Ramon Anger
(ramon.anger@capgemini.com)
arbeitet als Technischer Architekt im Bereich Public der Capgemini Deutschland GmbH. Er beschäftigt sich insbesondere mit neuen Technologien, Softwarearchitektur und agilen Methoden.

Matthias Ehlert
(matthias.ehlert@capgemini.com)
arbeitet als Enterprise Architekt im Bereich Public der Capgemini Deutschland GmbH. Er beschäftigt sich mit der Gestaltung von Anwendungslandschaften, leichtgewichtigem Enterprise Architekturmanagement und Lösungsarchitekturen im Kontext von BPM.

Änderbarkeit auf Lebenszeit: Lean-Prinzipien in der Architektur

Eine gute Architektur ist leicht verständlich. Die Software, die sie beschreibt, kann mit vertretbarem Aufwand erstellt und angepasst werden, und zeigt ein angemessenes Laufzeitverhalten. Die leichte Änderbarkeit einer Software ist eine wesentliche Eigenschaft, die von ihrer Architektur unterstützt werden muss. Das zeigt sich daran, dass über die Hälfte der Kosten einer Software während ihrer Wartungsphase entsteht, also wenn die Software bereits produktiv eingesetzt wird. Dieser Artikel beschreibt mehrere Ansätze, die bei der Erstellung und Pflege von einfacher Softwarearchitektur unterstützen.

Wie muss eine Architektur gestaltet sein, die eine leichte Änderbarkeit der Software, die sie beschreibt, ermöglicht? Aus unserer Sicht sollte sie in erster Linie so einfach wie möglich sein. Zur Erstellung und Erhaltung einer einfachen Architektur ist ein kontinuierlicher Prozess über den gesamten Lebenszyklus einer Anwendung hinweg erforderlich.

In diesem Artikel beschreiben wir mehrere Ansätze, die bei der Erstellung und Pflege von einfacher Softwarearchitektur unterstützen. Die beschriebenen Ansätze wenden Prinzipien aus dem *Lean Management* an. Lean Management ist ein System zur Optimierung von Wertschöpfungsketten aus Sicht von Kunden und Anbietern, wobei überflüssige Tätigkeiten vermieden werden. Zu den Lean-Prinzipien zählt zum Beispiel, Überflüssiges zu vermeiden und zu entfernen, sich kontinuierlich zu verbessern sowie das Team zu Entscheidungen zu befähigen. Durch die Anwendung dieser Prinzipien entstehen effiziente Prozesse, die flexibel auf Veränderung reagieren können. Im Ergebnis entsteht eine schlanke Architektur, die eine leichte Änderbarkeit der Software unterstützt.

YAGNI versus BUFD

Eine einfache Architektur ist in erster Linie eine Architektur ohne Verschwendung. In agilem Umfeld arbeiten wir nach dem Ansatz: *You Ain't Gonna Need It (YAGNI)* – du wirst es nicht brauchen. Die Architektur entwickelt sich hier schrittweise, getrieben durch konkrete Anforderungen. Architekturentscheidungen werden damit so spät wie möglich getroffen. Die Lösung ist, bezogen auf das konkrete Problem, optimal. Folgt man Kent Becks Aus-

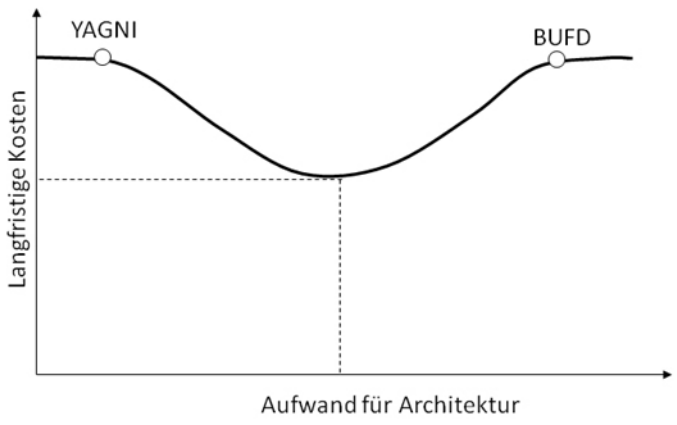

Abb. 1: Gegenüberstellung von YAGNI und BUFD (vgl. [Cop10]).

