Ralf Westphal
(ralfw@ralfw.de)
ist freiberuflicher Berater, Projektbegleiter, Referent, Autor und Trainer für Themen rund um Softwarearchitektur und die Organisation von Softwareteams. Er ist Mitgründer der Initiative „Clean Code Developer" (CCD) für mehr Softwarequalität (clean-code-developer.de), propagiert kontinuierliches Lernen mit der CCD School (ccd-school.de) und möchte mit ich-verspreche.org zu mehr Zuverlässigkeit motivieren.

Aufblühende Strukturen

Software hat eine Struktur. Sie dient der Erfüllung nicht-funktionaler Anforderungen wie Skalierbarkeit oder Wandelbarkeit. Es ist die Aufgabe der Rolle Softwarearchitekt, diese Struktur zu entwerfen. Über die Jahrzehnte haben sich Muster für die Grundstruktur von Software herausgebildet. Model-View-Controller, Schichtenarchitektur, N-Tier Architecture, Hexagonal Architecture, Clean Architecture, Pipes-and-Filters sind einige Beispiele. Gewöhnlich wird zu Beginn eines Projektes entschieden, welchem solcher Architekturmuster eine Software entsprechen soll. Dieses Muster wird dann von Anfang an implementiert. Ein Vorgehen dieser Art liegt nahe. Die nicht-funktionalen Anforderungen sind ja bekannt. Eine passende Architektur kann und sollte daher gleich von Anfang an gewählt werden. Es ist ja auch gut, von vornherein eine klare Struktur zu haben. Oder?

Starre Architekturbilder

Mir erscheint dieses Vorgehen zunehmend suspekt. Darin spiegelt sich die eine Konstante der Softwareentwicklung schlicht nicht wider: der Wandel. Welche Anforderungen in drei Monaten, einem Jahr oder drei Jahren zu erfüllen sind, ist eben nicht klar. Auch wenn die Fachdomäne sich nicht grundsätzlich über die Lebenszeit einer Software verändern mag, so steht ansonsten doch jeder andere Aspekt ständig auf dem Prüfstand. Überall kann der „Blitz" des Kundenwunsches einschlagen und die Struktur zur Veränderung zwingen.

Zu einem gegebenen Zeitpunkt hat Software natürlich nur eine bestimmte Struktur. Doch was war vorher, was wird nachher sein? Dass es überhaupt einen Strukturwandel gibt bzw. zumindest geben könnte, ist an den tradierten Architekturmustern nicht ablesbar. Sie sind wie eingefroren.

Darin, so glaube ich, ist ein Grund zu sehen, dass unter Softwaresystemen keines (oder nur wenige) mit einer *Screaming Architecture* zu finden ist. Robert C. Martin hat diesen Begriff geprägt: „Just as the plans for a house or a library scream about the use cases of those buildings, so should the architecture of a software application scream about the use cases of the application. [...] Your architectures should tell readers about the system, not about the frameworks you used in your system" [Mar11].

Ein CRM-System (Customer Relationship Management, CRM) hat keine CRM-Architektur, sondern vielleicht eine MVC-Architektur (MVC, englisch für Modell-Präsentation-Steuerung) genauso wie eine Arztpraxissoftware. Oder eine Baufinanzierungssoftware hat keine Baufinanzierungsarchitektur, sondern vielleicht eine Schichtenarchitektur genauso wie eine Maschinensteuerung.

Man mag einwenden, dass aber doch Baufinanzierungssoftware und Maschinensteuerung vielleicht dieselben nicht-funktionalen Grundanforderungen erfüllen müssen und es deshalb naheliegt, sie demselben Architekturmuster folgen zu lassen. Das ist sicher richtig – doch es bleibt die Frage: Warum ist dennoch die Domäne oft so wenig erkennbar in einer Softwarestruktur? Gebäude unterliegen ja auch denselben Grundanforderungen: sie müssen eine bestimmte Anzahl Menschen aufnehmen und Schwerkraft wie Windkraft standhalten. Dennoch unterscheiden sich die Architekturen von zum Beispiel Krankenhäusern und Bibliotheken deutlich.

Das Paradoxe am Mangel an *Screaming Architecture*: Sie fehlt, weil Softwarearchitekturmuster wie Gebäudebaupläne verstanden werden. Gerade die durch den in der Software benutzten Begriff „Architektur" entstehende Nähe zum Hochbau verhindert, dass sich die Domäne in Softwarestrukturen deutlich ausdrückt. Denn so wie im Hochbau Architekturpläne fix sind, so werden auch Softwarearchitekturen wohl meist noch als fix angesehen: einmal Schichtenarchitektur, immer Schichtenarchitektur.

