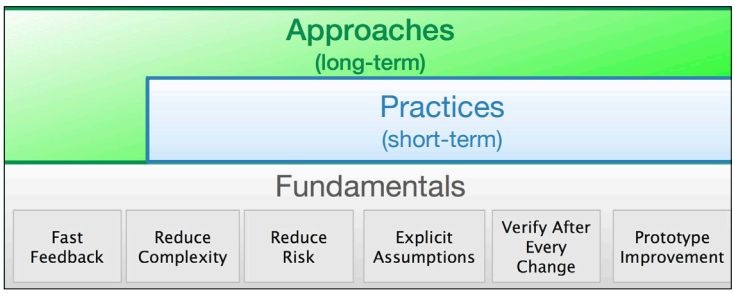
Bild 2 stellt die Zusammenhänge der Begriffe dar, mit denen aim42 arbeitet. „Issue" ist ein freundliches Wort für ein beliebiges Problem: Es verursacht Kosten (Cost of Problem) und könnte Symptom einer Ursache (Cause) sein.

Zu einem Problem gibt es eine oder mehrere verschiedene Maßnahmen, die es lösen (Improvement), wie Bild 3 nahelegt. Jede dieser Maßnahmen adressiert möglicherweise mehrere Issues. Natürlich kostet es etwas, eine Lösung umzusetzen (Cost of Improvement).

Damit stehen Probleme und Lösungen in einem bestimmen Verhältnis zueinander. Das allein wäre nun nicht schlimm, wären da nicht noch die ▶

möglichen Wechselwirkungen von Verbesserungsmaßnahmen untereinander: Ein Entwickler hat beispielsweise durch Caching und Denormalisierung das Laufzeitverhalten seiner DB-Abfragen deutlich gesteigert, das vorherige Problem damit vielleicht sogar vollständig behoben. Leider ist dabei die Komplexität im Code deutlich gestiegen – was wiederum die Wartungs- und Änderungsfreundlichkeit des Systems verschlechtert.

So ist es mit vielen Maßnahmen, die beim Entwickeln getroffen werden: Sie führen zu negativen Konsequenzen an anderen, oftmals a priori nicht bekannten Stellen.



Kurzfristige Praktiken und langfristige Ansätze bilden die Improve-Phase (Bild 5)

Analyze: Breitensuche

Nun zu den drei erwähnten zentralen Phasen, in in die sich der Verbesserungsprozess einteilen lässt. Systematische Verbesserung sollte mit einer Bestandsaufnahme beginnen, um eine Übersicht der Situation, insbesondere der Probleme zu gewinnen. Dazu zählen natürlich die bekannten Fehler und „Workarounds" aus dem Issue-Tracker, andererseits auch die Ergebnisse statischer Codeanalysen, also Abhängigkeiten, Komplexität, Einhalten von Codekonventionen, Testabdeckung und so weiter.

Zusätzlich ist es besonders wichtig, auch einmal die beteiligten Personen nach den aus ihrer Sicht wichtigen Problemen zu befragen. Die Kollegen wissen in der Regel viel über das System, seinen Aufbau und insbesondere die dazugehörigen Entwicklungs- und Betriebsprozesse.

Die Erfahrung des Autors aus zahlreichen Analysen und Reviews zeigt, dass die Suche nach Problemen gleichzeitig eine Menge über deren Lösung lehren kann. Insbesondere in Gesprächen mit den beteiligten Stakeholdern kommen oft konkrete Vorschläge zur Sprache, wie sich die bestehende Situation verbessert ließe. Das sind gute Kandidaten für das „Improvement Backlog".

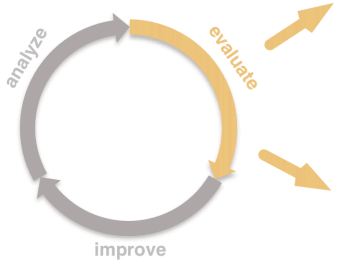
aim42 hält eine Vielzahl von Praktiken bereit, um die Suche nach Problemen zu unterstützen; der Kasten „Beispiele für Analyze-Praktiken" gibt einen Überblick.

Evaluate: Kosten schätzen

Eingangs ging es bereits um die Schwierigkeiten, betriebswirtschaftlich orientierte Stakeholder von Verbesserungen der inneren Qualität von Software zu überzeugen. Da diese Änderungen nur selten direkt „von außen" wahrnehmbar sind, mangelt es im Management oft an Verständnis.

Vergleichen Sie also systematisch Probleme und Lösungen, indem Sie deren betriebswirtschaftliche Werte schätzen, etwa in Euro oder in Aufwandsgrößen. Auf diese Weise können Sie Probleme und mögliche Maßnahmen priorisieren – und auch das Management wird diese Prioritäten gut verstehen.