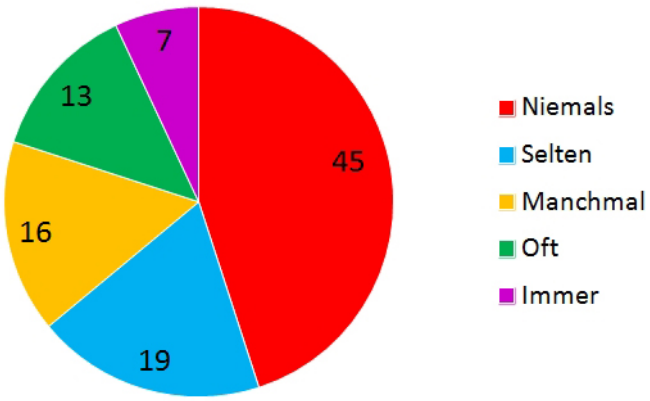

Abb. 2: Wie häufig werden Features eines Systems durchschnittlich genutzt (vgl. [Sta94])?

spruch, „Do the simplest thing that might possibly work“ (vgl. [Bec99]), gibt es keine Verschwendung. Bei diesem Ansatz besteht allerdings das Risiko, dass sich die Lösung bei sich verändernden Anforderungen nur schwer anpassen lässt.

Das *Big Up-Front Design (BUFD)* legt die Architektur im Vorfeld der Umsetzung komplett fest. Der Auftraggeber will vor Beginn des Baus einen vollständigen Bauplan haben. Hierbei müssen häufig Annahmen über noch unbekannte Anforderungen getroffen werden. Unsere Erfahrung zeigt, dass diese Annahmen nur bedingt zutreffen. Die BUFD-Architektur enthält damit im positiven Fall nicht benötigte Teile, die die Komplexität des Systems unnötig erhöhen. Im negativen Fall erschweren die vorgegebenen Strukturen die Änderbarkeit des Systems massiv.

So wurden beispielsweise beim Design eines UML-basierten Entwicklungswerkzeugs die Aktivitätsdiagramme sehr restriktiv im Sinne der UML gestaltet. Das erste Feedback des Marktes ergab, dass diese restriktive Auslegung in der täglichen Projektarbeit nicht praktikabel war. Die an vielen Stellen im Design eingebauten Restriktionen mussten aufwendig wieder entfernt werden. Die Auslieferung einer verbesserten Version wurde dadurch um mehrere Monate verzögert.

Über einen längeren Zeitraum macht es sich bezahlt, einen Mittelweg zwischen BUFD und YAGNI zu finden (siehe Abbildung 1). Aus unserer Sicht ist das ideale Maß für vorausschauende Planung und kurzfristige Entscheidung vom konkreten Problem abhängig und kann nicht pauschal bemessen werden. Bewährt hat sich, dass zu Beginn der Entwicklung die wesentlichen Architekturziele und die daraus abgeleiteten Prinzipien festgelegt werden. Sie definieren einen stabilen Rahmen für die späteren Architekturentscheidungen und reduzieren damit das Risiko von Umbauten. Zu diesen Prinzipien gehört beispielsweise, dass die stabilen Informationsmodelle und die Fachlogik von der flexibleren Prozesslogik entkoppelt werden oder dass Geschäftsregeln so beschrieben und gekapselt werden, dass sie über das gesamte System hinweg wiederverwendet werden können. Als Beispiel sei hier ein Softwareprodukt genannt, das in einer ersten Version als Rich-Client-Lösung implementiert wurde. Das Prinzip einer vollständig logikfreien Präsentationsschicht und dessen strikte Einhaltung ermöglichten eine schnelle Umstellung des Produkts auf eine Architektur, bei der im Server die Präsentationssichten generiert wurden und der Anwender mit einem Thin Client auskommt.

