Xpert.press

Die Reihe **Xpert.press** vermittelt Professionals in den Bereichen Softwareentwicklung, Internettechnologie und IT-Management aktuell und kompetent relevantes Fachwissen über Technologien und Produkte zur Entwicklung und Anwendung moderner Informationstechnologien.

Dieter Masak

Legacysoftware

Das lange Leben der Altsysteme

Mit 149 Abbildungen und 39 Tabellen

Springer

Dieter Masak
plenum Systems
Hagenauer Str. 53
65203 Wiesbaden
dieter.masak@plenum.de

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

ISSN 1439-5428
ISBN-10 3-540-25412-9 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-25412-6 Springer Berlin Heidelberg New York

Springer ist ein Unternehmen von Springer Science+Business Media
springer.de

Satz: Druckfertige Daten des Autors
Herstellung: LE-TEX, Jelonek, Schmidt & Vöckler GbR, Leipzig
Umschlaggestaltung: KünkelLopka Werbeagentur, Heidelberg
Gedruckt auf säurefreiem Papier 33/3142 YL – 5 4 3 2 1 0

Danksagung

. . . für Christiane . . .

Dr. Dieter Masak

Prolog

Was du bist hängt von drei Faktoren ab:
Was du geerbt hast,
was deine Umgebung aus dir machte
und was du in freier Wahl
aus deiner Umgebung
und deinem Erbe gemacht hast.

Aldous Huxley
1894-1963

Inhaltsverzeichnis

1

Einleitung

I have travelled the length
and breadth of this country
and talked with the best people,
and I can assure you
that data processing is a fad
that won't last out the year.

Editor Business-Books
Prentice-Hall, 1957

1.1 Lamento

In den letzten Jahrzehnten haben sich Forschung und Lehre innerhalb der Softwareentwicklung primär mit der Erstellung von Software befasst, die Maintenance wie auch die systematische Weiterentwicklung wurden stets stark vernachlässigt. Für die frühen Phasen des Lebenszyklus wurden von großen Softwareherstellern viele Werkzeuge geschaffen, aber nur wenige für Themen wie Reengineering, Migration oder auch Maintenance. Es ist eine Schieflage entstanden, welche die tatsächliche Situation verzerrt wiedergibt.

Informatikstudiengänge sowie Softwareengineeringkurse vermitteln den Studenten und Teilnehmern Methodiken, um Software quasi „auf der grünen Wiese zu bauen"– Konzepte wie Architektur, Effektivität, Robustheit oder auch Performanz werden mit dem Fokus auf der Synthese und selten auf Analyse vermittelt. Die Auseinandersetzung mit tatsächlich eingesetzter Software gilt als „schmutzig" und wird sehr selten durch Lehrkörper und Seminarleiter unterstützt. Dies steht in krassem Gegensatz zu anderen Gebieten wie dem Bauingenieurwesen, dem Städtebau oder der klassischen Architektur. Diese sehen es als ihre Pflicht an, bestehende Objekte eingehend zu studieren, um aus der Vergangenheit zu lernen. Nicht so die Softwareentwicklung! Hier werden stets „neue" Methoden entwickelt, um „neue" Applikationen zu schaffen. Dabei werden alle bisher genutzten Methoden als überholt angesehen. Neue Vorgehensweisen werden oft mit einem Fanatismus vertreten, welcher erstaunliche Parallelen zu religiösen Zeloten aufweist.[1] Durch diese Negati-

[1] Bezeichnenderweise hat die Firma Sun „Java-Evangelisten" und Microsoft „.NET-Evangelisten".

on der Erfahrung wird das angesammelte Wissen über Softwareentwicklung weder tradiert noch genutzt und „alte“ Fehler werden permanent wiederholt.

Selbst in Organisationen, welche schon jahrzehntelange Erfahrungen im Umgang mit ihren eigenen Altsystemen haben, ist zu beobachten, dass man Lippenbekenntnisse für diese Systeme abgibt und beteuert, sich der Problematik bewusst zu sein, es wird jedoch nicht in einer geordneten Art und Weise konstruktiv mit diesen Altsystemen umgegangen. Ziel dieses Buches ist es, Wege für den Umgang mit diesen Altsystemen aufzuzeigen.

In der angelsächsischen Literatur werden für Altsysteme oft die Begriffe „*legacy software*“ und „*legacy systems*“ genutzt. Dieses Buch folgt diesem Sprachgebrauch und nutzt für die Software in Altsystemen den Begriff *Legacysoftware* und für das Altsystem als Ganzes den Begriff *Legacysystem*.

1.2 Legacysystem

Was ist jedoch ein Legacysystem, oder was charakterisiert es?

> *Ein Legacysystem ist ein soziotechnisches System, welches Legacysoftware enthält.*

Folglich machen Legacysysteme nur innerhalb einer Organisation Sinn. Umgekehrt lässt sich ein solches System nicht abstrakt, quasi im Vakuum, ohne Kenntnis der dazugehörigen Organisation beurteilen. Beide, das Legacysystem wie auch die Organisation, sind parallel miteinander und aneinander gewachsen und schwer trennbar verwoben.

> *Legacysoftware ist eine geschäftskritische Software, welche nicht, oder nur sehr schwer, modifiziert*[2] *werden kann.*

Die Lebensdauer eines Softwaresystems kann sehr unterschiedlich sein und viele Großunternehmen haben Software im Einsatz, welche über 30 Jahre alt ist. Die meisten dieser großen Softwaresysteme sind geschäftskritischer Natur, d.h. sie sind im Tagesgeschäft unabdingbar. Ein Ausfall oder Wegfall würde den Zusammenbruch des Unternehmens, oder doch zumindest herbe Verluste, bedeuten. Diese Softwaresysteme nennt man Legacysysteme und die in ihnen enthaltene Software Legacysoftware. Zwar ist die Legacysoftware immer Bestandteil eines Legacysystems, doch kann das Legacysystem als Ganzes auch neuere Software enthalten, was der Regelfall ist.

Die Legacysysteme sind naturgemäß nicht dieselben Systeme, die vor 30 Jahren in Betrieb genommen wurden. Nein, sie haben sich oft sehr stark gewandelt und wurden, in der Regel, viel komplexer als ursprünglich antizipiert.

[2] Salopp formuliert: Legacysoftware ist Software, bei der wir nicht wissen, was wir mit ihr tun sollen, die aber noch immer eine wichtige Aufgabe erfüllt.

Ein großer Teil der Legacysoftware ist überhaupt nicht für diese lange Lebensdauer konzipiert worden.[3] Diverse äußere und innere Quellen waren für die Veränderungen an der Software verantwortlich; speziell die Veränderungen, welche durch das Geschäftsumfeld ausgelöst wurden, zwangen alle Softwaresysteme, sich anzupassen oder obsolet zu werden. Diese Form des Darwinismus führte zu einer Auslese, deren Ergebnis wir heute als Momentaufnahme sehen. Der Selektionsdruck wurde primär durch die Fähigkeit zur Anpassung ausgelöst, d.h. nur eine Legacysoftware, welche in der Lage war, sich in den letzten 30 Jahren anzupassen, ist heute noch existent. Neben den äußeren Einflüssen haben sich auch die unterschiedlichsten Softwareentwickler an der Legacysoftware versucht. Bei einer durchschnittlichen Betriebszugehörigkeit von 5-7 Jahren sind das immerhin 5 Generationen von Softwareentwicklern bei einem Systemalter von 30 Jahren. So ist es sehr ungewöhnlich, einen Menschen anzutreffen, der ein komplettes Verständnis des gesamten Legacysystems besitzt.

Die sehr lange Koexistenz von Legacysoftware und den Geschäftsprozessen innerhalb des Unternehmens hat zu einer Verknüpfung beider geführt, die jenseits der Dokumentation und der Wahrnehmung liegt. Die Geschäftsprozesse haben „gelernt“, die Stärken der Legacysysteme zu nutzen, bzw. ihre Schwächen zu umgehen. Geschäftsregeln, Abläufe, Ausnahmen und vieles andere sind in die Legacysoftware eingeflossen, ohne dass nach so langer Zeit noch ein einzelner Softwareentwickler dies vollständig weiß. Von daher birgt jeder Ersatz das Risiko, etwas vergessen zu haben.

Neben den hohen Risiken für eine Ablösung des Legacysystems selbst, stehen die stetig steigenden Kosten für die Maintenance der Legacysoftware. Diese Kosten sind nicht zu unterschätzen; so betrugen die Kosten für den Betrieb im Jahre 2002 nach Angaben von McKinsey in großen Unternehmen 65-75 % des gesamten IT-Budgets.

Die Unternehmen stehen also vor einem Dilemma.

- Auf der einen Seite ist es risikoreich das Legacysystem zu ersetzen.
- Auf der anderen Seite steigen die Unterhaltskosten stetig an!

Ein Legacysystem ist nicht nur einfach ein Stück „alte“ Software, sondern ein hochkomplexes soziotechnisches Gebilde, bestehend aus den unterschiedlichsten Teilen, s. Abb. 1.1.

Die logischen Teile eines solchen Systems sind:

- Hardware – In vielen Fällen wurde die Legacysoftware für eine Mainframehardware konzipiert und gebaut, welche heute obsolet ist[4] oder nicht

[3] Softwareentwickler haben für die lange Lebensdauer solcher Systeme das Sprichwort *Provisorien leben am längsten* geprägt.

[4] In den letzten Jahren ist allerdings eine Renaissance der Mainframes zu beobachten. Nach vielen Jahren eines „Weg-von-der-Mainframe“-Programms wird die Mainframe als Backendsystem heute wieder gesellschaftsfähig.

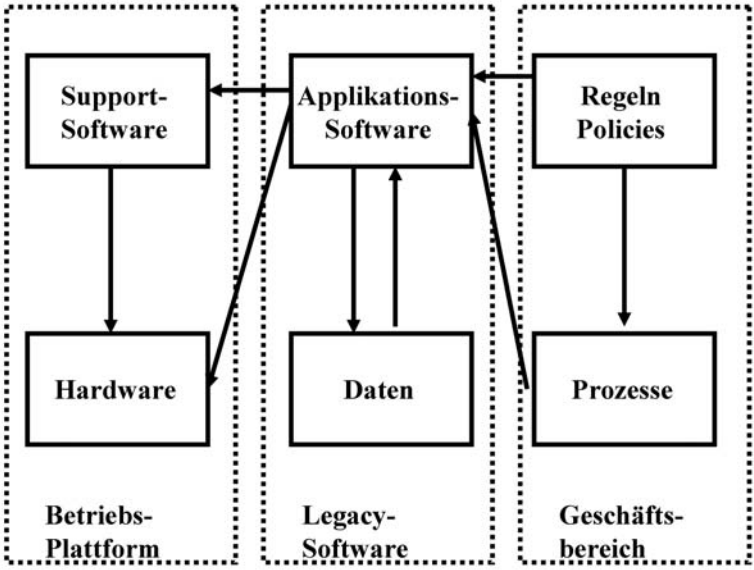

Abb. 1.1: Das Legacysystem als soziotechnisches Gebilde

mehr zur aktuellen strategischen Ausrichtung des betroffenen Unternehmens passt.

- Supportsoftware – In der Regel verlässt sich die Legacysoftware auf bestimmte Betriebssystemeigenschaften, sowie Utilities, welche vorhanden sein müssen. Dass ein Betriebssystemwechsel Probleme verursachen kann, ist nicht nur in der Mainframewelt bekannt, sondern war auch in der Windowswelt diverse Male schmerzhaft spürbar.
- Applikationssoftware – Dies ist die eigentliche Legacysoftware. Oft wird hierfür fälschlicherweise auch der Begriff Legacysystem benutzt.
- Daten – Hierbei handelt es sich um die Daten, welche von der Legacysoftware verarbeitet und genutzt werden. Diese können entweder datei- oder datenbankbasiert existieren.
- Prozesse – Die Geschäftsprozesse werden von den Unternehmen eingesetzt um ein Ergebnis zu erhalten; sie dienen der Durchführung von Tätigkeit um das gegebene Unternehmensziel zu erreichen.
- Regeln – Innerhalb der Geschäftswelt existiert ein reicher Regel- und Verhaltenscodex für Tätigkeiten innerhalb einer Domäne. So unterliegt beispielsweise die Buchhaltung strengen Wirtschaftsprüferregeln.

Das Legacysystem besteht aus allen diesen Teilen und alle verändern sich bzw. lösen Veränderungen aus, insofern ist ein Legacysystem ein hochkomplexes Gebilde. Als solches lässt es sich nicht auf einfache Lösungschemata reduzieren.

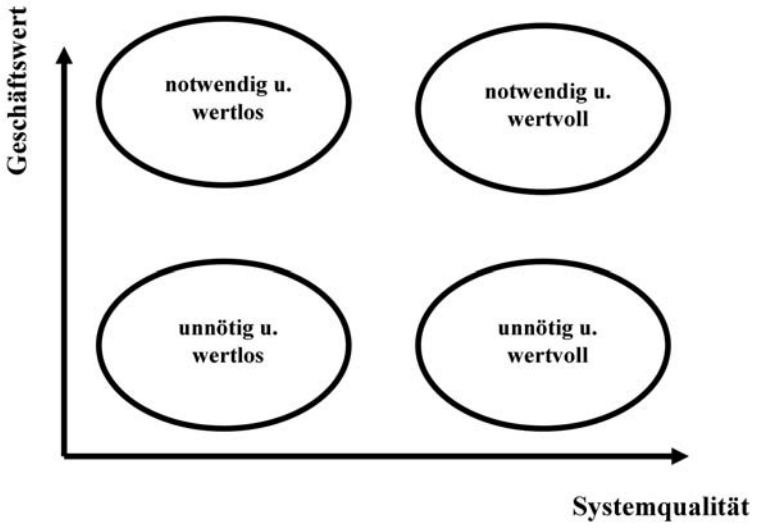

Abb. 1.2: Das Legacysystem-Assessment

1.3 Assessment

Bevor man sich überhaupt Gedanken über eine mögliche Strategie im Umgang mit der Legacysoftware macht, sollte man ein Assessment, d.h. eine Bewertung des Legacysystems, vornehmen. Eine solche Bewertung ist nicht nur für Legacysysteme unabdingbar, sondern sollte grundsätzliche für alle Systeme in Betracht gezogen bzw. durchgeführt werden. Alle im Einsatz befindlichen Softwaresysteme, nicht nur solche vom Typus Legacy, fallen in vier Kategorien, s. Abb. 1.2, wobei die Dimensionen, hier der Geschäftswert und die Qualität der Software, auf der Ordinate bzw. der Abszisse dargestellt sind. Diese Kategorien sind:

- unnötig und wertlos – Solche Systeme sind teuer und tragen nicht wirklich zum Geschäftserfolg bei. Im engeren Sinne handelt es sich hierbei nicht um Legacysysteme, da diese per definitionem geschäftskritisch sind, was Systeme mit niedrigem Geschäftswert nicht sein können.
- notwendig und wertlos – Eine ganze Kategorie von Legacysystemen fällt in diese Klassifikation, da solche Systeme ihren heutigen Zustand oft über eine sehr lange Evolutionsphase erreicht haben, s. Kap. 4. Im Sinne der unterschiedlichen Migrationsstrategien, s. Kap. 5, sind dies ideale Kandidaten für einen Ersatz.
- unnötig und wertvoll – Hierbei handelt es sich nicht um Legacysysteme im eigentlichen Sinne, da der notwendige geschäftskritische Anteil für eine

Legacysoftware nicht gegeben ist. Trotzdem wird man diese Systeme einer langen und sorgfältigen Maintenance, s. Kap. 7, unterziehen. Aus diesem Blickwinkel können ähnliche Mechanismen wie bei der Legacysoftware auf diese Systeme angewandt werden.

- notwendig und wertvoll – Für diese Systeme gilt dasselbe wie für die vorhergehende Klassifikation mit dem einzigen Unterschied, dass es sich hierbei tatsächlich um Legacysysteme handelt.

1.4 Dualismen

Ein Teil der Probleme, welche heute mit den Legacysystemen auftauchen, gehen auf einen fundamentalen Wandel zurück, der Ende der achtziger Jahre stattgefunden hat. Es ist der Wandel in der Betrachtung von Software. Das Verhältnis zwischen Hard- und Software kann von zwei unterschiedlichen Blickwinkeln aus gesehen werden:

- Die Software **steuert** die Hardware.
- Die Hardware **führt** die Software **aus**.

In der ersten Betrachtungsweise ist die Software ein Kontrollprogramm, welches die Hardware, sprich den Computer, kontrolliert. In der anderen Sichtweise ist es genau umgekehrt. Der Computer stellt eine Ablaufumgebung zur Verfügung und er „kontrolliert“ das Programm. In den sechziger und siebziger Jahren war die erste Denkweise weit verbreitet und kann noch heute in Realtime- bzw. *Embedded*-Systemen wiedergefunden werden.

Die Spuren der ersten Denkweise lassen sich oft in der Legacysoftware wiederfinden. Dies ist nicht besonders verwunderlich, da ein großer Teil der Softwareentwickler, die an den entsprechenden Legacysystemen beteiligt waren, in den frühen Jahren der IT ihr Handwerk erlernten. Der Versuch den Computer zu kontrollieren führt zu spezifischen Konstrukten, welche nur für einen bestimmten Compiler oder eine besondere Hardwareplattform Sinn machen. Für Softwareentwickler, welche die zweite Denkweise verinnerlicht haben, erscheinen diese Konstrukte sehr verwirrend, da in der zweiten Denkweise die Hardware nicht berücksichtigt wird oder werden muss.

Ein zweiter Dualismus ist im Zusammenhang mit Legacysoftware zu beobachten: Wir sind in der Lage, uns sehr schnell an schnelle Änderungen anzupassen, aber wir sind fast gar nicht in der Lage, uns auf langsame Veränderungen einzustellen. Ein Beleg dafür ist die Entwicklung des World Wide Web oder der mobilen Telefonie: Wir konnten uns sehr schnell an diese schnelle Veränderung anpassen, doch die Daten der meisten Großunternehmen befinden sich noch im Zustand der siebziger Jahre!

