Evolution Service-Orientierter Systeme

Harry M. Sneed
ANECON GmbH
Alserstrasse 4, A-1090 Wien
Harry.Sneed@T-Online.de

Abstract: In diesem Beitrag geht es um die Evolution, bzw. Wartung und Weiterentwicklung, Service-orientierter Systeme (SoS). Zunächst werden die besonderen Eigenschaften solcher Systeme vor dem Hintergrund einer Service-orientierter Architektur(SoA) erläutert. Danach werden drei Probleme vorgestellt, die im Zusammenhang mit der Evolution Service-orientierter Systeme stehen: die Mehrsprachigkeit, die gestiegene Komplexität und die vermehrte Abhängigkeit von externen Lieferanten. Der Beitrag geht darauf ein, welchen Einfluss diese Probleme auf die Evolution der Software haben werden. Anschließend werden alternative Möglichkeiten aufgezeichnet mit diesen Problemen fertig zu werden.

Schlüsselbegriffe: Software-Evolution, Web-Services, service-orientierte Architekturen, Wartung, Weiterentwicklung, Reverse-Engineering, Regressionstest.

1. Software-Evolution

Der Begriff Software-Evolution wurde bereits in den 70er Jahren von Les Belady und Manny Lehman geprägt. Im Jahre 1976 erschien ihre erste Studie über die Evolution des OS-Betriebssystems bei der IBM [BL75]. Schon damals wurde festgestellt, dass Software sich wandeln muss wenn sie überleben will. Anderseits, je mehr sie sich wandelt, desto brüchiger wird sie. Die Komplexität steigt und die Qualität sinkt. Irgendwann lässt sich die Software nicht mehr verändern und muss ausgemustert werden [LB85]. Allerdings haben Belady und Lehman dieses Erosionsprozesses grob unterschätzt. Sie konnten damals nicht ahnen, dass er Jahrzehnte dauern könne. Sie haben auch die Möglichkeit zur Sanierung der Software unterschätzt. Durch regelmäßige Überholungen lässt sich die Lebensdauer eines Softwaresystems verlängern. Als sie ihre Gesetze der Software-Evolution formulierten, steckte die Reengineering-Technologie erst in den Kinderschuhen. Heute haben wir Reengineering-Techniken mit denen wir sowohl die Komplexität einer Software abbauen als auch die Qualität steigern können [Sn02].

Trotzdem behalten die Gesetze der Software-Technologie noch ihre Gültigkeit. Nichts auf dieser Welt ist unsterblich, auch Software nicht. Sie wird immer älter weil die Technologie in der sie implementiert ist, immer weiter hinter den neuen Technologien zurückfällt und keine Reengineering-Techniken der Welt kann sie erneuern [Sn04]. Ein in Java migriertes COBOL System bleibt in seinem Wesen ein COBOL System obwohl es in Java kodiert ist. Der Geist der COBOL Urväter lässt sich immer erkennen, nicht nur in den Datenstrukturen und in den Algorithmen, sondern gerade in der Gesamtarchitektur des Systems. Wir können also den Tod eines Softwaresystems nicht verhindern, aber wir können ihn durch geeignete Maßnahmen noch lange vor uns herschieben, vielleicht bis die zuständigen Mitarbeiter pensioniert sind. Dieser Autor hat beobachtet, dass Systemablösungen in der Regel mit einem Generationswechsel in einer Firma verbunden sind. Wenn die Alten gehen folgt ihnen ihre alte Software nach. Dies könnte als Ergänzung zu den Gesetzen der Software-Evolution angesehen werden. Ein Softwaresystem lebt nur so lange wie seine Väter.

Jedenfalls ist Software-Evolution heute ein fester Begriff, inzwischen auch in dem deutschsprachigen Raum. Seine Bedeutung geht über das hinaus was allgemein unter Softwarewartung verstanden wird. Softwarewartung impliziert die Erhaltung der Software in seiner ursprünglichen Form. Sie wird zwar korrigiert, adaptiert und optimiert, aber sie bleibt in ihrer Funktionalität wesentlich gleich [vH05]. Unter Software-Evolution verstehen wir hingegen Wartung und Weiterentwicklung. Die Software wird nicht nur erhalten, sondern ständig erweitert und ergänzt um neue Funktionalitäten und neue technische Eigenschaften. Sie kann sogar in eine neue Sprache und/oder in eine neue Umgebung versetzt werden. Wichtig dabei ist, dass ihr betrieblicher Wert für ihre Benutzer im Sinne der Werttheorie zunimmt.

Nach Bennett und Rajlich ist auch die Zeit für die Software-Evolution begrenzt. Irgendwann lohnt es sich nicht mehr in ein System weiter zu investieren, nämlich dann wenn die Kosten der Erweiterung den Nutzen übersteigen. In dem Moment tritt das System in eine Erhaltungsphase in der es nur noch korrigiert und angepasst wird. Diese Phase der Wartung kann eine Weile dauern, dann wird das System durch die technologische Weiterentwicklung überholt und muss abgelöst werden. Es lohnt sich auch nicht mehr das System zu erhalten [BR00].

