www.bt-magazin.de 1.2012 Heft 8 Deutschland: € 14,90 Österreich: € 16,80 Schweiz: sFr 29,80 Luxemburg: € 16,90

Business Technology

Business Technology

Architektur & Management Magazin

BT-GEPRÜFT

Expertenwissen für IT-Architekten, Projektleiter und Berater

Qualität

Mythos Qualitätsmanagement

Höhere Qualität, bessere Software

„Funktioniert“ ist nicht genug

4 191858 414906 01

Mit Fokus auf Qualität bessere Software schaffen

Quality-driven Software Architecture

Qualität bildet die Existenzberechtigung für Softwarearchitekten: Zuverlässig, performant, skalierbar und benutzerfreundlich sollen unsere Systeme sein, das alles kosteneffektiv und zukunftssicher. Jeder ITler weiß, dass diese Kombination von Eigenschaften harte Arbeit bedeutet. Lesen Sie hier, wie Softwarearchitekten systematisch Qualität konstruieren können.

AUTOR: DR. GERNOT STARKE

Wollen wir die Qualität eines Systems prüfen, so müssen wir vorher die spezifischen Qualitätsanforderungen der maßgeblichen Stakeholder kennen, allgemeine Anforderungen helfen uns nur bedingt weiter. Auftraggeber und Anwender erwarten von Software heutzutage eine Vielzahl von Qualitätsmerkmalen, von denen die „korrekte Funktion" nur eines unter vielen ist. Im klassischen Softwareentwurf wird allerdings gerade die Funktionalität häufig ins Zentrum der Entwurfs- und Implementierungstätigkeiten gestellt – mit möglicherweise fatalen Folgen. Lassen Sie uns das anhand eines kleinen Beispiels nachvollziehen.

FUNKTION ALLEIN GENÜGT NICHT

Seit dem Siegeszug digitaler Kleinkameras besteht für praktisch jeden von uns die Notwendigkeit, die vielen einzelnen Bilder am Computer irgendwie zu organisieren. Verwenden wir dies als Beispiel einer Anforderungsbeschreibung für ein Softwaresystem: Digitalfotos verwalten. Versetzen Sie sich in die Rolle eines Softwarearchitekten, der von einem Kunden die Anforderungen erklärt bekommt: „Wir möchten Fotos in Ordnern organisieren und mit Schlüsselworten versehen. Später müssen wir nach verschiedenen Suchkriterien (etwa Datum, Schlüsselwort etc.) Bilder suchen und anzeigen können. In unserer privaten Sammlung kommen wir mit einigen tausend Fotos aus, ein gesondertes Mengengerüst (heißt, eine nicht funktionale Anforderung) geben wir daher nicht an.

Ganz einfach könnte unser gedachter Kunde diese Anforderung anhand von **Abbildung 1** erklären: „So etwas wie dort abgebildet hätten wir gerne." Jetzt sind Softwarearchitekten gefragt, aus diesen (zugegeben, sehr groben) Anforderungen eine Lösung zu konstruieren. Dabei hilft ein fachliches Modell im Sinne des Domain-driven Design [1], das wir aus einem exemplarischen Foto (**Abb. 2**) ableiten können. Ein Foto verfügt lediglich über eine Handvoll Attribute (Datum, Ort, Datei etc.). Die Schlüsselworte bilden wir als Liste ab, damit jedes Foto mehrere haben kann. Das Domänenmodell mutet in unserem Beispiel nahezu trivial an (**Abb. 3**). Nur zwei kleine Entitäten genügen, um die gewünschte Funktionalität aus einer rein fachlichen Perspektive abzubilden. Ort, Datum, Ordner und Dateiname bilden wir über Key-Value-Paare in der Klasse *Metadata* ab, die damit gleichzeitig noch die Liste der Schlüsselworte enthält. Einfacher geht's kaum. Softwarearchitekten und Entwickler können auf Basis eines solchen fachlichen Modells ruhig schlafen, weil Domänenmodelle oft hilfreiche Abstraktionen eines Problembereichs darstellen. Auf Basis solcher Modelle können wir oftmals Softwaresysteme sauber strukturieren und entwickeln. In unserem Beispiel müssten wir die klassischen CRUD- Aufgaben (create, read, update, delete) für unsere beiden Domänenentitäten implementieren, zusätzlich noch ein paar technische Aufgaben lösen, um das Fotosystem zum Leben zu erwecken:

Abb. 1: Beispielhafte Anforderungen an Fotomanagement

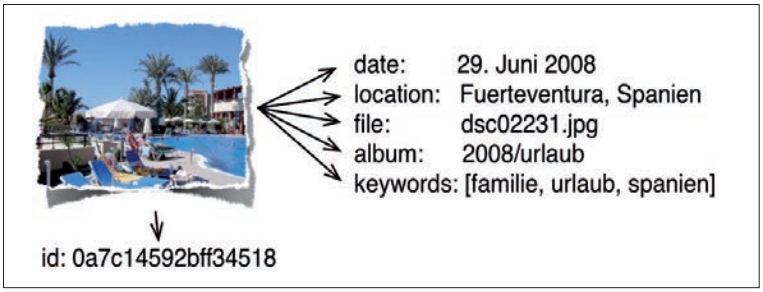

Abb. 2: Foto als Grundlage für Domänenmodell

- Anzeige von *jpg*-Dateien
- Layout und Implementierung einer grafischen Oberfläche

Für alle diese Aufgaben bieten die Standardbibliotheken der bekannten Programmiersprachen (Java, C#, Python etc.) bewährte Konstrukte an, sodass der Implementierung unseres Bei-

spielsystems keine großen Risiken drohen. Geschickte Entwickler können dieses System in kurzer Zeit realisieren. In unserem Beispiel würden Sie als Softwarearchitekt möglicherweise vorschlagen, ein fertiges Werkzeug einzusetzen, also „buy“ statt „make“ [2]. Nehmen wir an, dass Sie unserem hypothetischen Kunden eine dieser fertigen Lösungen präsentieren. Sie erfüllt sämtliche Funktionen, sieht dazu noch ansprechend aus. Der Kunde zeigt sich begeistert.

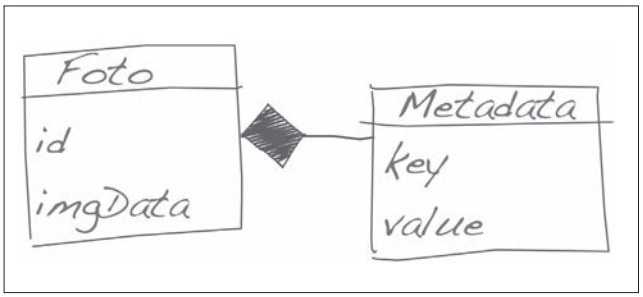

Abb. 3: Domänenmodell für das Fotobeispiel

ABER ...

Ein kleines, unbedeutendes Wort, dieses „aber“. Unser von der Vorführung begeisterter Kunde ergänzt seine Anforderungen: „Wir wollen unser Fotomanagement weltweit ausrollen und jedem Benutzer 50 Gigabyte Speicherplatz bereitstellen. Rechnen Sie mit mehreren Millionen Benutzern. Keine Sorge, die Funktionalität bleibt identisch...“ Zwei nicht funktionale Anforderungen, Benutzerzahl und Datenvolumen kommen also neu auf uns zu. Interessanterweise bleibt das Domänenmodell auch in unserer erweiterten Aufgabenstellung ganz simpel: Eine einzige Domänenklasse kommt dazu, nämlich der Eigentümer (*Owner*) des jeweiligen Bildes (**Abb. 4**). Genau genommen haben wir für diese neue Klasse jetzt auch ein wenig neue Funktionalität zu implementieren, nämlich diese *Owner* anzulegen und zu bearbeiten. Gegenüber den einfachen funktionalen Anforderungen bekommen nun selbst hartgesottene Architekten und Entwickler plötzlich Bedenken: Millionen Benutzer und Petabyte an Daten konfrontieren ein Entwicklungsteam mit völlig neuen Aufgaben:

- Skalierbarkeit, um das System auch bei den geplanten Benutzerzahlen noch performant und reaktiv zu halten. Hierzu gehört Clustering und Load Balancing, um kurzfristig auf Lastspitzen reagieren zu können.
- Speicherung großer Datenmengen (neudeutsch Big Data), was mit konventionellen Datenbanken oder Dateisystemen nicht zufriedenstellend funktioniert.
- Verteilte Datenhaltung, weil aus Performance- und Risikogründen der gesamte Datenbestand über mehrere Standorte oder Rechenzentren verteilt sein sollte.
- Hochverfügbarkeit, weil große Zahlen internationaler Benutzer keine langen Wartungsfenster dulden.
- Übertragungskosten, beispielsweise für die Replikation der Bilddaten.