Wir Architekten und Entwickler neigen dazu, Lösungen komplexer zu gestalten, als es erforderlich ist, und müssen uns selbst an die Angemessenheit unserer Lösungen erinnern: „Lass es! Denk darüber nach, wenn diese Erweiterung angefordert wird. Vorher wirst du sie nicht brauchen.“ Manchmal entscheiden wir uns gegen die Vernunft. Wenn die betroffene Funktionalität tatsächlich einmal angepasst wird, was unserer Erfahrung nach in etwa 20 Prozent der Fälle geschieht, ist unsere vorhergesehene Erweiterung unbrauchbar – und dies wiederum in fast 100 Prozent aller Fälle. Änderungen finden grundsätzlich dort statt, wo wir es uns nicht vorstellen können.

Angemessenheit der Lösung

Ein häufig beobachtetes Muster beim Entwurf eines Systems ist, dass die Angemessenheit der zu implementierenden Features nicht hinterfragt wird. Wenn folgende Punkte nicht abgeklärt werden, ist beispielsweise die Wahrscheinlichkeit hoch, dass die Features des Systems dem in Abbildung 2 dargestellten Schema entsprechen:

- Sind die zu unterstützenden Prozesse optimal auf die Leistungserbringung zugeschnitten?
- Befasst sich das System mit mehr als einer Aufgabenstellung?
- Enthalten die Anforderungen die notwendigen Features, gibt es überflüssige Features oder fehlen Features?
- Ist das System klar von seiner Umwelt abgegrenzt?

Die Erhebung in Abbildung 2 stammt aus dem Jahr 1994. Sie mag veraltet erscheinen, unserer Erfahrung nach sind die Zahlen immer noch aktuell. Im Kontext *Lean Architecture* erscheint folgende Interpretation angebracht: Zwischen 20 und 36 Prozent der Features eines Systems werden wirklich benötigt. Nur diese Features sind wichtig. Die Komplexität des Systems reduziert sich auf etwa ein Drittel der Gesamtmenge der Features, was jedoch nicht mit einem Drittel des angenommenen Umfangs verwechselt werden darf. Diese verringerte Komplexität beeinflusst hoffentlich die Architektur des betroffenen Systems, da sie mit dem Wissen um die wichtigen Features ebenfalls weniger komplex ausfällt.

Der Schnitt zwischen wichtigen und unwichtigen Features lässt sich im Vorfeld einer Systemimplementierung nur schwer zuverlässig ermitteln. Ein iteratives Vorgehen, bei dem nach jeder Iteration erneut abgewogen wird, ob weitere Features fehlen, und wenn ja, welche, stellt sicher, dass das System nicht von Beginn an überfrachtet ist. Ein konsequentes Ausbauen nicht benötigter Features würde diesen Zustand über den gesamten Software-Lebenszyklus hinweg erhalten. Beim zweiten Punkt hilft uns ein angemessenes Logging. Dieses kann uns verraten, welche Funktionalität in einem System tatsächlich verwendet wird.

In einem Projekt konnten wir den Kunden mit Hilfe von Log-Informationen über einen längeren Zeitraum davon überzeugen, dass es sich lohnt, über die Notwendigkeit neuer Funktionalität auf Basis von Nutzungshäufigkeiten nachzudenken. Auslöser war eine aufwendig hergestellte Funktionalität, die im Verlauf von zwei

Jahren von insgesamt etwa 500 Benutzern genau einmal verwendet worden war. Informationen über die Nutzungshäufigkeit einer Funktionalität sind aus unserer Sicht eine sinnvolle Ergänzung zum Geschäftswert-Ansatz in der agilen Welt: eine Funktionalität nur dann zu entwickeln, wenn für das betroffene Softwaresystem ein Nutzen entsteht.

Evolutionäres Design

Evolutionäres Design zeichnet sich durch einen regelmäßigen Abgleich zwischen Problem- und Lösungsdomäne aus (siehe Abbildung 3). Die Problemdomäne stellt im Kontext mit Architektur der Verbund aus Anforderungen und Rahmenbedingungen dar. Die Lösungsdomäne umfasst alle Aktivitäten, die mit der Erfüllung dieser Anforderungen verbunden sind.