Diese unterschiedlichen Sichten auf Software und ihre Entwicklung sind nicht die einzigen Dualismen, die im Umfeld der Legacysysteme vorkommen. Ein anderes Spannungsfeld wird durch den Unterschied zwischen kommerzieller und individueller Software bzw. ihrer Nutzung und Weiterführung innerhalb der Legacysysteme aufgebaut. Die kommerzielle Software, s. Kap. 10,

besitzt zwar eine Reihe von vermeintlichen Vorzügen, hat aber auch eine Menge von realen Nachteilen. Der tatsächliche Einsatz von kommerzieller Software anstelle oder innerhalb von Legacysystemen verlangt, da es sich stets um soziotechnologische Systeme handelt, notwendigerweise auch ein gewisses Maß an organisatorischen Anpassungen.

Neben der Frage nach dem Kauf von Software steht Legacysoftware mittlerweile auch beim Out- oder Insourcing, nicht nur in Bezug auf eine Neuentwicklung, sondern auch bezüglich der Maintenance, in der Diskussion. Outsourcing und dabei speziell das Offshoring, s. Kap. 8, hat zum Teil drastische Konsequenzen, welche nicht unbedingt a priori bekannt sind.

Zusätzlich zu den bisher aufgezählten Punkten macht sich der Methodenstreit zwischen den „klassischen“ Vorgehensmodellen, s. Abschn. 11.2, und den agilen Verfahren, s. Abschn. 11.4, auch bei der Legacysoftware bemerkbar. Es ist im Grunde der Streit zwischen einer prozesszentrierten und einer produktzentrierten Sicht.

Die Summe dieser unterschiedlichen Denkweisen und Strömungen hat über die letzten 30 Jahre hinweg deutliche Spuren in den Legacysystemen hinterlassen.

2

Messbarkeit

Gebt, so wird euch gegeben.
Ein volles, gedrücktes, gerütteltes und überfließendes
Maß wird man in euren Schoß geben;
denn eben mit dem Maß,
mit dem ihr meßt,
wird man euch wieder messen.

Lukas **6**, 38

Im Umgang mit der Legacysoftware ist es notwendig, quantifizierbare Größen bestimmen zu können. Neben einer rein subjektiv geprägten Qualitätsbetrachtung bieten quantifizierbare Größen die Möglichkeit, sowohl den Zustand des Legacysystems zu bestimmen sowie Steuerungsgrößen direkt oder indirekt zu berechnen.

Erst die Messbarkeit von Eigenschaften macht eine Legacysoftware vergleich- und bewertbar. Dies wurde in der Physik am deutlichsten 1891 durch den bekannten Physiker Lord Kelvin beschrieben:

> *When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot express it in satisfactory kind; it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science.*

Die Softwareentwicklung hat zwei Ursprungslinien, zum einen Softwareentwicklung als Ingenieurleistung und zum anderen Softwareentwicklung als „Kunstwerk“, mehr eine Art Handwerksleistung. Die einen verstehen sich als Künstler, die anderen als Ingenieure. Methodiken wie Patterns, Best Practices oder auch die agilen Vorgehensweisen fallen stärker in den „künstlerischen“ Sektor, während Model Driven Architecture, 4GL-Sprachen, Unified Process und Unified Modelling Language dem ingenieurmäßigen Sektor zuzuordnen sind. Die Verwendung von Metriken jedoch entstammt eindeutig der ingenieurmäßigen Disziplin, da diese sich sehr gerne auf quantifizierbare Größen beruft.

Eine Metrik, s. Abb. 2.1, braucht immer folgende fünf Größen:

- Eine Messvorschrift, die vorgibt, was und wie gemessen werden soll.
- Ein Modell, das Parameter kennt, um aus dem Modell, den Parametern und der Messung eine Vorhersage zu erzeugen.

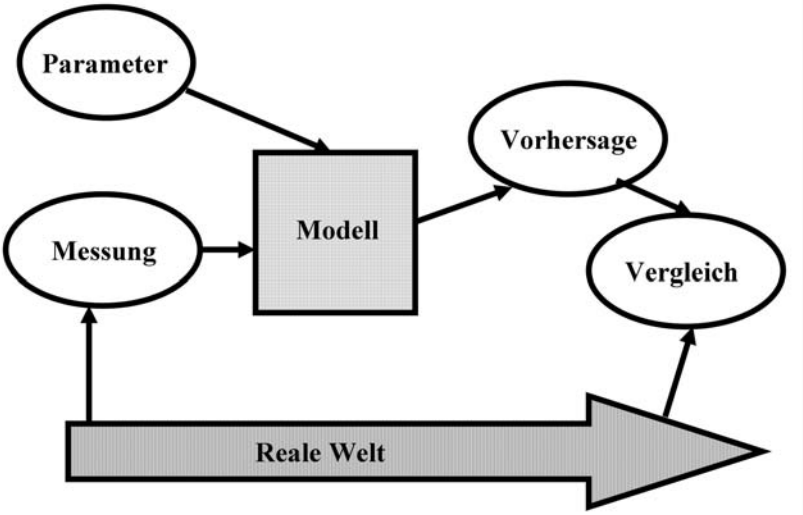

Abb. 2.1: Metriken brauchen Modelle und Messvorschriften

- Einen Satz von Parametern, damit das generische Modell konkretisiert werden kann.
- Eine Spezifikation der Bedeutung und Interpretation der Metrik.
- Den Vergleich der Vorhersage des Modells mit der realen Welt. Ohne eine solche, prinzipiell existente, Vergleichsmöglichkeit bleibt die Metrik mehr oder minder metaphysisch.

Die angesprochene Messung besteht immer aus einer Messvorschrift und einer Einheit und stellt den Versuch dar, eine Abbildung aus der realen, physischen Welt in eine mathematische Modellwelt[1] vorzunehmen. Aus der Physik sind Einheiten wie kg, m, Pa, h, usw. bekannt. Beim Messen von Software sind die Einheiten recht einfach: Sie sind entweder dimensionslos – damit haben sie die Einheit 1 – oder sie sind mit der Zeit, bzw. der inversen Zeit (Hertz: $1\text{Hz} = \frac{1}{s}$) verknüpft. Die durchgeführte Messung liefert stets eine Momentaufnahme des Messgegenstandes. Die Messergebnisse können genutzt werden, um Aussagen über die Legacysoftware zu gewinnen.

Leider wird aber bei Messungen ein zweiter Punkt oft übersehen:

> Kein Ergebnis einer Messung ist exakt. Es existiert immer ein gewisses Maß an Unsicherheit in der Messung.

[1] Speziell bei Metriken wie den Funktionspunkten ist zur Zeit noch unklar, was eigentlich das zugrunde liegende Modell ist. Oder anders formuliert: Wie wird eigentlich ein Funktionspunkt definiert?

Recht schnell stellt sich die Frage: Was und wie soll gemessen werden und wie wird das Ergebnis genutzt? Es existieren heute, speziell im objektorientierten Umfeld, eine große Anzahl von verschiedenen Messverfahren. Alles Mögliche kann gemessen werden, nur ist völlig unklar, was das Ergebnis der jeweiligen Messung wirklich bedeutet. Im vorliegenden Buch werden die unterschiedlichsten Metriken betrachtet, allerdings werden die Ergebnisse stets für tendenzielle Aussagen genutzt.

Für die Begutachtung bzw. weitere Entwicklung von Legacysoftware sind im Grunde nur drei Metriktypen entscheidend:

- Komplexitätsmetriken wie McCabe und Halstead
- der Maintainability Index
- die Entropie

Diese drei Metriktypen ermöglichen es, die Entwicklung und den aktuellen Zustand von Legacysoftware zu begutachten.

Die Messungen an einer Legacysoftware werden durch die Metriken bestimmt und durchgeführt. Mathematisch gesehen ist eine Metrik M der Abstand zweier Punkte in einem Raum mit den Eigenschaften:

$$\begin{aligned} M(\mathbf{a},\mathbf{b}) &\geq 0, && \forall \mathbf{a},\mathbf{b}, \\ M(\mathbf{a},\mathbf{a}) &= 0, && \forall \mathbf{a}, \\ M(\mathbf{a},\mathbf{c}) &\leq M(\mathbf{a},\mathbf{b}) + M(\mathbf{b},\mathbf{c}), && \forall \mathbf{a},\mathbf{b},\mathbf{c}. \end{aligned} \tag{2.1}$$

Die so definierte Metrik ist stets positiv definit mit $M(\mathbf{0}) = 0$. Für Softwaremetriken hat es sich bewährt, als Referenzvektor den Nullvektor zu wählen, was dazu führt, dass die Gleichungen 2.1 übergehen in:

$$\begin{aligned} M(\mathbf{a}) &> 0, && \forall \mathbf{a} \neq \mathbf{0}, \\ M(\mathbf{a},\mathbf{b}) &\leq M(\mathbf{a}) + M(\mathbf{b}), && \forall \mathbf{a},\mathbf{b}. \end{aligned} \tag{2.2}$$

Jede Metrik muss den in den obigen Gleichungen 2.2 gewählten Eigenschaften genügen.

Es existiert eine große Anzahl von unterschiedlichen Softwaremetriken; leider sind alle in ihrer Vorhersagekraft und Vergleichbarkeit nicht besonders ausgeprägt. Dafür gibt es folgende Gründe:

- Viele Metriken haben keine exakte Definition. Es ist völlig unklar, was Qualität oder Kohäsion bedeutet. Auch bei der Frage der Komplexität gibt es Widersprüche. Daher werden Metriken im Rahmen dieses Buches sehr viel stärker qualitativ als quantitativ betrachtet.
- Oft fehlt für die publizierten Metriken das dazugehörige Modell oder der Parametersatz, s. Abb. 2.1.
- Viele Metriken können in der Praxis überhaupt nicht eingesetzt werden, da ihnen jede praktische Relevanz fehlt.

Tab. 2.1: Die Zahl der Statements pro Funktionspunkt nach *Jones*

Programmiersprache	Statements
Assembler	320
C	128
Fortran 77	107
COBOL 85	107
C++	64
Visual Basic	32
Perl	27
Smalltalk	21
SQL	13

2.1 Komplexitätsmetriken

Der Versuch die Komplexität einer Software zu messen und daraus Strategien ableiten zu können hat innerhalb der Softwareentwicklung eine lange Tradition. Hauptsächlich werden Komplexitätsmetriken zur Bestimmung der Anzahl von Testfällen oder für die Prognose der Softwareherstellungskosten genutzt. Als steuernde Elemente für die Weiterentwicklung von Legacysoftware sind sie jedoch ungleich günstiger.

2.1.1 LOC

Die Bestimmung der Anzahl der Programmzeilen, der „Lines of Code“, ist vermutlich die älteste Metrik, um die Größe von Software zu bestimmen. Ihre einfache Messbarkeit macht sie zu einem Idealkandidaten für Metriken. Es wird ganz einfach die Zahl der Zeilen im Editor gezählt. Alternativ dazu könnte man auch das Gewicht des Programmlistings nehmen, da es linear mit der Zahl der Programmzeilen wächst. Es ist jedoch strittig, wie eine Codezeile zu bestimmen ist; insbesondere, wie folgende Typen von Zeilen behandelt werden sollen:

- Leerzeilen
- Kommentare
- Datendefinitionen
- Schnittstellendefinitionen
- zeilenüberspannende Codeteile

Die Vergleichbarkeit verschiedener Programme in einer Sprache ist folglich nur gewährleistet, wenn nicht die physischen, sondern die logischen Zeilen gezählt werden.

Es gibt natürlich Zweifel daran, wie sinnvoll die „Lines of Code“-Metrik n_{LOC} ist, da die Verwendung von Leerzeilen oder Kommentaren oder der Formatierstil einen großen Einfluss auf n_{LOC} haben. Aus diesen Gründen wird

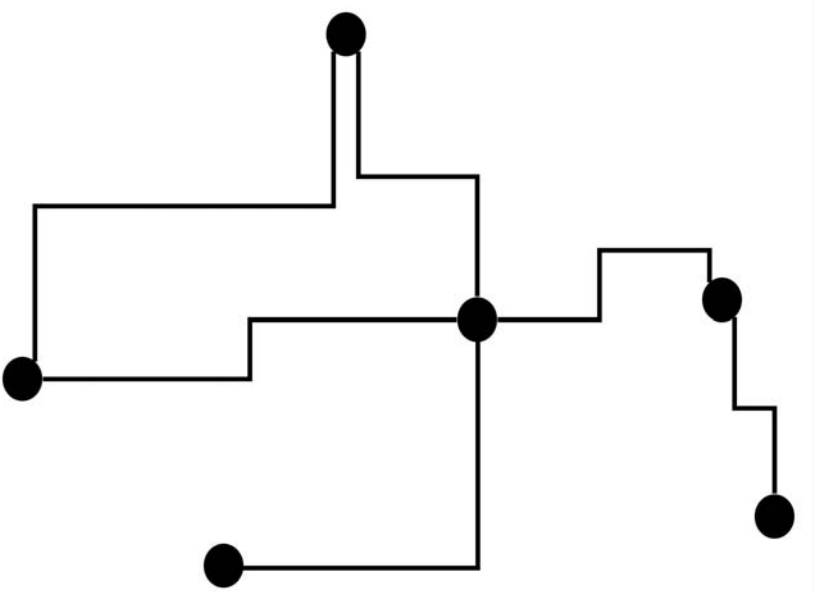

Abb. 2.2: Ein Graph mit Knoten und Kanten

die Metrik der „Lines of Code“ heute nicht mehr besonders ernst genommen. Trotzdem wird sie oft noch bei Programmen mit angegeben.[2] Der Vergleich zwischen unterschiedlichen Programmiersprachen ist recht aufwändig und wenig aussagekräftig, s. Tab. 2.1.

2.1.2 Graphen

Die einfachste Möglichkeit, Aussagen über Legacysoftware bzw. Software allgemein zu treffen, ist es, sich die Software in Form eines Graphen zu modellieren. Ein Graph besteht immer aus einer Menge von Knoten und Kanten, s. Abb. 2.2.

Wie ein solcher Graph aus der Legacysoftware entstehen kann ist nicht eindeutig, dafür gibt es, je nach Abstraktionsebene, unterschiedliche Möglichkeiten zur Modellierung.

- Batchgraph – Beim Batchgraphen stellen die einzelnen Programme und Dateien oder Devices, welche in einem konkreten Batch verwendet werden, die Knoten dar und die Abläufe in den Batchsteuerdateien[3] sind die Kanten.

[2] Dies geschieht wohl mehr aus psychologischen Gründen. Dahinter steckt die Einstellung: „Mein Programm ist größer als dein Programm. So gesehen bin ich besser oder wichtiger . . . “

[3] Oft in Form von JCL, Job Control Language.

- Transaktionsgraph – Hier werden die einzelnen Transaktionsprogramme als Knoten und die tatsächlichen Aufrufe, seien sie statisch oder dynamisch, als Kanten abgebildet.
- Callgraph – Der Aufrufgraph wird dadurch gebildet, dass innerhalb eines ausführbaren Programms alle möglichen Unterprogramme sowie das Hauptprogramm als Knoten und die möglichen Aufrufe als Kanten erscheinen.
- Logikgraph – Bei dieser, wohl die am meisten verbreiteten Form eines Graphen werden in einer Sourcedatei, nach der Expansion von Headern oder Copybooks, alle logisch möglichen Verzweigungen als Knoten und die möglichen Pfade als Kanten dargestellt, s. Abb. 2.3.

2.1.3 Average Node Degree

Der Average Node Degree eines Graphen, $\bar{N}_d$, wird rein topologisch definiert, s. Gleichung 2.3.

$$\bar{N}_d = 2\frac{n_{Knoten}}{n_{Kanten}}. \tag{2.3}$$

Die Knoten und Kanten in Gleichung 2.3 sind die Knoten und Kanten des jeweiligen Graphen. Bei einfachen Topologien lässt sich $\bar{N}_d$ direkt bestimmen, Gl. 2.4 - 2.7, wobei n die Zahl der Knoten angibt.

$$LineareKette = 2 + \frac{2}{n-1}, \tag{2.4}$$

$$Ring = 2, \tag{2.5}$$

$$Gitter = \frac{1}{1-\frac{1}{\sqrt{n}}}, \tag{2.6}$$

$$Baum = 2 + \frac{2}{n-1}. \tag{2.7}$$

Für den Fall, dass zwei Graphen A und B über $A \cup B$ miteinander verknüpft werden, ergibt sich der Average Node Degree zu:

$$\bar{N}_d\,(A \cup B) = \frac{N_{Kanten}(A) + N_{Kanten}(B) + N_{Kanten}(A \cap B)}{N_{Knoten}(A) + N_{Knoten}(B) - N_{Knoten}(A \cap B)}, \tag{2.8}$$

$$\approx \frac{1}{2}\left(\bar{N}_d(A) + \bar{N}_d(B)\right), \qquad fallsA \approx B. \tag{2.9}$$

Für den Spezialfall, dass beide Graphen etwa gleich groß sind und keine gemeinsamen Knoten haben, ergibt sich aus Gl. 2.8 näherungsweise Gl. 2.9.

2.1.4 McCabe-Metrik

Die McCabe-Metrik lässt sich direkt aus der Graphentheorie ableiten. Die McCabe-Metrik, auch cyclomatische Komplexität genannt, wird definiert durch:

Tab. 2.2: Beispielprogramm für die McCabe-Metrik

```
int max(int n, int m) {
        int value  = n;
        if (n<m) {
             value = m;
        }
        return value;
}
```

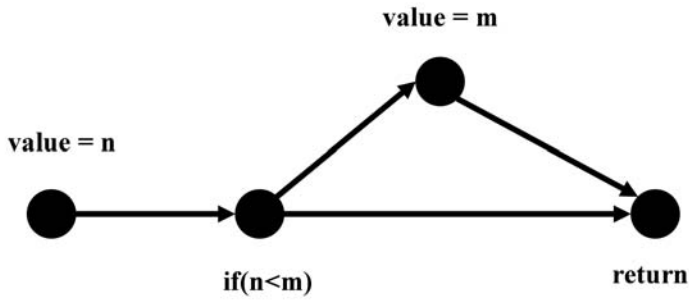

Abb. 2.3: Der Graph des Programms 2.2 mit 4 Kanten und 4 Knoten resultierend in $\gamma = 4$

$$\gamma = n_{Kanten} - n_{Knoten} + 2. \tag{2.10}$$

Im Grunde misst die McCabe-Metrik die Zahl der unabhängigen Ausführungspfade in einem Programm. Für ein einfaches Programm, wie z.B. Tab. 2.2 und Abb. 2.3, lässt sich die Metrik direkt ablesen und ergibt sich zu $\gamma = 2$.