2. Service-orientierte Systeme

Heute stehen wir vor einem erneuten Aufbruch in der Software-Technologie – der Übergang zu service-orientierten Systemen. Viele reden davon, einige Wenige haben ihn schon vollzogen. Hier stellt sich nun die Frage, was ist eigentlich ein Service-orientiertes System und wie unterscheidet es sich von bisherigen Systemen.

Ein Service-orientiertes System könnte man als ein System mit einer service-orientierter Architektur bezeichnen, aber das wäre zu kurz gegriffen. Denn die Architektur bestimmt den Rahmen innerhalb dessen das jeweilige System operiert. Eine SOA ist wie ein Mega-Betriebssystem, manche betrachten sie sogar als Betriebssystem des 21 Jahrhunderts [KBS05]. Sie fungiert als Plattform für die einzelnen Benutzerapplikationen. Ein System ist wiederum aus der Sicht seiner Benutzer eine einzelne Applikation wie z.B. ein Buchungssystem, ein Kontoführungssystem oder ein Lagerhaltungssystem. Das Applikationssystem benutzt nur die Architektur. Demnach ist ein Service-orientiertes System ein Applikationssystem, das in einer Service-orientierter Architektur eingebettet ist und Services, sprich Web-Services, benutzt [SL07].

Ein Software-Service ist eine Software-Komponente die eine allgemeingültige Funktion erfüllt, die mehrere andere Software-Komponente benutzen können [NL05]. Früher hat man sie im Deutschen als Dienstprogramme und im englischen als Utilities bezeichnet. Ein typisches Dienstprogramm war ein Sort, das ein Applikationsprogramm aufrief, um eine Menge Objekte nach einem bestimmten Kriterium zu sortieren. Das Program übergab dem Dienstprogramm eine ungeordnete Menge mit Duplikaten und bekam eine geordnete Menge ohne Duplikate zurück. Derartige software-technische Dienste sind schon seit dem Anfang der elektronischen Datenverarbeitung ein fester Bestandteil fast aller Applikationssysteme. Es beschränkt sich auch nicht auf reine technische Dienste. Es wurden ebenfalls schon seit Beginn der Programmierung allgemeingültige fachliche Dienste benutzt. Ein hervorragendes Beispiel ist die FORTRAN mathematische Subroutinen-Bibliothek. Darin gab es für jede erdenkliche mathematische Funktion eine FORTRAN Subroutine, die über eine Call-Schnittstelle aufgerufen wurde. Das nutzende Programm gab die Argumente als Eingangsparameter vor und bekam das Ergebnis als Rückgabewert zurück. Allerdings musste man diese Bibliothek bestellen und in die eigene FORTRAN Umgebung einbinden.

Ähnliche Subroutinen-Bibliotheken gibt es auch für statistische und graphische Funktionen. In Objekt-orientierten Systemen erfüllen diese Rolle die Klassenbibliotheken [Lo93]. Es gibt Klassenbibliotheken für alles möglich – für statistische Funktionen, für mathematische Funktionen, für graphische Anzeigen und auch für fachliche Dienste wie Kalenderführung. Kein Programmierer würde auf die Idee kommen selber alles zu programmieren wenn sovieles vorprogrammiert vorliegt. Auch wenn er auf diese völlig irrationale Idee käme, würde er sich zunächst eine eigene Klassenbibliothek mit allgemeingültigen Klassen aufbauen um sich später die Arbeit zu erleichtern.

So gesehen ist das Service-Konzept absolut nichts Neues und verdient keine besondere Aufmerksamkeit. Was neu ist, ist wie die Dienste angeboten werden. Bisher waren die Dienste mit der jeweiligen Programmiersprache verbunden. Es gab eine FORTRAN Common Subroutine-Library oder die PL/I eingebaute Funktionen oder die COBOL Dienstprogramme. Später gab es C++ und Java Klassenbibliotheken. Es war nicht ohne weiteres möglich Dienste außerhalb des eigenen Sprachraums zu benutzen. Der erste Ansatz den Zugriff auf fremdsprachliche Dienstprogramme zu ermöglichen war die IBM Cross System Product Architektur im Rahmen der AD-Cycle Entwicklungsumgebung, aber dieser Ansatz war proprietär [Mo91]. Das IBM Systems Application Architecture (SAA) war ein Vorgriff auf SoA. Es folgte die Open Systems Foundation und die Common Object Request Broker Architektur – CORBA – die eine Verbindung über Sprach- und Betriebssystemgrenzen hinweg zugelassen hat [SC99]. Damit konnten C++ Programme auf einer UNIX Plattform auf COBOL Programme auf einem Mainframe-Rechner zugreifen. Dennoch hat aber auch dieser Ansatz noch nicht die Bezeichnung „Service-orientiert“ verdient.

