EMERGENTE ARCHITEKTUR: DER MACHTLOSE ARCHITEKT

der autor

Matthias Bohlen
(E-Mail: mbohlen@mbohlen.de)
ist unabhängiger Coach für Softwareentwicklungsteams, die hochproduktiv werden wollen. Er ist Mitglied des OBJEKTspektrum-Redaktionsteams.

Die Softwarearchitektur ist ein wichtiges Artefakt der Softwareentwicklung. Softwarearchitekten müssen kommunikationsstark sein, für die Durchsetzung der Architektur in den Projekten sorgen usw. Das ist die öffentliche Meinung. In der Projektrealität sieht es meist anders aus: Architekturdokumente werden mühsam erstellt, doch im Alltag nicht gelesen. Architekten versuchen, durch Predigen die Einhaltung der Architektur zu erreichen. Teams entwickeln, auch ohne auf die Architektur zu achten. Was wäre, wenn man Architektur emergieren lassen würde, quasi als „Ergebnis des Schwarms", anstatt sie mit viel Kraft in Einzelleistung zu erstellen? Die folgende Geschichte ist ein Kondensat meiner Erfahrung aus vielen Projekten. Lesen Sie, warum Architektur letztlich nicht von Einzelnen gemacht werden kann, und erfahren Sie, was nötig ist, damit Emergenz möglich wird.

Der Neustart

Ludwig Angerer war der Leiter der Abteilung Anwendungsentwicklung. In seiner Abteilung fand das strategisch wichtige Projekt statt. Als er eines Abends über die Flure seiner Abteilung spazierte und den Mitarbeitern zusah, die in den Zimmern noch arbeiteten, fiel ihm die Stille dort auf, und er fing an, über den aktuellen Stand seines Projektes nachzudenken.

Dem Projekt ging es nach dem erneuten Start relativ gut, aber eben nur relativ. Das Vorprojekt war gescheitert, man hatte die „Lessons Learned" dokumentiert und das jetzige Folgeprojekt gestartet. Angerer hatte sich bemüht, es diesmal „richtig zu machen", und darauf geachtet, jedem Mitarbeiter seine Ziele genau zu erklären, sie messbar zu machen und aufzuschreiben. Doch seine Intuition sagte ihm, dass etwas nicht in Ordnung war. Sie waren schon sechs Monate unterwegs, doch die Architektur des neuen Systems, das sie schaffen sollten, stand noch nicht fest. Außerdem war die Stimmung im Projekt etwas matt, die Teams seltsam antriebslos. Er hatte schon einen Motivationsworkshop und eine Befragung unter seinen Mitarbeitern durchführen lassen, doch so richtig konnte das die Ursachen nicht aufdecken.

Angerer entschloss sich, externe Hilfe zu holen, und engagierte Andreas Carlsen, einen agilen Coach und Softwarearchitekten. Vielleicht würde der etwas herausfinden.

Der Coach fragt

Carlsen kannte solche Situationen – er hatte sie in vielen Projekten erlebt. Er kam in die Abteilung und machte sich mit den Entwicklungsteams und der Führungsmannschaft bekannt. Carlsen ließ sich den Prototyp des neuen Systems vorführen, fragte nach dem Quellcode und der Dokumentation und danach, ob es interne Kommunikationsmedien, z. B. ein Wikiweb, gebe.

Er ging an den vielen ruhigen Büros vorbei und blieb beim Büro des Softwarearchitekten Stefan Anders stehen: „Gehen wir zusammen einen Kaffee trinken?"

Anders nahm das Angebot gern an – er hatte richtig Lust, sich mit einem anderen erfahrenen Architekten auszutauschen. Carlsen erkundigte sich: „Wie seid ihr organisiert? Wer macht bei euch was, und wie ist die Zuständigkeit für die Architektur geregelt?"

„Jedes Komponententeam hat bei uns einen Chief Developer. Der ist der Erfahrenste von allen, die in dem Team arbeiten, und ist für das Design seiner Komponente verantwortlich. Er entwickelt selbst auch mit und spricht das Design der Komponente regelmäßig mit mir als verantwortlichem Architekten ab", so Anders.