Während des Designs kommt es nicht darauf an, ein Problem vollständig zu beschreiben und anschließend eine Lösung für dieses Problem zu suchen. Ziel ist es, Problem und Lösung kontinuierlich zu entwickeln und so dafür zu sorgen, dass sich Problem- und Lösungsraum mit fortschreitender Entwicklung immer mehr annähern (vgl. [Bro11]).

Evolutionäres Design ist ein *Domain Driven Design*, das heißt, dass dabei die Designentwicklung maßgeblich von der umzusetzenden Fachlichkeit der Anwendungsdomäne beeinflusst wird. Neben neuen fachlichen Komponenten in der Anwendung müssen dann häufig neue Kommunikationskanäle und weitere externe Schnittstellen eingebunden werden.

Als Beispiel sei hier der Hersteller eines Tools zur Unterstützung des Projektmanagements angeführt. Bei den ersten Softwareversionen lag der fachliche Schwerpunkt auf der Planung von Projekten. Da in diesem Kontext eine sehr gute Useability erforderlich ist, wurde ein Rich Client für das Tool entwickelt. In späteren Versionen sollte auch das Controlling von Projekten unterstützt werden. Die dazu bereitgestellte Erfassung von Zeiten und Aufwand der Projektmitglieder erforderte nur eine einfache Benutzungsoberfläche. Diese wurde als Web-Client umgesetzt.

Refactoring

Unter Refactoring verstehen wir den Prozess zur Anpassung der inneren Struktur beziehungsweise des inneren Verhaltens einer Software, während ihr äußeres Verhalten unverändert bleibt. Diese interne Anpassung erfolgt zur Verbesserung des Softwaredesigns (vgl. [Fow05]).

Ändern sich Anforderungen oder kommen neue Anforderungen hinzu, muss der bestehende Code häufig angepasst werden, damit das System diesen neuen oder geänderten Anforderungen gerecht werden kann. Wird dabei ausschließlich Wert auf die Realisierung der Anforderungen gelegt, leidet die Qualität des Codes. Die Entwickler hinterlassen im System technische Schulden. Die Wartbarkeit der Software verschlechtert sich, Änderungen können im Lauf der Zeit nur mit immer größerem Aufwand vorgenommen werden und die Auswirkungen sind kaum noch abschätzbar. Spätestens jetzt sind Refactorings zwingend erforderlich, um die Software wieder in einen wartbaren Zustand zu überführen.

Refactorings sollten kontinuierlich und geplant erfolgen. Die für das Refactoring notwendige Zeit geht von der Zeit für funktionale Veränderungen ab. Aus unserer Sicht darf die Veränderung der inneren Struktur einer Software mit Refactoring nicht mit einer Änderung der Fachlichkeit verbunden werden, da das Risiko neuer Fehler zu hoch erscheint. Der dafür notwendige Zeitrahmen innerhalb eines Inkrements bzw. Releases muss also überschaubar bleiben.

Voraussetzung für jedes Refactoring sind angemessene automatisierte Tests. Neben aufwendigen manuellen Tests sind sie die einzige Möglichkeit, näherungsweise sicherzustellen, dass die Software nach dem Refactoring noch das kann, was sie davor geleistet hat.

Wir wurden von einem Kunden darum gebeten, sein strategisch wichtigstes System zu erweitern. Damit waren tiefe Einschnitte in die bestehende Softwarearchitektur des Systems verbunden. Unsere erste Idee war ein Refactoring des Systems. Da aber keinerlei automatische Tests existierten, wäre nach der Erweiterung nur ansatzweise nachweisbar gewesen, ob das System nach dem Refactoring funktional noch wie gewohnt arbeitet. Eine Alternative war es, die fehlenden Tests sehr aufwendig nachzuziehen und mit ihnen die Wirkung des Refactorings sichtbar zu machen. Beide Varianten kamen für uns nur bedingt in Frage. Wir überzeugten den Kunden von einer Neuentwicklung des Systems – diesmal mit einem testgetriebenem Vorgehen.

