Technology Bridge

Object Engineering

Herausgeber: Object Engineering GmbH • www.objeng.ch • Mai 2021 • Nr. 2

Eine «alte» Applikation zu betreiben gilt beinahe schon als ein Tabu, denn ein Unternehmen will doch als modern wahrgenommen werden und möchte nicht mit «alten» Applikationen in Verbindung gebracht werden. Also verschliessen viele CTOs die Augen vor der unter Umständen immensen Aufgabe der Modernisierung ihrer Applikationslandschaft, gerade wenn Firmen ihre Investitionen aus globalwirtschaftlichen Gründen einschränken. Im Folgenden zeigen wir ein einfaches Vorgehen auf, welches die Entscheidungsfindung und Umsetzung eines Modernisierungsvorhabens aus praktischer Sicht unterstützt und mit Tipps aus der praktischen Erfahrung angereichert ist.

Aufmerksamkeit des Managements gefordert

Oft ist man sich nicht bewusst, welches schleichende Risiko das Unternehmen eingeht, wenn solche Modernisierungsvorhaben auf Eis gelegt werden. Das Management weiss unter Umständen gar nicht, was sich im «Keller» anbahnt. Flexibel auf neue Anforderungen wie die Digitalisierung der Prozesse zu reagieren, ist nur eines der Risiken, welchen ein Unternehmen gegenübersteht. Man muss eine Anwendung, welche vor zehn bis fünfzehn Jahren erstellt wurde und seither immer problemlos und ohne nennenswerte Probleme betrieben wird, als eines der grössten, verdeckten Betriebsrisikos einstufen. Kaum noch jemand ist mit der Machart vertraut oder hat es «vergessen», und wenn je ein wirkliches Problem auftritt, fehlen Fachkräfte, welche die Anwendung noch genügend kennen.

«Wir lassen's beim Alten, es läuft ja»

Es kann auch sein, dass man eine Modernisierung nicht einmal in Betracht zieht, da ein Architekt oder Produkt-Owner sich scheut, das alte System genau zu kennen.

Das führt entweder dazu, dass man es so belässt oder sich direkt für eine Neuentwicklung der Anwendung entscheidet. Dabei wird vergessen, dass oft ein Grossteil der bestehenden Funktionalität auch im neuen System benötigt wird und erneut implementiert werden muss.

Mit einer entsprechenden Modularisierung wäre es nach eingehender Analyse aber möglich, bestehende Teile, welche über die Jahre fast fehlerfrei wurden und entsprechend im Einsatz bewährt waren, zu übernehmen, auch wenn die technologische Grundlage gewechselt wird.

Bei einer Neuentwicklung ist aber nicht garantiert, dass die neue Lösung weniger komplex, flexibler und wartbarer wird. Die für die Entwicklung verwendeten Frameworks bringen oft über Bibliotheken mehr Komplexität mit.

Um den Reigen abzuschliessen, muss man unbedingt auch den Umfang an Komplexität und Code von bestehenden Systemen anschauen. Um ein über zehn und mehr Jahre entstandenes System zu ersetzen, braucht man auch mit einem grossen Team schnell einmal mehrere Jahre. Hier braucht es sowohl technische und entwurfsmässige, aber auch organisatorische Genialität. Viele Softwaresystem-Modernisierungen sind an ihrer Grösse gescheitert.

Worauf man achten soll

Jeder CTO - und vor allem CEO und CFO - tut gut daran, sich die Gründe für eine Neuentwicklung vorlegen zu lassen und diese kritisch zu hinterfragen, bevor das Budget gesprochen wird.