Damit versucht eine McCabe-Metrik, die Topologie des Programmflusses zu messen. Ihr Ergebnis reagiert sehr viel stärker auf strukturelle Komplexität als das einfache Zählen von Programmzeilen n_{LOC}. Hauptsächlich wird die McCabe-Metrik benutzt um Risiken und Testkosten zu bestimmen. Es hat sich eingebürgert, Programme vor dem Testen nach der McCabe-Metrik zu klassifizieren. Ein Beispiel für Ergebnisse einer solchen Klassifikation ist in Tab. 2.3 dargestellt.

Tab. 2.3: Liste der Risikoschwellwerte für die McCabe-Metrik. Programmiersprachen, die hierbei betrachtet wurden, waren C und C++. Bei anderen Sprachen können die Werte zum Teil drastisch abweichen. C und C++ eignen sich gut für die McCabe-Metrik, da hier die Funktionen meist recht klein sind. FORTRAN liefert ähnliche Werte.

McCabe-Wert	**Risikoklasse**
$1-10$	niedrig
$11-20$	mittel
$21-50$	hoch
>50	sehr hoch

Bei der erweiterten cyclomatischen Komplexität werden auch Schleifen und logische Pfade berücksichtigt.

$$\widetilde{\gamma} = \gamma_{inkl.Loopsundlogische AND/OR}. \tag{2.11}$$

Der Vorteil der McCabe-Metrik ist, dass sie recht einfach berechnet werden kann, wenn der Graph sich vergrößert. Wenn zwei Graphen A und B kombiniert werden, so gilt für die McCabe-Metrik:

$$\gamma(A \cup B) = \gamma(A) + \gamma(B) - \gamma(A \cap B). \tag{2.12}$$

Bei den ganz einfachen Topologien lässt sich γ direkt bestimmen, Gl. 2.13 - 2.16, wobei n die Zahl der Knoten angibt.

$$LineareKette = -1, \tag{2.13}$$

$$Ring = 0, \tag{2.14}$$

$$Gitter = n + \sqrt{n}, \tag{2.15}$$

$$Baum = 1. \tag{2.16}$$

2.1.5 Card-Metrik

Die Card-Metrik ist eine Design-Metrik, welche ursprünglich für die Bewertung von Modulen eingeführt wurde. Die Komplexität γ ist definiert durch:

$$\gamma_i = \delta_i + \beta_i, \tag{2.17}$$

$$\delta_i = \frac{1}{m} \sum_{j=1}^{m} \phi_i^2(j), \tag{2.18}$$

$$\beta_i = \frac{1}{m} \sum_{j=1}^{m} \frac{\nu_i(j)}{\phi_i(j)+1}, \tag{2.19}$$

für das Modul i, mit den Größen: m als Anzahl der internen Prozeduren des Moduls, ϕ als Zahl der Aufrufe des Moduls nach außen und ν als Zahl der I/O-Variablen im Modul.

Der erste Term δ_i, s. Gl. 2.17 bzw. Gl. 2.18, wird oft als strukturelle Komplexität und der zweite Ausdruck in Gl. 2.17, s. Gl. 2.19, als Datenkomplexität bezeichnet.

Die Card-Metrik ist für die Messung der Komplexität eines Programms mit allen seien Unterprogrammen bzw. für die Komplexität einer Schnittstelle geeignet.

2.2 Halstead-Metriken

Die Halstead-Metriken stellen einen Satz von Metriken dar, um die Rechenkomplexität, die „Computational Complexity“, eines Programms zu bestimmen. Sie gehören zu den ältesten erstellten Metriken und werden in der Literatur sehr unterschiedlich bewertet. Auf Grund der Tatsache, dass sie sich auf die Rechenkomplexität beziehen, ist die Vergleichbarkeit zwischen verschiedenen Sprachen nicht gegeben. Eine Sprache mit einer mächtigen Klassenbibliothek oder mit wenigen, aber sehr fachspezifischen Befehlen schneidet bei den Halstead-Metriken immer gut ab.

Der Satz der Halstead-Metriken, das Volumen Θ_V, die Schwierigkeit Θ_D und der Aufwand Θ_E sind neben der Programmlänge N_S und dem Vokabular η_V definiert durch:

$$N_S = N_{total}(Operatoren) + N_{total}(Operanden), \tag{2.20}$$

$$\eta_V = N_{distinct}(Operatoren) + N_{distinct}(Operanden), \tag{2.21}$$

$$\Theta_V = N_S \log_2(\eta_V), \tag{2.22}$$

$$\Theta_D = \frac{N_{total}(Operanden)}{N_{distinct}(Operanden)} \times N_{distinct}(Operatoren), \tag{2.23}$$

$$\Theta_E = \Theta_V \Theta_D. \tag{2.24}$$

Wobei die Zahl der verschiedenen Operatoren bzw. die Gesamtzahl der Operatoren wie auch die Zahl der Operanden, d.h. die Anzahl der Variablen und Konstanten, in ein Programm eingehen. Ein sehr einfaches Programmstatement der Form:

$$x = \sqrt{y}.$$

enthält zwei Operatoren: $=$ und $\sqrt{}$ sowie zwei Operanden x und y. Dieses einfache Beispiel führt zu einem Satz von Halstead-Metriken:

$$\begin{aligned} N_S &= 4, \\ \eta_V &= 4, \\ \Theta_V &= 8, \\ \Theta_D &= 2, \\ \Theta_E &= 4. \end{aligned}$$

Tab. 2.4: Beispielprogramm für die Halstead-Metriken

```
      SUBROUTINE SORT(A,B)
      INTEGER A(100),N,I,J,SAVE,M
    C SORTIERROUTINE
      IF (N.LT.2) GO TO 40
      DO 30 I=2,N
      M=I-1
      DO 20 J=1,M
      IF(A(I).GT.A(J)) GO TO 10
      GO TO 20
   10 SAVE=A(J)
       A(J)=SAVE
   20 CONTINUE
   30 CONTINUE
   40 RETURN
       END
```

Ein zweites Beispiel für die Anwendung der Halstead-Metriken ist der Sourcecode in Tab. 2.4. Für diesen Sourcecode ergeben sich die Werte, s. Gl. 2.20 - Gl. 2.24, zu:

$$\begin{aligned} N_S &= 93, \\ \eta_V &= 27, \\ \Theta_V &= N_S \log_2 \eta_V, \\ &\approx 442, \\ \Theta_D &= 38.5, \\ \Theta_E &= 10045. \end{aligned}$$

Die Halstead-Metriken sind sensitiver auf die lexikalischen und textuellen Komplexitäten eines Programms, weniger auf die strukturellen oder ablauftechnischen Schwierigkeiten ausgerichtet. Programme mit einem hohen Anteil an Berechnungslogik lassen sich mit den Halstead-Metriken besser beurteilen als mit der McCabe-Metrik. Die Ratio hinter dieser Beobachtung ist, dass Berechnungslogik von den meisten Softwareentwicklern sehr linear ausprogrammiert wird und daher wenig strukturelle Komplexität enthält.

Es gibt eine interessante Ergänzung zu den empirischen Halstead-Metriken. Wird nämlich berücksichtigt, dass ein Mensch eine bestimmte Anzahl β von Unterscheidungen pro Sekunde machen kann, so ergibt sich die Zeit, welche ein Programmierer für ein Programm benötigt, zu:

$$T = \frac{\Theta_D}{\beta}.$$

Tab. 2.5: Funktionskategorien und ihr jeweiliges Gewicht

	Komplexität		
Funktionstyp	**Einfach**	**Mittel**	**Schwer**
Inputtransaktion	3	4	6
Outputtransaktion	4	5	7
Abfragetransaktion	3	4	6
Logische Funktion	7	10	15

Tab. 2.6: Funktionskomplexitätskategorien für Inputfunktionen

	Datenelemente		
Dateien	1-4	5-15	>15
0-1	Einfach	Einfach	Mittel
2	Einfach	Mittel	Hoch
>2	Mittel	Hoch	Hoch

In der Psychologie werden Werte für β zwischen 5 und 20 zitiert. Folglich würde der Aufwand für das Beispielprogramm Tab. 2.4 zwischen $8\frac{1}{2}$ und 34 Minuten für die Programmierung liegen.

2.3 Funktionspunkt-Metrik

Die Funktionspunktanalyse ist ein wohlbekanntes Modell zur Messung der Funktionalität eines Legacysystems aus Benutzersicht. Die Menge der Funktionspunkte wird in einem zweistufigen Verfahren ermittelt:

- Das Funktionsvolumen eines Legacysystems wird berechnet, indem die einzelnen Funktionen ein Gewicht bekommen. Die Summe dieser Gewichte ist das „Unadjusted Function Points“-Volumen.

$$V_{UFP}^{i} = \sum_{j} w_{j}^{i}, \qquad \{j \in \mathit{Applikation}(i)\}\,. \tag{2.25}$$

- Auf der Ebene des kompletten Legacysystems werden die unterschiedlichen Applikationen i gewichtet. Dies geschieht durch den „Value Adjustment Factor“, welcher von der relativen Wichtigkeit der unterschiedlichen Applikationen abhängt:

$$\sum_{i} V_{VAF}^{i} = 1. \tag{2.26}$$

Aus diesen beiden Faktoren ergibt sich dann das „Adjusted Function Points“-Volumen zu:

$$V_{AFP}^{i} = V_{VAF}^{i} V_{UFP}^{i}. \tag{2.27}$$

Für die konkrete Bewertung der einzelnen Funktionspunkte gibt es bisher kein universal festgelegtes Schema. Allerdings lässt sich die Menge der unterschiedlichen Funktionspunkte auf ein recht überschaubares Maß reduzieren, wenn anstelle einer fachlichen Funktion die konkrete Implementierungsfunktionalität in Betracht gezogen wird, s. Tab. 2.5 bzw. Tab. 2.6, wobei die Komplexität von der Zahl der Dateien oder Datenbanktabellen abhängt. Die Funktionspunktkomplexität, s. Tab. 2.5, hängt wiederum von der Art der Funktion und deren jeweiliger Komplexität ab.

Die Ergebnisse dieser „Messungen“ sind durchaus nicht eindeutig, da eine unterschiedliche Anzahl von Funktionspunkten berechnet wird, je nachdem, was als Ausgangspunkt für die Zählung genommen wird:

- Fachliche Spezifikation
- Technisches Design
- Benutzerdokumentation
- Sourcecode

Diese unterschiedlichen Ansätze führen zu einer unterschiedlichen Anzahl von Funktionspunkten. Da der Unterschied zwischen diesen unterschiedlichen Zählweisen systematischer Natur ist – er rangiert im Bereich von 0 bis 30% – darf nicht einfach ein Mittelwert aus den ganzen Zahlen gebildet werden. Dieser würde nur bei statistischen Fehlern korrigierend wirken. Sourcecodeanalyse kann eingesetzt werden, wenn es eine Korrelation zwischen der Anzahl der Sourcecodezeilen und den Funktionspunkten gibt.

2.4 Small-Worlds

Lange Zeit war man der Ansicht, dass große Systeme auf regulären Graphen aufbauen, so beispielsweise ein Kristallgitter. Aber neuere Forschungen, speziell im Bereich der Soziologie, haben gezeigt, dass die meisten beobachtbaren komplexen Systeme andere Eigenschaften haben.

Große Softwaresysteme leben stets in einem Spannungsfeld zwischen zwei Extremen, zum einen ein völlig zufälliges Netzwerk, zum anderen ein hochsymmetrisches rigides System, wie beispielsweise eine lineare Kette.

Ob ein gegebener Graph, und damit das gesamte System, nur zufällig ist oder eine inhärente Struktur besitzt, kann anhand von zwei verwandten Größen bestimmt werden:

- mittlerer Abstand zweier Knoten
- Clusterkoeffizient oder die Wahrscheinlichkeitsverteilung der Kanten

Ein rein zufälliger Graph, bei dem mit der Wahrscheinlichkeit p eine Kante entsteht, folgt in seiner Wahrscheinlichkeitsverteilung einer Poissonverteilung, s. Abb. 2.4, d.h.

$$P(k) = \mathrm{e}^{-pN} \frac{(pN)^k}{k!}, \tag{2.28}$$

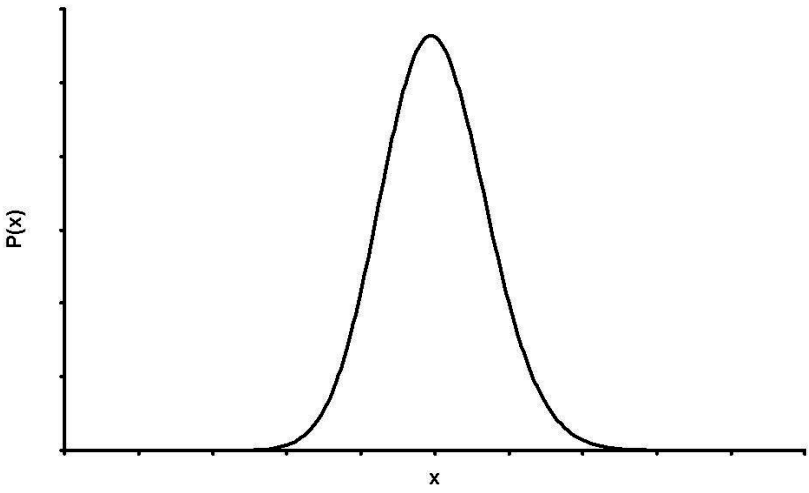

Abb. 2.4: Poissonverteilung

hierbei ist N die Gesamtzahl der Knoten und $P(k)$ gibt die Wahrscheinlichkeit an, einen Knoten mit genau k Kanten zu finden. Diese Poissonverteilung hat einen Average Node Degree von

$$\bar{N}_d = pN.$$

Der mittlere Abstand zwischen zwei Konten wird definiert durch:

$$d = \overline{\min d(i,j)},$$

wobei hier der minimalste Abstand zwischen zwei Endknoten i und j gemessen und dann über alle Endknoten i, j im Graphen gemittelt wird. Der mittlere Abstand in einem rein zufälligen Graphen ergibt sich zu:

$$d \approx \frac{\log_2 N}{\log_2 \bar{N}_d}.$$

Interessanterweise besitzen alle bisher untersuchten großen Systeme nicht eine Poissonverteilung, s. Gl. 2.28 und Abb. 2.4, sondern eine Verteilung der Form:

$$P(k) = Ak^{-\gamma} \mathrm{e}^{\frac{k}{k_c}}. \tag{2.29}$$

Die Wahrscheinlichkeit, dass ein gegebener Knoten eine Anzahl von Verknüpfungen k besitzt, ist gegeben durch:

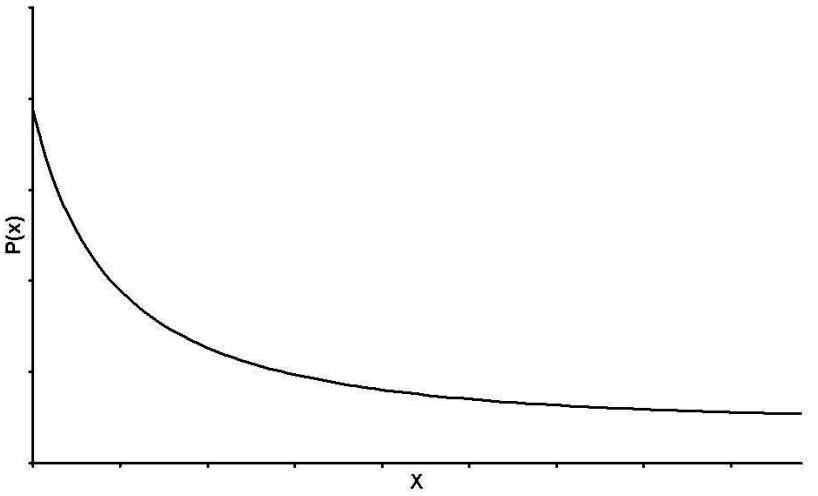

Abb. 2.5: Small-Worlds-Verteilung

$$P_{links}(k) \sim \frac{1}{k^{-\tau}}. \tag{2.30}$$

Eine solche Verteilung wird als Small-Worlds-Verteilung bezeichnet, was auf einen Versuch in den sechziger Jahren zurückgeht. Hierbei wurde nachgewiesen, dass jeder Einwohner der USA einen beliebigen anderen Einwohner über eine Kette von 6 Menschen kennt.

Die Zahl der untersuchten Systeme rangiert über Erdbeben, Hollywoodschauspieler, Softwaresysteme, Literaturreferenzen bis hin zu Stromnetzen, s. Tab. 2.7.

Bei der untersuchten Software lässt sich die Relation zwischen der Anzahl der Verknüpfungen und der Anzahl der Knoten darstellen durch:

$$N_{Kanten} \sim N_{Knoten}^{1.17}$$

Diese Relation scheint über einen großen Bereich von N_{Knoten} Gültigkeit zu besitzen.

Solche Small-Worlds-Netzwerke sind deswegen interessant, weil sie zwei Bedingungen genügen:

- Neue Knoten werden zufällig hinzugefügt.
- Neue Knoten verbinden sich mit vorhandenen unter Bevorzugung von Knoten mit bereits hohen Verbindungszahlen.

Beide Eigenschaften machen Small-Worlds-Netzwerke zu idealen Modellen für Legacysoftware, da diese, bedingt durch die langen Evolutionszeiten, sehr ähn-

Tab. 2.7: Exponentialkoeffizient τ in Small-Worlds-Netzwerken

Netzwerktyp	τ
Erdbeben und Richterskala	2
Hollywoodschauspieler und Filme	2.3
Internet	2.1
Literaturreferenzen	3
Stromnetz	4
E-Mail	1.8
JDK	2.4-2.55
GTK	2.5
Yahoopops	2.7
Linux Kernel	2.85
Mozilla	2.72
XFree86	2.79
Gimp	2.55

liche Charakteristika haben. Es gibt eine weitere, dynamische Eigenschaft von Small-Worlds-Netzwerken, welche der von Legacysoftware ähnelt:

- Das Versagen eines zufälligen Knotens hat einen vernachlässigbaren Effekt auf das Gesamtsystem. Dies entspricht dem Zustand eines Legacysystems, das ein riesiges Backlog an Defekten hat, aber trotzdem sehr aktiv genutzt wird.
- Es gibt wenige zentrale Knoten, Hubs genannt, deren Versagen das gesamte System beeinflusst. Auch dies entspricht der Erfahrung mit Legacysoftware. Es existieren dort meist einige katastrophale Defekte, welche sich aber nur auf wenige Module konzentrieren.