Architekturwerkzeuge: Weniger ist mehr

Softwareentwicklung und damit auch der Architekturentwurf spielt sich in einer Welt voller Tools ab. Jedes Werkzeug bringt dabei ein Stück weit eine eigene Methodik und Architektursichten mit. Diese innerhalb des Werkzeugkastens abzustimmen, erfordert oft mehr Aufwand als der damit verbundene Nutzen. Der Architekt muss deshalb in einem ersten Schritt die für die beteiligten Stakeholder notwendigen Architektursichten bestimmen. Die Werkzeuge für den Architekturentwurf müssen diese Architekturansichten auf dem richtigen Abstraktionsniveau und in einer für die Stakeholder verständlichen Notation unterstützen. Der Architekturentwurf ist in vieler Hinsicht auch ein Kommunikationsprozess.

Häufig führt ein Werkzeug-fokussiertes Entwicklungsvorgehen zu einer unnötigen Detailtiefe in der Betrachtungsweise, durch die aber keinerlei Nutzen für we-

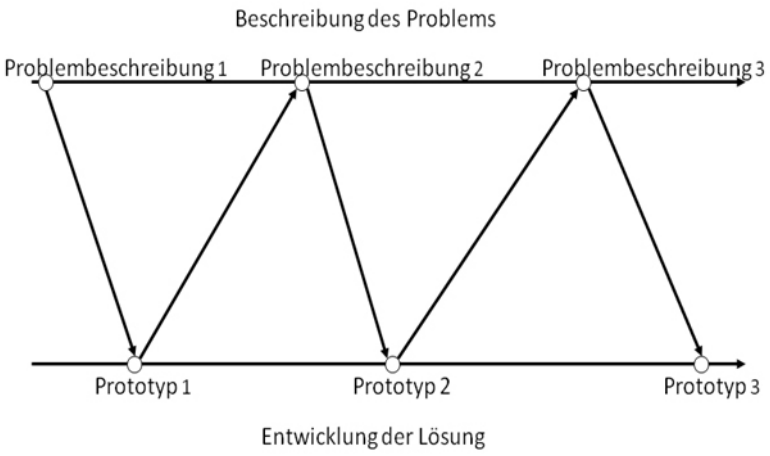

Abb. 2: Koevolutionsmodell des Designs nach Maher, Poon und Boulanger von 1996 (vgl. [Bro11]).

sentliche Designentscheidungen entsteht. Die durch das Modellierungswerkzeug angebotene Verfolgbarkeit auf Attributebene zwischen Analyse- und Designmodell hilft nicht, wenn man mit dem Tool nicht in der Lage ist, aus dem Modell die wesentlichen Informationsobjekte und deren Beziehungen zu identifizieren.

Unserer Erfahrung nach haben Tools kaum Auswirkungen auf die Qualität der Architektur. Daher bleibt dem Architekten aus unserer Sicht wenig anderes übrig, als den eigenen Verstand für die Lösung von Problemen beim Architekturentwurf zu nutzen. Nicht nur im Kontext schlanker Architektur ist es ein valider Ansatz, Dinge zu hinterfragen und eine Optimierung voranzutreiben. Legt man zu viel Wert auf den Einsatz von Tools, besteht die Gefahr, die Einschränkungen des Werkzeugs als die Grenzen der Anpassungsfähigkeit eines Prozesses, einer Idee oder einer Methode zu begreifen.

James Coplien, Autor eines Buchs zum Thema "Lean Architecture", meint dazu recht treffend: "Lean doesn't mean less work but more thinking" (vgl. [Cop10]).

In einem unserer agilen Projekte setzten wir User-Storys zur Anforderungsbeschreibung ein. Bei mehreren fachlichen Sachverhalten mit vielen Beziehungen zwischen User-Storys stießen wir aber an die Grenzen dieses Werkzeugs. Für einen der fachlichen Aspekte gab es beispielsweise 24 alternative Teilabläufe mit vielen Querbeziehungen zwischen den insgesamt knapp 100 User-Storys. Dieses Geflecht war mit dem Tool für User-Storys nur schwer beherrschbar. In der Not versuchten wir, diesen fachlichen Aspekt mit klassischen Use-Case-Beschreibungen zu formulieren, und erhielten ein Geflecht aus sieben Use-Cases auf insgesamt knapp 30 A4-Seiten. Verglichen mit den User-Storys waren die Beziehungen in dieser Darstellung gut verständlich. Seitdem empfehlen wir Kollegen und Kunden, die in agilen Projekten zwischen User-Storys und Use-Cases wählen sollen, immer, diese Entscheidung von der Beziehungskomplexität fachlicher Aspekte abhängig zu machen.