Nicht jede bestehende Anwendung kann vernünftig modernisiert werden, aber es ist in mehr Fällen möglich, als man aus dem Bauchgefühl denken würde. Beim Modernisieren gilt es folgende Aspekte zu beachten:

- Pragmatische Lösungen in Betracht ziehen, aber Systematik nicht beiseitelassen.
- Ehrlichkeit bei Aufwandsschätzungen, welche oft zu tief angesetzt sind.
- Verbesserung der Abstraktion und Vereinfachung des zu modernisieren Systems.
- Hinterfragen der Gründe und der Wirtschaftlichkeit für eine Modernisierung (inkl. Neuerstellung und Lebenserwartung).
- Das neue System muss dem Unternehmen und den Benutzer dienen und wirtschaftlicher sein als das Bestehende.
- Aufteilen in Teilprojekte und -phasen, und verhindern eines «Big-Bang» Ansatzes.
- Wertvolle Komponenten erhalten und Unbrauchbares ersetzen.
- Organisation über das ganze Vorhaben unter Einbezug aller Stakeholders.
- Eingebaute Umgehungslösungen so früh wie möglich wieder entfernen.

Vorgehen in drei Schritten

Kein System ist wie das andere, dennoch folgen viele Systeme einem ähnlichen Muster. Dadurch kann man in der Regel mehr erreichen, als man erwarten würde.

In verschiedenen von den Autoren begleiteten Projekten, welche sowohl Plattform- als auch Technologie-Epochen-übergreifend waren, hat sich immer wieder ein Vorgehen bewährt und wird wohl in ähnlicher Weise von vielen so angewendet. Es besteht aus folgenden Schritten:

Schritt 1: Wissen was man hat.
Schritt 2: Wissen was man will.
Schritt 3: Wissen wie man es tut.

Wissen was man hat

Wenn eine Modernisierung oder eine Neuerstellung einer bestehenden Anwendung überhaupt auf die

Copyright © 2021 by Object Engineering GmbH CH-8142 Uitikon-Waldegg

Agenda gesetzt wird, müssen Beweggründe oder Probleme vorhanden sein. In dieser Phase geht es darum, herauszufinden, was genau die Situation ist:

- Was stört den Benutzer oder was macht ihn ineffizient?
- Welche Einbussen an Finanzen, Ansehen oder suboptimalen Geschäftsprozessen hat das Unternehmen durch die fragliche Applikation?
- Wie ist das Betriebsverhalten der bestehenden Anwendung?
- Wird die vorhandene Funktionalität überhaupt noch benötigt?
- Wie hoch sind Betriebs- und Wartungskosten der Applikation?
- Wie hoch ist die Anzahl der regelmässigen Benutzer sowie der täglichen Transaktionen zugreifenden Umsysteme?
- Wie passt die Anwendung in die strategische Unternehmensarchitektur und wie hoch ist ihre Wichtigkeit darin?
- Gibt es eine Cloud-Auslagerungs-Strategie?

Man muss bei diesen Punkten zwischen organisatorischen und technischen Aspekten unterscheiden. Im Folgenden möchten wir hauptsächlich die technischen Aspekte behandeln. Trotzdem beeinflussen die organisatorischen Aspekte unvermeidlich die Architektur und den Modernisierungsplan.

Grosse unübersichtliche Codebasis

Ein technischer Aspekt hierbei ist die Codebasis. Dabei spielen Codeumfang, Technologie und Programmiersprache eine wichtige Rolle. Bei Applikationen, die über mehrere Jahrzehnte weiterentwickelt wurden, ist die Codebasis in der Regel sehr gross. Muss man diese manuell sichten, sollte man den Wert einer maschinellen Analyse des Codes [Thurner-Koch19] nicht unterschätzen. Man möchte am Schluss ja aufgrund von Fakten entscheiden können, was vom Bestehenden noch übernommen werden kann und was neu erstellt werden muss.

Ein hybrides Vorgehen hat sich in der Praxis bewährt: In einem ersten Schritt manuell den Code zu begutachten und Muster zu finden, welche dann in einer automatisierten Analyse flächendeckend gesucht werden. Man sollte auch nicht unterschätzen, welche wichtigen Informationen ein beiläufiges Gespräch mit den Anwendungsverantwortlichen bringen kann. Man kann dabei aber auch auf Einstellungen treffen, wie: «Eine Analyse der Anwendung erübrigt sich, wir kennen doch die Funktionalität genau». Diese Einstellung kann sich sowohl nachteilig für eine Modernisierung wie auch für eine Neuerstellung zeigen und sollte nicht akzeptiert werden.