2.5 Entropie

Die Entropie stellt ein Maß für die Unordnung in einem System dar. Ursprünglich im Rahmen der klassischen Thermodynamik definiert, wurde der Begriff auf die statistische Mechanik ausgedehnt. Die Entropiedefinition der statistischen Mechanik kann auf die Informationstheorie übertragen werden. Der Physiker Helmholtz entlehnte den Begriff Entropie aus dem griechischen Wort $\epsilon\nu\tau\rho\epsilon\pi\iota\nu$ mit der Bedeutung von „umkehren“ oder „umwenden“.

Die Entropie eines Graphen ist definiert als:

$$S = -\sum_{j=1}^{N} p_j \log_2 p_j, \tag{2.31}$$

wobei p_j die Wahrscheinlichkeit ist, dass ein Knoten zur Klasse j gehört. Die Klassifikation kann unterschiedlich gewählt werden, so beispielsweise die Zahl der eintreffenden und ausgehenden Kanten eines Knotens.

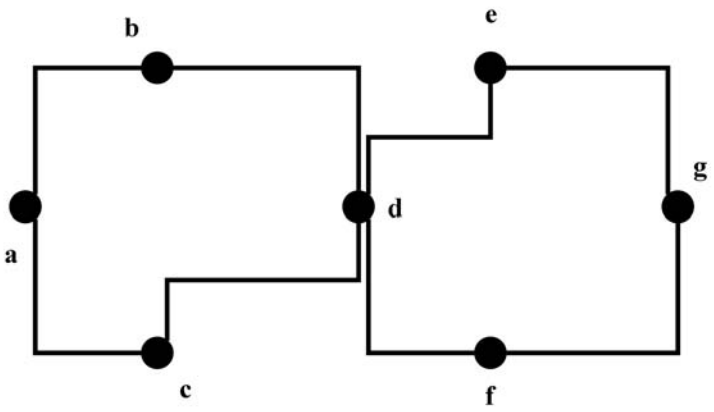

Abb. 2.6: Einfacher Graph mit mehreren Knotentypen

Wenn alle Knoten identisch sind, so gilt $p_j = 1$ und damit folgt:

$$S = 0 \quad wenn \quad p = 1.$$

Sind in einem Graphen alle Knoten unterschiedlich, so gilt:

$$p_j = \frac{1}{N}.$$

Hieraus resultiert die Entropie von:

$$\begin{aligned} S &= -\sum_{j=1}^{N} \frac{1}{N} \log_2 \frac{1}{N}, \\ &= \log_2 N. \end{aligned}$$

Daraus folgt, dass für jedes System gilt:

$$0 \leq S \leq \log_2 N.$$

Die Berechnung der Entropie kann an folgendem Beispiel erläutert werden:

Die enthaltenen Knoten $N = 7$ können anhand der Zahl der hinführenden und ausgehenden Kanten in die Kategorien: $\{a, b, e, g\}$, $\{c, f\}$ und $\{d\}$ klassifiziert werden. Diese Kategorien führen mit ihren jeweiligen Wahrscheinlichkeiten zu einer Entropie:

$$\begin{aligned} S(Graph) &= \frac{4}{7}\log_2\frac{4}{7} + \frac{2}{7}\log_2\frac{2}{7} + \frac{1}{7}\log_2\frac{1}{7}, \\ &= 0.4149. \end{aligned}$$

Wie verändert sich die Entropie, wenn einem Graphen ein neuer Knoten hinzugefügt wird? Zwar lässt sich diese Frage im Einzelfall nur durch eine exakte Berechnung mit Hilfe der Klassifikation beantworten, für sehr große Systeme mit $N \gg 1$ folgt näherungsweise

$$\begin{aligned} \Delta S &= S - S_0, \\ &\approx \log_2(N+1) - \log_2(N), \\ &= \frac{1}{\ln 2}\ln\left(1 + \frac{1}{N}\right), \\ &\approx \frac{1}{\ln 2}\left(1 - \frac{1}{N}\right). \end{aligned}$$

Wenn zwei Graphen A und B zusammengefügt werden, so ergibt sich die gemeinsame Entropie näherungsweise zu:

$$S(A \cup B) \approx S(A) + S(B) + \frac{1}{N}\log_2 N. \tag{2.32}$$

Bei sehr großen Systemen ist die Zahl N so groß, dass sich durch das Hinzufügen eines Knotens die Entropie um eine Konstante erhöht.

$$\Delta S = \frac{1}{\ln 2}.$$

Die so gewählte Definition über den Graphen berücksichtigt jedoch nicht die innere Entropie der Knoten. Wenn die innere Entropie der Knoten mit ins Kalkül gezogen wird, ergibt sich die Entropie zu:

$$S = S(Graph) + \sum_{Knotentyp} S(innere). \tag{2.33}$$

Für den Fall, dass mehrere Systeme miteinander verglichen werden müssen, empfiehlt es sich die Entropie zu normalisieren:

$$S^\dagger = \frac{1}{\max(S)}S, \tag{2.34}$$

$$= -\frac{1}{\log_2 N}\sum_l imits_{i=1}^{N} p_i \log_2 p_i, \tag{2.35}$$

mit der Folge, dass für die normalisierte Entropie gilt:

$$0 \leq S^\dagger \leq 1.$$

Die reguläre Entropie, s. Gl. 2.31, wird für den Rest des Buches als Beispiel für ein Komplexitätsmaß dienen.

2.6 Volatilität

Unter Volatilität wird das Risiko eines Legacysystems verstanden, auf Änderungen instabil zu reagieren. Eng verknüpft hiermit ist der Begriff der Stabilität. Je stabiler ein Legacysystem, desto weniger beeinflusst eine kleine Änderung das Gesamtsystem und desto niedriger ist die Volatilität.

Unter Stabilität wird hierbei die langfristige Stabilität der implementierten Geschäftsprozesslogik verstanden. Kurzfristige Änderungen, beispielsweise der Benutzeroberflächen, oder das Hinzufügen einzelner Attribute müssen von dem Legacysystem gut verkraftet werden. Üblicherweise wird der so genannte Volatilitätsindex V definiert durch:

$$V = \frac{\sum C_{structuralchange}}{\sum C_{structuralchange} + \sum C_{physicalchange} + \sum C_{userinterfacechange}} \tag{2.36}$$

Der Volatilitätsindex ist definiert über die Kosten C der Änderungen. Dies ist eigentlich ein indirektes Maß, da hier die Kosten gerechnet werden, welche zur Wiederherstellung der Aktionsfähigkeit des Gesamtsystems dienen. Die einzelnen Größen in Gleichung 2.36 sind:

- $C_{structuralchange}$, stellt die Kosten für die Änderung der Struktur dar. Diese Kosten werden auch als Costs of Deep Structural Changes bezeichnet, da sich die Kosten auf die Veränderung der Geschäftsprozesslogik und damit auf Änderungen im logischen „Herz“ beziehen.
- $C_{physicalchange}$, diese Größe bezeichnet den Aufwand für die Einführung neuer Attribute oder Datenbanksysteme, ohne dass hierbei die grundlegende Geschäftsprozesslogik verändert wird.
- $C_{userinterfacechange}$, die Kosten für die Veränderung der Benutzerschnittstellen gehören zu den einfachen Veränderungen im System.

Diese Einteilung der Arten und damit auch der Kosten der Veränderungen entspricht den unterschiedlichen Entwicklungsgeschwindigkeiten. So sind die verschiedenen Änderungszeiten für die einzelnen Gebiete heute gegeben durch:

$$\frac{1}{\nu_{userinterface}} = 2\,Jahre,$$
$$\frac{1}{\nu_{physicalchange}} = 5\,Jahre,$$
$$\frac{1}{\nu_{structuralchange}} = 10 - 20\,Jahre.$$

Durch die obige Definition des Volatilitätsindexes, s. Gl. 2.36, ergibt sich als obere und untere Grenze für den Wert des Volatilitätsindexes:

$$0 \leq V \leq 1,$$

Tab. 2.8: Sprachvolatilität

Sprache	**Wert**
Assembler	1
COBOL	2
Fortran	2
4GL	3
Java	4
C++	4

Obwohl der Volatilitätsindex primär durch die Kosten gemessen wird, lässt sich empirisch belegen, dass er sich auch an anderen Systemgrößen messen lassen kann.

$$V = \mathcal{F}\left(t_{Programmiersprache}, t_{Systemalter}, n_{LOC}\right), \tag{2.37}$$

wobei die einzelnen Parameter definiert sind durch:

- $t_{Programmiersprache}$, das Alter der eingesetzten Programmiersprache, das in den Stufen:
  klassifiziert ist.
- $t_{Systemalter}$, das durchschnittliche Alter der Applikationen,
- n_{LOC}, die Größe des Systems in Anzahl der Codezeilen.

Gefunden wurde ein empirischer linearer Zusammenhang, d.h.

$$V \approx c_0 + c_1 t_{Programmiersprache} + c_2 t_{Systemalter} + c_3 n_{LOC} + \epsilon, \tag{2.38}$$

wobei der Korrekturparameter ϵ sich als klein herausstellte.

Da die Größe n_{LOC} des Systems mit der Zahl der Knoten korreliert und bei einer mehr oder minder homogenen Umgebung die beiden Alter konstant sind, lässt sich der empirische Volatilitätsindex, Gleichung 2.38, annähern durch:

$$V \approx const. + c_3 n_{LOC}. \tag{2.39}$$

Folglich steigt in dieser Näherung die Volatilität des Systems mit der Zahl der Codezeilen N_{LOC} an.

2.7 Maintainability Index

Zusätzlich zur Maintenance, s. Kap. 7, definiert IEEE in ihrem Standardglossar des Software Engineering auch die Maintainability, d.h. die Fähigkeit von Software, gewartet zu werden:

> **maintainability**, ... The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.

Tab. 2.9: Maintenance Index Parameter für Pascal und C

Parameter	Wert
α_1	171
α_2	5.2
α_3	0.23
α_4	16.2
α_5	50
α_6	2.4

Maintainability ist folglich ein Maß für den Widerstand, welchen eine Legacysoftware der Maintenance entgegensetzt. Zurzeit existiert noch immer kein geschlossenes Modell für Maintainability, dennoch können die Basismetriken, insbesondere die Halstead-Metriken, s. Abschn. 2.2, und die McCabe-Metriken, s. Abschn. 2.1.4, bei der Beurteilung bzw. Annäherung an ein Modell zu Maintainability hilfreich sein. Allerdings existiert auch der so genannte Maintainability Index, s. Gl. 2.40. Dieser Maintainability Index leitet sich aus empirischen Untersuchungen an großen Softwareprojekten des amerikanischen Verteidigungsministeriums ab. Neben der rein statischen Messung des Maintainability Index ist seine zeitliche Entwicklung ein wichtiger Indikator für Vorhersagen oder zukünftige kritische Momente. Der Maintainability Index ist insofern dem Volatilitätsindex verwandt, s. Gl. 2.36.

Der Maintainability Index $\mathcal{M}$ ist definiert durch:

$$\mathcal{M} = \alpha_1 - \alpha_2 \ln\left(\overline{\Theta}_V\right) - \alpha_3 \overline{\gamma}(g') - \alpha_4 \ln \overline{n}_{lines} + \alpha_5 \sin\left(\sqrt{\alpha_6 \frac{\overline{n}_{comments}}{\overline{n}_{lines}}}\right), \tag{2.40}$$

mit den einzelnen Parametern:

- $\overline{\Theta}_V$ ist das mittlere Halsteadvolumen pro Modul, s. Gl. 2.22.
- $\overline{\gamma}(g')$ beschreibt die erweiterte, mittlere, cyclomatische Komplexität pro Modul, s. Gl. 2.11.
- $\overline{n}_{lines}$ ist die mittlere Anzahl der Codezeilen pro Modul.
- $\overline{n}_{comments}$ ist die mittlere Anzahl der Kommentarzeilen.

Die Parameter α_1 bis α_6 sind letztendlich von der Architektur und der jeweiligen Programmiersprache abhängig. Gesicherte Werte für die Parameter existieren für die Sprachen Pascal und C im Umfeld von so genannter *Defense Software*. Für diese spezielle Umgebung wurden die Parameter geeicht, s. Tab. 2.9. Dabei gelten Werte für den Maintainability Index im Bereich unter 50 für ein einzelnes Modul als schlecht und höhere Werte als gut.

Interessant ist die Tatsache, dass nur das Verhältnis der Codezeilen zu den Kommentarzeilen eine Rolle spielt, s. Abb. 2.7. Die absolute Zahl der Codezeilen pro Modul geht nur schwach[4] ein, allerdings sollte berücksichtigt werden,

[4] Die Wirkung der beiden Beiträge ist in Abb. 2.7 negativ dargestellt.

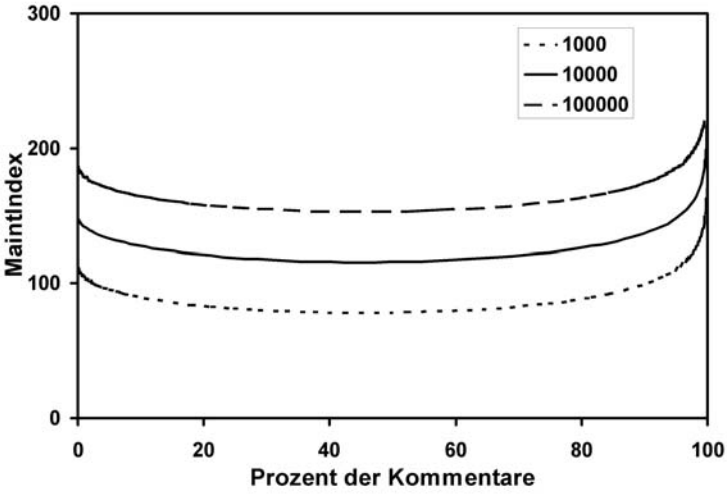

Abb. 2.7: Die Auswirkungen auf den Maintainability Index $\mathcal{M}$ durch die Veränderung der Anzahl der Kommentarzeilen, bei konstanter Komplexität, s. Gl. 2.40, $\alpha_4 \ln \overline{n}_{lines} - \alpha_5 \sin\left(\sqrt{\alpha_6 \frac{\overline{n}_{comments}}{\overline{n}_{lines}}}\right)$

dass größere Module mit mehr Codezeilen auch stets eine höhere Komplexität besitzen. Der Anstieg bei kleiner Anzahl von Kommentaren in Abb. 2.7 lässt sich auf die genutzte empirische Glättung für den Maintainability Index zurückführen. In den meisten Legacysystemen ist der prozentuale Anteil der Kommentarzeilen für faktisch alle Module relativ konstant, von daher wird sich nur in einem relativ engen Bereich der Abszisse in Abb. 2.7 bewegt.

Der Maintainability Index lässt sich auf diverse Arten nutzen. Zunächst einmal sollte angemerkt werden, dass allein schon die Messbarkeit und damit direkt die Quantifizierbarkeit von Maintenance ein wichtiger Schritt ist. Die offensichtlichen Verwendungsformen des Maintainability Index sind:

- Die Beobachtung der zeitlichen Entwicklung des Maintainability Index $\mathcal{M}$, um in der Lage zu sein, das Lebensende oder das Einsetzen einer intensiven Restrukturierung vorhersagen zu können.
- Während der Maintenance, sogar während der Entwicklung, kann der Maintainability Index $\mathcal{M}$ proaktiv benutzt werden, um negative Tendenzen frühzeitig zu bemerken und die entsprechenden korrigierenden Maßnahmen einzuleiten.

- Der Maintainability Index $\mathcal{M}$ kann dafür genutzt werden, eine Weak-Spot-Analyse durchzuführen, d.h. die Frage zu beantworten, welche Teile der Legacysoftware besonders stark gefährdet[5] sind.
- Mit der Beurteilung der Maintainability eines Fremdsystems auf einer quantifizierbaren Basis lassen sich die langfristigen Kosten für den Einsatz von Fremdsystemen oder auch COTS-Software, s. Abschn. 10.4, abschätzen.

Obwohl das Modell des Maintainability Index heuristisch ist, besitzt es doch die interessante Eigenschaft, dass sich der Maintainability Index $\mathcal{M}$ durch eine automatische Sourcecodeübersetzung kaum verändert[6]:

$$\mathcal{M}(\widehat{\Lambda}_{\Psi\to\Upsilon} m) \approx \widehat{\Lambda}_{\Psi\to\Upsilon} \mathcal{M}(m)$$

Bei näherer Betrachtung bedeutet dies, dass eine deterministische Umsetzung von einer Sprache Ψ in eine Sprache Υ die Maintainability nicht verändert.[7] Es zeigt sich, dass die Maintainability sich nicht auf die Sprache reduzieren lässt, d.h. andere Größen wie Architektur, Komplexität und die betrachtete Domäne sowie das Alter des Sourcecodes spielen eine sehr viel größere Rolle. Allerdings kann es sich in Extremfällen, wie z.B. dem Wegfall einer Produktplattform oder dem Aussterben einer Programmiersprache trotzdem lohnen, den Sourcecode automatisch zu übersetzen. Nur wird er deswegen nicht leichter wartbar!

2.8 Metrikbasierte Verbesserungen

Ziel hinter dem Einsatz einer Metrik ist nicht nur eine aktuelle Bestandsaufnahme über den Zustand, sondern auch die Fähigkeit die Zukunft zu beeinflussen. Neben der Veränderung der Legacysoftware an sich kann die Metrik auch eingesetzt werden, um damit den Prozess der Maintenance selbst, s. Kap. 7, zu verbessern, s. Abb. 2.8. Der Zyklus beginnt auf der linken Seite mit dem Problem oder der Zielsetzung. Die Organisation analysiert das Problem und ermittelt eine Reihe von möglichen Ursachen für das Problem. Für diese Form der Analyse wird eine Kombination aus Erfahrung und allgemein zur Verfügung stehendem Wissen, im Sinne einer Theorie, zur Verfügung gestellt. Im nächsten Schritt wird ein Messprogramm, eine Metrik bzw. eine Interpretation einer Metrik aufgebaut, um sich der Ursache besser zu nähern. Die Messungen werden durchgeführt und so kann man einer möglichen Lösung

[5] Im Sinne einer Risikoanalyse.

