Phillip Ghadir

(E-Mail: phillip.ghadir@innoq.com)
ist CTO und Principal Consultant bei der innoQ Deutschland und baut am liebsten tragfähige, langlebige Softwaresysteme. Er ist Mitglied der Geschäftsleitung bei innoQ und hat sich früh auf Architekturen für verteilte, unternehmenskritische Systeme spezialisiert. Darüber hinaus ist er Mitbegründer und aktives Mitglied des iSAQB, des International Software Architecture Qualification Board.

Stefan Tilkov

(E-Mail: stefan.tilkov@innoq.com)
ist Geschäftsführer und Principal Consultant bei der innoQ Deutschland und beschäftigt sich dort mit leichtgewichtigen Ansätzen für Entwicklung und Architektur von komplexen IT-Systemen. In den letzten Jahren liegt sein Fokus auf SOA, Cloud Computing und dem sinnvollen Einsatz von Web-Technologien für Anwendungen und Anwendungsintegration.

Softwarearchitektur im Großen

Architekturmuster für Individualsoftware finden sich reichlich. Ob im Java/Java EE- oder im .NET-Umfeld: Architekten sind es gewohnt, eine einheitliche technologische Basis für das Gesamtsystem vorauszusetzen. Spätestens nach den ersten Versionen gilt für große Systeme jedoch häufig, dass sie zu schwerfälligen Monolithen werden, und die anfangs positive Homogenität wird zum Nachteil. Es ist daher häufig ein gute Idee, die Entwicklung eines Systems damit zu starten, dass man versucht, es in einzelne, voneinander weitgehend unabhängige Teilsysteme zu zerlegen – also eine Transformation in ein System von Systemen vornimmt. Daraus ergibt sich eine Reihe von Herausforderungen, denn nur einige Architektur- und Entwurfsmuster sind auf diese Ebene übertragbar und darüber hinaus werden neue Ansätze erforderlich – in technologischer, aber auch in organisatorischer Sicht.

Fragen Sie Softwarearchitekten nach dem, was sie für am wichtigsten halten, werden diese entweder antworten: „Der Fokus liegt auf der Konstruktion eines einzelnen Systems“ oder „Es geht um die Integration einer Reihe bestehender Systeme“. Aber aus wie vielen Systemen besteht die IT eines Unternehmens? Woraus ergeben sich die Grenzen eines Systems? Wie kommen wir dazu, einen bestimmten Systemschnitt durchzuführen?

Häufig liegt der Grund im Kontext eines Projektes: Die Aufgabenstellung bestimmt, oft basierend auf niemals hinterfragten Entscheidungen aus der Vergangenheit, was wir als System betrachten. Viele Architekten kommen zu Beginn eines Projektes gar nicht auf die Idee, zu hinterfragen, ob sie tatsächlich nur ein einziges oder mehrere Systeme entwickeln.

Für unsere Diskussion meinen wir mit „System“ ein eigenständiges Informationssystem mit eigener Datenhaltung, eigener Benutzeroberfläche und Fachlogik. Wir sind davon überzeugt, dass es sich fast immer lohnt, ein großes System in eine Reihe kleinere Systeme zu unterteilen, bevor für jedes einzelne davon eine interne Strukturierung vorgenommen wird.

Nachfolgend beschäftigen wir uns deshalb bewusst mit der Architektur von großen Systemen, unabhängig von Programmiersprachen, Bibliotheken, Frameworks oder Integrationsprodukten und Werkzeugen. Unser Fokus liegt dabei zunächst auf der Konstruktion neuer Systeme, nicht der Integration bestehender. Auf den Bezug zur Integration werden wir später zurückkommen.

Architekturebenen großer Systeme

Versetzen Sie sich einmal in die Lage des Architekten, der den allerersten Systemschnitt vornimmt: Auf dieser Ebene ist es relativ unerheblich, ob die Realisierung der einzelnen Systeme mit .NET oder Java erfolgt und welche Frameworks eingesetzt werden. Sie müssen entscheiden, ob Sie ein, drei oder sieben Systeme entwickeln, welche Aufgaben (Funktionen und Daten) in welchem System angesiedelt werden und wie diese – wenn es notwendig ist – miteinander kooperieren.