Das Resultat dieses Schrittes ist die Grundlage, welche es dem Modernisierungs-Architekten ermöglicht zu entscheiden, ob und welche Teile der bestehenden Anwendung übernommen werden können und wie die Lösungs- respektive die Migrations-Architektur aussehen soll. Nach diesem Schritt sollten die ersten Kosten- und Zeitschätzungen für die Durchführung ermittelt werden können.

Wissen was man will

Nach abgeschlossener Ist-Aufnahme sollten genügend Informationen für die Verantwortlichen vorhanden sein, um sowohl Lösungsarchitektur, als auch Vorgehen bestimmen zu können. Weiter sollten alle Stakeholder und vor allem auch die Benutzer ihren Bedarf angemeldet haben und das neue System sollte die Unternehmensstrategie besser unterstützen.

Wenn man sich dafür entscheidet, Teile der bestehenden Anwendung zu übernehmen, dann muss auch dafür ein Plan ausgearbeitet werden, wie man Neu und Alt in Einklang bringt.

In jedem Fall ist die Modularisierung ein wichtiger Teil. Wenn nur Teile der bestehenden Applikation übernommen werden sollen, sollten diese eigentlich als Komponenten isoliert werden, was durch eine Code-Abhängigkeits-Analyse ermittelt wird.

Wissen wie man es tut

Eine Modernisierung ohne eine sinnvoll detaillierte Architektur in Angriff zu nehmen, ist wie ohne Karte und Kompass eine Seereise in Angriff zu nehmen.

Die Ziel-Architektur, meistens in mehrere Phasen aufgeteilt, ist die Grundlage für den Aktivitätsplan der Migration. Dazu gehört eine «sinnvoll-detailliert» gewählte Granularität auf der Basis von Komponenten in Form von Services oder Bibliotheken. Im Bereich von externen oder bestehenden Systemen sind Schnittstellen auch für die Architektur relevant und benötigen dort in der Regel eine feinere Detaillierung.

Bei der Neuerstellung, also (Teil-)Ersatz des Alt-Systems, ist ein paralleles Hochziehen des neuen Systems und ein temporärer Parallelbetrieb beider Systeme in der Regel sinnvoll.

In einer Ablösung eines älteren Unix-basierten, in Cobol geschriebenen Logistik-Systems hatte diese Koexistenz den Vorteil, dass die Benutzer immer den Eindruck hatten, dass die Erneuerung voranschritt, auch wenn sie über die gesamte Migrationszeit mit zwei unterschiedlichen Systemen arbeiten mussten. Benutzer sind in der Regel tolerant, solange sie ihre Arbeit effizient durchführen können und laufende Verbesserung am Horizont sehen.

Horizontal oder vertikal

Sieht man sich die typische Applikationslandschaft an, dann kann man diese grob in vier Bereiche aufteilen (s. Abb.1):

- Benutzer-Interaktion,
- Business-Logik,
- Daten-Bereich und
- Integration der Umsysteme.

Das Vorgehen wird nach diesem Modell ausgerichtet. Dabei ist es wichtig, ein einfaches Modell (nicht mit Schichten zu verwechseln) zu verwenden, damit alle Beteiligten damit denken können.

Hat man einen Monolithen vor sich, dann sind die Funktionalitäten und auch Datenzugriffe oft über alle Bereiche verteilt. Bei (erschreckend) vielen Applikationen aus der Epoche der 4-GL-Systeme (4th Generation Language) fehlt oft ein expliziter Business-Logik-Bereich.

Ob man *bottom-up* oder *top-down* vorgeht (horizontal) oder eher den Durchstich (vertikal) vorsieht, muss situativ entschieden werden. Der Nachteil von bottom-up liegt darin, dass über einen langen Zeitraum wenig Fortschritt sichtbar ist. Geht man eher top-down vor, so gibt es auf der Benutzeroberfläche schneller eine sichtbare Veränderung mit dem Nachteil, dass der Eindruck entstehen kann, man sei bereits am Ziel.