[6] Diese Beobachtung stammt aus der Umsetzung eines großen Migrationsprojekts der US Airforce, bei dem FORTRAN-Code in C-Code umgewandelt wurde.

[7] Das Marketing der Hersteller von automatischen Übersetzern demonstriert die Fähigkeiten ihrer jeweiligen Werkzeuge an mehr oder minder trivialen akademischen Beispielen in hochgradig kontrollierter Umgebung.

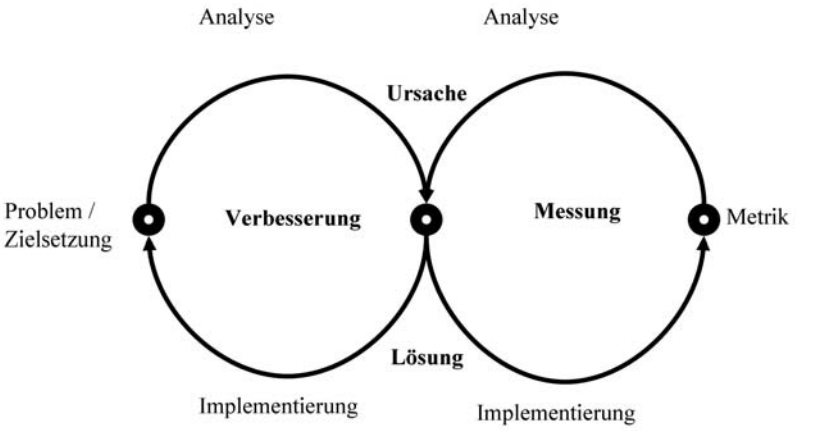

Abb. 2.8: Metrikbasierte Verbesserungen

oder einer besseren Präzisierung der Ursachen näher kommen. Schließlich reichen die gesammelten Daten aus, um den Prozess selbst zu verändern. Diese Veränderung wird dann dem Problem gegenübergestellt, d.h. die Veränderung verändert auch das Problem oder es wird durch das Verfahren gelöst.

Messungen sollten für jedes Unternehmen einen Mehrwert produzieren, wobei der Mehrwert von außerhalb, d.h. jenseits der Messung, kommen muss. Die Hauptgründe für die Implementierung eines Messprogramms, s. Abb. 2.8, sind:

- Reporting – Oft ist ein Reporting auch Bestandteil eines Service Level Agreements. Das Reporting ermöglicht bis zu einem gewissen Grad auch Vorhersagen, ob additive Schwellenwerte erreicht werden oder ob bestimmte Ereignisse vorliegen.
- Performanzmonitoring – Die Beobachtung der aktuellen Performanz geschieht in der Regel intern, im Gegensatz zum Reporting, welches meistens ex post erfolgt und oft auch externe Abnehmer hat.
- Lernen – Durch die Zunahme an Informationen über ein jeweiliges Gebiet lassen sich besser Planungen und Vorhersagen über zukünftige Aufgabenstellungen treffen.
- Performanzverbesserung – Hinter diesem Ansatz steckt die Idee, dass es eine Korrelation zwischen Messgrößen auf der einen Seite und einem Prozess oder Produkt auf der anderen Seite gibt. Folglich kann vorhergesagt werden, was eine Veränderung der Messgröße produzieren würde. Die Ziel-

setzung ist dann, ein Optimum für eine Größe, beispielsweise Aufwand, zu finden.

- Organisationsgüte – Eine Metrik kann im Sinne von CMM, s. S. 157, auch genutzt werden, um damit den „Reifegrad“ oder die Güte einer Organisation zu bestimmen.

3

Lebenszyklus

... translate thy life into death,
thy liberty into bondage:
I will deal in poison with thee,
or in bastinado, or in steel;
I will bandy with thee in faction;
I will o'errun thee with policy;
I will kill thee a hundred and fifty ways:
therefore tremble and depart.

As You Like It,
William Shakespeare

Jede Software befindet sich immer in einem definierten Zustand, d.h. implizit, dass jede Software, auch Legacysoftware, einen Lebenszyklus durchlebt. Der Lebenszyklus ist die definierte zeitliche Abfolge von Zuständen. Anhand des Lebenszyklusmodells werden die möglichen Zustände aufgezeigt, welche die Summe aller Softwaresysteme annehmen können. Insofern ist der Lebenszyklus ein generisches, idealtypisches Modell der Veränderung von Software.

Grundlage des Lebenszyklusmodells der Software ist die Feststellung, dass jede Veränderung der Software einen Übergang von einem Zustand der Software in einen anderen Zustand darstellt:

$$\hat{O}(t, t_0)\psi(t_0) \rightarrow \psi(t), \tag{3.1}$$

wobei der zeitliche Unterschied $\delta t = |t - t_0|$ hinreichend klein sein sollte.

Wenn die beiden Zustände $\psi(t_0)$ und $\psi(t)$ sehr nahe beieinander liegen, so gehören beide zum gleichen Zustand des Lebenszyklusmodells. Mathematisch gesehen bedeutet dies, dass es eine Schranke ϵ gibt, so dass für eine gegebene Metrik, s. Kap. 2, gilt:

$$M\left(\psi(t_0), \psi(t)\right) \leq \epsilon.$$

Ein solches metrisches Vorgehen lässt aber die Vergleichbarkeit unterschiedlicher Softwaresysteme als sehr fragwürdig erscheinen. Genau aus diesem Grund werden die möglichen Zustände nicht nach den internen, quantifizierbaren Eigenschaften der Software eingeteilt, sondern nach den äußeren, mehr qualitativen Eigenschaften. Genauer gesagt werden die Zustände nach der jeweiligen Form der Entwicklung $\hat{O}$, s. Gl. 3.1, klassifiziert; obwohl dies zu einem gewissen Grad willkürlich ist, hat sich in der Praxis ein hohes Maß

an Übereinstimmung über alle Softwaresysteme hinweg gezeigt. Die Zustände werden sehr viel stärker nach der Art ihrer Entstehung bzw. Verwendung klassifiziert. Diese Form der Einteilung ermöglicht es eine systemübergreifende Vergleichbarkeit sicherzustellen.

3.1 Zustände

Von der ersten Konzeption über die initiale Version bis hin zu einem aktiven Produkt, welches schließlich irgendwann einmal obsolet wird – so sieht der übliche Lebensweg von Software aus. Mit den frühen Phasen des Lebenszyklus befasst sich die Literatur sehr intensiv, allein die späten Phasen werden oft nicht ausführlich behandelt, so auch im Vergleich der Vorgehensmodelle RUP, s. Abschn. 11.2, und EUP, s. Abschn. 11.3. Für die Betrachtung von Legacysoftware sind die späteren Phasen interessant; zwar werden in den frühen Phasen, das sind solche vor der initialen Inbetriebnahme, grundlegende Eigenschaften definiert und quasi zementiert, jedoch lassen sich diese Eigenschaften in einem Legacysystem nur noch schwer verändern. Speziell Größen wie die der Legacysoftware unterliegende fachliche und technische Architektur lassen sich nur schwer nachträglich abändern.

Relativ klar wird das Lebenszyklusmodell, s. Abb. 3.1, wenn es als ein Stufenmodell betrachtet wird, das sich zunächst nur in eine Richtung, d.h. entlang der Zeitachse, bewegen kann. Bewegungen in andere Richtungen werden detaillierter in Abschn. 3.3 betrachtet. Die so entstehenden Stufen sind:

- Initiale Software
- Evolution
- Servicing
- Ausphasung
- Abschaltung

Hierbei wird zunächst von der zusätzlichen Problematik paralleler Releases abgesehen, s. Abschn. 3.2.

Obwohl die Zustände Evolution und Servicing meistens länger andauern als die initiale Phase, gibt es auch Ausnahmen; bei sehr komplexen Systemen kann die Umgebung sich so schnell verändern, dass die Software schon sehr schnell nach der Einführung obsolet geworden ist.

Initialer Software-Zustand

Die initiale Software ist die erste Lieferung, welche in Betrieb genommen wird. Sie hat typischerweise noch nicht alle fachlichen Funktionen, falls es sich um ein neues Produkt handelt, oder sie ist durch ein Reengineering eines bestehenden Produktes entstanden. Eine Eigenschaft ist jedoch charakteristisch: Die initiale Software besitzt schon die Architektur der Software, die während

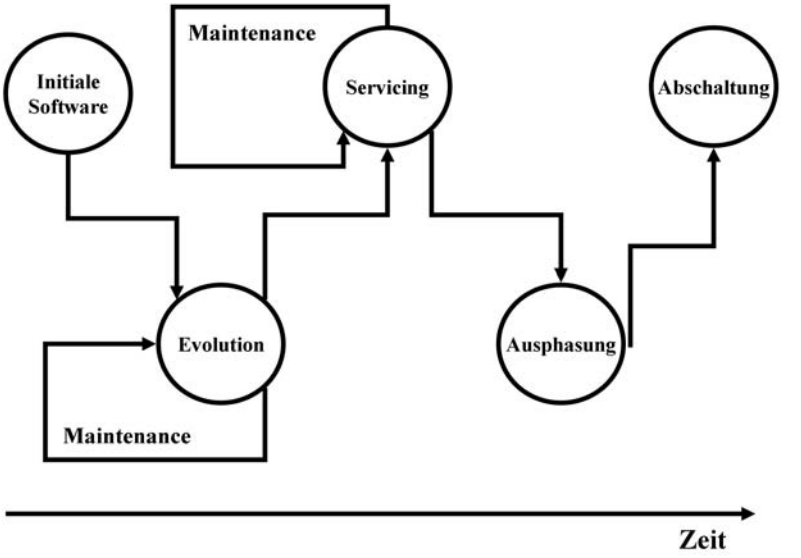

Abb. 3.1: Der einfache Lebenszyklus

des gesamten Lebenszyklus erhalten bleiben wird. Ein Reengineering mit einer neuen Architektur bedeutet immer auch eine neue initiale Software. Was hier als Zustand der initialen Software bezeichnet wird, ist in der Realität meistens sehr viel komplexer als hier dargestellt, aber eine Legacysoftware hat per definitionem stets ein gewisses Alter erreicht und ist damit auch immer über den Zustand der initialen Software hinweg. Einzige Ausnahme ist das oben erwähnte komplette Reengineering des Legacysystems, das zu einem initialen Software-Zustand zurückführen kann.

Ein zweites Charakteristikum ist das Wissen, welches das Entwicklungsteam zur Erstellung der initialen Software akquiriert hat. Dieses Wissen ist die notwendige Voraussetzung um die späteren Phasen überhaupt erst zu ermöglichen, wobei es für die Legacysysteme geradezu ein Kennzeichen ist, dass genau dieses Wissen in der Breite verloren gegangen ist.

Evolutionszustand

Erst eine erfolgreiche Einführung – d.h. die initiale Software steht zur Verfügung, hat eine Architektur und es wurde ausreichendes Domänenwissen aufgebaut – ermöglicht den Einstieg in die Evolutionsstufe. Das Ziel der Evolutionsstufe ist es, sich an die stetig wandelnden Benutzeranforderungen sowie Veränderungen der Umwelt anzupassen. Auch die eventuell vorgefundenen Defekte werden in dieser Stufe behoben. Allerdings sollte beachtet werden, dass diese Stufe primär durch die von der Anforderungsevolution ausgelöste

Co-Evolution, s. S. 185, bestimmt ist. In dieser Stufe wird das zunehmend gesammelte Wissen der Endanwender über die Software, sowie das Wissen der Softwareentwickler über die Domäne zur treibenden Kraft, um die Software selbst zu verändern. Aus Sichtweise der Betriebswirtschaft ist die Software im Markt erfolgreich und das Bedürfnis der Endanwender die Software zu erwerben ist stark ausgeprägt.

Das Domänenwissen und die Architektur machen diese Evolution erst möglich; erst beide zusammen ermöglichen eine Veränderung der Software, ohne dass allzu große Teile beschädigt werden. Wenn eine der beiden notwendigen Voraussetzungen verschwindet, das Wissen über die Domäne oder die Architekturintegrität, geht die Software in die nächste Stufe, das Servicing, über. Legacysoftware ist, leider, sehr selten in der Stufe der Evolution, sondern befindet sich, in der Regel in der Stufe des Servicing oder darüber hinaus.

Da die Evolutionsstufe durch zwei notwendige Voraussetzungen bedingt ist – Wissen und Architektur – eignet sie sich relativ schlecht zum Outsourcing, s. Kap. 8, außer in dem speziellen Fall, dass das originäre Erstellungsteam beim Outsourcing mitwandert. Wenn Schlüsselpersonen – wie Architekten und Designer – das Team verlassen, bewegt sich die Software sehr schnell auf die Stufe des Servicing hin.

Servicingzustand

Hat die Software das Ende der Evolutionsstufe erreicht, so tritt sie in die Stufe des Servicing ein. In der Servicingstufe finden auch noch Veränderungen statt, welche die Software durchaus beeinflussen können, allerdings liegt hier der Schwerpunkt sehr viel stärker auf dem Bereich der taktischen Änderungen. Typischerweise spricht man hier von „Patches“ und „Fixes“ für die korrektiven Teile, welche an die entsprechenden Endanwender ausgeliefert werden. Diese Stufe in der Entwicklung wird auch als „Software Maturity“ bezeichnet. Für einen COTS-Software-Hersteller, s. Kap. 10, ist dies der Zustand der „Reife“. Es werden kaum noch Investitionen getätigt und die Aufwände sind gering. Gleichzeitig können Lizenzen für ein fachlich ausgereiftes Produkt gut verkauft werden.[1]

Da diese Stufe durch das Wegbleiben eines der beiden notwendigen Voraussetzungen der Evolutionsstufe definiert ist, stellt sich hier die Frage, wie sich diese beiden Voraussetzungen verhalten. Das Wissen über die Software und die Domäne, wie auch die Architektur des Legacysystems befinden sich in einem stark negativen Feedbackloop:

[1] Es existieren Softwarefirmen, welche sich genau darauf spezialisiert haben, ein anderes Unternehmen aufzukaufen, das Produkte im Servicingzustand hat, s. Kap. 10. Nach dem Aufkauf wird ein großer Teil der Softwareentwickler entlassen und es werden nur noch Lizenzen verkauft und anschließend das Produkt „End of Life“ gestellt.

- Nachlassen der Architektur – Je weniger erkennbar eine Architektur in der Software ist, desto mehr Wissen über die Domäne und die konkrete Implementierung wird gebraucht, um in der Lage zu sein, die Legacysoftware weiter zu evolvieren.
- Nachlassen des Wissens – Je weniger über die konkrete Implementierung oder die Domäne bekannt ist, desto größer ist die Wahrscheinlichkeit, dass die Software sehr viel komplexer wird und damit, implizit, die Architektur verloren geht.

Das Feedback zwischen diesen beiden Kräften geht sehr schnell in negativer Richtung und führt zu einem rapiden Verfall der Software. Viele der heute beobachtbaren Legacysysteme befinden sich in diesem Zustand des Servicing.

Ausphasungszustand

Während der Ausphasung werden auch keine „Patches“ mehr geliefert, wie noch im Zustand des Servicing; damit unterliegt die Legacysoftware keinerlei Maintenance mehr. Eine solche Ausphasung kann unterschiedliche Gründe haben. Ein möglicher Grund ist die Ausphasung einer Software im Rahmen einer Migration, s. Kap. 5, in diesem Fall lässt es sich für eine gewisse Zeit mit einer Reihe von Fehlern leben, ohne diese zu beheben. Ein anderer möglicher Grund ist, speziell bei der COTS-Software, s. Kap. 10, dass der Hersteller die Software eingestellt hat.[2] Eine längerfristige Benutzung der Ausphasungssoftware durch die Endbenutzer führt zu einer Anhäufung von „Work-Arounds“. Auf Grund der Tatsache, dass die Software sich nicht mehr verändert, werden Teile in ihr semantisch reinterpretiert und anders benutzt, als ursprünglich intendiert war. Solche „Work-Arounds“ sind auch in früheren Stufen zu beobachten, nur werden sie dort durch die Geschwindigkeit bzw. durch die Nichtreaktion der jeweiligen Softwareentwicklung auf Kundenbedürfnisse ausgelöst und führen oft zu Konflikten. Mit der Einführung eines „Work-Arounds“ verändert der Endanwender unbewusst das Legacysystem; schließlich ist es ein soziotechnisches Gebilde, bei dem ein Teil der technischen Funktionalität auch durch organisatorische Maßnahmen ersetzt werden kann. Dadurch wird der „Work-Around“ zu einer Quasifunktionalität des Legacysystems. Eine softwaretechnische Veränderung des Systems führt dazu, dass der „Work-Around“ eventuell nicht mehr funktioniert, was Frustrationen auf Seiten der Endanwender zur Folge hat. Umgekehrt produzieren diese „Work-Arounds“ in der Regel große Probleme bei der Datenqualität, s. Abschn. 4.18, was für die Softwareentwicklung Frustration und erhöhten Aufwand bedeutet.

[2] Äußere Ereignisse – dass sind solche, die dem Hersteller nicht angelastet werden können, wie beispielsweise die Euroumstellung oder das Jahr 2000 – waren beliebte Auslöser zur Elimination solcher toter COTS-Software.