Wir können daraus Architekturebenen ableiten: Auf der Mikroebene befassen wir uns mit der internen Architektur eines einzelnen, nach außen über eine oder mehrere Schnittstellen kommunizierenden Systems, auf der Makroebene beschäftigen wir uns mit der Gesamtsystemlandschaft, die durch mehrere solcher untereinander kommunizierender Einzelsysteme gebildet wird. Beide Ebenen sind wichtig, aber es ist keine gute Idee, sie miteinander zu vermischen.

Um zu steuern, auf welche Art und Weise unterschiedliche Systeme miteinander integriert werden sollen, ist das Festlegen von Spielregeln wichtig. Es gibt dafür kein universell gültiges Muster, aber oft ist eine Unterscheidung nach Systemarten

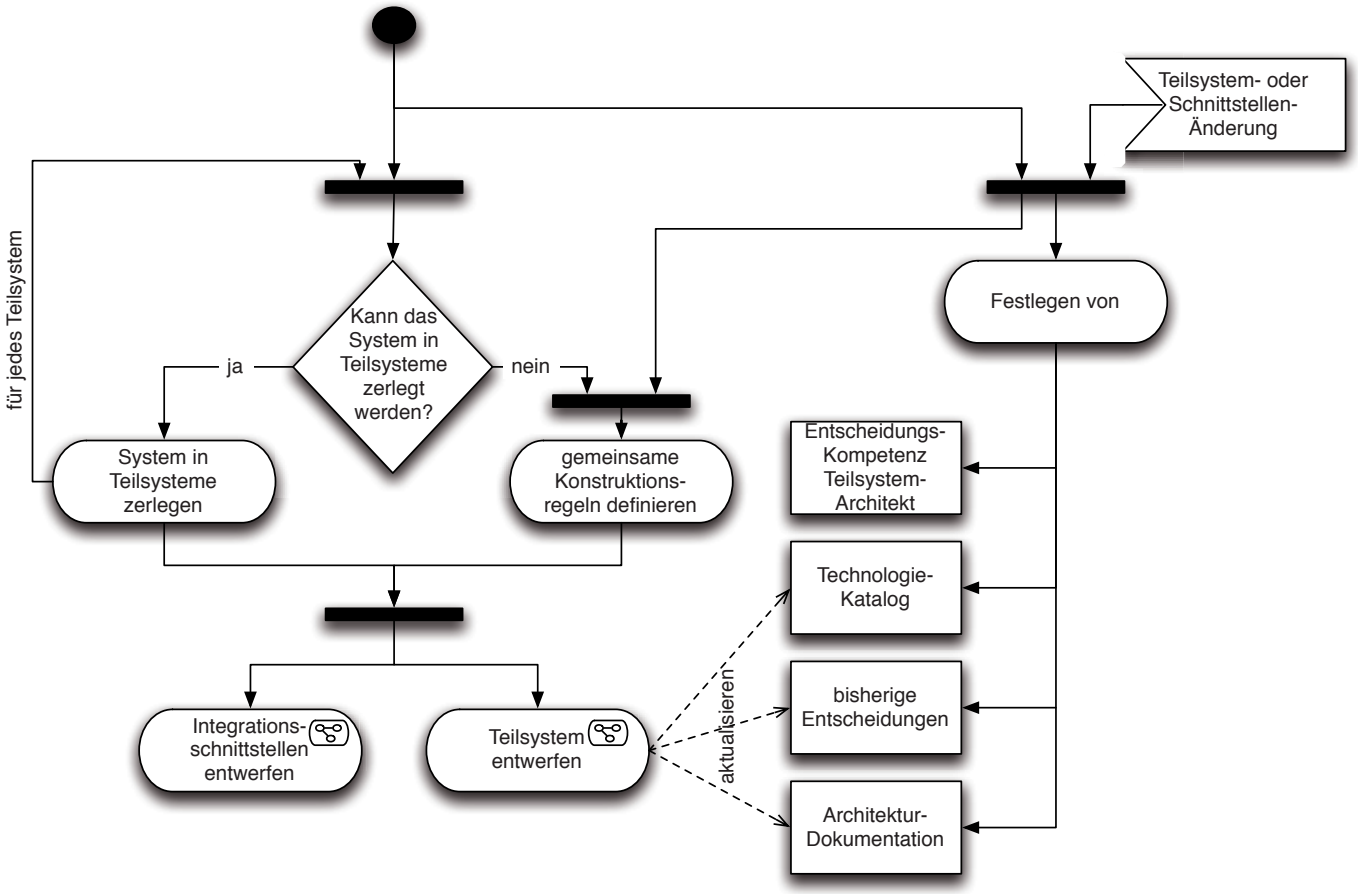

Abb.: Vorgehen bei der Architektur auf Makroebene

(Treiber, Client, Agent, Analysesystem, Online-Transaktionssystem usw.) sinnvoll.

Wenn wir, je nach Systemart, spezielle Spielregeln definiert haben, erfolgt damit der Entwurf und die Konstruktion jedes Systems unter Einhaltung gewünschter Rahmenparameter, die uns helfen, das große Ganze zu beherrschen.

Gesamtarchitektur und Implementierungsvorgaben

Das bislang Beschriebene unterstreicht die Erkenntnis, die wir beim Design solcher Systemlandschaften gewonnen haben: Es ist selbstverständlich wichtig, durch technologische Vorgaben eine möglichst homogene Gesamtlandschaft zu erzeugen, für den Systemschnitt sind diese jedoch weitgehend irrelevant. Innerhalb der einzelnen Systeme kann es durchaus sinnvoll sein, unterschiedliche Ansätze zu verfolgen.