Ein rein domänenorientiertes Vorgehen bei Entwurf und Entwicklung unseres Fotobeispiels hätte die wesentlichen nicht funktionalen Anforderungen (Benutzeranzahl und Datenvolumen) außer Acht gelassen. Allgemein besteht bei zu starkem Fokus auf Funktionalität und Fachdomäne das Risiko, wichtige Qualitätsmerkmale zu übersehen oder lange Zeit zu ignorieren. Falls Sie der Meinung sind, die hier genannten Qualitätsanforderungen seien ja viel zu groß, weil Sie selbst niemals Millionen von Benutzern haben werden, wählen wir eine ganz andere, neue Anforderung: Ihr Kunde möchte für einige Dutzend Mitarbeiter Bilder speichern, jeweils nur höchstens hundert. Das ergibt insgesamt ein Datenvolumen, das sich problemlos auf einer einzigen Festplatte speichern lässt, und wenige Dutzend Benutzerkennungen können wir auch ohne Probleme managen. Allerdings sollen jetzt die gespeicherten Bilder streng geheim sein – nennen wir das mal „military grade security“. Schon haben wir es wieder mit Problemen jenseits der reinen Funktionalität zu tun: Sichere Verschlüsselungsalgorithmen, sichere Übertragung der Bilder zu den Benutzern, Key-Management oder Authentifizierung der Benutzer. Auch diese Aufgaben lassen sich nur schwer in einer bereits fertigen Software nachrüsten. Kehren wir vom Beispiel zur Frage zurück, was Qualität von Software denn insgesamt bedeutet, und wie wir sie erreichen können. Dazu möchte ich zuerst den Begriff „Qualität“ von Software genauer untersuchen.