Service-orientierte Systeme enthalten nicht nur allgemeingültige Dienstkomponente, sie sind auch noch verteilt. Die Services, bzw. die Server-Komponente, sind nicht auf dem gleichen Rechner wie die Client-Komponente. Sowohl die Server-Komponente als auch die Client-Komponente sind auf verschiedene Netzknoten verteilt. Verbunden werden sie miteinander über eine Middleware-Infrastruktur. In klassischen Client/Server Systemen war diese Middleware proprietär. Um seine verteilte Komponente miteinander zu verbinden, musste der Anwender ein Middleware-Produkt kaufen, ein sogenanntes Enterprise Application Integration Framework [Ke02]. Enterprise Application Integration – EAI – zielte darauf hin, eine Schicht von vorgefertigten Software Services, bzw. allgemeingültige Dienstkomponente, bereitzustellen [CHK06].

Inzwischen kann jeder über das Internet auf entfernte Web-Services zugreifen. Er muss nur eine Berechtigung haben. Man braucht keine aufwendige EAI-Infrastruktur, es sei denn für Sicherheitszwecke. Das World Wide Web ermöglichte den unbegrenzten Zugang zu entfernten Dienste am anderen Ende der Welt. Das Web wurde zum wichtigsten Instrument der Systemintegration und alles bewegte sich in diese Richtung [SnSn03]. Falls man aber einen entfernten Dienst in eine seiner Applikationen einbauen möchte, sollte man zuerst sichern, dass der entfernte Dienst immer verfügbar bleiben und immer die gleichen Ergebnisse liefern wird. Es ist prinzipiell möglich Web Services aus völlig unterschiedlichen Quellen in einer einzigen Service-orientierter Applikation einzubinden. Damit bekommen wir aber nicht nur ein Problem mit diversen Sprachen sondern noch ein Problem mit diversen Lieferanten.

Hinzu kommt ein drittes Problem auf uns zu und zwar die steigende Komplexität derartig hybriden Systeme. Komplexität wächst mit der Zahl der Beziehungen. Je mehr fremde Services in eine Applikation, bzw. in einen Geschäftsprozess, eingebaut werden, desto mehr Beziehungen wird es geben. Vor allem dann wenn viele Prozesse die gleichen Services verwenden, wird die Zahl der Netzbeziehungen relativ zur Zahl der Netzknoten ansteigen. Jede zusätzliche Nutzung eines bereits benutzen Services belastet die Schnittstelle zu dem Service und erhöht die Komplexität des Gesamtsystems [CA88].

Das was das heutige Service Angebot ausmacht ist dass die Services über das Web erreichbar sind. Die Tatsache, dass sie eine allgemeingültige, neutrale Schnittstellensprache haben, nämlich WSDL, macht sie von der Sprache der Benutzerkomponente unabhängig. Es spielt keine Rolle mehr in welcher Sprache die Services verfasst sind. Sie können in jeder beliebigen Sprache geschrieben werden, sie müssen nur die neutrale Schnittstellensprache bedienen können. Das macht es möglich Programmiersprachbarriere zu überwinden. Moderne Service-orientierter Systeme sind also mehrsprachlich. Das unterscheidet sie von konventionellen IT-Systemen, die in der Regel mit einer Sprache implementiert wurden.

Ein weiterer Unterschied ist die Verfügbarkeit der Services. In einem klassischen Client/Server System gehören die Server-Komponenten zum Anwender. Er kann darüber bestimmen was mit ihnen geschieht. Er trägt auch die Verantwortung für deren Wartung und Weiterentwicklung. Web Services sind hingegen irgendwo im Cyberspace. Sie könnten im Besitz des Anwenders sein, müssen es aber nicht. In vielen Fällen werden sie von dem einzelnen Benutzer nur benutzt. Falls mehrere Web-Services zum gleichen Zweck angeboten werden, können sie nach Bedarf ausgetauscht werden. Der Vorteil für den Benutzer ist dass er sie nicht entwickeln und pflegen muss. Der Nachteil ist, dass er sie nicht über sie frei verfügen kann. Sie haben einen eigenen Lebenszyklus, der nicht mit dem der benutzenden Systeme übereinstimmt. Das Web-Service, das der Benutzer heute beansprucht muss nicht mit dem gleichen Service, den er gestern benutzt hat, identisch sein. Er könnte sich geringfügig anders verhalten und andere Ergebnisse liefern. Eventuell wurde er über Nacht durch eine neue Version ersetzt. Wenn der Benutzer kein Service Level Agreement hat muss der Service nicht einmal die gleiche Funktionalität bringen, geschweige denn zur gleichen Qualität. Die Verantwortung für die Anpassung an der neuen Schnittstelle liegt beim Benutzer [TGH04].