So könnte z. B. in einem System, dessen Kernfunktion die Unterstützung von Analyseprozessen ist, eine nach Datawarehouse-Prinzipien entworfene Datenhaltung, vergleichsweise einfache Geschäftslogik und eine datenzentrierte Benutzeroberfläche sinnvoll sein; in einem auf wenige Benutzer zugeschnittenen System, das sich um komplexe Fachlichkeit kümmert, ist vielleicht eine mehrschichtige Architektur mit einem zentralen, in einer OO-Sprache implementierten Domänenmodell die richtige Wahl; für ein vor allem auf Leseoperationen ausgerichtetes Websystem passt vielleicht eine darauf optimierte Shared-Nothing-Architektur [SNA].

Der Idee, in jedem System die passenden Ansätze und Werkzeuge einzusetzen, steht natürlich der Wunsch nach einer Begrenzung der Optionen entgegen. Schließlich wollen Sie den Einarbeitungsaufwand minimieren und Skills, Werkzeuge und Bibliotheken wieder verwenden können.

Das ist völlig verständlich, und wir schlagen nicht vor, völlige Beliebigkeit walten zu lassen. Wichtig ist uns jedoch, dass die beiden Themen – Gesamtarchitektur und Implementierungsvorgaben – unabhängige Lebenszyklen haben: Ersteres ist sehr langlebig, während letzteres sich schneller ändern kann. Eine Vermischung der beiden Aspekte führt zu einer Lähmung, die sich oft erst nach einigen Jahren manifestiert – auf einmal wird schon eine Aktualisierung einer Logging-Bibliothek zum Problem, weil sie alle Systemkomponenten auf einmal betrifft, von einer Ablösung eines UI- oder Persistenzframeworks oder gar einem Ersatz eines Applikationsservers ganz zu schweigen.

Unsere Empfehlung ist daher: Lassen Sie besondere Strenge bei den Vorgaben walten, die sich aus dem Systemschnitt ergeben – und gewähren Sie im Gegenzug etwas mehr Freiheit für die Implementierung der einzelnen Systeme. Damit ist nicht gemeint, dass bei der Implementierung jeder machen kann, was er will: Wie stark Sie hier reglementieren wollen, hängt von vielen Parametern ab, unter anderem auch von so weichen Faktoren wie der etablierten Firmenkultur.