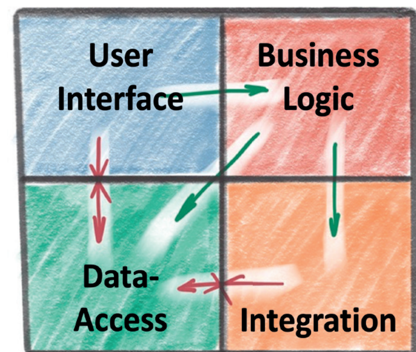

Abb. 1: Konzeptionelles Bereichs-Modell

Modernisierung

Alle vier Bereiche müssen modernisiert werden.

Benutzer-Interaktion

Im Bereich der Benutzer-Interaktion kann sehr wohl ein Technologiewechsel anstehen. Besonders dann, wenn man die Applikation zum Beispiel für die Cloud-Fähigkeit webtauglich machen möchte.

Hier stellt sich die Frage, ob man bei einem Technologiewechsel auch ein Redesign der Benutzerabläufe und der Masken ins Auge fassen möchte. Dies hängt stark von der Anzahl und Art der Benutzer ab. Sehr oft sind die bestehenden Maskenabläufe auf Effizienz und Verständlichkeit ausgelegt. In einer solchen Situation ist man gut beraten, den Aufbau zu belassen und nur einen Technologiewechsel durchzuführen.

In einem Projekt [ThKoKo13], bei dem ein Technologiewechsel von einem Fat-Client auf Web-SPA gemacht wurde, hatte man sich entschieden, die Masken unverändert zu belassen. Dank einer teilautomatisierten Umstellung konnten rund 50% des geschätzten manuellen Aufwandes eingespart werden.

Ist der Technologiewechsel vollzogen kann man mit der neuen Darstellungs-Funktionalität in weiteren Schritten auch Verbesserungen für den Benutzer einführen. Es empfiehlt sich jedoch, diese Verbesserungen und Veränderungen des UX zu Gunsten des Benutzers in kleinen Schritten umzusetzen.

Hinter der Kulisse und in der Regel vor einem Technologiewechsel von Fat-Client zu Web-Client gibt es aber noch genügend zu bereinigen. So sind gerade bei solchen PC-basierten und auch X-Windows-Applikationen oft Funktionalitäten auf der Benutzer-Interaktions-Layer, die eigentlich in die Business-Ebene gehören. Bei 2-Schichten-Applikationen müssen auch die Datenzugriffe durch entsprechende Business-Funktionszugriffe ersetzt werden. Handelt es sich vor allem um CRUD-Operationen, können diese meistens auch automatisiert umgestellt werden.

Business-Logik

Im Bereich der Business-Logik haben wir nun die Möglichkeit, modular, statt monolithisch zu werden. Services als Komponenten zu verwenden, hat erfahrungsgemäss den Vorteil, flexibel und ausbaubar zu bleiben. Dies kombiniert mit *Domain Driven Design* [Vernon] kann zum Dream-Team werden. Natürlich ist die Wahl der Granularität, Funktionalität und auch der Schnittstelle des Service der Teil, welcher vom Design respektive der System-Architektur vorgegeben werden sollte. Wenn man dann noch auf die lose Kopplung achtet, und wo es möglich und sinnvoll ist, Event-basiert arbeitet, ist die Flexibilität fast garantiert.

Die oben erwähnte Bereinigung gilt aber auch für die bestehende Funktionalität. Dabei soll Business-Funktionalität aus dem UI-Bereich und aus dem Datenbank-Bereich in den richtigen Service migriert werden. Das Aufwands/Ertrags-Verhältnis darf nicht ausser Acht gelassen werden.