Die Tatsache, dass die benutzten Services nicht im Besitz des Benutzers sind hat vielleicht den größten Einfluss auf die Evolution Service-orientierter Systeme, sogar mehr als den der Mehrsprachigkeit. Bisher waren die Dienstprogramme, bzw. die allgemeingültigen Subroutinen und die Klassenbibliotheken, im Besitz des jeweiligen Anwenders. Entweder hat er sie gekauft oder, wie im Falle des Open Source, geschenkt bekommen. Jedenfalls konnte er mit ihnen machen was er wollte. Somit konnte er sie als fertige Komponente in seine Anwendungssoftware einbauen. Sie wurden einen festen Bestandteil davon und konnten zusammen mit den eigenentwickelten Bestandteilen fortgeschrieben werden. Oder sie blieben unverändert sowie der Benutzer sie bekommen hat. Er musste nicht fürchten, dass sie sich über Nacht verändern.

Falls der Benutzer darauf beharrt die Service-Komponente selber zu besitzen, wird es keinen wesentlichen Unterschied zu den klassischen Client/Server Systemen geben, es sei denn das er jetzt das Internet benutzt um auf sie zuzugreifen. Statt über eine Client/Server Schnittstelle wie CORBA-IDL werden sie über eine Webschnittstelle angestoßen. Was die Evolution der Systeme anbetrifft bleibt alles gleich. Der Benutzer muss dafür sorgen, dass die Evolution der Services mit der Evolution der Klienten synchronisiert ist.

Wenn also von Service-orientierter Systeme die Rede ist, müssen wir unterscheiden zwischen Service-orientierten Systemen die im vollen Besitz des Benutzers sind und Service-orientierten Systemen bei denen die benutzten Services nicht der Kontrolle des Benutzers unterliegen, z.B. Software on Demand [Sn07]. Im letzteren Fall stellen sich neue Herausforderungen für die Evolution der Anwendungssysteme heraus, Herausforderungen jenseits der Verteilung und der Mehrsprachigkeit. Alles hat seinen Preis. Der Preis für die Nutzung von Services, die nicht im eigenen Besitz sind, ist das Abhängigkeitsverhältnis zu den Providern jener Services. Zum Einen wird die Evolution, sprich Weiterentwicklung, der eigenen Software von deren Evolution bedingt und zum Anderen wird der Benutzer gezwungen die Evolution der benutzten Services nachzuvollziehen. Kurzum, der Benutzer ist nicht mehr autark [NV00].

Zusammenfassend ist hier festzustellen, dass der Übergang zu service-orientierten Systemen die IT-Benutzer einerseits von der lästigen Pflege und Weiterentwicklung sämtlicher Software-Bausteine befreit, sie jedoch anderseits mit neuen übergeordneten Problemen belastet. Die drei Hauptprobleme sind:

- Die Mehrsprachigkeit,
- Die Schnittstellenkomplexität und
- Die Lieferantenabhängigkeit

Wer Service-orientierte Systeme einsetzen möchte, muss diese drei Probleme im Griff bekommen. Der Rest dieses Beitrages befasst sich mit möglichen Lösungen zu diesen Problemen.

3. Das Problem der Mehrsprachigkeit

In früheren IT-Systemen hatten Anwender in der Regel mit nur einer Programmiersprache zu tun. Alle Entwickler haben diese eine Sprache mehr oder weniger beherrscht. Es gab COBOL Betriebe, C Betriebe oder Java Betriebe. Hinzu kamen allenfalls eine Jobsteuerungssprache, eine Datenbankzugriffssprache wie DLI, ADABAS oder SQL und eine Benutzeroberflächensprache wie BMS, MFS oder HTML. In einem service-orientierten System werden viele Sprachen und damit vielen Technologien zusammenkommen. Die Systeme werden multi-kulturell.

Für die Benutzeroberflächen kommen außer HTML auch XHTML, XSL und neuere Internetsprachen wie PHP und Flex ins Spiel. Für die Systemschnittstellen können neben IDL, XML und WSDL verwendet werden. WSDL ist die Sprache für die Anbindung der Web Services. Die Web Services selbst können in jeder beliebigen Sprache verfasst werden – PL/I, COBOL, C++, C#, Java, ADA, PERL, Natural, ABAP, usw. Schließlich gibt es neben der traditionellen Clientsprachen wie C++, C#, Java und PHP die neuen Geschäftsprozesssprachen wie BPEL, BPMN und JCom sowie weitere Domain Specific Sprachen. Der Turm von Babel lässt grüßen [Jon03].