Wir haben gute Erfahrungen damit gemacht, die Wahl der Implementierungstechnologien durch definierte Zusicherungen, Rahmenparameter und Varianten zu reglementieren und verwendete wie ausgeschlossene Technologien in einem projektweiten Technologie-Katalog aufzuführen und zu diskutieren. Für alle Projektbeteiligten soll damit leicht ersichtlich sein, was für oder gegen den Einsatz einer Technologie in einem definierten Kontext spricht.

Andere Unternehmen legen eine Liste erlaubter Technologien, Werkzeuge und Bibliotheken fest, und nur die in diesem Katalog aufgelisteten dürfen verwendet werden. Aber auch wenn Sie so weit gehen, sollten Sie sich zu jedem Zeitpunkt bewusst sein, dass dies keine Relevanz auf Ihre Gesamtarchitektur hat – schon allein deswegen, weil sich diese Liste mit einer anderen Frequenz ändern muss, wenn Sie sich nicht selbst paralysieren wollen.

Strategic DDD

Wenn Sie nach Mitteln zur Strukturierung großer Systeme suchen, können Sie manchmal an unterwarteten Stellen fündig werden. Domain Driven Design (DDD, [DDD]) ist vielen Architekten als Ansatz für den Entwurf von Systemen bekannt: Ein auf Basis erprobter Muster entworfenes, technologieunabhängiges Domänenmodell wird ins Zentrum gestellt, die darin enthaltenen Klassen und Methoden als gemeinsame Sprache für Entwicklung und Fachseite genutzt. Oft wird jedoch der aus unserer Sicht eigentlich interessantere Teil des DDD-Ansatzes, „Strategisches DDD", ignoriert. Unter diesem Oberbegriff werden Konzepte zusammengefasst, die explizit für den Entwurf großer Systeme anwendbar sind (***siehe Kasten***).

So werden abgegrenzte Kontexte (Bounded Contexts) verwendet, um ein großes fachliches Modell in mehrere einzelne zu zerlegen. Kontexte definieren nicht nur ihr eigenes (Teil-)Modell, welche Komponenten für die Erbringung von Diensten verwendet werden sollen, sondern z. B. auch wie Schlüsselwerte zu interpretieren sind.

Zur übergreifenden Kollaboration können eine Reihe unterschiedlicher Muster eingesetzt werden – auf der von uns vorgeschlagenen Makro-Ebene sind vor allem Open Host Service und Published Language relevant, da sie zu unabhängigen, über Interprozessschnittstellen kommunizierenden Systemen passen. Shared Kernel ist ein gutes Mittel für die interne Entkopplung, ein Anti Corruption Layer oft die richtige Wahl zur Isolation gegenüber sich stark ändernden externen Systemen.

Entwurfsprinzipien im Großen

Beim Softwareentwurf finden wir immer wieder dieselben Konzepte als Leitlinien für „gutes" Design: Kapselung, lose Kopplung, hohe Kohäsion und Trennung nach Verantwortlichkeiten usw. Für objektorientierte Software hat sich das von Robert Martin („Uncle Bob") geprägte Akronym „SOLID" etabliert, das für die bekanntesten Prinzipien steht. Betrachten wir diese aus dem Kontext des Systemschnitts, können wir feststellen, dass sie leicht abgewandelt auch auf unserer Makroebene gelten:

Mit dem *Single Responsibility Principle* ist beim OO-Entwurf gemeint, dass jedes Objekt (bzw. jede Klasse) Verantwortung für genau eine einzelne Aufgabe haben sollte. Im Kontext des Systementwurfs gilt analog: Die einzelnen Systeme sollten eine klar abgrenzbare Verantwortung haben, also für eine zusammenhängende Menge von Aufgaben – Daten, Funktionen, Benutzeroberflächen usw. – zuständig sein, und auf der anderen Seite soll eine Verantwortung auch genau einem System zugeordnet werden.