Dabei sind Gebäude und Software in ihrem Wesen diametral entgegengesetzt: Gebäude sind statisch. Über Jahre und Jahrzehnte verändern sie sich nicht oder wenn, dann nur in überschaubarem Maße. Menschen ziehen ein und aus, Wände werden eingerissen oder neu arrangiert. Doch ganz fundamental bleiben die meisten Gebäude wohl so, wie sie ursprünglich auf einer Blaupause beschrieben wurden.

Software hingegen unterliegt vom ersten Tag an extremen Veränderungen. Neue funktionale Anforderungen führen zu ihrer Ausdehnung, geänderte nicht-funktionale Anforderungen führen zu ihrer Umstrukturierung. Alles ist ständig in Bewegung. Das ist normal.

Doch von diesem Wandel gibt es bei Architekturmustern keine Vorstellung. Sie stehen still, sind Momentaufnahmen.

Storyboards für fließende Architektur

Ich halte daher den Hochbau für eine wenig hilfreiche Analogie. Er zwingt der Softwareentwicklung eine zu statische Sichtweise auf. Diese immobilisiert das Denken. Das färbt den gesamten weiteren Prozess negativ für die zentrale Eigenschaft Wandelbarkeit ein.

Viel günstiger und der Softwareentwicklung verwandter scheint mir die Biologie. Dort geht es per definitionem um Entwicklung, um Veränderung, um Wachstum, um Evolution, um Anpassung an sich ändernde Umstände.

Software zu bauen und dabei an ein Haus zu denken, widerspricht der Realität ihrer Entwicklung. Das bessere Bild ist für mich das einer Entfaltung. Software erblüht. Aus einer Knospe wird eine Blüte. Etwas zunächst Kleines wächst zu etwas Größerem, gar Großem – und verändert dabei immer wieder seine Grundstruktur.

Nicht ein Architektur-*Bild* braucht die Softwareentwicklung, sondern ein Architektur-*Storyboard*. Softwarearchitektur sollte nicht auf einem Standpunkt beharren, sondern eine Entwicklungslinie verfolgen. Die Zeit sollte Einzug halten in die Vorstellung von Softwarestrukturen – und damit in ihre Veränderung.

Entwicklung in der Biologie bedeutet einerseits Wachstum im Sinne von Mengenzunahme. Erst eine Zelle, dann zwei, dann vier, dann acht usw. Andererseits findet eine Differenzierung statt. Organismen bestehen aus einer großen Vielfalt von Zellen hoher Spezialisierung. In Muskelzellen und Nervenzellen drückt sich dies beispielhaft in Größe und Form aus.

Die Genese eines Organismus beginnt mit nur einer Zelle. Sie ist der Beginn einer Entwicklungsgeschichte zunächst in Bezug auf Menge, dann Vielfalt und dann Struktur. Vielfalt und Struktur stehen nicht am Anfang, sondern ergeben sich aus Notwendigkeiten. Es sind Reaktionen, wenn auch keine ganz freien. Die Reaktionsbreite ist durch die organismusspezifische DNS begrenzt.

Software-Genesis

Wie könnten nun aber die Entwicklungsstadien von Software aussehen? Gibt es eine Folge von Bildern, die zu einer Geschichte von Wachstum und Differenzierung aneinandergereiht werden könnten?

Ja, ich glaube, solch ein Storyboard lässt sich zeichnen. Hier ist die Geschichte dazu:

Am Anfang da ist… nichts – nur ein amorphes Problem (siehe Abbildung 1, links oben). Doch dann wird eine Software beauftragt und die Problemwelt wird geschieden in ein **System**, das das Problem löst, und eine immer noch amorphe **Umwelt**, in die es eingebettet ist (siehe Abbildung 1, rechts oben).

Bei näherer Betrachtung zeigt die Umwelt jedoch eine grobe Struktur: sie besteht aus anderen Systemen, **Umweltsystemen**, die in **Kommunikation** mit dem problemlösenden stehen (siehe Abbildung 1, links unten).

Die Kommunikation besteht aus einem Strom von einfließenden Reizen und ausfließenden Reaktionen. Um als eigenständiges System adressierbar zu sein und Bestand zu haben, ist es nötig, dass das Lösungssystem eine Kontur entwickelt, eine Grenze. In einer ersten inneren Differenzierung trennt es daher eine **Membran** von einem **Innenraum** (siehe Abbildung 1, rechts unten). Die Membran kapselt die Feinheiten des Empfangs von Umweltreizen sowie ihrer Umsetzung in Nachrichten innerhalb des Systems und umgekehrt die Darstellung von inneren Nachrichten als Reaktionen auf der Systemoberfläche. Der Innenraum wird damit unabhängig von der Umwelt.