Ein IT-Betrieb der die Sourcen seiner Service-orientierten Systeme zwecks der Messung oder der Nachdokumentation analysieren möchte muss Werkzeuge einsetzen, die alle seiner benutzten Sprachen abdecken. Denn es bringt wenig allein nur die Clientseite zu analysieren ohne die Services und deren Schnittstellen einzubeziehen. Man will schließlich wissen wie groß und wie komplex das ganze System ist oder wie ist das ganze System zusammengesetzt, und das kann man nur herausbekommen wenn man alle Bestandteile analysiert.

Eine weitere Folge der Mehrsprachigkeit ist der erhöhte Personalbedarf. Ein einzelner Mitarbeiter wird kaum mehr als zwei Sprachen beherrschen. Viele werden nur eine einzige Sprache beherrschen. Dennoch braucht der Systembetreiber für die Evolution des Systems Spezialisten für jede Sprache damit er jede Sprache fortschreiben kann. Auch wenn die einzelnen Spezialisten meistens nicht ausgelastet sind, werden sie für alle Fälle gebraucht. D.h. der Anwender braucht nicht weniger sondern noch mehr Wartungspersonal als bisher der Fall war. Die Erhaltungskosten Service-orientierter Systeme steigen.

Eine besondere Herausforderung stellt die Pflege und Weiterentwicklung von Legacy Softwarebausteine, die als Web Services gekapselt und angeboten werden. Dies wird öfters notwendig sein wenn ein Standard Service nicht verfügbar und die Entwicklung einer neuen Service zu teuer ist. In dem Fall muss der Anwender auf seine bestehende Software zurückgreifen und sie web-fähig gestalten. Die Kapselungstechnologie ist eine ausgereifte und weit verbreitete Methode zur Wiederverwendung bestehender Softwarekomponente als Services [Sn06]. Sie ist zudem schnell und preiswert, vor allem wenn die Kapselung automatisch stattfindet. Sie hat aber einen Haken. Der alte Code bleibt und lebt weiter als Bestandteil, bzw. als Organtransplant, des neuen service-orientierten Systems. Als solcher muss er korrigiert, adaptiert und möglicherweise erweitert werden. Der Anwender braucht also Personal, das mit dem alten Code umgehen kann. Auch wenn diese eingebauten Altbausteine sich nur geringfügig ändern, braucht man trotzdem Personal um sie zu betreuen. Ihr Know-How ist für die Erhaltung dieser Services, sprich für die Kontinuität der Dienstleistung, unentbehrlich.

Die Lösung zu diesem Personalproblem liegt in der Konzentration der Ressourcen. Nur große Organisationen können es sich leisten so viele teure Spezialisten parat zu halten. Die Anwender werden gezwungen ihre Personalressourcen in größere IT-Service Zentren zusammen zu fassen. Kleine und mittelgroße Anwender, die es sich nicht leisten können so viel Know-How anzueignen und zu halten, müssen zumindest Teile der Evolutionsarbeiten outsourcen. Entweder benutzen sie fertige Services on Demand oder sie übergeben ihre maßgeschneiderten Services an einen externen Partner, der sie für sie wartet. Sie werden selber nur Teile ihrer Gesamtsysteme, wo möglichst die Clientkomponente selber pflegen können. Der Preis der anspruchsvollen und diversifizierten Service-Technologie ist die Aufgabe der Selbstständigkeit. Nur die größten Anwender werden in der Lage sein alles selber zu machen. Dem Rest wird nicht anders übrig bleiben, wenn sie mitziehen wollen, als in die Abhängigkeit externer Dienstleister zu geraten.

4. Das Problem der steigenden Komplexität

Auch die großen IT-Anwenderbetriebe werden sich mit der zunehmenden Komplexität der Service-orientierten Systeme schwer tun. Sämtliche Evolutionsaufgaben, sei es Fehlerkorrekturen, Änderungen oder der Einbau neuer Funktionalität müssen über Teilsystemgrenzen und Schnittstellen hinweg miteinander abgestimmt werden. Sonst besteht die Gefahr dass alles auseinander läuft. Zu viel sind die gegenseitigen Abhängigkeiten zwischen den verteilten Services auf der einen Seite und den verteilten Clientprozessen auf der Anderen.

Um überhaupt Herr dieser komplexen Lage zu werden braucht die zuständige Entwicklungsorganisation eine detaillierte, graphische Landkarte der IT Landschaft [BSB08]. Aus dieser Karte geht hervor welche Prozesse welche Services verwenden, welche Services welche anderen Services in Anspruch nehmen und welche Prozesse mit einander direkt verbunden sind. Dazu muss man sehen können, auf welche Datenbanken die Services zu greifen und wie diese Datenbanken zusammenhängen. Schließlich sollten alle Schnittstellen zu fremden Systeme erkennbar sein. Diese graphische Landkarte ist die sichtbare Oberfläche zu einer darunter liegenden Repository, die automatisch aus der Analyse etlicher Sourcen, Modelle und Dokumente gefüttert wird. Jedes Mal wenn ein neues Release der Software zusammengestellt wird, ist die Repository über die statische Analyse der geänderten Komponente auf den neuesten Stand zu bringen [SHT04].