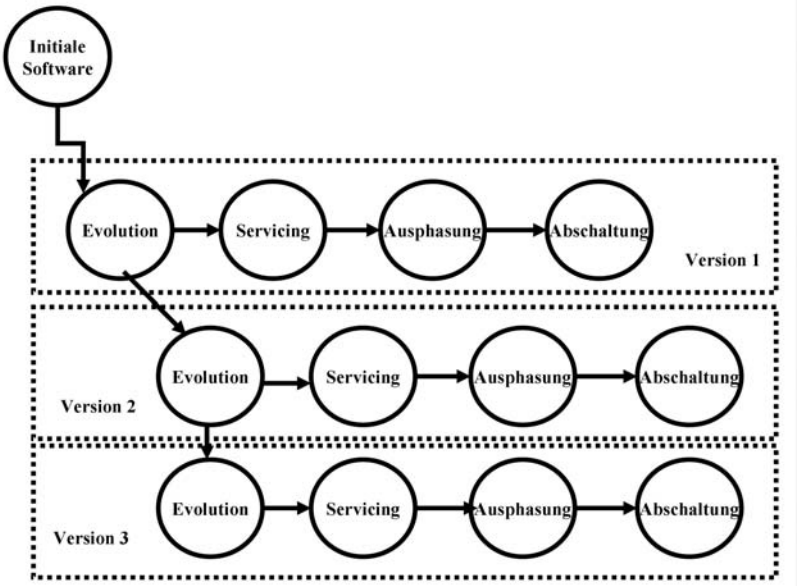

Abb. 3.2: Lebenszyklus mit mehreren Versionen

Abschaltungszustand

Die Abschaltung ist der finale Schritt. Die Software wird entfernt und steht nicht mehr zur Verfügung. In den meisten Fällen wird zwar versucht, die Daten zu retten, dies gelingt jedoch nicht immer. Häufig sind diese Daten nachträglich weder zugänglich noch sinnvoll interpretierbar. Viele Unternehmen unterschätzen den Wert der Daten innerhalb des abgeschalteten Legacysystems drastisch.

3.2 Versionierung

Eine Erweiterung des einfachen Stufenmodells des Lebenszyklus, s. Abb. 3.1, ist in Abb. 3.2 zu sehen. Hierbei werden einzelne Versionen freigegeben, welche sich aber nicht mehr evolvieren lassen, sondern nur noch in die Stufe Servicing übergehen. Die Evolution geschieht in diesem Modell durch den Übergang von einer Version auf die nächste.

Mit diesem Stufenmodell lässt sich der Lebenszyklus von COTS-Software recht gut beschreiben, wobei hier die Servicingstufe oft übergangen wird. Die Servicingstufe wird in der Regel dann übergangen, wenn die neue Version schneller produziert werden kann, als die Endanwender bereit sind, auf einen „Patch“ zu warten. Die Softwarehersteller ihrerseits haben keinerlei Interesse daran, dass zu viele Versionen der gleichen Software simultan auf dem

Markt sind – schließlich bindet jede im Einsatz befindliche Version einer Software nicht nur Zeit und Aufwand in der Entwicklung, sondern auch Kräfte im Support, im Produktmanagement und im Marketing. Von der Seite des COTS-Software-Herstellers aus wird Druck auf die Kunden ausgeübt, neuere Versionen zu erwerben und diese zu installieren, parallel dazu nimmt der Hersteller alte Versionen vom Markt.

Es stellt sich bei genauerer Betrachtung dieses Lebenszyklusmodells die Frage: Wie kann eine Architektur entwickelt werden, die zukünftige Änderungen verlässlich implementierbar macht? Genauer gefragt, welche Architektur ermöglicht es unvorhergesehene Änderungen in einer verträglichen Art und Weise zu verkraften? Die vorhersehbaren Änderungen sind in aller Regel unproblematisch, da sie sehr stark der noch nicht implementierten Funktionalität ähneln. Diese Frage wird, wenn auch nicht abschließend geklärt, auf dem Gebiet der *Enterprise Architekturen* erörtert.

Ein spezieller Fall tritt ein, wenn die Software nach den Gesichtspunkten einer Produktlinie, s. Kap. 9, entwickelt wird. In diesem Falle existieren mehrere parallele Versionen, zum einen der eigentliche Produktlinienkern, zum anderen die diversen Adaptionen. Es ist am sinnvollsten, die Adaption als Servicingstufe zu betrachten und die Evolution allein auf dem Produktlinienkern vorzunehmen, da sonst das betriebswirtschaftliche Argument für die Produktlinien ad absurdum geführt wird.

3.3 Operationen

Bisher wurde das Lebenszyklusmodell nur aus der „üblichen“ Richtung betrachtet, d.h. entlang des Weges der zunehmenden Entropie. Es sind jedoch auch andere Übergänge beobachtbar, siehe Abb. 3.3.

Aus der Menge der möglichen Übergänge hier einige, welche besonders hervortreten und speziell für Legacysoftware wichtig sind, s. Kap. 5 bzw. Kap. 10:

- Replacement – Das Replacement ist ein Ersatz für die Software; dabei macht ein Ersatz in der Regel erst dann Sinn, wenn mindestens die Servicingstufe erreicht ist. Während der Evolutionsstufe ist ein solches Replacement nur durch äußere Einflüsse, so beispielsweise Übernahmen, Fusionen oder Geschäftsaufgabe, erklärbar. Hier wird jedoch die bestehende Software durch eine neue initiale Software ersetzt. Die meisten stabilen Unternehmen warten mit einem solchen Schritt bis auf das Eintreten der Ausphasungsstufe.
- Reengineering – Ein groß angelegtes Reengineering ist der Versuch, aus einer Servicingstufe eine neue initiale Software zu erzeugen. Zwar gibt es kleinere Formen des Reengineerings, quasi partielles Reengineering; dies zählt jedoch stärker zur regulären Evolution.
- Transformation – Die beste Möglichkeit „alter“ Software neues Leben einzuhauchen ist die Transformation. Hierdurch wird eine Legacysoftware von

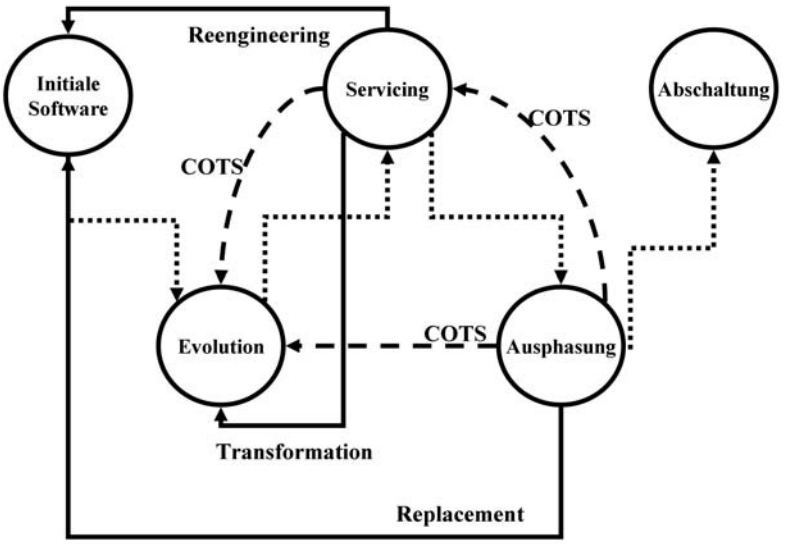

Abb. 3.3: Lebenszyklus mit komplexen Übergängen

der Stufe des Servicing auf die der Evolution zurückgestuft bzw. verbleibt in der Evolutionsstufe.

- COTS – Der vollständige Ersatz einer Software durch eine COTS-Software ist ein Sprung aus einer beliebigen Stufe in die Evolutions- oder Servicingstufe des COTS-Software-Herstellers. Üblicherweise wird dieser Übergang nicht in der Evolutionsstufe vorgenommen, sondern meist in den späteren Stufen. Allerdings sind auch Fälle bekannt, wo direkt von der initialen Softwarestufe in die Servicingstufe einer COTS-Software migriert wurde, was nur bei extremer Unzufriedenheit mit der ersten Version einer Software geschieht.

4

Softwareevolution

I have lived long enough: my way of life
Is fall'n into the sear, the yellow leaf;
And that which should accompany old age,
As honour, love, obedience, troops of friends,
I must not look to have; but, in their stead,
Curses, not loud but deep, mouth-honour, breath,
Which the poor heart would fain deny, and dare not
...

MacBeth,
William Shakespeare

Die Evolution eines Systems im Allgemeinen ist eines der komplexesten Gebiete der Systemtheorie und auch nur bedingt erforscht worden. Auf dem Gebiet der Softwareevolution wurde die Pionierarbeit von Lehman[1] geleistet. Da die Steuerung und Vorhersage der Softwareevolution eine der großen Herausforderungen der Softwareentwicklung darstellt, wird die Evolution leider oft auf die Frage der Steuerbarkeit[2] reduziert. Dies ist deutlich zu wenig, da ein großer Teil der Evolution durch die Produkte, ihre Verwendung und Vorgehensweisen ausgelöst wird. Es ist viel hilfreicher, die Evolution aus dem Blickwinkel der erzeugten Softwaresysteme zu betrachten.

Für die Evolution eines Softwaresystems gibt es durchaus verschiedene Sichtweisen. Zielführend ist es, zunächst die unterschiedlichen Evolutionsräume, welche in Zusammenhang mit der Software auftauchen, zu betrachten, s. Abb. 4.1. In dieser Darstellung ist die Evolution in zwei Dimensionen gruppiert. In der Horizontalen ist die zeitliche Achse, beginnend auf der linken Seite, mit dem Design, bis hin zur rechten Seite der Nutzung. Auf der Vertikalen ist der Ort innerhalb des Unternehmens aufgetragen, rangierend von „hart", d.h. direkt in der Hardware oder auch in der Software bis hin zu „weich", in den Köpfen der Mitarbeiter. Die verschiedenen Evolutionsräume lassen sich in fünf Bereiche einteilen:

- Softwareevolution – Die klassische Veränderung der Legacysoftware – es ist genau diese Softwareevolution, welche direkt im Legacysystem an sich erkennbar ist.

[1] Gl. 4.3 wurde nach Lehman benannt.
[2] Evolution as a management issue.

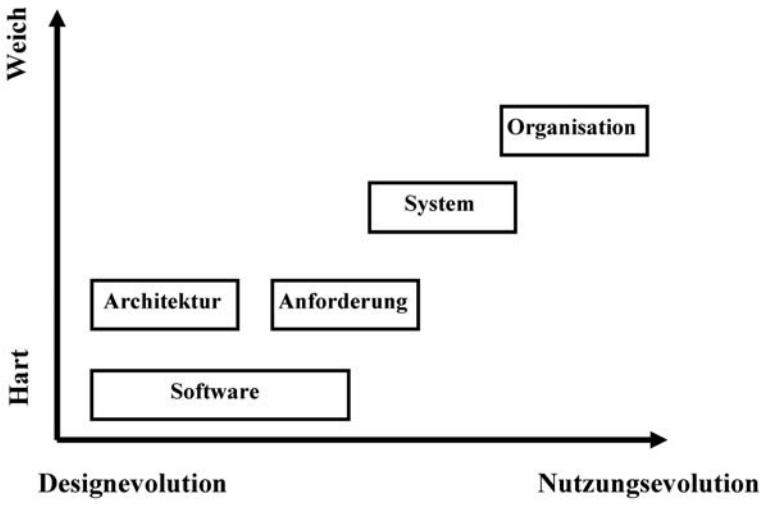

Abb. 4.1: Die Evolutionsräume

- Architekturevolution – Dieser Begriff bezeichnet die Veränderung der strukturellen Muster innerhalb des Legacysystems. Diese Form der Evolution ist nur durch eine abstrakte Betrachtung sichtbar zu machen. Ihre indirekten Auswirkungen sind jedoch sehr schnell in Form der Softwareevolution erkennbar.
- Anforderungsevolution – Die Veränderung der fachlichen Anforderungen an das jeweilige Legacysystem. Diese Veränderungen sind in den meisten Fällen der Auslöser für Maintenance und Migration. Hierunter kann auch eine Technologieevolution, im Sinne von technisch neuen Teilsystemen, Schnittstellen, etc. fallen.
- Systemevolution – Hierunter wird die Veränderung des Gesamtsystems verstanden. Das Gesamtsystem besteht aus den Geschäftsprozessen und dem Legacysystem.
- Organisationsevolution – Evolution des gesamten Unternehmens, seines Aufbaus, seiner Ablauforganisation. Da diese Ebene dem obersten Management unterstellt ist, finden hier oft Veränderungen statt. Diese „Reorganisationen“ werden aus zwei Gründen durchgeführt. Zum einen signalisiert eine Reorganisation, dass die Unternehmensleitung etwas arbeitet und zum anderen stammen die meisten oberen Führungskräfte aus dem organisatorischen Bereich. Ironischerweise haben die meisten Unternehmen jedoch ihre Probleme eine oder zwei Ebenen tiefer, s. Abb. 4.1.

Alle diese Formen bestimmen die Weiterentwicklung der Legacysoftware. Obwohl die Architekturevolution wichtig ist, sollte man sie doch nicht als Evolution, sondern sehr viel stärker als Revolution betrachten, denn die Architektur tendiert dazu, der stabilste Teil eines Legacysystems zu sein.[3]

Die Änderungen in der Architektur haben stets massiven Einfluss auf das ganze Legacysystem, folglich haben Veränderungen der Architektur revolutionären Charakter. Es wurde schon öfters versucht, die Anforderungsevolution durch Projektmanagementmethoden in den Griff zu bekommen. Alle Versuche mit Hilfe einer so genannten „Frozen Zone“, d.h. die Anforderungen werden während eines langen Zeitraumes „eingefroren“, haben sich als gescheitert herausgestellt. Es scheint besser zu sein, die Anforderungen zu kategorisieren, und zwar in die Kategorien stabil und instabil, und sich auf die Veränderungen der instabilen Teile einzustellen. In einigen Vorgehensmodellen ist dies Bestandteil der Philosophie des Vorgehensmodells, so z.B. bei den agilen Methodiken, s. Abschn. 11.4.

Traditionell wird die Architektur stets mit den frühesten Designphasen innerhalb einer Softwareentwicklung identifiziert. Diese Tatsache schlägt sich explizit in der Normenformulierung nieder. So definiert die IEEE 1471-2000:

> *... architecting contributes to the development, operation, and maintenance of a system from its initial concept until its retirement from use. As such, architecting is best understood in a life cycle context, not simply as a singular activity at one point in that life cycle.*

Die Veränderung, welche Software direkt jenseits der ersten Implementierungsphase, des initialen Zustandes, erfährt, bezeichnet man als Weiterentwicklung. Die beiden möglichen Veränderungstypen im Lebenszyklus der Software sind die Evolution und die Revolution. Unter Evolution wird das kontinuierliche, quasi infinitesimal inkrementelle Abändern und unter Revolution das diskontinuierlich spontane Verändern der Software verstanden. Eine typische Revolution in der Software wäre beispielsweise die Einführung einer Architektur. Solche Revolutionen führen immer zu Diskontinuitäten, welche sich dann anhand der Metriken, s. Kap. 2, nachweisen lassen. Genauer gesagt, evolutionäre Vorgänge bilden stetige Veränderungen der messbaren metrischen Größen, während revolutionäre zu nichtstetigen Veränderungen dieser Messgrößen führen, daher auch die Bezeichnung Diskontinuität, s. Abb. 4.2. Besonders stark sichtbar werden solche Diskontinuitäten bei der Betrachtung der Entropie. Diese Diskontinuitäten in der Entropie, aber auch im Maintainability Index, sind die eigentlichen Signaturen der Revolution.

Die Softwareevolution ist eng verknüpft mit dem Prozess der Maintenance. Den meisten Führungskräften in Unternehmen ist mittlerweile bewusst, dass Softwaremaintenance in Bezug auf den zeitlichen Aufwand und den Einsatz

[3] Die Hartnäckigkeit und Immobilität der Architektur ist geradezu ein Charakteristikum für Legacysysteme.

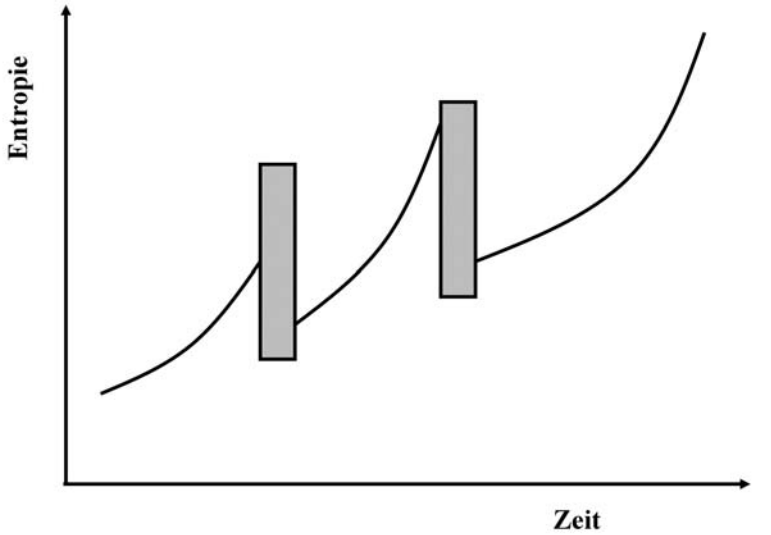

Abb. 4.2: Evolution und Revolution

anderer Ressourcen sehr teuer ist. Schätzungen über die Kosten von Maintenance rangieren zwischen 50 und über 80% der Kosten für den gesamten Lebenszyklus eines Softwareproduktes. Diese Aufwände müssen während der Maintenancephase, s. Kap. 7, erbracht werden. Die Maintenance ist jedoch nur ein Teil der Softwareevolution. Weitere Teile sind der Softwareausbau und das Lifecycle-Enablement. Unter dem Begriff Softwareausbau oder *Software Enhancement* wird die Erweiterung der Funktionalität gegenüber der ursprünglich konzipierten Funktionalität eines Softwareproduktes verstanden. Das Lifecycle-Enablement subsumiert alle Aktivitäten, welche notwendig sind, damit eine Legacysoftware überhaupt in der Lage ist, an der Maintenance oder dem Softwareausbau teilzunehmen, s. Abb. 4.3. Im Lebenszyklus, s. Kap. 3, ist das Enablement der Übergang von Servicing oder Ausphasung zurück in den Evolutionszustand, während das Enhancement in der Regel einen Übergang des Evolutionszustandes auf sich selbst darstellt.