Architekturdokumentation

In Softwareentwicklungsprojekten entsteht ein großer Teil der Kosten bei der Dokumentation – wobei wir darunter eine entwicklungsnahe Dokumentation, wie Grob- oder Feinkonzept, aber auch Dokumentation der Software- beziehungsweise Systemarchitektur, wie Big Picture oder Architekturübersicht, Grobarchitektur- und Detailarchitekturbeschreibung, verstehen. Je besser eine Software dokumentiert ist, desto leichter kann sie potenziell von einem neuen Entwickler oder Architekten verstanden werden.

Ist eine Software mangels Dokumentation nicht leicht verständlich, steigt der Kommunikationsaufwand, um die fehlenden Informationen auf anderem Weg zu erlangen. Werden beispielsweise Architekturentscheidungen nicht festgehalten, besteht das Risiko, dass für gleichartige Probleme jedes Mal erneut abgewogen wird, welcher Weg eingeschlagen werden soll.

Auf der anderen Seite stellt Dokumentation, die ihren Zweck nicht erfüllt, Verschwendung dar, die nicht nur im Kontext von Lean vermieden werden sollte. Ziel muss es also sein, eine bedarfsgerechte Architekturdokumentation zu betreiben, wobei der Aufwand für jedes Projekt individuell abgewogen werden muss. Nach unserer Erfahrung lohnen sich vor der Erstellung jeder Architektursicht folgende Fragen:

- Wer benötigt diese Information?
- Was ist das Minimalset an Information, das dem Empfängerkreis helfen kann?
- Kann dieser Personenkreis die Information auch aus einer anderen Quelle erhalten?
- Ist die Information an der richtigen Stelle und redundanzfrei dokumentiert?

Wenn wir für ein Architektur-Artefakt keinen Interessenten identifizieren können, fangen wir gar nicht erst mit der Erstellung an.

In einem Projekt wurden die Geschäftsregeln, die sich auf einzelne bzw. Gruppen von Informationsobjekten bezogen, in den Geschäftsfunktionen dokumentiert. Die Regeln waren damit weder konsistent noch redundanzfrei. Dieses Problem wurde erst in der Designphase identifiziert und musste in einem aufwendigen, ungeplanten Zwischenschritt behoben werden. Erst danach konnte ein korrektes Design erstellt werden.

Aus unserer Sicht ist es in jedem Fall sinnvoll, die Notwendigkeit zur Erstellung von Architekturdokumentation weiter zu reflektieren: Statt sklavisch ein gefordertes Template auszufüllen, weil es zum definierten Prozess gehört, lässt es sich vielleicht auf die Teile reduzieren, die im aktuellen Kontext wirklich wichtig sind. Einen pragmatischen Ansatz zur Architekturdokumentation unter Berücksichtigung von YAGNI bietet beispielsweise „arc42" (vgl. [Hru13]). arc42 beschreibt einen einfachen und flexiblen Prozess zur Erhebung, Dokumentation und Kommunikation von Software- und Systemarchitekturen.

Continuous Integration and Deployment

Die leichte Auslieferbarkeit einer Software steht unserer Meinung nach in Beziehung zu ihrer leichten Änderbarkeit. In der Praxis treffen wir noch sehr häufig diese Situation an: Der Aufwand für die Fehlerbehebung beträgt nur wenige Tage, der für den Test, die Erstellung und Lieferung des neuen Release hingegen mehrere Wochen. Die Ursachen liegen oft darin, dass:

- kein automatischer Build möglich ist,
- die Software manuell installiert werden muss,
- kein automatischer Regressionstest möglich ist,
- der Lieferprozess viele Formalismen und Beteiligte enthält.