Bisher sind Membran und Innenraum direkt verbunden. Das macht eine flexible unabhängige weitere Wandlung schwer. Deshalb bildet sich an der Berührungsfläche ein „Bindegewebe“ heraus. Das entkoppelt beide voneinander – und hält sie gleichzeitig doch beisammen. Seine Aufgabe ist die **Integration** der Teile zu einem Ganzen. Das System zerfiele sonst entweder in lose Einzelteile oder bestünde nur als starre Einheit von abhängigen Bereichen (siehe Abbildung 2, links oben).

Die Umweltsysteme gehören zwei Gruppen an: solchen, die vom Lösungssystem etwas wollen (**Akteure**) und solchen, von dem es selbst etwas will (**Ressourcen**) (siehe Abbildung 2, rechts oben). Dadurch entsteht eine Asymmetrie in der Struktur des Lösungssystems: den Akteuren zugewandt ist die Membran eine Vermittlungsschicht für Reize nach innen, ressourcenseitig hingegen erzeugt die Membran selbst Reize.

Da Akteure sehr verschiedene Ansprüche an das System haben können, also es in sehr unterschiedlicher Weise reizen wollen, und Ressourcen ebenfalls nicht alle über einen

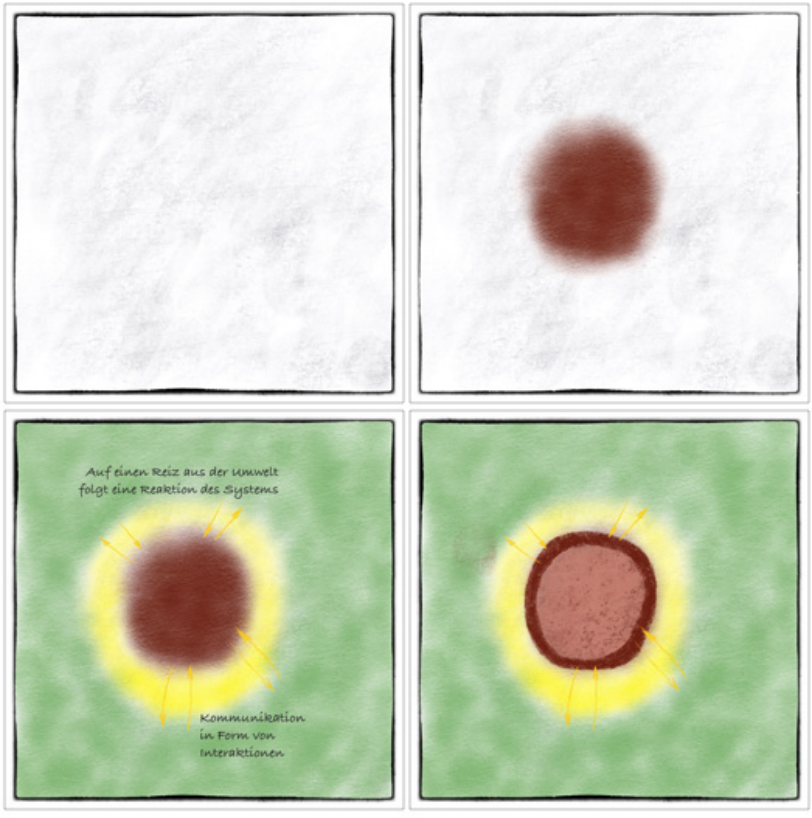

Abb. 1: Ein Softwaresystem grenzt sich von der Umwelt ab und tritt mit ihr in Interaktion.