Sämtliche Fehlermeldungen, Änderungsanträge und neue Anforderungen müssen zunächst auf der großen Landkarte des betroffenen Systems fixiert werden. Innerhalb der Repository werden sie dann durch das System vom Baustein zu Baustein verfolgt und alle betroffene Knoten im Netzwerk der Prozesse, Klienten und Services identifiziert bis der Auswirkungsbereich der Änderung genau abgesteckt ist. Die Größen der betroffenen Bausteine werden aus der Metrikdatenbank geholt und summiert um die Summe dann durch die Änderungsrate zu glätten [Sn01]. Dadurch lässt sich der Umfang der geplanten Änderung herausbekommen. Dies sei erforderlich um die Kosten zu schätzen. Diese Kosten werden mit dem verrechneten Nutzen verglichen um zu sehen ob die Änderung sich lohnt. Nur lohnende Änderungen sind in solchen komplexen Systemen erlaubt, denn in einem Service-orientierten System sind die Risiken einer Änderung viel zu hoch [Sn10].

Wenn einmal entschieden ist das System zu ändern, muss der zu ändernden Bausteine bezüglich deren Abhängigkeiten genau untersucht werden, um ja keine unerwünschte Nebeneffekte zu verursachen. Änderungen zu Schnittstellen sind besonders heikel. Es muss vorher geklärt werden, welche Services welche Schnittstellen benutzen. Dies ist wohl nur mit einer Repository möglich. Deshalb ist ein mächtiges Repository eine unabdingbare Voraussetzung für die Evolution eines Service-orientierten Systems.

Nach der Vollendung der Änderungen sind zunächst die geänderten Softwarebausteine in einer künstlichen Umgebung neu zu testen. Ihre Verknüpfungen nach Außen werden dabei durch Stubs simuliert. Erst wenn jeder Netzknote für sich getestet worden ist, werden die Knoten verbunden und in ihrer Gesamtheit getestet. Es wäre zu aufwendig werden jede Interaktion zu den Services einzeln zu testen. Dies wird nur gemacht wenn ein Problem auftritt, z.B. wenn sich der Auftrag von einem Prozess-Schritt an ein Service nicht korrekt abgehandelt wird. Der Integrationstest beschränkt also auf den Test einzelner Interaktionen. Ansonsten ist das Ganze als System zu testen.

In der Evolution wird mit den angereicherten Daten der vorhergehenden Version getestet. Die Daten werden lediglich entsprechend den Änderungen mutiert. D.h die Tester müssen wissen welche Auswirkung die Änderungen auf ihre Testdaten haben werden. Um dies zu erfahren brauchen auch sie Zugang zum System-Repository. Über das Repository werden sie erfahren welche Testfälle sie fahren müssen um die Zielschnittstellen zu treffen. Es wird allzu leicht sich im Test zu verzetteln wenn blind getestet wird. Es komme darauf an gezielt zu testen und dies sei nur möglich mit Hilfe geeigneter Analysewerkzeuge [NJH03].

Zusammenfassend ist festzustellen, dass die Komplexität Service-orientierter Systeme nur mit leistungsfähigen Werkzeugen zu bewältigen ist. Diese werden reichen von einfachen Analyse Tools bin hin zum einem allumfassenden Repository und zum vollautomatisierten Regressionstest. Wer also Service-orientierter Systeme einführen will, muss bereit sein in eine voll ausgebaute Werkzeuginfrastruktur zu investieren.

5. Das Problem der Lieferantenabhängigkeit

In der IT-Welt mit ihren vielen Spezialgebieten bei denen jedes ein eigenes Teilwissen voraussetzt, hat es schon immer Abhängigkeiten gegeben. Die Analytiker sind von den Programmierern abhängig, da sie den Code beherrschen. Die Programmierer sind von Datenbankadministratoren abhängig weil diese über die Daten verfügen, auf die die Programmierer zugreifen müssen. Alle drei sind von den Systemadministratoren abhängig, weil sie die Herren jener Ressourcen sind welche die Anderen zu ihrer Arbeit brauchen. Wir sind in der IT-Welt gewohnt von Anderen abhängig zu sein. Diese Abhängigkeiten waren aber alle innerhalb der Anwenderorganisation und konnten intern geregelt werden. Am Ende des Tages hatten alle den gleichen Chef.