Komponententeams gibt es schon sehr lange. Sie erschienen zu einer Zeit logisch, in der die Versionskontrolle brüchig war, man nur selten und verzögert integrierte und nur schwache Testwerkzeuge und Testmethoden hatte. Komponententeams entwickelten sehr spezielle Skills und konnten schnell arbeiten, wenn man es lokal betrachtet und den Durchsatz von fertigen Features, die für den Kunden wertvoll sind, außer Acht lässt. Diese Spezialisten zerbrachen den Code nicht so schnell, und es gab keine Konflikte mit Änderungen aus anderen Teams. Die Kehrseite der Medaille: Personen, die längere Zeit am selben Stück Code arbeiten, werden weniger empfindlich für dessen Nachteile und Designfehler. Sie bauen schneller Workarounds ein, die Qualität der Komponente sinkt schneller, als wenn andere Personen auch einen Blick darauf werfen. Außerdem führen Komponententeams fast automatisch zu wasserfallartigem Vorgehen, denn wenn man nur für eine Komponente zuständig ist, muss ja jemand vorher die Anforderungen analysieren, jemand muss die Architektur entwerfen und jemand muss zum Schluss den Gesamttest machen: alles separate Personen oder Teams, denen die Ergebnisse wie im Wasserfall zugereicht werden müssen.

Feature-Teams sind schlagkräftiger. Sie haben Analysten, Designer, Entwickler und Tester an Bord und sind dafür verantwortlich, ein ganzes Feature – also eine Menge von Systemfunktionalität, die der Kunde gewünscht hat – lauffähig zu machen und fertig getestet bereitzustellen. Durch eine enge Kommunikation benötigen diese Teams keine Übergabepunkte und können schnell reagieren. Mit den Entwicklungsmethoden und Werkzeugen von heute spielen die Konflikte, die dadurch möglicherweise auf den Komponenten entstehen, eine gegenüber den Vorteilen vernachlässigbare Rolle (vgl. [Lar08-a], [Lar08-b]).

Kasten 1: Komponenten- vs. Feature-Teams.

Value Stream Mapping (dt. etwa „Wertstromanalyse") zeichnet die Wertschöpfungskette der Softwareentwicklung auf. Das Verfahren stellt dar, wo produktiv gearbeitet wird, also Wert im Sinne des Abnehmers oder Kunden erzeugt wird, und wo Wartezeiten entstehen, also kein Wert geschaffen wird. Das Verfahren stammt aus der Familie der *Lean*-Methoden und hat das Ziel, die Durchlaufzeit für ein beauftragtes Feature und den Anteil von Wartezeit an der Durchlaufzeit darzustellen, um Engpässe in der Entwicklungsorganisation zu finden (vgl. [Rot99], [Pop03]).

Kasten 2: *Value Stream Mapping.*

Carlsen fragt nach: „Komponententeams, sagst du? Jedes Team hat genau eine Komponente unter sich?"

„Naja, manche haben auch zwei, aber jede Komponente wird von genau einem Team entwickelt. Jedes Team heißt auch so wie seine Hauptkomponente."

Carlsen schwante Übles. Doch er entschloss sich, erst einmal weiter zu fragen, um das Bild abzurunden: „Wer sorgt denn für die Konsistenz und die richtige Zusammenarbeit der Komponenten untereinander?"

Anders sah das Stirnrunzeln bei Carlsen und erklärte: „Das bin ich. Ich spreche mit allen die Schnittstellen ab und dokumentiere die Architektur als UML-Modell. Das Modell ist für alle zugänglich, jeder kann es sich jederzeit ansehen."

„Und woher nimmst du den Ablauf, die User-Story oder den Use-Case, in dem alle Komponenten wie ein Orchester mitspielen müssen?"

„Den liefert mir Udo-Carsten Ammann, unser Use-Case-Analyst. Er beschreibt jeden Anwendungsfall, den er mit den Stakeholdern analysiert hat, im selben UML-Modell, in einem anderen Package", antwortet Anders.

Carlsen hatte schon einen Verdacht, doch er wollte es zuerst ganz genau wissen: „Wie gut funktioniert denn das Ganze? Machen alle begeistert mit?"

Anders konnte sich endlich einmal richtig aussprechen: „Nein, es funktioniert überhaupt nicht gut. Die Leute interessieren sich nicht für meine Architektur. Auch die Use-Cases liest keiner, Ammann und ich müssen sie immer mühevoll erklären. Die Teams bauen die Komponenten, wie sie gerade Lust haben, ich muss ständig an die Einhaltung der Schnittstellen erinnern und komme vor lauter Code-Reviews nicht dazu, das UML-Modell weiterzuentwickeln. Ich gerate sogar ins Hintertreffen und muss Architektur aus dem Code herauslesen und nachdokumentieren. Man stelle sich das vor! Und wenn Angerer vorbeikommt und nach der Architektur fragt, muss ich wieder zugeben, im Rückstand zu sein!"