Open/Closed Principle: „Offen für Erweiterung, geschlossen gegenüber Veränderungen" – diesem Grundsatz wird bei der Objektorientierung vor allem durch Polymorphie Rechnung getragen. Im Kontext unserer Makroarchitekturebene können Mechanismen, wie eine Plugin-Architektur, ein Registrierungs- und Callback-Mechanismus bzw. eine Pub/Sub-Architektur diese Rolle übernehmen.

Liskov Substitution Principle: Dies ist vielleicht das OO-spezifischste der Prinzipien, da es einen direkten Bezug zur Typsystemtheorie hat. Es besagt, dass Instanzen von Subklassen überall dort einsetzbar sein müssen, wo eine Instanz der Superklasse erwartet wird. Auf Systeme lässt sich dieses nur relativ abstrakt abbilden, z. B. bei den oben erwähnten Plugin-Architekturen.

Klarer ist die Übertragbarkeit beim *Interface Segregation Principle*: Kommunizieren Systeme untereinander, kommen häufig Service-Schnittstellen zum Einsatz, die eine Reihe von Funktionen bündeln (z. B. mehrere Operationen in einem einzelnen WSDL-Porttype beim Einsatz von Web-Services). Im OO-Kontext ist es sinnvoll, ein komplexes Interface in mehrere einzelne zu zerteilen, damit Nutzer, die eine Abhängigkeit nur zu einer einzelnen Operation haben, nicht ständig von Änderungen an anderen Operationen betroffen sind.

Das Gleiche gilt auch in unserem systemübergreifenden Kontext – einzelne Schnittstellen, die auf einen spezifischen Fall zugeschnitten sind, reduzieren Abhängigkeiten. (Eine Alternative ist übrigens der Einsatz von RESTful Web-Services, da hier eine so generische Schnittstelle verwendet wird, dass sie sich im Idealfall nicht ändert. Eine nähere Diskussion würde hier den Rahmen sprengen, siehe dazu [TuG06].)

Das *Dependency Inversion Principle* schließlich fordert, dass Abhängigkeiten immer vom Spezifischen hin zum Allgemeineren gerichtet sein sollen. Auch dies ist ein Prinzip, das auf allen Architekturebenen gilt und in beiden Fällen durch den Einsatz geeigneter Mechanismen, insbesondere dem Einführen von abstrakten Schnittstellen, adressiert werden kann.

Organisatorische Aspekte

Große Systeme unterliegen häufig unterschiedlichen Strömungen. Der Zeitgeist wandelt sich im Laufe der Entwicklung, Kompetenzverschiebungen in der Organisation führen softwareseitig zu grundlegenden Änderungen.

Der Hauptunterschied beim Entwurf von großen Systemen liegt wohl darin, dass es systemweit nicht die eine für einen Informationskontext alles entscheidende Autorität gibt, sondern mehrere. Architekten für große Systeme müssen daher von Anfang an mit konkurrierenden, widersprüchlichen oder redundanten Funktionen und Daten rechnen und diese geeignet kanalisieren.

Dazu ist die Aufteilung in einzelne, voneinander möglichst unabhängige Systeme ein probates Mittel, denn sie legt auch eine entsprechende Organisationsstruktur nahe. Nach dem Gesetz von Conway [MMM] spiegelt die Architektur eines Softwaresystems die Organisationsstruktur wider, in der sie entstanden ist. Im Umkehrschluss ist die Architektur ein Mittel, eine Organisationsstruktur vorzugeben – einzelne Systeme können von verschiedenen Teams konzipiert und entwickelt werden.

Command Query Responsibility Segregation (CQRS)

Große Systeme verfügen häufig über verschiedene Benutzergruppen, die das System auf unterschiedlichste Arten nutzen: Während einige Tausend Sachbearbeiter pro Tag im Akkord Daten pflegen, recher-

chieren, Auskünfte einholen und Vorgänge bearbeiten, möchte eine – sowohl unbekannte als auch theoretisch unbegrenzte – Anzahl von Endkunden direkt Teile des Systems als Self-Service nutzen, während von einer kleinen Gruppe von Unternehmenslenkern die aktuellen Daten zur Echtzeit-Auswertung aufbereitet werden.