Mit der Einführung Service-orientierter Systeme treten neue Abhängigkeiten auf. Wer fremde Services in seine Geschäftsprozesse einbaut ist von den Lieferanten jener Services abhängig. Möglicherweise ist dieser eine interne Dienststelle, aber oft wird es sich um einen externen Lieferanten handeln. Wie früher mit den mathematischen Subroutinen, werden viele Services besondere Kenntnisse voraussetzen, die nur wenige Service Provider haben. Das ist letztendlich der Sinn von Services. Der Benutzer holt sich verpacktes Wissen nach Bedarf, Wissen das er im eigenen Unternehmen nicht zur Verfügung hat. Daraus entsteht ein Abhängigkeitsverhältnis. Darum ist es wichtig mit jedem externen Service Lieferant ein Service Level Agreement – SLA – abzuschließen [Sn05]. Auch mit internen Dienststellen die Services bereitstellen, empfiehlt es sich ein SLA zu haben. Die Service Benutzer müssen dafür Sorge tragen, dass die sie ins Leben rufen weiterentwicklungsfähig sind. Diese können jedoch nur so bleiben wenn alle ihrer Bestandteile ebenfalls weiterentwickelt werden. Jede Service-Komponente wird ein eigener Lebenszyklus haben. Die Wenigsten werden so bleiben wie sie sind, die Meisten werden regelmäßig erneuert. Über das Service Level Agreement kann sich der Benutzer eines Services sichern, dass die Evolution des Services mit der Evolution seiner eigenen Software im Einklang bleibt [Bö08].

In der SLA gilt es zu regeln wie oft die Services erneuert werden und wie der Benutzer darüber zu informieren ist. Es darf nicht vorkommen, dass ein Service ohne adäquate Vorwarnung geändert wird. Der Benutzer braucht genügend Zeit seine eigene Software auf das veränderte Service einzustellen und genügend Zeit die Interaktion mit dem veränderten Service zu testen. Das Testen geänderter, bzw. erneuerter, Services wird die meiste Zeit in Anspruch nehmen. Um diese Zeit möglichst kurz zu halten, braucht der Benutzer genaue Angaben darüber was sich geändert hat. Zur Sicherung der Änderungen bedarf es eine besondere Schnittstellenverwaltung. Je mehr unter unterschiedliche Services beansprucht werden, desto aufwendiger diese Verwaltung.

Das Service Level Agreement sollte sowohl die Funktionalität als auch die Qualität des Services regeln. Was die Funktionalität anbetrifft sind:

- Die einzelnen Operationen
- die Eingangsparameter und
- die zu erwarteten Ergebnisse

zu spezifizieren. Bezüglich der Qualität sind u.a.

- die Zuverlässigkeit
- die Verfügbarkeit
- die Sicherheit
- die Antwortzeit und
- die statische Qualität der Schnittstelle

zu regeln.

Die Zuverlässigkeit wird im Bezug zur Anzahl Abstürzen und fehlerhafte Ergebnisse pro 1000 Aufträge definiert. Die Verfügbarkeit wird im Prozentsatz der Betriebszeit definiert. Die Sicherheit wird anhand der vereinbarten Sicherheitsmaßnahmen vorgegeben. Die Antwortzeiten werden in Sekunden zwischen dem Absenden eines Auftrages und dem Erhalt der Antwort festgelegt. Die statische Qualität der Service-Schnittstelle wird über eine Reihe Regel und Metrik zur Gestaltung der Schnittstellen vereinbart [Sn07].

Es wird aber nicht genügen eine SLA auf Papier zu haben. Die Einhaltung der SLA muss überwacht werden. Die geringste Abweichung von der Vereinbarung ist sofort zu registrieren und zu melden. Daraufhin tritt der vorbestimmte Eskalationsprozess in Gang. Auch dieser Prozess muss in der SLA beschrieben sein. Kurzum das Service Level Agreement spielt eine zentrale Rolle bei der Evolution eines Service-orientierter Systems. Folglich wird eine der Haupttätigkeiten eines Projektleiters sein, solche SLAs zu verfassen und mit den Service Providern auszuhandeln. Anschließend ist er verpflichtet die Umsetzung der SLAs zu verfolgen. Dazu braucht er einen Produktleitstand mit dessen Hilfe er die Einhaltung der einzelnen Services kontrollieren kann. Der Produktleitstand soll an dem Repository angeschlossen sein, und über alle Informationen zu dem Produkt verfügen. Über die Key Produktindikatoren (KPIs), wie z. B., Antwortzeitverhalten und Fehlerrate, sowie die Prozessindikatoren, wie z.B., Zeit zur Fehlerkorrektur und Zeit bis zur Änderungsumsetzung, wird er in der Lage sein den Evolutionsprozess zu steuern.