Auf dem Wege zur Lösung

Carlsen bat Anders: „Habt ihr irgendwo ein Whiteboard, auf dem wir die Situation einmal aufmalen können?"

„Ja klar, komm mit in den Besprechungsraum."

Carlsen zeichnete auf, was er gerade erfahren hatte (siehe Abb. 1).

„Schau, der Value Stream sieht bei euch so aus: Hier links ist Ammann mit seinen Use-Cases. Er gibt sie an dich, du machst die Architektur und die Schnittstellen der Komponenten, das gibst du an die Komponententeams, die machen das Detaildesign und die Codierung, richtig?"

„Ganz genau!"

Carlsen fragte nach: „In Wirklichkeit gibt es anscheinend noch einen zweiten Weg, oder? Die Use-Cases fließen direkt von Ammann zu den Komponententeams, die dann schon mal loslegen, noch ehe du so richtig die Architektur aufschreiben konntest."

„Ja, daher kommt ja das Problem! Und zum Schluss muss die Qualitätssicherung alles wieder ausbügeln und die Qualität hinein testen!"

Carlsen bremste: „Moment, lass uns erst noch die Wartezeiten einfügen!"

„Wartezeiten?"

„Jeder, der arbeitet, hat eine eingehende und eine ausgehende Warteschlange. Im Eingang kommt das an, was zu erledigen ist, in den Ausgang kommt das, was fertig ist. Dazwischen wird gearbeitet."

Anders hielt das für offensichtlich: „Ach so, na klar. Was soll's?"

„Ihr habt hier einen wunderschönen Wasserfallprozess mit einem völlig überfüllten Eingang bei dir und leer laufenden Eingängen bei den Komponententeams. Der Sog der Komponententeams ist groß – kein Wunder, dass immer der zweite Weg genommen wird und du mit deiner Architektur allein da stehst. Und schau dir mal die gesamte Zykluszeit an!"

Anders hielt auch das für glasklar: „Das sag' ich dem Ludwig Angerer doch schon die ganze Zeit: Wir brauchen mehr Architekten!"

Carlsen warnte: „Das würde das Warteschlangenproblem zwar beheben, doch ihr habt noch ein anderes Problem: Die Konsistenz der Architektur, das Zusammenspiel der Komponenten wäre weiterhin getrennt von der Implementierung des Codes. Ihr als Architekten wäret immer noch zuständig dafür, dass das, was die Teams programmieren, auch