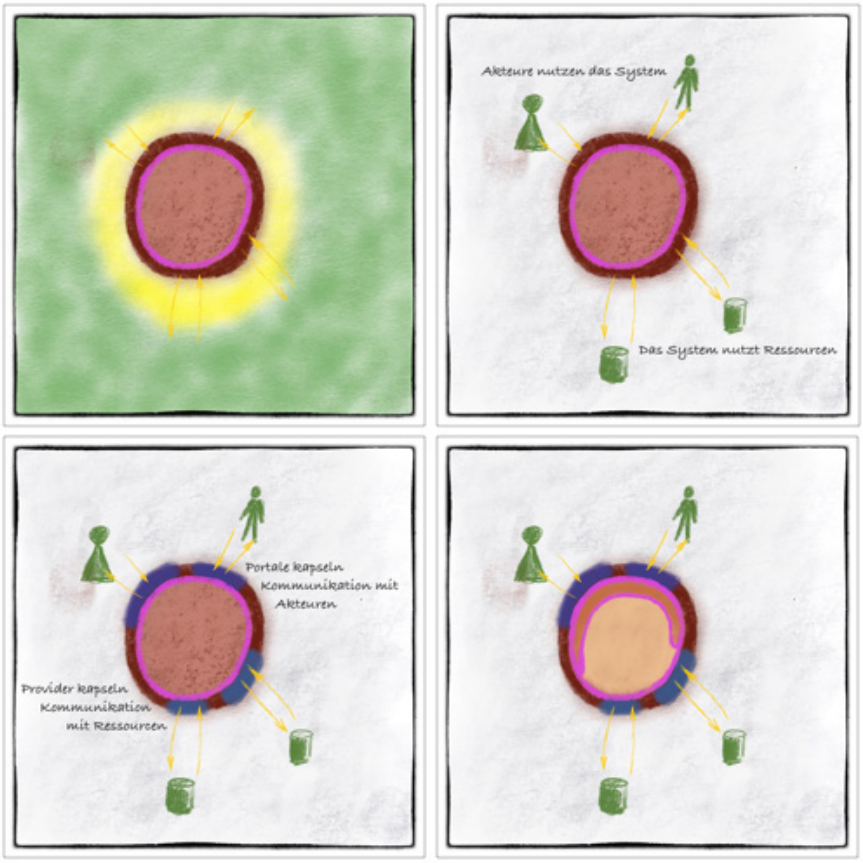

Abb. 2: Differenzierung und Entkopplung gehen Hand in Hand.

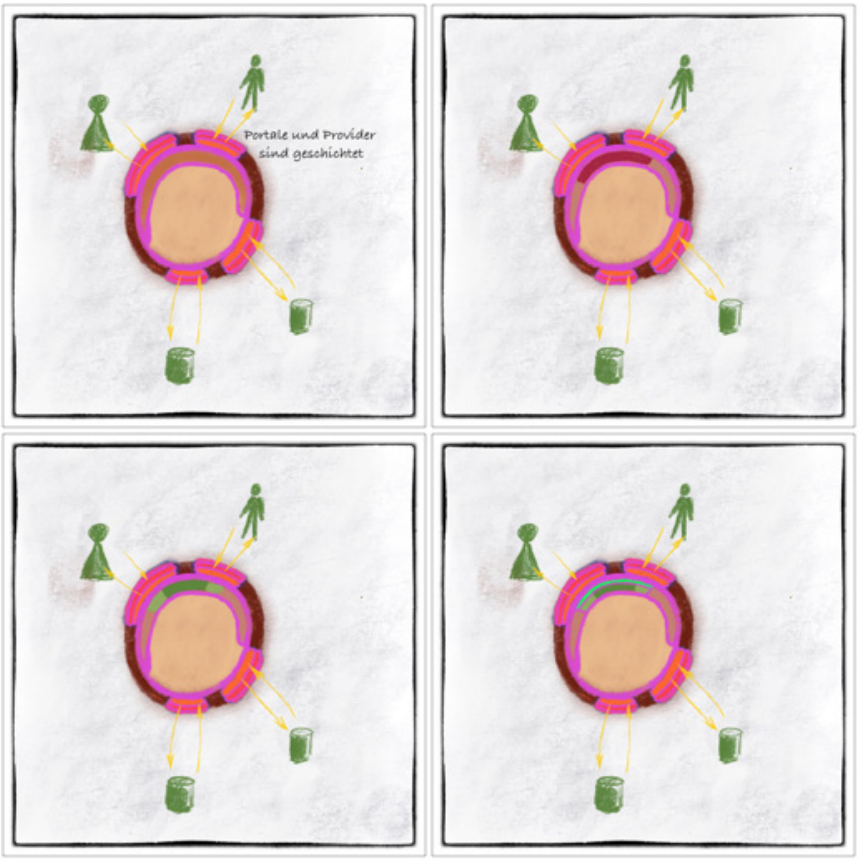

Abb. 3: Differenzierung der Kommunikation zieht Differenzierung ihrer Behandlung nach sich.