Der Erfahrung des Autors nach tun sich viele Entwickler mit dem Thema „Kosten schätzen" sehr schwer, insbesondere bei den Kosten der Probleme (Issue Cost). Eine einfache Abhilfe ist das Schätzen in Intervallen (Minimum, Maximum), wobei die Breite des Intervalls die Unsicherheit der Schätzung ausdrückt: Große Intervalle drücken hohe Unsicherheit aus, enge Intervalle hohe Zuversicht. Bild 4 zeigt das Vorgehen schematisch. Sie erkennen darin die Kosten sowohl für Probleme wie auch für Maßnahmen.



Probleme („Issues")

ID	Beschreibung	Kosten min	Kosten max	Maß-nahmen
P1	Build/Test Cycle zu lang	€800/T	€3000/T	M2;M3
P2	Billing-Komponente...	€500/W	€2000/W	M1
P3	Falsche Daten in UserTab	€100€/W	€1000/W	M4;M5

Maßnahmen

ID	Beschreibung	Kosten min	Kosten max	Issues
M1	Billing -> MicroServices	€50T	€80T	P2
M2	Hot-Code-Replace Tool	€6000	€8000	P1
M3	Praktikanten testen lassen	€0	€400/M	P1
M4	Autokorrektur für UserTab	€5000	€25000	P3
M5	Manuelle Korrektur UserTab	€4000	€10000	P3
...				

Das Schätzen von Kostenintervallen zeigt, wo in einem Projekt die Unsicherheiten liegen (Bild 4)

Annahmen offenlegen

Bei Schätzungen sind zuerst die Einflussgrößen (Parameter) einer Schätzung zu ermitteln: Was und wer ist überhaupt betroffen? Legen Sie Ihre Annahmen darüber offen und beschaffen Sie sich im Zweifel die Ansichten anderer Stakeholder.

Ein Beispiel aus der Erfahrung des Autors: Ein Entwicklungsteam klagte über unsäglich lange Build-, Deploy-, und Testzyklen. Die Entwickler mussten für jeden Feature-Test zehn bis fünfzehn Minuten auf das Re-Deployment des Systems warten. Um die Kosten dieses Problems zu schätzen, waren zunächst mögliche Parameter zu sammeln:

- Wie viele Entwickler waren betroffen? (Hier: 30 bis 35 Personen)

- Wie häufig bauten/testeten diese jeden Tag? (Hier: im Mittel vier bis sechs Mal)
- Was könnten Entwickler während der langen Wartezeiten alternativ tun? (Hier: höchstens Dokumentation schreiben – aber die war für das System nicht so wichtig.)
- Welche anderen Seiteneffekte würden durch die langen Wartezeiten auftreten? (Hier: Entwickler testen seltener, dadurch werden Fehler später erkannt und benötigen mehr Aufwand bei der Behebung).

Für das Team kamen bei geschätzten Personalkosten von 400 bis 800 Euro pro Tag „Problemkosten" (Cost of Problem) von 2000 bis 8000 Euro täglich zusammen. Das Management hatte das zugrunde liegende Problem lange Zeit als Luxusproblem der Entwickler abgetan. Doch nach der Schätzung stellte es plötzlich auch ein Budget für Verbesserungen zur Verfügung (hier: Anschaffung von Lizenzen für ein Hot-Code-Replace-Framework, das die Build-/Deploy-/Testzyklen in der Entwicklung auf unter drei Minuten gesenkt hat). Die teuren Lizenzkosten hatten sich innerhalb einer einzigen Woche amortisiert.

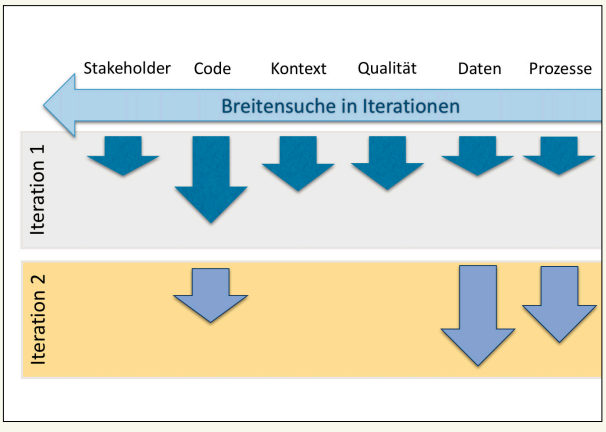
Sie sollten jetzt für einige Ihrer gravierenden oder größeren Probleme jeweils eine (Intervall-)Schätzung der Problemkosten haben – und diese mit dem Management, zum Beispiel den Product Owner, diskutieren.