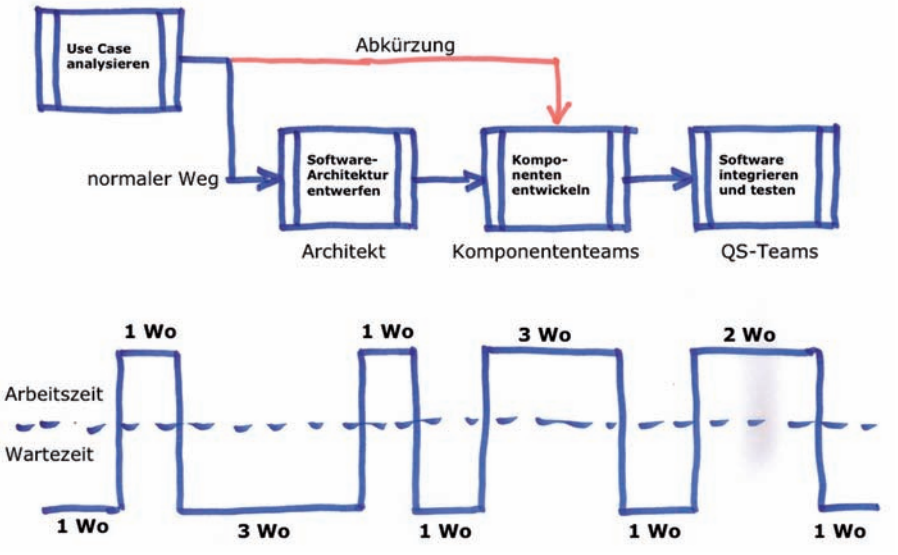

Abb. 1: *Ursprüngliche Wertschöpfungskette.*

Jeder Vorgesetzte versucht, seine Mitarbeiter dazu zu bringen, die übergeordneten Ziele, die seine Organisationseinheit hat, zu erreichen. Je nachdem, welche Aufgaben die Einheit hat, erfordert das unterschiedliche Mechanismen. Sehr einfache Aufgaben kann man durch Kommandieren und Kontrollieren managen, etwas komplexere durch Zielvereinbarungen und Erfolgsmessung und noch komplexere durch eine charismatische Führung. Bei hoch komplexen Aufgaben – wie Softwareentwicklung und Softwarearchitektur – setzt man am besten auf Selbstorganisation. Der Leiter muss es den Mitarbeitern ermöglichen, sich selbst zu organisieren, denn keiner kann mit Komplexität allein umgehen, man benötigt die „Weisheit des Schwarms". Diese emergiert aus der Befolgung einfacher Regeln für das Individuum, kombiniert mit Feedback und sehr rascher Kommunikation (Diskurs) zwischen mehreren Individuen. Regeln, Feedback und Diskurs müssen aktiv gefördert werden. Fehlt eines davon, ist die Gefahr groß, dass das soziale System zur Unordnung zurückkehrt (vgl. [Sim07], [Sim08]).

Kasten 3: *Selbstorganisation.*

zusammen läuft. Ihr müsstet im Vorhinein ein komplexes Design erstellen, was nachher auch wirklich funktioniert. Das wird auch bei mehr Architekten nicht klappen. Außerdem macht das den Teams keinen Spaß – es degradiert sie zu reinen Ausführungsinstrumenten der Architektur."

Anders runzelte die Stirn: „Und was machen wir stattdessen?"

Jetzt musste Carlsen Farbe bekennen: „Löst die Komponententeams auf, sie verstärken den Wasserfall. Macht Feature-Teams daraus! Ein Feature-Team ist dafür verantwortlich, dass ein komplettes Feature, also ein Paket von Abläufen, Storys, oder Use-Cases, die von den Kunden gewünscht werden, in sich geschlossen funktioniert. Die Komponenten werden dann zum Besitz von allen: Jedes Team bekommt das Recht, jede Komponente zu bearbeiten. Und die Architektur wird von allen gemeinsam erstellt."

Carlsen zeichnete die gewünschte Situation an das Whiteboard (siehe Abb. 2).

Für Anders war das zu revolutionär: „Wer sorgt dann dafür, dass eine Komponente in sich noch konsistent bleibt? Viele Köche verderben den Brei!"

Selbstorganisation und Emergenz

Carlsen wusste, jetzt kommt das, was am schwierigsten zu verstehen ist: „Alle auf einmal tun das, ohne zentrale Regie. Das Problem, das ihr hier lösen wollt, ist so komplex, dass es nicht durch einen Chef oder einen Architekten oder viele Architekten gelöst werden kann. Auch mit zentral vergebenen Zielen und einem Belohnungssystem für erreichte Ziele bekommt ihr das nicht hin. Die Problemkomplexität ist so hoch, dass ihr das nur mit Selbstorganisation schaffen werdet."

Anders Gesicht war die Verwirrung anzusehen. Carlsen fuhr fort: „Ihr als Entwicklungsorganisation müsst euch als System begreifen, das ein Verhalten entwickeln muss, das die gewünschte Qualität selbst hervorbringt – ohne Anweisungen von außen. Wie ein Ameisenhaufen, der sehr geordnet funktioniert, obwohl alle Ameisen scheinbar wild durcheinander laufen. Das Verhalten des Haufens ist mehr als die Summe des Verhaltens jedes einzelnen Individuums. Man nennt das Emergenz. Dazu gibt es eine eigene Wissenschaft mit vielen Ergebnissen."

„Angerer wird mich fragen, wie wir das machen sollen. Erklär' mir, wie wir das angehen und warum das funktionieren soll. Vorher kann ich so etwas nicht ernsthaft vorschlagen."

Carlsen musste etwas ausholen: „Soziale Systeme entwickeln Verhalten aufgrund von Regeln, Feedback und Diskurs. Nimm mal an, du tust etwas und verletzt dabei eine Regel. Jemand anders gibt dir Feedback, du wunderst dich darüber und fragst warum. Ihr kommt ins Gespräch, anschließend ändert sich euer beider Verhalten – ihr entwickelt ein neues, gemeinsam gewünschtes Verhalten. Der Zyklus fängt danach von vorn an, je mehr Leute du im Alltag triffst. Schließlich verändert die gesamte Gruppe ihr Verhalten."