Kamm geschoren werden können, ist es nur natürlich, dass sich die Membran ihnen gegenüber sehr individuell ausprägt. Sie entwickelt **Portale** für die Kommunikation mit Akteuren und **Provider** für die mit Ressourcen (siehe Abbildung 2, links unten). Beide Membranregionen stellen Adapter dar und kapseln die APIs der Umweltsysteme. Die unterschiedliche Richtung des Kontrollflusses durch die Membran legt allerdings nahe, sie namentlich zu unterscheiden. Akteure kontrollieren das System, das System kontrolliert Ressourcen.

Es stellt sich die Frage, wo die Reize der Umwelt behandelt werden. Die Aufgabe der Portale ist lediglich der Empfang von Reizen und die Darstellung von Reaktionen mittels einer Hardware (z. B. Tastatur, Maus, Bildschirm, Netzwerkkarte) und zugehörigem API. Ihre Verarbeitung hingegen ist Sache des Innenraums. Der differenziert sich daher in Richtung der Portale in eine Schale zur Behandlung von Reizen (**Handler**) und einen Kern (**Core**) (siehe Abbildung 2, rechts unten). Das betont die Asymmetrie im Umgang mit den Umweltsystemen.

Die Umwelt vom Innenraum zu entkoppeln, ist nicht immer einfach. Die Membran spaltet sich daher in drei Schichten: **Wrapper**, **Mapping** und **Validation** (siehe Abbildung 3, links oben). Der Wrapper verbirgt die Abhängigkeit zum API, das Mapping sorgt für die Übersetzung der rohen Daten vom API in etwas für den Innenraum strukturell Bekömmliches; sie werden damit unabhängig vom API. Die Validation schließlich sorgt für eine erste Absicherung gegen inhaltliche Unverträglichkeit. Die ist jedoch naturgemäß vor allem dort ausgeprägt, wo Reize unaufgefordert einfließen, d. h. in Richtung Akteure.

Bei genauerem Hinsehen wird erkennbar, dass Reize und Reaktionen in zwei Kategorien zerfallen; die Interaktionen der Akteure mit dem System sind nicht homogen. Die einen stellen konkrete Aufträge an das System dar (**Request**) und kommen mit der Erwartung an eine Antwort einher (**Reply**). Die anderen sind schlicht Beschreibungen von Ereignissen (**Events**) und gar nicht an ein konkretes System adressiert. Darauf reagiert das System mit einer Differenzierung der Handler-Schale. Die zerfällt in getrennte **Request-Handler** und **Event-Handler** (siehe Abbildung 3, rechts oben).

Und bei noch genauerer Betrachtung sind sogar die Requests mit ihren Replies nicht alle gleich. Vielmehr zielen die einen auf eine Veränderung des Zustands des Systems ab (**Command**) und erwarten