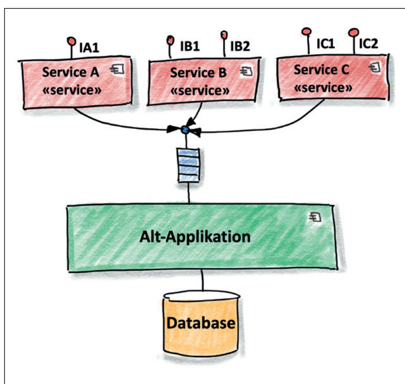

Abb.: 2 Wrapping Legacy-Komponente

Aber aufgepasst, nicht alles lässt sich sinnvoll oder einfach auf Services auf- und verteilen. Ist die zu modernisierende Applikation eine sehr stark auf Datentransaktionen ausgelegte Batch-Applikation, welche grosse Datenmengen in kurzer Zeit im Hintergrund und ohne Online-Transaktionen verarbeitet, dann muss dem Design der Services grosse Beachtung geschenkt werden. Die Interprozess-Kommunikation kann bei hoher Belastung und falschem Design wie eine Bremse wirken. Hier kann es durchaus sinnvoll sein, mit dedizierten, weniger verteilten Services und eher grober Funktionsgranularität zu arbeiten. Services richtig eingesetzt, können andererseits aber auch einen guten Skalierungseffekt bewirken.

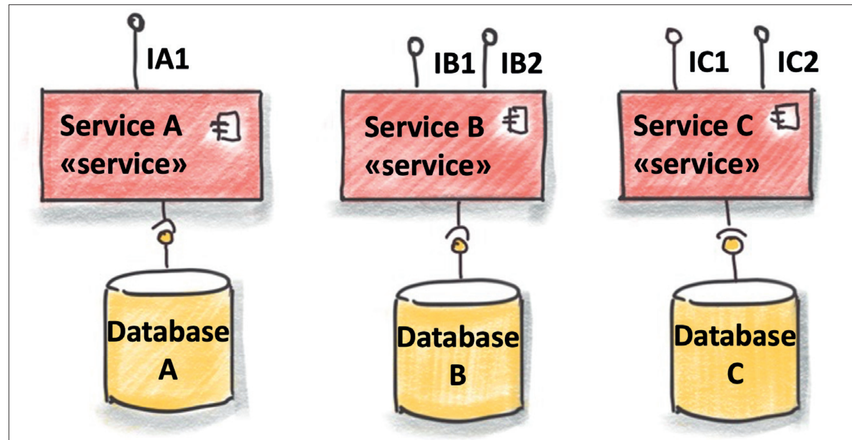

Abb. 3: Ersatz der Legacy-Komponente

Mit Wrapping kann man auch Zeit gewinnen, um das System in eine komponentenorientierte Architektur überführen zu können. Dabei sind die Services in einem ersten Schritt nur Adapter und stellen die applikatorischen Schnittstellen zur Verfügung.Die Funktionalität wird jedoch an die bestehende Applikation delegiert (s. Abb. 2). In der «neuen» Welt greifen die neuen Komponenten auf diese Service-Schnittstellen zu.

Wenn man die Funktionalität dann in die entsprechenden Services überführt hat, so dass diese dann auf die entsprechenden Datenbanken-Entitäten zugreifen, kann die alte Applikation abgelöst werden (s. Abb. 3). Dieses Prinzip hat sich bereits bei der Überführung zu verteilten Applikationen in den 90er-Jahren bewährt.

Modularität ist der Erfolgsfaktor auf dieser Ebene, aber es lohnt sich, die Aufteilung mit Design und Umsicht und einem angemessenen Design zu tun.

Daten-Bereich

Bei der Applikationsmigration ist es oft sinnvoll vorerst die Business-Funktionalität nach Domänen auf Komponenten aufzuteilen und sicherzustellen, dass nur die für die Domäne zuständigen Komponenten auf die Daten der Domäne zugreifen dürfen. Auch hier müssen die Transaktionsrate und Datenmenge wie oben erwähnt berücksichtigt werden.