Anders fand das zu abstrakt. Carlsen ging mehr ins Detail: „Ich erkläre es dir mal an einem Beispiel: Gib den Feature-Teams deine Designregeln mit. Sag ihnen, was du unter einer guten Komponente verstehst. Du weißt schon, der ganze *S.O.L.I.D.*-Kram: *Single Responsibility Principle, das Open/Closed-Priciple, Liskov Substition Principle* usw. Eines der Teams wird vielleicht eine Komponente so verändern, dass sie für ein anderes Team nicht mehr wartbar ist. Das andere Team gibt Feedback und macht auf die Verletzung der Regel aufmerksam, z. B. dass SRP verletzt ist. Beide diskutieren das und finden ein besseres Design. Du hilfst ihnen dabei, mit deinem Architektur-Know-how. Nachher gehen sie wieder an ihre Arbeit, dokumen-

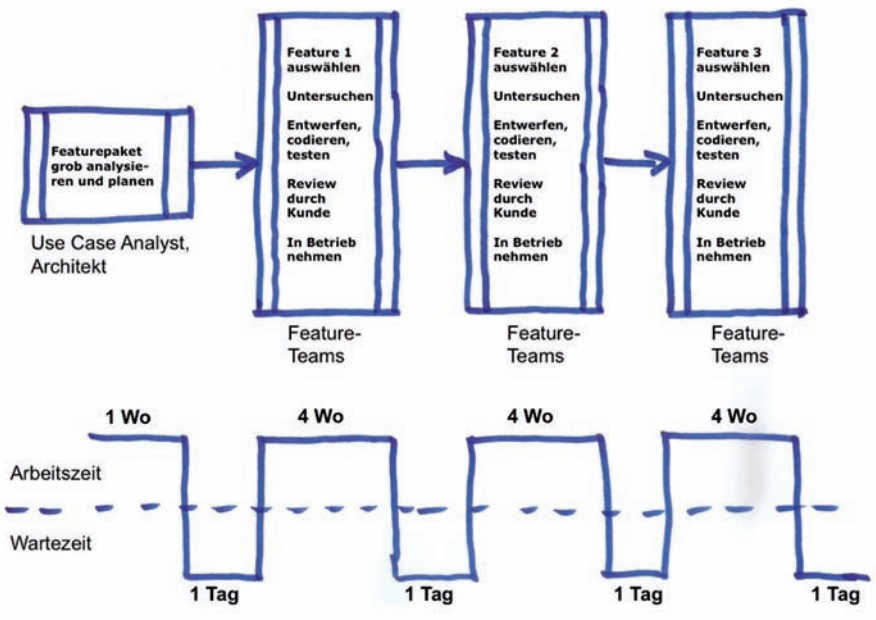

Abb. 2: *Wertschöpfungskette mit verbessertem Arbeits-Wartezeit-Verhältnis.*

Der Begriff „Emergenz" kommt von dem lateinischen Wort „emergere" (auftauchen, hervorkommen, sich zeigen). Das Wort bezeichnet die spontane Herausbildung von Eigenschaften eines Systems auf der Makroebene, auf der Grundlage des Zusammenspiels seiner Elemente. Dabei lassen sich die emergenten Eigenschaften des Systems nicht offensichtlich auf Eigenschaften der Elemente zurückführen, die diese isoliert betrachtet auf der Mikroebene aufweisen (vgl. [Wik]).

Die Emergenz-Theorie ist interdisziplinär und findet beispielsweise Anwendung in der Biologie, Soziologie, Kognitionswissenschaft, der Theorie der Finanzmärkte und der Physik. Sehr faszinierend sind Emergenz-Effekte in soziologischen Strukturen, wie z. B. Entwicklungsteams, zu beobachten. Meines Erachtens haben wir erst angefangen, die Vorteile der Emergenz zu erkennen, geschweige denn, diese konsequent zu nutzen.

Kasten 4: Emergenz.

tieren die Änderung und du bist entlastet." Anders verstand sofort: „So würden wir alle gemeinsam die Architektur erschaffen und brauchen keinen Chef wie mich mehr. Die Architektur wäre dann emergent, sagt man das so? Ich wäre dann nur noch der Coach, der du jetzt bist."

Carlsen freute sich: „Genau so – jetzt müssen wir das nur noch vervollständigen und dann Ludwig Angerer beibringen."

Carlsen schlug noch vor, nicht nur Regeln für die Ergebnisse (Code und Architektur) herauszugeben, sondern auch minimale Regeln für die Kommunikation, die dann Feedback und Diskurs anregen würden:

1. Die Architektur und deren Qualität für alle sichtbar machen.
2. Ein ehrliches Feedback ohne Angriffe einholen und geben.
3. Wöchentliche Meetings für Designabstimmungen vorsehen.
4. Eine interessante, besondere Art der Architekturdokumentation einführen.