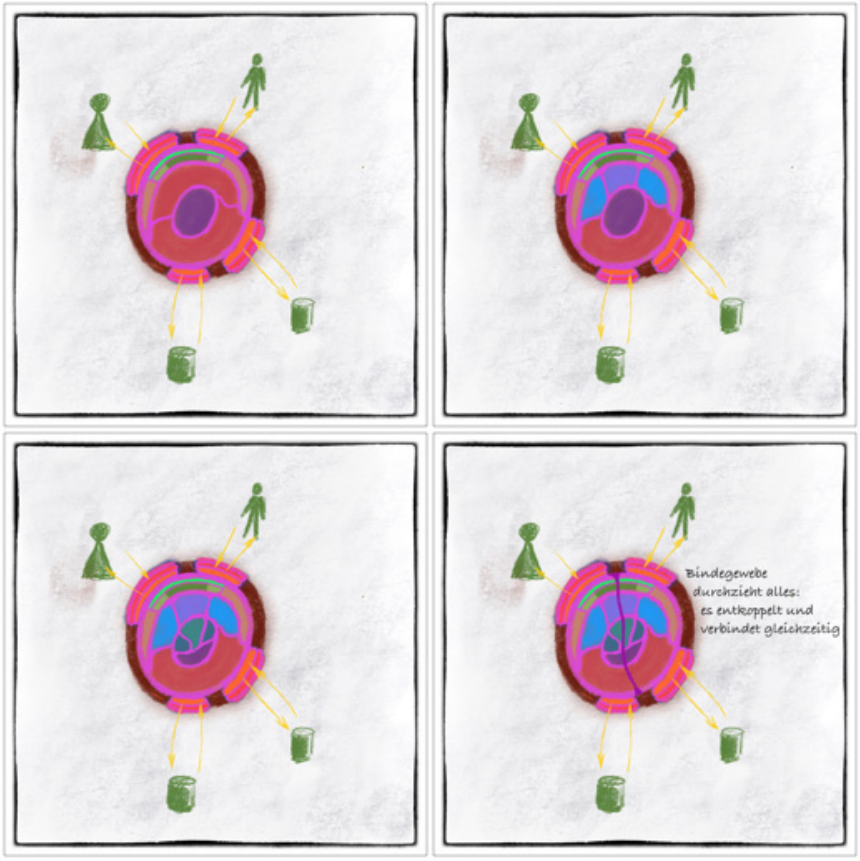

Abb. 4: Daten fließen und werden in domänenspezifischen Strukturen gehalten.

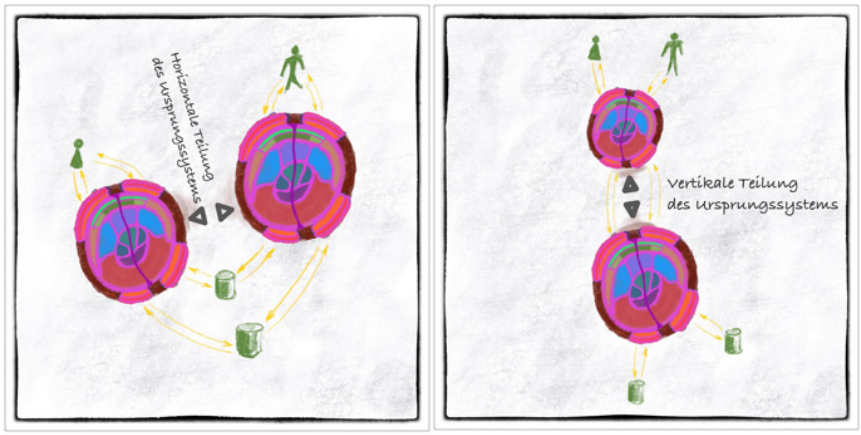

Abb. 5: Ein Softwaresystem als selbstähnliche Struktur kann in kleinere, autonome Systeme zerfallen, um Anforderungen an Agilität oder Effizienz zu erfüllen.

dazu eine Vollzugsmeldung (**Status**). Die anderen hingegen wollen das System nicht verändern, sondern nur unter Umständen seinen Zustand nutzen, um auf eine Frage (**Query**) eine Antwort (**Result**) zu erhalten. Wieder reagiert die Handler-Schale auf solche Differenziertheit mit einer weiteren Trennung: Request-Handler zerfallen in **Command-Handler** und **Query-Handler** (siehe Abbildung 3, links unten).

Ein direkter Kontakt zwischen Request-Handlern und Membran kann die Differenzierung von Requests in Commands und Queries behindern. Die Membran – das UI – könnte mit einer steigenden Granularität der Handler zu koordinierenden Aufgaben gezwungen werden. Außerdem würden manche Benutzungszusammenhänge schwerer testbar. Deshalb spalten sich von den Request-Handlern vorgelagerte **Use-Case-Handler** ab (siehe Abbildung 3, rechts unten). Sie erlauben auch die Kommunikation mit der Außenwelt über undifferenzierte Requests, die durch die Use Cases erst in Sequenzen von Kommandos und Queries gegenüber den Request-Handlern übersetzt werden.

Die vielfältigen Ansprüche der Akteure üben Druck auf die Differenzierung des Lösungssystems von außen nach innen aus. Bisher ist der Kern nicht weiter ausgeprägt; Membran und Handler-Schalen haben ihn isoliert. Doch auf weiter steigenden Druck hin zerfällt der Kern natürlich auch. Daten fließen nicht nur zwischen den schon entstandenen Bereichen bzw. zwischen Umwelt und Kern (Messages), sondern es werden auch Daten im Kern gehalten (State).

War diese Datenhaltung bisher amorph und implizit, so entstehen nun deutlich getrennte Bereiche: einerseits explizite „reine“ Datenstrukturen (**Abstract Data Type**), deren Aufgabe die Datenhaltung, Konsistenzsicherung und Traversierung ist, andererseits Verhaltensstrukturen (**Workflow**), die auf Daten operieren, die durch sie fließen, bzw. die Daten haben, aber keine Daten sind (siehe Abbildung 4, links oben).

Innerhalb von Workflows setzt sich die Differenzierung weiter fort durch Unterscheidung von Command-Verarbeitung (**Aggregate**), also Zustandsveränderung, und Query-Behandlung (**Processor**), d. h. Zustandsnutzung (siehe Abbildung 4, rechts oben).