4.1 Alterungsprozess

Jede Software, die sich jenseits der ersten Implementierung, jenseits des initialen Zustandes, befindet, altert! Die Puristen würden an dieser Stelle argumentieren, dass dies so nicht sein kann, da Software ja die technische Implementierung eines mathematischen Algorithmus sei und Algorithmen, wenn sie einmal als korrekt bewiesen und ordnungsgemäß implementiert wurden,

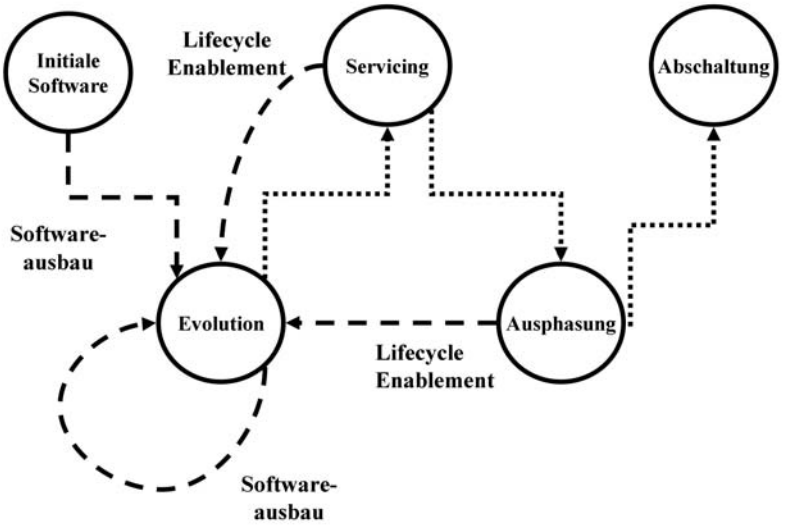

Abb. 4.3: Lebenszyklus mit Lifecycle-Enablement und Softwareausbau

auch in der Zukunft stets korrekt sind. So wahr diese Annahme ist, für die Betrachtung von Software ist sie leider irrelevant.

Die Legacysoftware zeigt, über die Zeit hinweg betrachtet, Eigenschaften, welche dem menschlichen Altern sehr nahe kommen. Legacysoftware ist mittlerweile zu einer Last für alle Unternehmen geworden, da immer größere Kosten durch den Softwarealterungsprozess entstehen. Obwohl das Phänomen als solches nicht neu ist, erlangt es immer größere wirtschaftliche Bedeutung, einfach durch die Tatsache, dass ein immer größerer Teil der vorhandenen Software in Unternehmen altert.

Die allzu menschliche Neigung zu glauben, dass neue Softwareprodukte nicht altern[4], ist falsch. Jedes Softwareprodukt, das eingesetzt wird, altert. Die Softwarealterung lässt sich am einfachsten an den Symptomen erkennen. Die beiden bekanntesten Symptome für Alterung sind: Zum Einen eine wach-

[4] Der Alterungsprozess ist unabhängig von der Programmiersprache. Daher sind Aussagen wie:

Unsere Software altert nicht, weil sie in Java[5] *geschrieben wurde ...*

oder

Unsere Software altert nicht, weil wir MDA[6] *einsetzen ...*

sinnlos und auf die Unkenntnis der Betroffenen zurückzuführen.

[5] ... oder C++ oder C# ...

[6] ... oder agile Verfahren oder RUP oder EUP oder J2EE oder .NET ...

sende Differenz zwischen der Erwartung der Endbenutzer und der tatsächliche Funktionalität der Software; zum Anderen eine sehr stark erhöhte „Zerbrechlichkeit[7]“ der Software. Was aber sind die Ursachen für die Alterung, und wie kann diesen begegnet werden?

Es gibt beim Softwarealterungsprozess zwei klar unterscheidbare Ursachen:

- Das Unvermögen, auf sich ändernde Anforderungen zu reagieren.
- Die Ergebnisse bzw. die Seiteneffekte von Änderungen, welche in der Software vorgenommen wurden.

In den meisten Fällen kommen beide Ursachen zusammen, was zu einer schnell degradierenden Spirale[8] für die Software führt.

Neben einer sich ändernden Geschäftswelt mit neuen oder veränderten Geschäftsprozessen hat sich in den letzten Jahrzehnten die Erwartungshaltung der Benutzer sehr stark verändert. Selbst Programme, welche vor über 30 Jahren entstanden sind und noch heute völlig ohne Veränderung lauffähig wären, würden von den Benutzern abgelehnt werden, da sie deren Erwartungshaltung nicht mehr befriedigen könnten. Zusätzlich zu der veränderten Erwartungshaltung ist oft die Hardware bzw. die entsprechende Betriebssystemplattform nicht mehr vorhanden. Neuere Software wird die alte immer ablösen, sobald die Vorteile – hierzu zählen auch die subjektiv wahrgenommenen Vorteile – die Kosten für Training und Migration übersteigen. Im Fall einer windowsbasierten Oberfläche im Vergleich zu einem charakter-orientierten Screendesign, überwiegen oft die subjektiven Vorzüge und beschleunigen das Altern der charakter-orientierten Oberfläche enorm. Die Hersteller von COTS-Software nutzen diese subjektiven Vorteile, verstärkt durch geschicktes Marketing, damit sie neue Releases oder Updates besser verkaufen können.

Obwohl die Veränderung der Software essentiell ist, um den Alterungsprozess zu stoppen oder zu verlangsamen, ist die zweite Quelle von Alterungsprozessen in Software die Summe der durchgeführten Veränderungen. Der Grund des Alterns liegt darin, dass eine Veränderung häufig falsch oder unsachgemäß durchgeführt wird. Der ursprüngliche Designer einer Applikation verfolgte bei der Implementierung der Software ein wohldefiniertes Konzept. Die nun pflegende Person ist meistens ignorant in Bezug auf das ursprüngliche Konzept. Die Folge sind Störungen und Fehlverhalten innerhalb der Software. Diese werden zumeist symptomatisch beseitigt, was zu einer stark zunehmenden Komplexität und einer Degradierung der Software führt. Nach vielen solchen Änderungen ist selbst der ursprüngliche Designer der Software nicht mehr in der Lage, die Applikation zu verstehen. Ist Software einmal in diesen Zustand geraten, steigen die Maintenancekosten exponentiell an. Veränderungen brauchen nun immer länger und haben ein immer höheres Risiko, Defekte in das System zu importieren. Dieser Vorgang wird zusätzlich noch dadurch verstärkt, dass die meisten Maintenanceprogrammierer ihre Veränderungen

[7] Volatilität

[8] Eine solche Spirale wird auch als negativer Feedback-Loop bezeichnet.

nicht dokumentieren. In dem Moment, wo die Defektbeseitigung zu dem Risiko führt, in großem Maße Folgedefekte zu produzieren, wird eine Legacysoftware ausgephast.

In dem Maße, wie Software altert, wächst sie auch in der Größe an.[9] Dieser Zuwachs resultiert primär aus dem Verhalten der Softwareentwickler. Für diese ist es einfacher, neuen Sourcecode in ein Programm aufzunehmen, anstatt bestehenden grundlegend abzuändern. Die Ursache dieses Verhaltens ist meistens, dass der zu ändernde Sourcecode weder gut verstanden noch wohldokumentiert wurde. In einer solchen Situation ist die Addition eines neuen Teils der Weg des geringsten Widerstandes. Mit wachsender Größe wird es immer schwieriger, Veränderungen durchzuführen. Am Ende des Lebenszyklus eines großen Softwarepakets kann die 10- bis 20-fache Menge der ursprünglichen Sourcecodezeilen vorhanden sein. Durch das starke Ansteigen der Menge der Sourcecodezeilen ist das Auffinden und systematische Abändern viel schwieriger geworden, auch die Wahrscheinlichkeit, dass der Sourcecode verstanden wird, sinkt drastisch ab. Außerdem lassen sich in den späten Phasen des Lebenszyklus Änderungen oft nicht mehr lokal, auf eine Stelle, beschränken, sondern breiten sich über den gesamten Sourcecode aus.

Ab einem gewissen Zeitpunkt entsteht ein so genanntes Maintenance Backlog, d.h. Anforderungen entstehen viel schneller, als die Software sich ändern kann. Als Reaktion hierauf wird in den meisten Fällen Entwicklungskapazität in die Maintenance transferiert, was umgekehrt zu einem so genannten Applikationsstau bei den Neuentwicklungen führt.

Neben der reinen Größe des Softwaresystems ändern sich auch Performanz und Verlässlichkeit drastisch. Beide Größen nehmen mit zunehmendem Alter der Software ab. Die zunehmende Codegröße impliziert einen höheren Hauptspeicherbedarf und ursprüngliche Designmaßnahmen für eine hohe Performanz werden durch lange Maintenance gestört. Die Folge ist ein drastischer Anstieg in den Antwortzeiten der Software, die ein Benutzer sofort wahrnimmt. Neben dem eigentlichen Alter der Software, verstärkt der in der Regel stetig wachsende Datenhaushalt dieses Phänomen. Die Verlässlichkeit der Software sinkt mit zunehmendem Alter, da jede Maintenance die Wahrscheinlichkeit birgt, Fehler in das System einzuführen. Im Laufe der Zeit wird sogar die Fehlerbeseitigung zu einem Risiko, da ab einem gewissen Alter das System so fragil ist, dass jedwede Fehlerbeseitigung im Mittel mehr als einen neuen Fehler hinzufügt. Ist ein solcher Zustand erreicht worden, so wächst die Zahl der Fehler explosionsartig an.

[9] Dasselbe gilt für die Zahl und Größe der Requirements.

4.2 Gesetze der Softwareevolution

Nach Lehman wird Software in drei Kategorien eingeteilt, s. Tab. 4.1:

- S-type: *Specifyable*, das entsprechende Problem kann formal geschlossen dargestellt werden, z.B. ein Algorithmus für die Wurzelfunktion $x \mapsto \sqrt{x}$. Diese Form der Software verändert sich nicht.
- E-type: *Embedded*, eine Software, welche in einem menschlichen Kontext eingebettet und inhärent evolutionär ist.
- P-type: *Problem Solving*, eine Software, zu der es keine geschlossen formale Darstellung gibt und die sich eventuell weiterentwickelt. In der Praxis ist jede P-type-Software entweder eine S-type- oder eine E-type-Software.

Eigentlich gelten diese Evolutionsgesetze nur für E-type-Software, aber da jede Legacysoftware eine E-type-Software ist, was durch den soziotechnischen Kontext wie auch das Alter offensichtlich wird, können wie hier die Gültigkeit annehmen.

Tab. 4.1: Eigenschaften von S-, P-, und E-type Software

Aspekt	**S-type**	**P-type**	**E-type**
Komplexität	klein	mittel – hoch	klein – hoch
Volumen	klein	groß	groß
Fokus	Problem	Problem	Prozess
Überdeckung	vollständig	partiell	kundengetrieben
Problemänderung	keine	keine	ja
äußerer Druck	nein	ja	ja
Druck auf Umgebung	nein	nein	ja

Die so genannten „Gesetze“ der Softwareevolution gehen auf Lehman zurück, welcher sie auf Grund der Beobachtungen bei der Entwicklung des Betriebssystems OS/360 aufstellte. Die sieben Gesetze der Softwareevolution sind:

I Kontinuierliche Veränderung
II Wachsende Komplexität
III Selbstregulierung
IV Erhaltung der organisatorischen Stabilität
V Erhaltung der Ähnlichkeit
VI Wachstum
VII Nachlassende Qualität

In ihrer Gesamtheit entsprechen sie Beobachtungen, welche jeder Einzelne in seinem Kontakt mit Softwareevolution schon einmal gemacht hat, allerdings ist es hilfreich, sie gemeinsam mit ihren Treibern und Interdependenzen

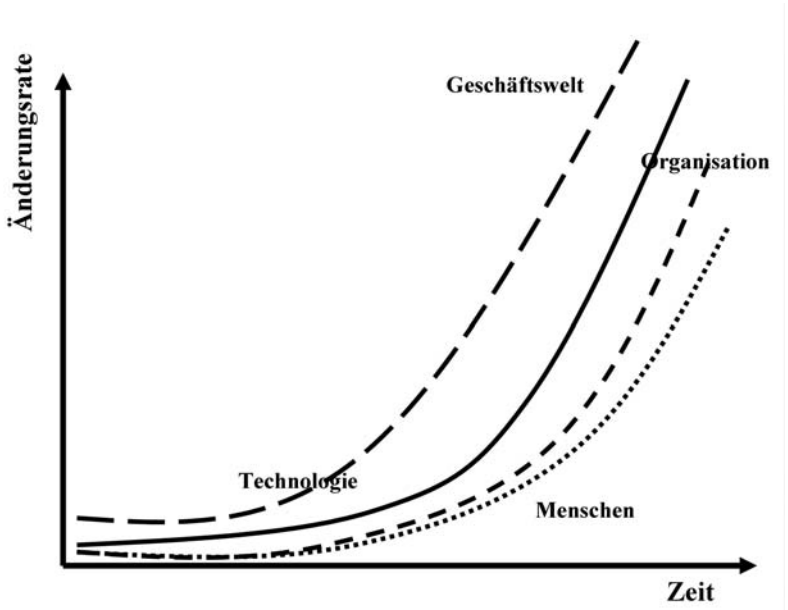

Abb. 4.4: Die verschiedenen Treiber hinter der Softwareevolution; sie verhalten sich in der Regel additiv

zu betrachten. Die meisten Menschen empfinden Technologieveränderungen als einen primären Treiber für die Veränderung von Software, doch das ist in der Regel nicht zutreffend. Technologie ändert sich auf kurze Sicht weniger, als erwartet wird, aber ihre langfristigen Auswirkungen sind sehr viel größer als angenommen. Typischerweise vergehen mehr als 5 Jahre, bis eine Technologie so weit allgemein akzeptiert ist, dass sie Eingang in die grundlegende Architektur findet. Viel häufiger werden die Softwareveränderungen durch Impulse aus der Geschäftswelt beeinflusst, s. Abb. 4.4.

4.3 Kontinuierliche Veränderung

I. Software, die genutzt wird, muss sich kontinuierlich anpassen, ansonsten wird sie sehr schnell nicht mehr nutzbar sein.

Dieses Gesetz formuliert eine universelle Erfahrung; unser gesamtes Leben wird von der Veränderung bestimmt. Im Fall der Software wird, im Gegensatz zu einem biologischen Organismus, der Veränderungsdruck auf die Software durch die Differenz zwischen der Anforderung der Domäne und der Eigenschaft der Software ausgelöst. Je größer diese Differenz, desto höher der Druck, die Software zu verändern.

Interessanterweise wird ein Teil des Druckes durch die Software selbst produziert. Jede neue Installation oder Version der Software ändert durch ihren

Einsatz die Domäne ab und erzeugt damit implizit eine Differenz zwischen der Anforderung der Domäne und der Software selbst. So negativ dies im ersten Moment klingt, dieser Druck ist wichtig, damit sich Software überhaupt entwickelt. Fehlt aus irgendwelchen Gründen dieser Druck, indem beispielsweise die Software „eingefroren“ wird, so wird die Software rapide obsolet.

Für den Fall von COTS-Software wird das Feedback aus der Domäne nur sehr indirekt erzeugt. Allerdings gibt es bei der COTS-Software noch einen zweiten Erzeuger von Veränderungsdruck: Die Konkurrenzprodukte. Das Vorhandensein von Konkurrenzprodukten führt auf Dauer zu einer „Ähnlichkeit“ s. Gesetz *V*, aller verwandten Produkte für eine bestimmte Domäne. Bei der Individualsoftware ist es meistens umgekehrt, hier liefert die Domäne den primären Druck und die möglichen COTS-Software-Produkte, welche in diesem Fall die Konkurrenz darstellen, wirken nur indirekt.

Dieses Gesetz resultiert sehr stark daraus, dass Veränderungen der Domäne einmal getroffene Annahmen ungültig machen, obwohl diese zu einem vergangenen Zeitpunkt durchaus korrekt gewesen sind. Die Endbenutzer verlangen nun eine Korrektur des von ihrer Warte „fehlerhaften“ Softwareprogramms. Es werden also die Auswirkungen des Feedbacks, bedingt durch den Einsatz durch dieses Gesetz, modelliert.

Interessanterweise haben die agilen Methoden, s. Abschn. 11.4, das erste Gesetz so verinnerlicht, dass dort die permanente Veränderung zum Prinzip erhoben wird. Auch andere Vorgehensmodelle, wie der Rational Unified Process, s. Abschn. 11.2, oder seine Erweiterung, der Enterprise Unified Process, s. Abschn. 11.3, versuchen, der Veränderung der Software Rechnung zu tragen.

4.4 Wachsende Komplexität

II. Die Komplexität einer Software wächst stetig an, außer es wird Arbeit investiert, um diese Komplexität zu reduzieren.

Diese Beobachtung entspricht dem zweiten Gesetz der Thermodynamik, dem der stetig wachsenden Entropie. Die Komplexität resultiert aus der stetig steigenden Zahl von Änderungen an der Software. Durch die Gesetze *I* und *VII* bedingt, werden Änderungen in das vorhandene System implementiert mit der Konsequenz, dass jede einzelne Änderung zu einer Erhöhung der Komplexität führt, da es eine gewisse Wahrscheinlichkeit für inkompatible und unstrukturierte Änderungen in der Software gibt.

Mit zunehmender Komplexität wird es immer schwerer, die einmal erreichte Komplexität auf dem Anfangsniveau zu Beginn der Veränderung zu halten, s. Gl. 4.3. Die einzige Maßnahme gegen dieses Anwachsen ist ein Refaktoring, welches die Komplexität absenken kann. Berücksichtigt man, dass im Allgemeinen die Ressourcen für eine Softwareentwicklung innerhalb eines Unternehmens limitiert sind, so kann daraus geschlossen werden, dass das Unternehmen sich nur für eine Strategie entscheiden kann, entweder die Strategie der Erweiterung oder die Strategie der Komplexitätsreduktion.

Wie entwickelt sich die Komplexität eines Softwaresystems im Laufe der Evolution? Hierfür ist es hilfreich, die Komplexitätsentwicklung einer großen Applikation zu betrachten.

Die Änderung der Komplexität, sprich das Komplexitätswachstum einer Applikation beim Hinzufügen eines neuen Moduls, ist infinitesimal gegeben durch:

$$\mathrm{d}\phi = \mathcal{K}(\phi)\mathrm{d}n, \tag{4.1}$$

wobei ϕ ein gegebenes Komplexitätsmaß, beispielsweise Card oder McCabe, s. Abschnitte 2.1.4 und 2.1.5, ist. Hierbei ist $\mathrm{d}\phi$ der Komplexitätszuwachs und $\mathcal{K}$ eine nichtverschwindende Funktion der Komplexität. Für kleine Systeme ergibt sich im Grenzfall:

$$\lim_{\phi \to 0} \mathcal{K}(\phi) = k_0 > 0,$$

so dass sich für kleine ϕ die Änderung des Komplexitätsmaßes zu

$$\mathrm{d}\phi \approx k_0 \mathrm{d}n,$$

ergibt.