Anders dokumentierte seine Architektur nämlich bisher gern, doch die Teams fanden die Dokumentation langweilig.

Carlsen schlug vor: „Wie war das im Kriminalroman? Der Mörder braucht drei Dinge: Motiv, Gelegenheit und Mittel! Gib euren Teams Motiv, Gelegenheit und Mittel, um Architektur zu gestalten. Architektur muss so interessant sein, dass jeder sich darum reißen wird, sie zu beschreiben. Lass uns bei Angerer einen Camcorder beantragen. Wir lassen in der wöchentlichen Designabstimmung jedes Mal ein Team über seine geänderten Komponenten berichten. Diese Referate nehmen wir mit dem Camcorder auf und stellen sie als Videodateien ins Wiki. So kann sie sich jeder anschauen und bekommt das Bild von einer Komponente viel breitbandiger mitgeteilt, als jedes noch so ausgefeilte Dokument allein es könnte!" Anders war von der Lösung ebenso begeistert wie Carlsen. Angerer, der Abteilungsleiter, erfasste die Vorschläge sehr schnell und erkannte die Vorteile, aber auch die Risiken. „Das ist eine erhebliche Änderung unserer Vorgehensweise, unserer ganzen Entwicklungskultur! Wir müssen gut aufpassen und Fallstricke vermeiden."

S.O.L.I.D. ist eine Familie von bewährten Entwurfsprinzipien, die innerhalb der objektorientierten Softwareentwicklung auf Klassen und Komponenten angewendet werden können. Die Abkürzung steht für folgende Prinzipien (vgl. [Mar]):

- *Single Responsibility Principle:* Eine Klasse soll genau einen Grund haben, sich zu ändern.
- *Open Closed Principle:* Eine Klasse soll offen für Erweiterungen sein, aber geschlossen gegen Modifikationen. Man soll in der Lage sein, ihr Verhalten zu erweitern, ohne die Klasse selbst zu verändern.
- *Liskov Substitution Principle:* Ein Subtyp muss sich immer wie sein Basistyp verhalten.
- *Interface Segregation Principle:* Clients sollen nicht mit Details belastet werden, die sie nicht benötigen. Einfache, schlanke, zweckorientierte Interfaces sind gut.
- *Dependency Inversion Principle:* High-Level-Klassen sollen nicht von Low-Level-Klassen abhängig sein, sondern beide von Interfaces. Interfaces sollen nicht von Details abhängig sein, sondern umgekehrt.

Kasten 5: S.O.L.I.D.

Fazit

Die obigen Szenen haben sich in manchen Projekten so abgespielt und könnten es in vielen heutigen Projekten tun. Emergente Architektur ist eine Lösung für die Anforderungen, die an eine moderne Softwarearchitektur gestellt werden. Immer mehr Teams erkennen, dass Architektur nicht im üblichen Sinne „gemacht" werden kann, sondern sich am besten aus der täglichen Entwicklungsarbeit ergibt. Dafür muss man dem System „Entwicklungsorganisation" einige wenige Regeln und Beschränkungen auferlegen – dann bildet es durch Feedback und Diskurs das Verhalten aus, das die Architektur erzeugt. Die Architektur emergiert aus der Weisheit des Schwarms und der Architekt wird zum Coach. ■

Literatur & Links

[LarO8-a] C. Larman, B. Vodde, Scaling Lean & Agile Development: Thinking and Organizational Tools for Large-Scale Scrum, Addison-Wesley 2008

[LarO8-b] C. Larman, B. Vodde, Kapitel über Feature- und Komponententeams aus dem Buch „Scaling Lean & Agile Development", siehe: www.infoq.com/articles/scaling-lean-agile-feature-teams

[Mar] R.C. Martin, Principles of object oriented design, siehe: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[Pop03] M. und T. Poppendieck, Lean Software Development, Addison-Wesley 2003

[Rot99] M. Rother, J. Shook, Learning to See – Value-stream mapping to create value and eliminate muda, Lean Enterprise Institute, 1999

[Sim07] F.B. Simon, Einführung in die systemische Organisationstheorie, Carl-Auer Compact, 2007

[Sim08] F.B. Simon, Einführung in Systemtheorie und Konstruktivismus, Carl-Auer Compact, 2008

[Wik] Wikipedia, Emergenz, siehe: http://de.wikipedia.org/wiki/Emergenz