Dem folgt eine Trennung der Daten. Ein bisher universelles Datenmodell zerfällt in **Write-Model** und **Read-Model**, um die Anforderungen von Commands bzw. Queries besser bedienen zu können (siehe Abbildung 4, links unten).

Und auch das „Bindegewebe“ entwickelt sich bei steigender Belastung. Während es bisher durch schieres Vorhandensein in Form von Integrationsmethoden entkoppelt hat, prägt es nun einen speziellen Mechanismus aus: einen **Bus** (siehe Abbildung 4, rechts unten). Dieser verbindet flexibler und weiter entfernte Bereiche.

Auch vor dem Lösungssystem selbst macht die Differenzierung nicht halt. Zu jeder Zeit kann es sich teilen. Vertikal ist eine Teilung in autonome Durchstiche möglich, die funktionale Anforderungen gegenüber Akteuren bündeln und voneinander trennen; es entstehen mehrere **Applications** (siehe Abbildung 5, links).

Oder das Lösungssystem entspricht mit einer horizontalen Teilung einer nichtfunktionalen Effizienzanforderung. **Client** und **Server** werden geboren (siehe Abbildung 4, rechts).

In Fällen der Teilung eines Lösungssystems bzw. eines seiner Subsysteme haben die resultierenden Systeme dieselbe grundsätzliche Struktur. Für Server ist jedoch kein

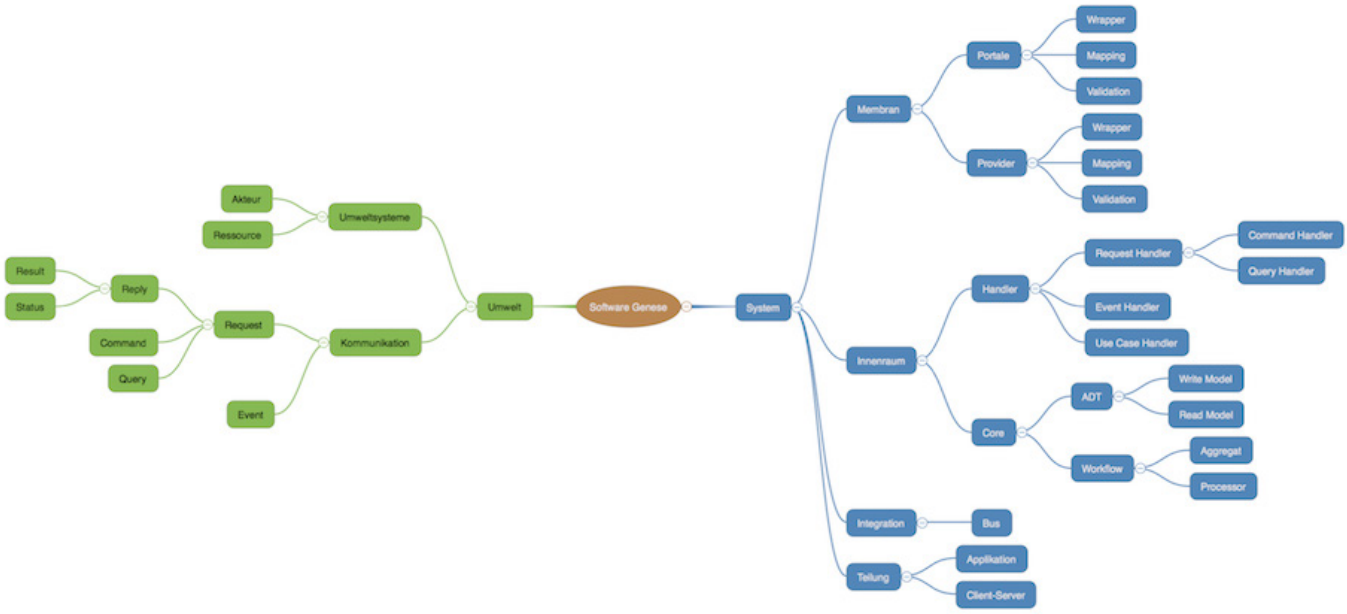

Abb. 6: Überblick über die Hierarchie der sich ausdifferenzierenden Softwaresystemstrukturen.

Akteur in der Umwelt die fordernde Instanz, sondern ein Client-System.

Organisches Wachstum

Das ist sicher eine für Sie ungewöhnliche Darstellung von Softwarearchitektur. Keine Kästchen, keine Abhängigkeitspfeile... So soll Software aussehen, innen?