Was bei bestehenden Applikationen aber untersucht werden sollte, sind die oft als *Stored Procedures* in der Datenbank vorhandenen Business-Funktionen. Man kann es nicht generell als schlecht bezeichnen, aber in vielen Fällen gehört ein Grossteil dieser Funktionen nicht in die Datenbank. Insbesondere, wenn man auf der Programmier-Ebene von Services auf Testabdeckung, CI/CD unter anderem schaut und mit Metriken und Entwicklungsprozessen für Reviews die Codequalität anzuheben versucht, sollte man Datenbank-Code nicht einfach unhinterfragt tolerieren. Wir haben Applikationen angetroffen, welche im Bereich von 200'000 bis mehrere Millionen Zeilen Business-Logik-Code in der Datenbank hatten. Eine Modernisierung der Datenmodelle in Schritten parallel mit der Modularisierung auf der Business-Logik-Ebene kann ein gangbarer Weg sein. Ohne eine eingehende Abhängigkeits-Analyse «aller» Applikationen und Datenbankprozeduren kann dies aber nicht gemacht werden. Will man die Business-Logik aus der Datenbank entfernen, heisst es zusätzlich einen Technologiewechsel zu bewältigen (SQL⇨Programmier-Sprache).

Integration der Umsysteme

Die Applikationen eines Unternehmens stehen ja nicht für sich alleine da, sondern sind immer öfters mit externen Systemen funktional verbunden. Beispiele für externe Systeme sind Buchungssysteme, EDIFACT-basierte Bestell- und Auftragssysteme, Zahlungssysteme und viele mehr.

Um einer organisch gewachsenen System-Welt von externen Abhängigkeiten gerecht zu werden, empfiehlt sich der Einsatz eines Gateway-Services pro Umsystem. Auch wenn der Aufwand höher scheint, merkt man schnell, dass man sich damit Flexibilität schafft.

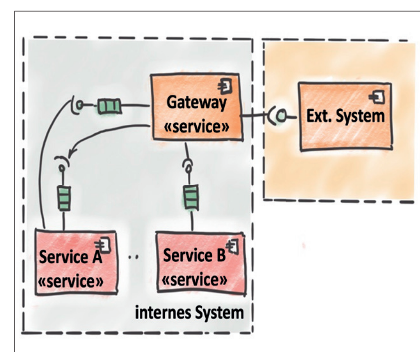

Abb. 4: Gateway für Integration

Object Engineering®

Das Gateway (s. Abb. 4) kennt das externe und das interne System und kann so die Protokollkonversion und das Daten-Mapping sicherstellen. Da oft die Änderungen des externen Systems nicht mit dem Release-Zyklus des internen Systems übereinstimmen, kann dieser Service wie eine Kupplung für die unterschiedliche Release-Drehzahl wirken. Ändert sich das externe System, muss nur der Gateway-Service angepasst werden. Dieser kann in der Regel auch ausserhalb des Release-Zyklus des internen Systems neu installiert werden. Umgekehrt, wenn das interne System eine, den Gateway beeinflussende Änderung bekommt, muss wiederum nur der Gateway-Service angepasst werden, ohne dass die Verarbeitung mit dem externen System davon gestört wird.

Praxis-Beispiel

Bei der Modernisierung des bestehenden Systems, ist eine phasenweise Umstellung empfehlenswert. Als Beispiel: Eine modulare C++-Applikation und die darin verwendete Middleware (CORBA) musste auf Java und Messaging umgestellt werden. Dies wurde in zwei Schritten gemacht (s. Abb 5):

Schritt 1: Umstellung des Frontends
Schritt 2: Umstellung des Backends

Diese Modernisierung wurde innerhalb von 8 Monaten vollzogen und in zwei Releases in den Betrieb gebracht. Wichtig bei einer solchen Umstellung ist die Existenz eines Fallback-Szenarios, damit im Falle unerwarteter Probleme notfalls der Schritt zurück möglich ist. Letztere sind gerade im Falle von schlecht testbaren Lastverhalten fast unvermeidlich.

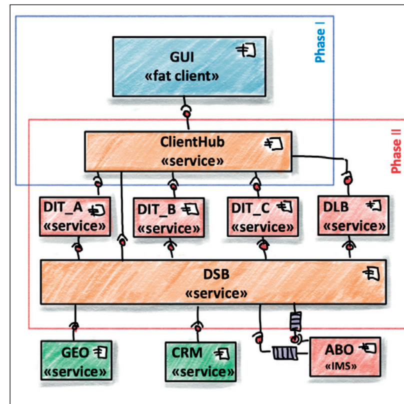

Abb. 5: Phasen-Modernisierung

Hilfreich war dabei eine vorhandene Komponente, welche den UI-Teil vom Backend trennt. Diese hat bei der Umstellung von CORBA zu REST im Frontend dies noch auf die alte Middleware im Backend adaptiert. In der zweiten Phase wurde diese dann auch auf die neue Message-basierte Middleware umgestellt, ohne dass der Frontend-Teil sich anpassen musste.
Die Wrapping-Methode (Strangler Pattern) kann auch beim Parallelbetrieb einer Neuerstellung oder Umstellung helfen. Die Form kann unterschiedlich sein.