Der Build- und Lieferprozess setzt dabei auf Artefakten auf, die direkt oder indirekt aus der vorliegenden Architektur entstanden sind. Der Architekturentwurf muss bereits berücksichtigen, in welcher Form eine Software für ihre Nutzung zur Verfügung gestellt werden kann. So sollten zum Beispiel Konfigurationsparameter weitestgehend standardisiert werden.

Unter *Continuous Integration and Deployment* wird ein Prozess verstanden, der eine Software automatisiert baut, testet und auf einem Zielsystem ausliefert. Unserer Erfahrung nach erfordert die Einrichtung eines solchen Prozesses bei komplexeren Systemen signifikanten Aufwand. Für das Projekt müssen der erforderliche Aufwand und die notwendigen Experten so eingeplant werden, dass der Prozess zu Beginn der Umsetzung zur Verfügung steht.

Continuous Deployment wird in agilen Projekten immer häufiger zur Realität. Wir stoßen aber auch auf agile Projekte, bei denen das Team die letzten Arbeitstage jeder Iteration für manuelle Builds, Tests und Deployment nutzt.

Die Kosten jeder einzelnen Lieferung betrugen in einem Projekt zwölf Tage – etwa 15 Prozent der Kapazität des Teams. Das Argument für die manuellen Tests

lautete: keine Zeit zur Erstellung automatisierter Tests. Also führte jeder der operativen Projektmitarbeiter am Ende einer dreiwöchigen Iteration etwa 1,5 Tage lang Tests durch. Da die Funktionalität des Projekts stetig wuchs, wuchs auch der Aufwand für den Test neuer und den Regressionstest bereits bestehender Funktionalität. Eine Schätzung für das Nachziehen der automatisierten Tests zum aktuellen Zeitpunkt ergab einen Aufwand von etwa 280 Tagen, also drei Iterationen, in denen keine Funktionalität geliefert, sondern lediglich die Voraussetzung für *Continuous Integration and Deployment* geschaffen wurde. Da es sich um ein für den Auftraggeber wichtiges System handelte, wurde entschieden, die Testfälle nachzuziehen und jegliche Weiterentwicklung an die Erstellung automatisierter Testfälle zu koppeln. Die Alternative – das Stoppen des Projekts – war für den Auftraggeber nicht tragbar.

Das nachträgliche Injizieren der Voraussetzung für *Continuous Integration and Deployment* – automatisierte Tests – kann beliebig aufwendig werden. Wir empfehlen, sie in jedem neuen Projekt von Beginn an zu schaffen und ihre Aufrechterhaltung der Schaffung neuer Funktionalität überzuordnen.

Fazit

Zur Erstellung und Erhaltung einer einfachen Architektur ist ein kontinuierlicher Prozess über den gesamten Lebenszyklus einer Anwendung erforderlich. Die Lean-Prinzipien bieten eine gute Basis, diesen Prozess erfolgreich auszuführen und dabei fortlaufend zu verbessern. Die in diesem Artikel dargestellten Ansätze wenden die Lean-Prinzipien an. Wir empfehlen, die Lean-Prinzipien als Leitlinien bei Veränderungen der Architektur- und Entwicklungsprozesse zu nutzen. Das führt nach unserer Erfahrung zu besseren Ergebnissen als beispielsweise die unreflektierte Einführung von derzeit gängigen agilen Frameworks ohne Fokus auf den Aspekt Architektur. ■

Literatur & Links

[Bec99] K. Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley1999

[Bro11]F.P. Brooks, Erfolgreiches Design, MITP 2011

[Cop10] J. Coplien, Lean Architecture for Agile Software Development, Wiley & Sons 2010

[Fow05] M. Fowler, Refactoring -- oder: wie Sie das Design vorhandener Software verbessern, Addison-Wesley 2005

[Hru13] P. Hruschka, G. Starke, ar42 – Resourcen für Software-Architekten, 2013, siehe: http://www.arc42.de/

[Pop07] M. Poppendieck, Implementing Lean Software Development – From Concept To Cash, Addison-Wesley 2007

[Sta94]Standish Group, 1994, siehe: http://blog.standishgroup.com/

Der Beitrag wurde in der Printausgabe 05/2013 des OBJEKTspektrums erstmals veröffentlicht.