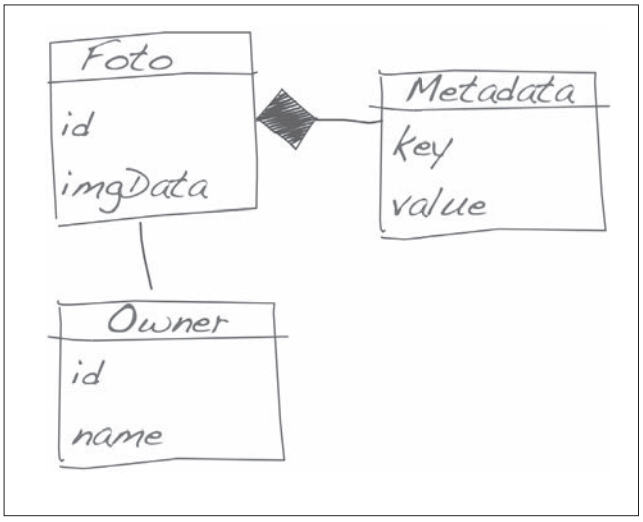

Abb. 4: Erweitertes Domänenmodell

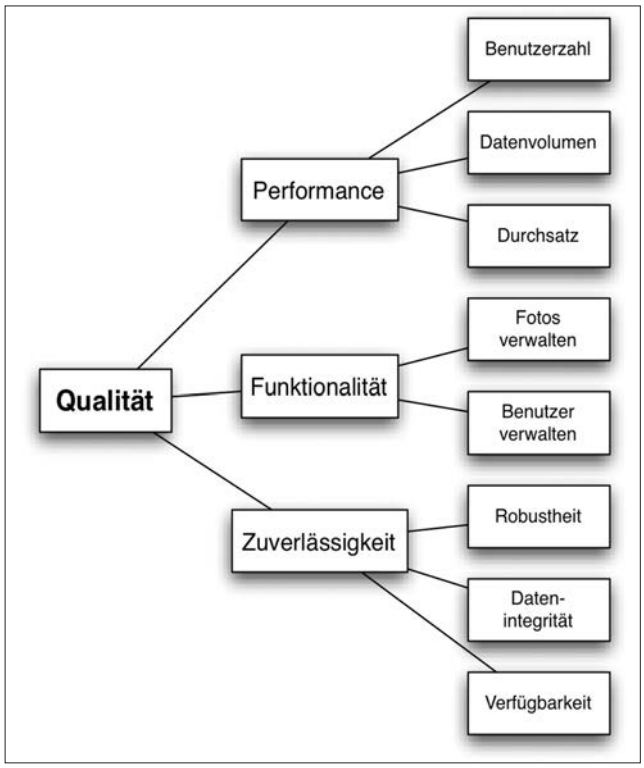

Abb. 5: Qualitätsbaum für das Fotomanagement

WAS IST QUALITÄT

In der Vergangenheit haben sich viele Forscher und Organisationen mit dem Begriff beschäftigt. Dabei sind diverse Definitionen und Qualitätsmodelle entstanden. Allen gemeinsam ist der Ansatz, den komplexen Begriff „Qualität" durch schrittweise Zerlegung zu präzisieren oder zu definieren. Qualität wird dabei in mehrere einzelne (Qualitäts-)Merkmale zerlegt, die ihrerseits wiederum verfeinert werden können. Als Beispiel dient unser Foto-Management-System aus der Einleitung: Wir beschreiben jetzt die (Qualitäts-)Anforderungen in Form eines so genannten Qualitätsbaumes (**Abb. 5**): Die geforderte Softwarequalität wird in Untermerkmale zerlegt, die ihrerseits bei Bedarf weiter zerlegt werden. Es entsteht eine hierarchische Struktur, die spezifische Qualitätsanforderungen an ein System beschreibt.

Der häufig verwendete Terminus „nicht funktionale Anforderungen" als Bezeichnung für Qualitätsmerkmale ist etwas ungenau, weil die gängigen Qualitätsmodelle „Funktionalität" als ein Merkmal enthalten. In **Abbildung 5** steht sie ebenfalls als Qualitätsanforderung vermerkt. Wie können Sie nun diese, häufig recht umfangreiche, Menge an Qualitätsanforderungen für ein System erreichen? Wir haben bereits gesehen, dass eine Konzentration auf die funktionalen oder fachlichen Merkmale, wie im Domänenentwurf vorgeschlagen, zu Problemen mit den übrigen Qualitätsmerkmalen führen kann.

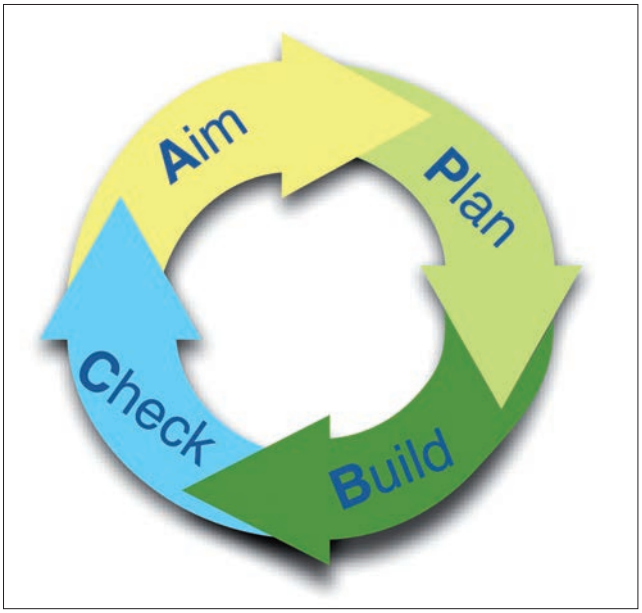

Abb. 6: Aim, Plan, Build, Check: Qualität systematisch konstruieren

QUALITÄT SYSTEMATISCH KONSTRUIEREN

Als Softwarearchitekt müssen Sie bei der Konstruktion und Implementierung Ihrer Systeme also von Beginn an großen Wert auf die Erreichung der wichtigen Qualitätsmerkmale legen. Zum Glück ist das in der Praxis mit ein bisschen Systematik auch ohne neue Werkzeuge gut machbar. Sie sollten erst mal Qualität zu Ihrem wichtigsten Entwurfsziel erheben und ihr dementsprechend Augenmerk widmen. Folgen Sie dann einem klassischen Planen-Handeln-Prüfen-Zyklus (analog dem PDCA aus [3]) mit den folgenden Schritten:

1. **Aim** = zielen: Konkretisieren und priorisieren Sie die notwendigen Qualitätsanforderungen.
2. **Plan** = Maßnahmen planen: Definieren Sie (gemeinsam mit Ihrem Team) Maßnahmenpakete für die jeweiligen Qualitätsanforderungen. Priorisieren Sie auf Basis fachlicher und technischer Gegebenheiten.
3. **Build** = bauen: Setzen Sie die hoch priorisierten Maßnahmen um. Dies bedeutet meistens, Teile des Systems zu implementieren.
4. **Check** = prüfen: Überprüfen Sie die Wirksamkeit Ihrer Maßnahmen. Hierzu müssen Sie möglichst nah am laufenden System oder Teilen davon messen und testen. Bei Bedarf können Sie nun nachsteuern. Fangen Sie dazu beim ersten Schritt wieder an.