Fazit

Auch wenn man versucht ist zu denken, dass die Neuerstellung einer bestehenden Anwendung der einfachere Weg ist, sollte man dies erst nach eingehender Analyse tun.
Die Modularisierung bietet typisch in Form von Services den Schlüssel zur Langlebigkeit und Flexibilität eines Systems. Die Granularität der Services ist aber immer noch eine Designfrage, was durch Domain-Driven Design gut unterstützt wird. Wie bei jeder Methode oder Technologie führt die rohe Anwendung auf jegliches Problem nicht zum Ziel, sondern wirft im Gegenteil neue Probleme auf. Jede der oben beschriebenen Ratschläge und Designvorschläge sollten für passende Problemstellungen angewendet werden. Überhaupt eine Zielarchitektur und einen Migrations- oder Modernisierungsplan zu formulieren, wirft bereits auf dem Papier Fragen auf, welche sonst erst bei der Umsetzung erkannt und teuer zu stehen kommen.
Ein schrittweises Vorgehen minimiert das Risiko eines Fehlschlags beträchtlich, gerade weil Grösse und Komplexität der bestehenden Systemumgebungen dem entgegenwirken. Darum sollte eine Modernisierung von Applikationen wie die Qualitätssicherung eine laufende Disziplin sein. Systematik, aber auch Pragmatik und eine gute Kosteneinschätzung gehören zu Modernisierungsunterfangen. Anwendungsmodernisierung sollte als eine Disziplin von Software Engineering angesehen werden.

Ersterscheinung OBJEKTspektrum 02/2021

Object Engineering GmbH
Birmensdorferstr. 32 CH-8142 Uitikon-Waldegg
www.object-engineering.ch
kontakt@object-engineering.ch
Member of the Solution Network Group
SNG
Object Engineering® ist ein eingetragenes Warenzeichen im Besitz der Object Engineering GmbH

Die Autoren

Andres Koch
(andres.koch@objeng.ch)
befasst sich seit mehr als 25 Jahren mit Software-Engineering und Parserbau für Legacy-Migration und Legacy-Integration.

Remo Koch
(remo.koch@objeng.ch)
befasst sich mit Graphen und Netzwerken und deren Anwendung im Bereich der automatisierten Analyse und im Reengineering von Applikationskomponenten.

Literatur und Links

[Rich19] C. Richardson, Microservices Patterns, Manning Publications, 2019
[Parn71] D. L. Parnas, On The Criteria To Be Used In Decomposing Systems Into Modules, in: Com. of the ACM, Dezember 1972
[ThurnerKoch19] R. Thurner, A. Koch, Legacy-Engineering: ein Repository-zentrierter Ansatz, in: OBJEKTspektrum, 06/19
[ThKoKo13] R. Thurner, A. Koch, R. Koch, Modellbasiertes Software-Reengineering: Teilautomatisiert Migration eines GUI in eine Web-Umgebung, in: OBJEKTspektrum, 06/14
[Vernon] V. Vernon, Domain-Driven Design kompakt, dpunkt.verlag, 2017
[WSRE19-1] K.-U. Herrmann, Teilautomatisiertes Architektur-Reengineering in einem Java EE Monolithen, in: 21. Workshop Software-Reengineering & Evolution, 2019, siehe: https://www.objeng.ch/content/5-wissen/2-whitepapers/1-whitepapers/ paper_hermannkaiuwe_teilautomatisiertes_architektur-reengineering_in_einem_javaee_monolithen.pdf

Copyright © 2021 by Object Engineering GmbH CH-8142 Uitikon-Waldegg