Beispiele für Analyze-Praktiken

Die Analyze-Phase besteht aus einer breit angelegten Suche nach Problemen. Beginnen Sie mit einer Analyse der wesentlichen Stakeholder und der externen Schnittstellen des Systems. Untersuchen Sie Code, aber betrachten Sie auch weitere Qualitäten des Systems wie Sicherheit, Laufzeitverhalten oder Robustheit, siehe Bild 6.

Einige stichwortartige Beispiele der aim42-Praktiken zur Analyse vermitteln einen Eindruck, wie Sie Probleme finden können:

- Stakeholder-Analyse
- Kontextanalyse
- Analyse des statischen Codes
- Laufzeitanalyse
- Analyse des Entwicklungsprozesses
- Analyse der Dokumentation
- qualitative Analyse (ATAM [6])
- Sicherheitsanalyse



Die Analyze-Phase ist von der breit angelegten Suche nach Problemen geprägt (Bild 6)

Improve: Probleme angehen

In der Improve-Phase von aim42 geht Sie ein oder mehrere Probleme mit konkreten Maßnahmen an. Das ist für die meisten Entwickler der interessanteste Teil, weil es um die Umsetzung konkreter, oft technischer Dinge geht.

Allerdings ist auch das Verbessern selbst in der Praxis oft schwierig, weil das leidige Tagesgeschäft ja ebenfalls erledigt werden muss. Die Krux der Verbesserung liegt also darin, die notwendigen Maßnahmen mit den Notwendigkeiten des Entwicklungsalltags so zu verzahnen, dass sich in beiden Bereichen ein deutlicher Fortschritt ergibt. Der Autor hat Teams erlebt, die beispielsweise 10 bis 20 Prozent der Scrum-Sprints für reine Verbesserungsmaßnahmen reserviert hatten und damit auf längere Sicht ziemlich gute Resultate erzielten.

aim42 differenziert die Improve-Phase in kurzfristige Praktiken und langfristige Ansätze (siehe Bild 5) – man könnte es auch taktische oder strategische Ansätze nennen. Zusätzlich formuliert aim42 noch einige Grundprinzipien, die analog zu den klassischen Entwurfsprinzipien (lose Kopplung, geringe Komplexität, hohe Kohäsion und so weiter) stets gelten sollten, die an dieser Stelle aber keine Rolle spielen.

Taktische Praktiken

Zuerst ein kurzer Blick auf die kurzfristigen Verbesserungsansätze: aim42 führt auch hier wiederum Kategorien zur Strukturierung ein. Einige Beispiele:

- Improve Processes: Optimieren der Anforderungs-, Entwicklungs-, Test- oder Betriebsprozesse, die für das System gelten. Oft sind technische Probleme nur die Symptome tieferliegender Prozessprobleme.
- Improve Architecture and Code: Hierzu gehören insbesondere eine bessere Modularisierung, die Einführung klarer fachlicher/technischer Grenzen sowie die Verbesserung technischer Konzepte. Alleine dieser Punkt umfasst mehr als ein Dutzend detaillierter Praktiken, unter anderem die klassischen Refactorings, aber auch einige Umbauten größeren Stils (siehe Kasten Verbessern von Architektur und Code).
- Improve Technical Infrastructure: Verbesserung der Entwicklungs- und Betriebsumgebung, Betriebssysteme, Middleware, Datenbanken oder Ähnliches.
- Improve Analysability: Hier geht es darum, das System zu instrumentieren, um bessere Problemanalysen (in der Analyze-Phase) zu ermöglichen. Dazu gehört etwa besseres Logging, besonders in verteilten Systemen.

Strategische Planung: Approaches

Manche Verbesserungen lassen sich nur durch langfristige Umbauten, die im großen Stil angegangen werden, erreichen. Auch dafür hält aim42 Praktiken bereit. Der Autor ▶

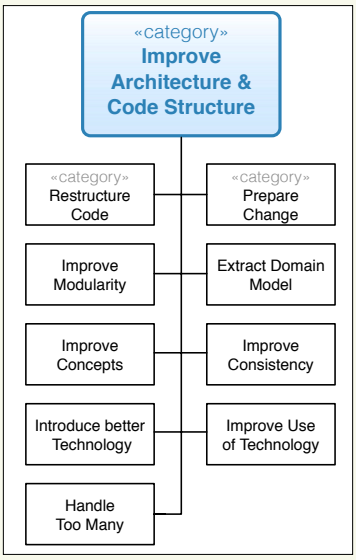
Verbessern von Architektur und Code

Die wohl umfangreichste Sammlung konkreter Praktiken in aim42 gehört zu den Verbesserungsvorschlägen für Code und Architektur, siehe Bild 7. Sie finden hier neben den klassischen Refaktorisierungsansätzen auch Ansätze zur verbesserten Modularisierung sowie zur Optimierung querschnittlicher Konzepte. Hier einige Beispiele:

Prepare Change: Bereiten Sie Code auf Umbauten vor, etwa durch automatisierte Tests oder durch interne Schnittstellen.