Dieses zyklische oder iterative Vorgehen, das in **Abbildung 6** grafisch veranschaulicht wird, erhebt die

Qualitätssteigernde Maßnahmen sind anfänglich teuer, aufwändig oder sogar riskant, zahlen sich langfristig aber aus.

systematische Überprüfung der Zielerreichung zum System [4]. Es stellt sicher, dass hoch priorisierte Qualitätsanforderungen rechtzeitig in der Konstruktion und Entwicklung berücksichtigt werden. Die Build- und Check-Aktivitäten sind ganz normales Architektur- bzw. Projektgeschäft, Aim und Plan bedürfen hingegen einer Erläuterung.

AIM: QUALITÄTSZIELE KONKRETISIEREN

In der idealen Welt bekommen Sie als Softwarearchitekt aus dem Requirements Engineering bereits konkrete und spezifische Qualitätsanforderungen als Vorgabe. Allerdings habe ich in vielen Jahren in der IT-Branche selten ideale Zustände angetroffen, regelmäßig musste ich die Qualitätsanforderungen an Systeme nacharbeiten. Das hat den einfachen Grund, dass die Funktionen und Abläufe eines Systems sich meist recht einfach beschreiben lassen, sich Auftraggeber und andere Stakeholder jedoch mit der konkreten, präzisen Beschreibung von Wartbarkeit, Flexibilität, Robustheit, Ergonomie und Effizienz meist sehr schwer tun. Dabei gibt es doch seit Langem Abhilfe gegen dieses Problem: Methoden wie Qualitätsszenarien ([5], eine deutsche Kurzfassung unter [6]) oder Planguage [7] helfen, Qualitätsanforderungen und -ziele pragmatisch und operationalisiert zu beschreiben. Ich möchte Ihnen die Szenarien, genauer gesagt Qualitätsszenarien, als nützliches Hilfsmittel vorstellen. Weiter oben haben wir den Oberbegriff „Qualität“ bereits in Form eines Qualitätsbaums verfeinert. Nun widmen wir uns den Blättern dieses Baumes. Wir konkretisieren diese Blätter mit Szenarien, um genauer zu beschreiben, was die jeweiligen Stakeholder mit diesem Begriff meinen. Dabei können sich durchaus mehrere unterschiedliche Szenarien auf ein einzelnes Qualitätsmerkmal beziehen. Wiederum sollen Ihnen statt grauer Theorie einige Beispiele (Tabelle 1) diesen Ansatz verdeutlichen. Allgemein beschreiben Szenarien die Reaktion eines Systems auf Ereignisse. Viele Qualitätsmerkmale lassen sich mithilfe von Szenarien konkretisieren, insbesondere Effizienz, Performance, Flexibilität, Erweiterbarkeit und Zuverlässigkeit.

Ich selbst habe gute Erfahrung damit gemacht, die Szenarien in Form einer Mind Map statt eines Baumes zu gruppieren, weil die Zuordnung eines Szenarios zu einem Qualitätsmerkmal oftmals mehrdeutig ist. Mind Maps bieten ein paar zusätzliche Strukturierungs- und Layoutmöglichkeiten und eignen sich hervorragend für interaktive Brainstorming-Workshops, in denen Qualitätsziele mit verschiedenen Stakeholdern erarbeitet werden. Letztlich ist die Anordnung reine Geschmacks-