Ich habe die Darstellung bewusst so gewählt, um auch mit der bildlichen Vorstellung von Software raus aus der Ecke von Hochbau oder Maschinenbau zu kommen. Die Darstellung soll organisch, unscharf sein, denn so ist das Leben. So ist auch Software, denn die wächst. Wenn sich das in dieser Diagrammfolge so zeigt, dann ist das ein Feature, kein Bug.

Aber es geht auch anders. Abbildung 6 zeigt Software sozusagen „in voller Blüte" als Mindmap. Aber schauen Sie bitte nicht nur auf die „Ausbaustufen" am Ende. Leiten Sie nicht ab, dass Sie am besten gleich am Anfang eines Projektes all die darin zu sehenden Teile herstellen. Die Abbildungen zeigen vielmehr mögliche Reaktionen auf Anforderungen. Wenn der „Druck" auf Software steigt, reagiert sie darauf mit einer Umstrukturierung und Differenzierung. Entscheidend ist, dass Sie das Potenzial sehen, in die eine oder andere Richtung zu gehen.

Sie mögen nun denken, „Aber wie halte ich die Software so flexibel, dass die Struktur langsam evolvieren kann?" Dafür ist das „Bindegewebe" zentral. In den Bildern habe ich es extra zwischen alle ausgeprägten Aspekte gezogen. Es umschließt, verbindet, hält zusammen. Das ist seine einzige Verantwortung. Die gilt es, in Software mit speziellen Modulen (Funktionen, Klassen) zu repräsentieren (vgl. hierzu auch [Wes17]).

Zusammenschau

Soweit mein derzeitiger Vorschlag für ein Architektur-Storyboard.

Erkennen Sie, wie grob vereinfachend, gar sträflich vereinfachend demgegenüber die bekannten Architekturmuster sind? Die kommen mit zwei, drei oder vielleicht acht verschiedenen Verantwortungsbereichen aus, die auch noch direkt voneinander abhängig sind. Das ist so, als würden Sie einen Menschen als Strichmännchen zeichnen: nicht ganz falsch, aber auch nicht wirklich realistisch. Weitere Anforderungsbereiche müssen sich dann in ein solch starres Muster zwängen. Guidance gibt es dafür nur wenig.

Das hier vorgestellte Bild, nein, das hier vorgestellte Storyboard einer Entwicklung von Softwarestruktur ist viel reichhaltiger. Es unterscheidet Verantwortlichkeiten viel feiner und auf verschiedenen Abstraktionsebenen. Seine Vorstellung von Softwareanatomie ist realistischer. Nicht Strichmännchen, sondern zumindest Anatomieabbildung wie in einem Kinderbuch. Die Realität ist natürlich noch vielfältiger.

Mit so differenziert ausgeprägten Verantwortlichkeiten lässt sich eine *Screaming Architecture* viel eher herstellen. Es gibt mehr und feingranularere Teile, die problemdomänenspezifisch arrangiert werden können.

Indem das Storyboard zeigt, dass Softwarestruktur nicht starr ist, sondern sich über die Zeit entwickelt, ausprägt, entfaltet, lädt es dazu ein, selbst weiterzumachen im Rahmen zweier Grundprinzipien: Die operierenden Einheiten kennen einander nicht, sie sind funktional unabhängig voneinander, und die operierenden Einheiten werden durch ein spezielles „Bindegewebe", die Integration, zusammengehalten. Damit ist eine grundsätzliche Entkopplung gegeben. Das Lösungssystem ist offen für Erweiterungen und Ausdifferenzierungen.

Außerdem ermutigt diese Sichtweise, nicht vorzeitig zu optimieren. Softwarestrukturen können einfach beginnen und bei Bedarf verfeinert werden. So kann die Struktur zu jedem Zeitpunkt die angemessene Reaktion auf die aktuellen Anforderungskräfte sein. Verschieben sich diese Kräfte, wird die Struktur nachgeführt.

Das ist für mich wahrhaft agile Softwarearchitektur. So wird der Natur von Software als sich ständig weiterentwickelndem Gebilde Rechnung getragen. Aufblühende Softwarelandschaften liegen dann vor uns. ■

Referenzen

[Wes17] Ralf Westphal, Die IODA-Architektur: Wandelbarkeit erreichen ohne funktionale Abhängigkeiten, OBJEKTspektrum 1/2017.

[Mar11] https://8thlight.com/blog/uncle-bob/2011/09/30/Screaming-Architecture.html