Im Kleinen hat Bertrand Meyer den Begriff Command Query Separation [CQS] geprägt: Eine einzelne Methode kapselt entweder eine Aktion, die eine Änderung bewirkt, ohne ein Ergebnis zu liefern, oder sie liefert Daten, ohne Seiteneffekte zu erzeugen. Im Großen können wir die Systemerstellung unterstützen, indem wir das auf CQS aufbauende CQRS für alle Schnittstellen anwenden, die zwischen den Systemen erstellt werden: Schnittstellen für die Manipulation von Daten bzw. das Ausführen von Aktionen werden von den Schnittstellen zur Abfrage getrennt. Dies erlaubt auf hoher Abstraktionsebene unterschiedliche Zusicherungen zu machen, die sich anders nur schwer einhalten ließen.

CQRS bedeutet im Wesentlichen genau das: Anstatt eine Schnittstelle mit beliebigen Operationen zu entwerfen, entwirft man nun Schnittstellen mit sich ändernden Operationen und Schnittstellen mit Abfragen separat.

Aus einer solchen Trennung heraus ergibt sich die Möglichkeit, die Systeme, die diese Schnittstellen implementieren, sehr unterschiedlich umzusetzen: So könnten lesende Operationen ohne große Umwege direkt aus einer Datenbank beantwortet werden, die genau dafür optimiert ist, während schreibende Operationen durch ein anderes System mit einer komplexen Geschäftslogikschicht bearbeitet werden. Die Notwendigkeit zur Datensynchronisation und die damit verbundene Komplexität wird dabei möglicherweise durch die getrennte Skalierbarkeit der beiden Aspekte aufgewogen – und tatsächlich ist ein Verhältnis Lesen zu Schreiben von 10:1 nichts Unübliches.

Übergreifende Themen

Die Welt ist natürlich nicht so einfach, dass sich ein Kontext einfach auf eine Menge von Systemen abbilden lässt, die vollständig voneinander isoliert sind. Es gibt auch bei einer Aufteilung in einzelne Systeme eine Reihe von Themen, die übergreifender Natur sind und daher mit dem Blick aufs Ganze adressiert werden müssen. Dazu zählen in aller Regel zumindest die Themen UI-Integration, Transaktionssteuerung, Authentisierung und Autorisierung sowie systemübergreifende Prozesse.

Die Themenbereiche UI-Integration und Authentisierung/Autorisierung ergeben sich aus dem verständlichen Wunsch eines Endanwenders, nicht durch die Architekturentscheidungen für einen bestimmten Systemschnitt belästigt zu werden: Da wir davon ausgehen müssen, dass in einem großen System einzelne Bereiche unterschiedliche Lebenszyklen haben, müssen wir damit rechnen, dass durch Zukäufe oder Neuentwicklungen mit aktuellen Technologien Arbeitsabläufe nicht nur innerhalb einer homogenen Anwendung abgearbeitet werden, sondern dass vielmehr mehrere technologisch-heterogene, spezialisierte Anwendungen innerhalb eines Flusses verwendet werden müssen. Die Integration an der Oberfläche ist daher ein häufiges Thema in heterogenen bzw. langlebigen Systemumgebungen.

Sowohl für die menschlichen Benutzer als auch für programmatische Konsumenten unserer Systeme gilt, dass Authentisierung einheitlich gelöst werden muss. Insbesondere für den Endanwender gilt, dass er sich nur einmal anmelden und nicht bei jedem System separat authentifizieren möchte.

Die Autorisierung stellt im Großen insbesondere aus Governance-Gesichtspunkten eine Herausforderung dar. Es ist nicht immer wünschenswert, für ein neues System völlig neue Rollen zu schaffen. Häufig wünscht man die Abbildung des Organisationsmodells und seiner Rollen auf die Softwaresysteme.