Merkmal	Untermerkmal	Szenario	Priorität
Zuverlässigkeit	Robustheit	Beim Upload eines korrumpierten *jpg*-Fotos gibt das System einen aussagekräftigen Hinweis ohne Absturz.	B
	Robustheit	Beim Upload eines unbekannten Dateityps (etwa *pdf*, *svg*, *tiff*) gibt das System eine entsprechende Meldung und speichert die Daten ohne Absturz.	A
Zuverlässigkeit	Datenintegrität	Das System wird unter keinen Umständen die von Benutzern hochgeladenen Fotos modifizieren oder beschädigen.	A
Performance	Benutzerzahl	Das System unterstützt bis zu 15 Millionen Benutzer, die jeweils bis zu 50 Gigabyte Speicher für Fotos und Metadaten nutzen dürfen.	B
Benutzerfreundlichkeit	Erlernbarkeit	Gelegenheitsnutzer sollten in weniger als zwei Minuten in der Lage sein, ihr erstes Foto zu speichern.	B
	Erlernbarkeit	Benutzer sollten auch komplexere Suchanfragen ohne Zuhilfenahme der Dokumentation erstellen können.	C

Tabelle 1: Beispiel von Qualitätsszenarien

sache. Bitte beachten Sie in Tabelle 1 die Spalte „Priorität“: Damit bringen wir die Qualitätsanforderungen in eine aus fachlicher Sicht angemessene Reihenfolge. Das fertige System wird sämtliche Qualitätsziele erreichen, die Priorität hilft während Architektur und Entwicklung bei Entwurfs- und Implementierungsentscheidungen. Nun haben wir die Qualitätsziele und -anforderungen geklärt und konkretisiert, jetzt geht es an den nächsten Schritt: Strategie und Taktik zur Erreichung dieser Ziele.

PLAN: QUALITÄTSMASSNAHMEN DEFINIEREN

„Dem Ingeniör ist nichts zu schwör“– das gilt für Daniel Düsentrieb und im Falle der jetzt konkretisierten Qualitätsziele auch für Softwarearchitekten: Kennen wir erst das Ziel, definieren wir spezifische Maßnahmen, Konzepte und Technologie zu seiner Erreichung. Dabei können wir uns neben den Best Practices des Software-Engineering auch auf gängige Strategien und Taktiken (Len Bass hat unter [5] dafür den Begriff „Quality Tactics“ eingeführt) für einzelne Qualitätsmerkmale stützen, brauchen also keinesfalls immer gleich das Rad neu zu erfinden. Bis dato fehlt zwar eine geschlossene Sammlung solcher Qualitätstaktiken, dennoch liefert Ihnen die gängige Literatur einen großen Fundus: Die seit einigen Jahren so verbreiteten Patterns helfen beispielsweise bei den Qualitätsmerkmalen Flexibilität, Robustheit, Performance und Sicherheit [8] etc. gut weiter – sofern Sie selbige mit dem Fokus auf qualitätssteigernde Maßnahmen betrachten. Quality-driven Software Architecture (QDSA) [9] hat den Anspruch, passende Qualitätsmaßnahmen, -taktiken und -strategien systematisch zu sammeln; steckt momentan allerdings noch in den Kinderschuhen. Die Methode QDSA verwendet die vorher definierten Qualitätsziele, um für jedes bekannte Qualitätsszenario oder -merkmal eine Menge möglicher Maßnahmen zu deren Erreichung zu sammeln. Sammeln Sie diese Maßnahmen mit Ihrem Team und schreiben Sie sie auf. Eine einfache Tabelle oder ein Flipchart genügen, sie brauchen keine besonderen Werkzeuge dafür.

Kommen wir ein letztes Mal auf unser Foto-Management-Beispiel zurück und widmen uns auf diese Weise

unserem anspruchsvollen Qualitätsziel „Hohe Benutzerzahl und große Datenmenge“: Folgende Vorschläge könnten positiv auf dieses Ziel hinwirken: NoSQL-Datenbank mit Auto-Replikation, Cluster-Hardware, frühzeitige Lasttests, Caching, Nutzung von Content Delivery Networks, Hardware-Load-Balancer und, und, und... Kein Grund also, vor den Millionen Benutzern zu kapitulieren. Manche dieser Vorschläge sind teuer, andere schwierig umzusetzen. Aber wenn Ihr Kunde hohe Anforderungen hat, müssen Sie manchmal große Sprünge wagen, um diese Ziele zu erreichen. Das Brainstorming, Sammeln und systematische Analysieren von Maßnahmenvorschlägen ist der Kern der Quality-driven Software Architecture. In diesem zentralen Schritt gilt es allerdings, an einer Stelle Vorsicht walten zu lassen: Einzelne Maßnahmen können leicht auf mehrere Qualitätsmerkmale Einfluss nehmen. Eine positive Auswirkung auf ein Merkmal, etwa Performance, kann leicht negative Auswirkungen auf andere Merkmale haben, etwa Speicherbedarf, Robustheit oder Einfachheit. Sie sollten vorab eine Konsequenzanalyse durchführen, ehe Sie mit der Umsetzung Ihrer Qualitätsmaßnahmen beginnen. Es hat sich nach meiner Erfahrung bewährt, solche potenziellen oder bekannten Konsequenzen direkt mit den Qualitätsmaßnahmen zu den jeweiligen Szenarien oder Qualitätszielen zu dokumentieren, das heißt sie auf jeden Fall schriftlich festzuhalten. Was jetzt noch fehlt, ist klassisches Projektgeschäft, nämlich die Maßnahmen im Projektalltag einzuplanen und anschließend umzusetzen. Ob Sie nun nach Scrum, dem V-Modell, RUP oder anderen Vorgehensmodellen arbeiten – bezüglich QDSA gibt es hier keinerlei Besonderheiten gegenüber normalem Vorgehen.