Extract Domain Model: Manchmal ist die Fachlichkeit (Domäne) von Systemen durch technische Bausteine, etwa Schichten oder Layer, verborgen. Machen Sie das Domänenmodel sichtbar, separieren Sie Entities, Dienste oder andere fachliche Bestandteile von technischen Teilen.

Improve Consistency: Viele IT-Systeme lösen wiederkehrende Probleme auf viele unterschiedliche Arten und sind deswegen insgesamt schwer verständlich und schwer zu ändern. Vereinheitlichen Sie solche Konzepte. Verbessern Sie die konzeptionelle Integrität, sodass der Code danach wirkt, als wäre er aus einem Guss.



Improve better Technology: Teilweise verwenden Systeme Infrastrukturen oder Frameworks nur noch aus historischen Gründen, obwohl es inzwischen bessere Alternativen gibt. Entledigen Sie sich dieser Altlasten.

Sie finden in Bild 7 außerdem die Kategorie Restructure Code. Dahinter verbirgt sich wiederum eine Vielzahl von Praktiken im Sinne von Refactoring oder Restrukturierung, etwa Introduce Interfaces, Extract Component, Hide unmaintainable Code, Anticorruption Layer, Remove unused Parts, Deprecate obsolete Parts und weitere.

An Verbesserungsvorschlägen bietet aim42 ein umfangreiches Arsenal (Bild 7)

warnt aufgrund vieler Risiken vor einem Neuschreiben des Codes von Grund auf, sozusagen einem „Big Bang“ (siehe dazu Joel Spolsky [3]), aber es geht zum Glück auch anders. Einige Beispiele:

- „Managed Evolution“ [4]: Auch „Value based Improvement“ genannt. Strebt nach einer Balance von Geschäftsnutzen und Verbesserung der Architektur und benötigt wirksame Leistungskennzahlen zur Erfolgsmessung. Für Unternehmen geeignet, die über ein funktionierendes Management der IT-Architektur verfügen.
- Change on Copy: Die Verbesserungen werden auf einer Kopie (einem Klon) des Systems eingearbeitet, um die Risiken zu senken.
- Change by Split: Diese Praktik ist für Systeme mit getrennten Nutzergruppen geeignet. Das System (mitsamt Code und Daten) wird mehrfach kopiert und dann jede Kopie für eine der Nutzergruppen optimiert.
- Strangulate bad Parts [5]: Problematische Teile des Systems werden systematisch über Schnittstellen isoliert und schrittweise ersetzt.

Es gibt noch weitere Praktiken (zum Beispiel Keep Data, Toss Code, Chicken Little, Butterfly) und wahrscheinlich werden Sie für Ihre Organisation und das System eine spezifische Strategie entwickeln müssen. Die genannten Ansätze haben dem Autor zumindest öfter als Gedankenanstöße geholfen.

Fazit

Eine systematische Verbesserung von Softwaresystemen funktioniert auch unter Zeitdruck und sogar mit wenig Budget. Eine methodische Vorbereitung, Iteration und die Übersetzung von Problemen in betriebswirtschaftliche Größen (also Geld) können dabei helfen. Der Open-Source-Ansatz aim42 berücksichtigt dies und bündelt eine Vielzahl etablierter Praktiken, aus denen Sie sich auch für Ihre Systeme und Situationen frei bedienen können. ■

[1] aim42, http://aim42.org
[2] aim42 – architecture improvement, https://github.com/aim42
[3] Joel Spolsky: Things You Should Never Do, Part 1, www.dotnetpro.de/SL1609aim421
[4] Murer, Bonati, Furrer: Managed Evolution. A Strategy for Very Large Information Systems, ISBN 978-3-642-01632-5
[5] Paul Hammant: Legacy Application Strangulation – Case Studies, www.dotnetpro.de/SL1609aim422
[6] Wikipedia: Architecture tradeoff analysis method, www.dotnetpro.de/SL1609aim423

Gernot Starke
(innoQ Fellow) betreibt Softwarearchitektur aus Leidenschaft. Er berät und coacht Kunden aus unterschiedlichen Branchen, ist Initiator und Maintainer von arc42 und aim42 sowie aktives Mitglied und Trainer im iSAQB e.V.
gernot.starke@innoq.com

dnpCode A1609aim42