Dies gestaltet sich nicht immer einfach, wenn verschiedene Softwaresysteme mit unterschiedlichsten Technologien und Berechtigungssystemen miteinander zu einem Gesamtsystem integriert werden sollen. Auch wenn das Thema Authentisierung/Autorisierung grundsätzlich gelöst ist, erfordert es immer wieder spezielle Aufmerksamkeit. Zum Beispiel stoßen bei der Berechtigungsprüfung im Prepaid-Umfeld althergebrachte Lösungen schnell an ihre Grenzen.

Systemübergreifende Prozesse erfordern von uns Architekten leider mehr als nur das Hinzukaufen einer Workflow-Komponente. Wir müssen uns über Kompensationen von Transaktionen genauso Gedanken machen wie über die fallbezogene Ansteuerung unterschiedlichster Anwendungen, die in manchen Fällen von der Belegung irgendwelcher fachlicher Daten abhängt. Dann müssen wir teilweise in der Lage sein, Anwendungen auf dem Arbeitsplatzrechner zu starten und zu einer definierten Maske oder in einen definierten Zustand zu bringen. All dies bietet auch heute noch höchst anspruchsvolle technische Herausforderungen.

Der Charme des Monolithen

Obwohl wir überzeugt sind, dass in der Mehrzahl der Fälle gerade große Entwicklungsvorhaben davon profitieren, wenn ein Ansatz mit mehreren unabhängigen Systemen gewählt wird, gibt es natürlich auch negative Seiten. So ist jede Form der losen Kopplung mit zusätzlichen Kosten behaftet, sowohl in der Entwicklung (durch initialen Mehraufwand) als auch zur Laufzeit (durch erhöhten Kommunikationsbedarf). Das Aufsetzen des Gesamtsystems sollten Sie gesondert softwaretechnisch unterstützen, um erforderliche mehrere Schritte fehlerfrei ausführen zu können.

Will man sich diesen Aufwand sparen, kann man durch Zusammenpacken der Software in ein Installationsmedium (eine Deployment-Unit o. ä.) diesen zumindest verringern. In dieser Beziehung ist der Monolith einer Menge von Systemen überlegen – wodurch sich die Vielzahl historisch gewachsener Monolithen erklärt, die aus heutiger Sicht besser als kollaborierende separate Systeme entwickelt worden wären.

Fazit

Systeme signifikanter Größe unterteilen wir am Besten in kollaborierende autarke Teilsysteme. Dies schützt vor unwartbaren und nicht zu beherrschenden Gargantua-Monolithen. Ob wir bewusst eine Trennung in Teilsysteme vornehmen oder nicht, das Gesamtsystem unterliegt unterschiedlichsten Änderungen.

In den einzelnen Bereichen werden sich diese Änderungen aber mit jeweils eigener Geschwindigkeit ergeben: Die Anforderungen aus Fachbereich A wechseln nicht so schnell wie die Anforderungen aus dem Vertrieb. Manche Technologien tragen für längere Zeit, andere ändern sich mit höherer Frequenz, gelegentlich

werden völlig neue Ansätze notwendig. So erleben diverse Unternehmen aktuell, dass das Öffnen einiger Funktionen für alle Internetnutzer andere Architekturmuster und ggf. einen Austausch der eingesetzten Komponenten erfordert.

Sie sollten daher frühzeitig darauf achten, in welchen Kontexten welche Aussagen und Zusammenhänge gelten. Wir orientieren uns hier am Strategic Domain-Driven Design, das uns verschiedene Hilfsmittel an die Hand gibt, um mit unterschiedlichen Arten von Abhängigkeiten umzugehen.

Handwerklich wenden wir beim Entwurf von großen Softwaresystemen die gleichen Prinzipien an, wie bei kleineren. Entscheidend ist allerdings, dass wir bei großen Systemen viel genauer auf die Gestaltung von Spielregeln und Zusicherungen Wert legen müssen, um ein möglichst konsistentes tragfähiges System zu erhalten. Die Festlegung auf erlaubte Technologien ist dafür nicht ausreichend. Darüber hinaus müssen wir sicherstellen, wie die Integration von Komponenten und Teilsystemen unabhängig von den Technologien erfolgen kann.