QUALITÄT HAT WERT

Qualität auf diese Weise systematisch zu konstruieren, schafft für Ihre Stakeholder Werte: Softwaresysteme, die spezifische Qualitätsziele erreichen. Wie oben bereits erwähnt, sind viele qualitätssteigernde Maßnahmen anfänglich teuer, aufwändig oder sogar riskant hinsichtlich anderer Ziele. Langfristig hingegen zahlt sich eine solche Investition in die Erreichung von Qualitätszielen praktisch immer aus. Bei reinen Kurzfrist-Managern ist solche Qualitätsorientierung daher oftmals unbeliebt, weil manche Investitionen sich eben nicht unmittelbar, sondern erst mit etwas Verzögerung amortisieren. Denken Sie daran, dass Qualität die Existenzberechtigung von Softwarearchitekten ist. Insbesondere die nicht funktionalen Merkmale lassen sich ohne systematische Konstruktion (Architektur) kaum erreichen. Und wenn Software-Engineering einfach wäre, würden es schon lange die Maschinen für uns erledigen.

Links & Literatur

[1] Evans: Domain-driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2003. Der ausführliche Klassiker zum Thema

[2] Mögliche Kandidaten wären hier: Google Picasa (http://picasa.google.com), xnview (http://www.xnview.de), Apple iPhoto (http://www.apple.com/de/ilife/iphoto) oder auch die Open-Source-Lösung PhotArk der Apache Software Foundation (http://incubator.apache.org/photark)

[3] Demings Plan-Do-Check-Act Zyklus: http://de.wikipedia.org/wiki/Demingkreis

[4] Hruschka, Starke: Knigge für Softwarearchitekten. Entwickler.press, 2012

[5] Bass et. al: Software Architecture in Practice. Addison-Wesley 2003

[6] Starke: „Effektive Software-Architektur – ein praktischer Leitfaden“, Hanser Verlag, 5. Auflage 2011

[7] Tom Gilb: http://www.gilb.com/tiki-download_file.php?fileId=39. Eine deutschsprachige Darstellung in: Chris Rupp und die Sophisten: Requirements Engineering und –management, Hanser Verlag, 5. Auflage, 2009

[8] Diverse Bücher zu Patterns aus der Wiley Software Pattern Serie. Insbesondere Buschmann et. al: A Pattern Language for Distributed Computing. Wiley, 2007, Hanmer: Patterns for Fault Tolerant Software, Schumacher et. al: Security Patterns, Kircher et. al: Patterns for Resource Management.

[9] Quality Driven Software Architecture (QDSA), online: http://www.qdsa.org

[10] Hofmeister et. al: Applied Software Architecture. Addison-Wesley 1999. Verfolgt konzeptionell einen ähnlichen Ansatz wie QDSA, jedoch unter der Bezeichnung Global Analysis

Dr. Gernot Starke

(innoQ. Diplom-Informatiker, Promotion über Software-Engineering) hat mehr als 20 Jahre Berufserfahrung in Konstruktion und Implementierung von IT-Systemen in unterschiedlichen Branchen. Er ist spezialisiert auf Softwarearchitektur und iterative Entwicklungsprozesse, Gründungsmitglied des iSAQB e. V. sowie Mitgründer und Committer von arc42, dem Portal für Softwarearchitektur. Autor mehrerer Bücher zu Softwarearchitektur und Entwicklungsprozessen.