6. Schlußfolgerung

In diesem Beitrag wurde angesprochen, welche Probleme in Zusammenhang mit der Evolution Service-orientierter Systeme zu erwarten und wie sie zu begegnen sind. Diese Probleme werden erst nach der Einführung der Systeme richtig in Erscheinung treten. Die erhöhte Komplexität verteilter und vernetzter Services mit deren vielen Schnittstellen zueinander wird die Fortschreibung solcher Systeme erschweren und die Risiken jeder Änderung steigern. Die Mehrsprachigkeit der beteiligten Servicekomponente wird die Kosten der Systemevolution weiter in die Höhe treiben. Der Betreiber der Systeme muss für jede verwendete Sprache Personal halten, das sich mit der jeweiligen Sprache auskennt. Hinzu kommt die zunehmende Abhängigkeit von den Service-Lieferanten. Der Benutzer muss sich gegen unerwartete Funktionsänderung und Service-Leistungsverfall schützen. Das Instrument dazu ist das Service Level Agreement. Damit steigt aber der Verwaltungsaufwand. In jedem Projekt muss es jemanden geben, der sich um die SLAs kümmert.

Literaturverzeichnis

[BL75] Belady, L. /Lehman, M.: „The Evolution Dynamics of Large Programs“, IBM Systems Journal, Nr. 3, Sept. 1975, S. 11

[LB85] Lehman, M., Belady, B.: “Program Evolution, Academic Press, London, 1985, S. 29

Seacord, R./Plakosh, D./Lewis, G.: Modernizing Legacy Systems, Addison-Wesley, Boston, 2003

[Sn02] Sneed, H: "Das Ende von Migration und Reengineering", Computerwoche Nr. 7, Feb. 2002, S. 18

[Sn04] Sneed, H.: "A Cost Model for Software Maintenance and Evolution", in Proc. of 20th ICSM, IEEE Press, Chicago, 2004, S. 264

[vH05] von Hahn:, E.: Werterhaltung von Software – Planung und Bewertung von Reengineering Maßnahmen am Beispiel von Standard Software, Deutscher Universitätsverlag, Wiesbaden, 2005, S. 173

[BR00] Bennett, K. H.; Rajlich, V. T.: Software Maintenance and Evolution: A Roadmap. In Finkelstein, A. [Hrsg.]: The Future of Software Engineering, Proceedings of the International Conference on Software Engineering, ACM Press, 2000, S. 73

[KBS05] Krafzig, D./Banke, K./Slama, D.: Enterprise SOA – Service-Oriented Architecture, The Coad Series, Prentice-Hall, Upper Saddle River, N.J. 2005, S. 9

[SL07] Smith, D.; Lewis, G.: "Standards for Service-oriented Systems", in Proc. of 11th European Conference on Software Maintenance and Reengineering, Amsterdam, 2007, S. 100

[NL05] Newcomer, E./Lomow, G.: Understanding SOA with Web Services, Addison-Wesley, Upper Saddle River, N.J., 2005, S. 10

[Lo93] Lorenz, M.: Object-oriented Software Development, Prentice-Hall, Englewood Cliffs, 1993, S. 125

[Mo91] Montgomery, S.: AD/Cycle – IBM's Framework for Application Development and CASE, Van Nostrand, New York, 1991, S. 17

[SC99] Sadiq, W., Cummins, F.: Developing Business Systems with CORBA, Cambridge University Press, Cambridge, 1999, S. 23-42

[Ke02] Keller, W.: Enterprise Application Integration – Erfahrungen aus der Praxis, dpunkt Verlag, Heidelberg, 2002, S. 5

[CHK06] Conrad, S./ Hasselbring, W./ Koschel, A./ Tritsch, R.: Enterprise Applcation Integration, Elsevier Spektrum Verlag, Heidelberg, 2006, S. 11

[SnSn03]Sneed, H./ Sneed, S.: Web-basierte Systemintegration, Vieweg Verlag, Wiesbaden, 2003, S. 91

[CA88] Card, D., Agresti, W.: "Measuring Software Design Complexity", Journal of Systems & Software, Vol. 8, 1988, S. 185

[TGH04] Tilley,S./Gerdes,J./Hamilton,T./Huang,S./Mueller,H./Smith,D./Wong,K.: "On the Business Value and technical Challenge of adopting web services", Journal of Software Maintenance and Evolution, Vol. 16, Nr. 1-2, Jan. 2004, S. 31

[Sn07] Sneed, H.: "Migrating to Web Services – A Research framework", in Proc. Of 11th CSMR, Amsterdam, March, 2007, S. 116

[ABC01] Abraham, N.; Bibel, U.; Corleone, P.: Formatting Contributions for LNI. In (Glück, H.I. Hrsg.): Proc. 7th Int. Conf. on Formatting of Workshop-Proceedings, New York 1999. Noah & Sons, San Francisco, 2001; S. 46-53.

[Az99] Azubi, L. et.al.: Die Fußnote in LNI-Bänden. In (Glück, H.I.; Gans, G., Hrsg.): Formattierung leicht gemacht – eine Einführung. Format-Verlag, Bonn, 1999; S. 135-162

[Ez99] Ezgarani, O.: The Magic Format – Your Way to Pretty Books, Noah & Sons, 2000.