Lohn des Mehraufwands gegenüber einem monolithisch entworfenen System sind die einfachere Parallelisierbarkeit in der Entwicklung, die bessere Beherrschbarkeit der einzelnen Teilsysteme in technischer, fachlicher und organisatorischer Sicht, vor allem aber die Möglichkeit zur geordneten Evolution des Gesamtsystems – und das oft nicht erst in Weiterentwicklung und Wartung, sondern bereits vor dem Produktivgang der ersten Version. ■

Für die Abgrenzung von eigenständigen Kontexten bietet DDD eine Reihe von Mustern:

Bounded Context	Ein abgegrenzter Kontext definiert einen Teilbereich innerhalb eines großen fachlichen Modells, der gegenüber anderen isoliert ist. Innerhalb verschiedener Kontexte können ähnliche oder gleiche Klassen enthalten sein, i. d. R. mit den jeweils für den Bereich relevanten Daten/Funktionen.
Customer/Supplier	Bei Customer/Supplier handelt es sich eher um ein organisatorisches als um ein fachliches Muster: Gemeint ist, dass ein Team ein Modell entwickelt, das von einem anderen genutzt wird. Beide Parteien definieren dabei gemeinsam, wie die Schnittstelle zwischen den Kontexten aussieht. Änderungen können vom Anbieter nicht einfach durchgeführt, sondern müssen mit dem Nutzer abgestimmt werden.
Conformist	Im Gegensatz zum kooperativen Customer/Supplier-Modell ist in diesem Fall der Nutzer ein reiner Konsument, der kein Mitbestimmungsrecht über die angebotene Schnittstelle hat und deshalb auf eine Transformation von Modellen verzichtet.
Anticorruption Layer	Wird aus einem Kontext heraus ein anderer genutzt, kann eine Antikorruptionsschicht verhindern, dass Änderungen aus dem „fremden“ Modell unmittelbar auf das eigene Modell durchschlagen. Verantwortlich für diese Schicht ist der nutzende Kontext, der damit seine eigene, typischerweise idealisierte Sicht auf den anderen Kontext realisiert.
Open Host Service	Mit diesem Muster ist das Anbieten einer Service-Schnittstelle gemeint, unabhängig von einer bestimmten Technologie, aber häufig auf Basis einer Interprozesskommunikation.
Published Language	Zwischen verschiedenen Kontexten erleichtern gemeinsam genutzte Datenformate den Austausch von Informationen. In der Regel wird dieses Muster zusammen mit Service-Schnittstellen verwendet, denkbar ist aber auch eine Nutzung zum Beispiel über Datei-Import/Export.
Shared Kernel	Mit einem gemeinsam genutzten Kern ist ein Teilmodell gemeint, dass von mehreren verschiedenen Kontexten genutzt wird, d. h., die Kontexte teilen sich gemeinsamen Code, den sie ggf. auf ihre spezifischen Bedürfnisse adaptieren.
Separate Ways	Das Beste aller Integrationsmuster – die Vermeidung der Integration – ist häufig nicht anwendbar, aber aus Sicht einer losen Kopplung naturgemäß optimal: Wenn zwei Kontexte nichts miteinander zu tun haben müssen, sollten sie auch nicht miteinander integriert werden.

Kasten

Referenzen

[AUP] http://catb.org/~esr/writings/taoup/html/
[CQS] http://martinfowler.com/bliki/CommandQuerySeparation.html
[CQRS] http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/
[DDD] Domain-Driven Design: Tackling Complexity in the Heart of Software, Eric Evans, Addison-Wesley 2003
[MMM] Gesetz von Conway in The MythicalManMonth, Essays on Software Engineering, Anniversary Edition by Frederick P. Brooks, Jr., Published by Addison Wesley 1995
[SOLID] http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
[SNA] http://de.wikipedia.org/wiki/Shared_Nothing_Architecture
[TuG06]: http://www.sigs-datacom.de/fachzeitschriften/objektspektrum/archiv/artikelansicht.html?tx_mwjournals_pi1%5Bpointer%5D=0&tx_mwjournals_pi1%5Bmode%5D=1&tx_mwjournals_pi1%5BshowUid%5D=1912