Auf der anderen Seite kann gezeigt werden, dass das Funktional $\mathcal{K}$ sich für große ϕ wie ein Potenzgesetz verhalten muss, d.h.

$$\mathcal{K}(\phi \gg 1) \sim \phi^{\nu}. \tag{4.2}$$

mit einer nichtnegativen Konstanten ν. Das Lehmansche Gesetz besagt, dass die Zahl der Quellmodule eines Softwarepakets einer einfachen Differentialgleichung genügt:

$$\frac{\partial \psi}{\partial t} = c_1 \psi + \frac{c_2}{\psi^2}. \tag{4.3}$$

Diese Differentialgleichung korreliert die Zahl der Quellmodule ψ mit der Zeit. Näherungsweise lässt sich die Differentialgleichung durch

$$\lim_{\psi \to 0} \psi \approx \sqrt[3]{3c_2 t}, \tag{4.4}$$

$$\lim_{\psi \to \infty} \psi \approx \mathrm{e}^{c_1 t}, \tag{4.5}$$

lösen, was auch als das Turski-Lehman'sche Wachstumsmodell bekannt ist.

Das Lehmansche Gesetz, Gleichung 4.3, entspricht einer Pareto-Verteilung:

$$P(X > x) = \left(\frac{k}{x}\right)^{\alpha}, \tag{4.6}$$

mit $\alpha = 1/3$.

Es wird manchmal behauptet, dass bei agilen Vorgehensweisen, speziell beim eXtreme Programming, s. Abschn. 11.4.1, das Wachstum der Komplexität nur linear sei, im Gegensatz zu Gl. 4.2. Dies ist nur scheinbar so, da hier

Tab. 4.2: Das Wachstum von Windows

System	Lines of Code
PC DOS[10] 1.0	4.000
NT 3.1	6.000.000
Windows 98	18.000.000
Windows 2000	35.000.000
Windows XP	45.000.000

verschiedene Zeitskalen miteinander verglichen werden. Beim eXtreme Programming wird ein fester Aufwand pro Zeiteinheit geleistet, welches zu einer praktisch linearen Steigerung der Komplexität führt, da der Aufwand näherungsweise linear mit der Komplexität verläuft und der Aufwand konstant ist. Sinnvollerweise kann man Software aus agilen Verfahren mit der Software aus anderen Verfahren nur dann vergleichen, wenn die Zeitachse durch die Releases definiert wird und nicht durch Verstreichen der kalendarischen Zeit.

Ein interessantes Beispiel für wachsende Komplexität ist, obwohl es sich hier nicht um die Komplexitätsentwicklung eines einzelnen Produktes handelt, die Größe des Betriebssystems Windows, s. Tab. 4.2. Der Faktor an Codezeilen zwischen DOS und Windows XP liegt bei über 10.000. Zwischen Windows XP und NT 3.1 beträgt er immerhin noch 7,5!

4.5 Entropie

Damit zwischen den einzelnen Legacysystemen und der Entwicklung ihrer Entropie eine gewisse Vergleichbarkeit möglich ist, empfiehlt es sich, als Zeitachse nicht die kalendarische Zeit, sondern die jeweilige Releasenummer der Legacysoftware zu nehmen. Dies führt zu einer Vergleichbarkeit der Releaseabfolge und einer Art Normierung auf der Zeitachse.

Wird die Entropieentwicklung zweier verschiedener Produkte auf diese Art und Weise miteinander verglichen, so lässt sich ein *managed* Produkt von einem *unmanaged* Produkt unterscheiden. Bei einer schlecht gesteuerten Entwicklung steigt die Entropie faktisch permanent an und führt zu immer größerem Chaos.

Da jede Änderung an dem Gesamtsystem eine Änderung der Entropie zur Folge hat und in erster Näherung eine Änderung sich in einem proportionalen Wechsel der Entropie niederschlägt, lässt sich die Entropieänderung, entsprechend Gl. 4.1, durch

$$\Delta S \sim S,$$

beschreiben. Die Folge ist, dass sich für die Entropie in erster Näherung eine Differentialgleichung der Form

[10] Im Fall von PC DOS handelt es sich um Assemblerzeilen.

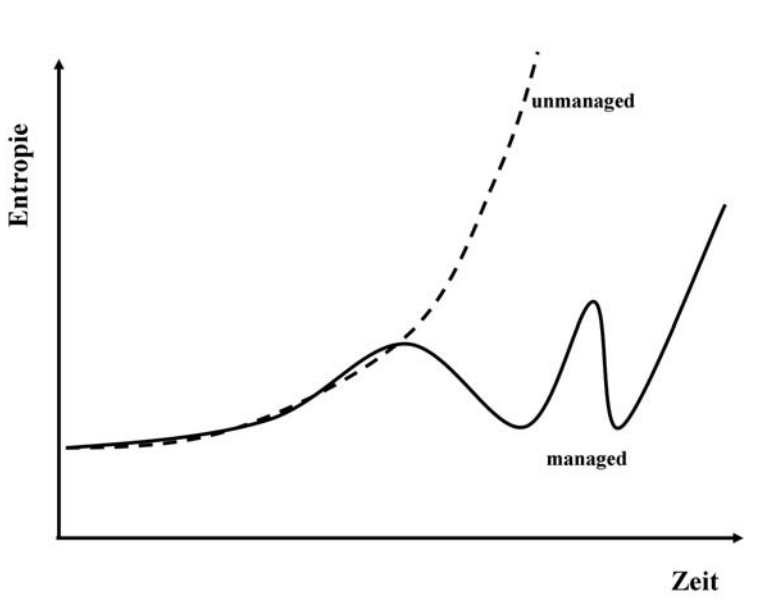

Abb. 4.5: *Managed* vs. *unmanaged* Entropie

$$\frac{\partial S}{\partial t} = \alpha S,$$

ergibt. Mit der Folge, dass in diesem vereinfachten Modell die zeitliche Entwicklung der Entropie als

$$S = S_0 \mathrm{e}^{\alpha t}, \tag{4.7}$$

formuliert werden kann.

Die Auswirkungen dieser einfachen Entropieentwicklung, s. Abb. 4.6, sind drastisch. Jeder noch so kleine Unterschied bei der Startentropie zum Zeitpunkt t_0 resultiert in einem drastischen Unterschied in der Endentropie, oder anders formuliert, die Grenze der sinnvollen Maintenance wird sehr viel früher erreicht, wenn mit einer höheren Entropie ab initio gestartet wird. Durch das exponentielle Wachstum der Entropie kann dieser zeitliche Unterschied recht hoch sein.

Interessant ist es hierbei auch, die Entwicklung der Steigerungsrate der Entropie zu betrachten, s. Abb. 4.7. Hierbei fällt auf, dass sich das Entropieproblem im Laufe der Zeit verschärft hat. Das heißt, dass heutige Applikationen sehr viel schneller altern als die Applikationen in den achtziger Jahren, da die rapide Veränderung der Geschäftswelt ihre direkten Spuren in der Software hinterlässt.

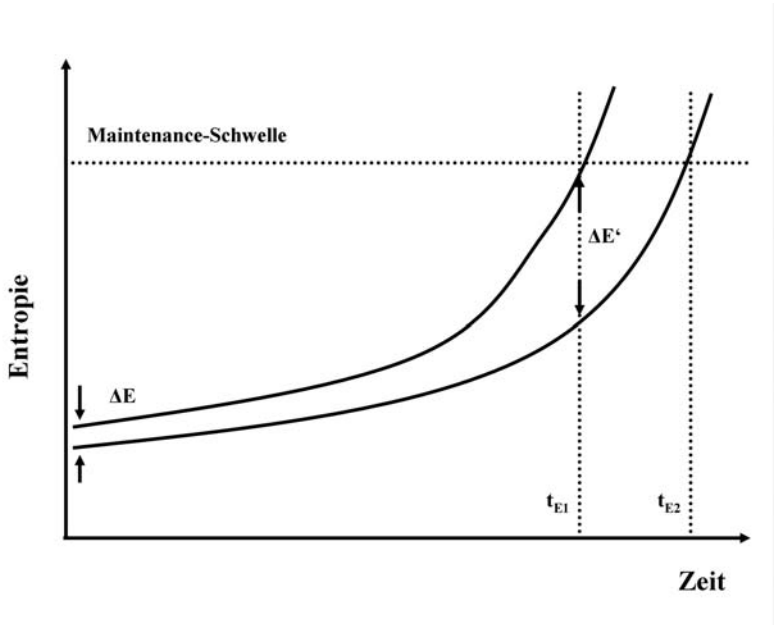

Abb. 4.6: Vereinfachtes Entropiemodell

4.6 Selbstregulierung

III. Die Evolution von Software ist selbstregulierend mit Produkt- und Prozessmetriken, welche nahe an einer Normalverteilung[11] *liegen.*

Die Implementierung von Software geschieht durch ein technisches Team, welches Teil eines Unternehmens ist. Die Ziele und Interessen des Gesamtunternehmens gehen weit über die konkrete Software hinaus, von daher existieren innerhalb des Unternehmens eine Reihe von Regularien und Kontrollmechanismen um sicherzustellen, dass die Unternehmensziele auf allen Ebenen erreicht werden. Eine große Zahl dieser Mechanismen wirken auf die Software als Treiber oder als stabilisierende Kräfte. Da die Zahl dieser einzelnen Kräfte relativ hoch ist, kann die Entwicklung der Software als ein Subsystem betrachtet werden, welches in ein großes System eingebettet ist. Dieses Gesamtsystem übt nun durch eine große Zahl von quasi unabhängigen Entscheidungen einen Druck auf das System aus, welcher einer Normalverteilung ähnlich ist. Wenn

[11] Eine Normalverteilung ist eine Wahrscheinlichkeitsverteilung $p(x)$, welche dem Gesetz:

$$p(x) = \frac{1}{\int\limits_{-\infty}^{\infty} \mathrm{e}^{-\alpha x^2}\, dx} \mathrm{e}^{-\alpha(x-x_0)^2}$$

genügt.

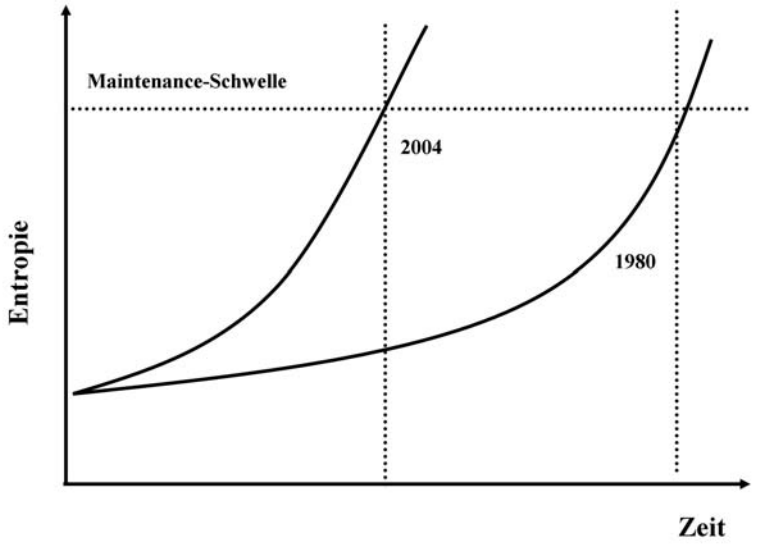

Abb. 4.7: Entwicklung der Entropiesteigerung

dieser Druck eine Zeit lang auf die Entwicklung und damit auch auf die Software angewandt wird, ist die Eigendynamik so stark, dass ein Equilibrium gefunden wird. Umgekehrt ändert sich ein komplexes System erst dann, wenn es weit von seinem Gleichgewichtspunkt entfernt wird. Wurde das System so weit von seinem Equilibrium entfernt, so kann es sich jetzt einen neuen Gleichgewichtszustand suchen und diesen einnehmen. In der Komplexitätstheorie wird das Vorhandensein mehrerer Gleichgewichtspunkte als Bifurkation bezeichnet. Für Legacysysteme ist es oft charakteristisch, dass sie sich in einem sehr engen Lösungsraum befinden und diesen nicht mehr verlassen können.

Aus dieser Gesetzmäßigkeit lässt sich auch ableiten, dass der Wegfall oder die Addition eines einzelnen Mitarbeiters, inklusive des Projektleiters, langfristig betrachtet keine großen Auswirkungen auf die Gesamtsoftware hat.

4.7 Erhaltung der organisatorischen Stabilität

IV. Die mittlere effektive Aktivitätsrate, welche für eine evolvierende Software angewandt wird, bleibt während der Lebensdauer der Software konstant.

Oder anders formuliert: Die mittlere Leistung $\overline{P}$, definiert aus der Arbeit pro Zeiteinheit, ist invariant:

$$\frac{\partial}{\partial t}\overline{P} = 0.$$

Diese Beobachtung ist zunächst verwirrend, da man geneigt ist anzunehmen, dass der Aufwand, welcher in Software investiert wird, primär ein Resultat von Managemententscheidungen ist. Dies ist in engen Grenzen auch so, allerdings sind die tatsächlichen Kostentreiber in Unternehmen meistens externer Natur: Verfügbarkeit von qualifiziertem Personal und Ähnliches. Auf der anderen Seite ist ein großer Teil des Aufwandes durch das dritte Gesetz vorgegeben: Um die Trägheit des Equilibriumpunktes zu überwinden, ist sehr viel Aufwand notwendig. Die Folge dieser hohen Trägheit ist eine quasi konstante effektive Aufwandsrate.

Eine praktische Folge dieses Gesetzes ist die Beobachtung, dass in schwierigen Projekten die Hinzunahme von zusätzlichen Projektmitarbeitern zu einer Reduktion der Produktivität führt, da der Equilibriumspunkt verlassen wurde. In manchen Fällen ist daher eine explizite Reduktion der Anzahl der Projektmitarbeiter zu empfehlen.

Zusammenfassend kann gesagt werden, dass der Aufwand durch eine solche Vielzahl von Quellen beeinflusst wird, dass Managemententscheidungen kaum noch ins Gewicht fallen.[12]

4.8 Erhaltung der Ähnlichkeit

V. Die Inhalte von aufeinander folgenden Releases innerhalb einer Software sind statistisch konstant.

Einer der wichtigsten Faktoren, welche die Veränderung von Software beeinflussen, ist das Wissen aller Beteiligten über die Zielsetzung hinter der Software. Das Produktmanagement tendiert dazu, in jedem der einzelnen Releases besonders viel an neuer Funktionalität unterzubringen, denn es möchte dem Kunden einen „Mehrwert“ durch das neue Release suggerieren. Dieses Vorgehen alleine würde zu einem Anwachsen des Inhaltes von Release zu Release führen. Die entgegengesetzte Kraft ist die Notwendigkeit, dass alle Beteiligten die große Menge auch verstehen und umsetzen müssen. Der Fortschritt und die Dauer ist auch beeinflusst durch die Menge an Information, welche gewonnen werden muss. Das Verhältnis zwischen Informationsmenge und Aufwand zur Gewinnung der Information ist nicht linear, sondern der Aufwand zur Informationsgewinnung scheint sich ab einer gewissen Größenordnung so drastisch zu erhöhen – vermutlich wird in dieser Konstellation eine Verhaltensänderung ausgelöst –, dass von diesem Moment an die Gesamtmenge an zu gewinnender Information unüberwindbar erscheint. Die Folge dieser beiden auf jedes Release wirkenden Kräfte ist eine statistische Konstanz bezüglich der Inhaltsmenge pro Release.

[12] Überspitzt kann dies auch anders formuliert werden:

Unternehmen sind nicht wegen, sondern trotz ihres Managements erfolgreich.

4.9 Wachstum

VI. Die funktionalen Inhalte einer Software müssen stetig anwachsen, um der Endbenutzererwartung auf Dauer gerecht zu werden.

Dieses Gesetz sieht dem ersten Gesetz über die kontinuierliche Veränderung sehr ähnlich, ist aber nicht identisch. Wenn eine neue Software eingeführt wird, existiert immer eine Differenz zwischen der implementierten Funktionalität und der tatsächlich gewünschten. Die Gründe für diese Unvollständigkeit sind divers, sie rangieren von Budget über Zeit bis hin zu der prinzipiellen Fähigkeit, eine Domäne zu verstehen. Folglicherweise gibt es eine Differenz zwischen den Endbenutzererwartungen und der tatsächlichen Software. In diesem Falle jedoch wird mit zunehmender Dauer der Ruf nach der Implementierung dieser fehlenden Funktionalität stärker, was über die Feedbackmechanismen zur Erweiterung der vorhandenen Funktionalität führt.

Mathematisch gesehen kann dieses Gesetz formuliert werden durch

$$\frac{\mathrm{d}V}{\mathrm{d}t} = \mathcal{M}(t), \tag{4.8}$$

wobei V das Volumen in einer geeigneten Form misst und $\mathcal{M}(t)$ eine positive, monoton steigende Funktion der Zeit ist.

4.10 Nachlassende Qualität

VII. Die Qualität der Software wird als nachlassend empfunden, solange keine massiven Anstrengungen zur Adaption vorgenommen werden.

Eine Software kann nie alle Anforderungen treffen, dafür ist die Zahl der Anforderungen zu groß. Während des Evolutions- und des Entwicklungsprozesses werden eine Reihe dieser Anforderungen als Rahmenbedingungen oder Systemgrenzen angenommen, s. Abb. 4.8. Aber da die reale Welt sich permanent verändert, steigt die Unschärfe zwischen der möglichen Gesamtmenge von Anforderungen an, während die der implementierten Anforderungen auf Grund der Systemgrenzen konstant bleibt. Die Unschärfe wächst also! Der Endbenutzer empfindet die so anwachsende Unschärfe als einen Verlust an Qualität. Je größer diese Unschärfe wird, desto schwieriger ist es für den Endbenutzer, das Verhalten der Software erwartungskonform vorauszusagen.

Im Umkehrschluss gilt: Eine aktive Behandlung und Reduktion dieser Unschärfe erhöht die wahrgenommene Qualität. Aus psychologischer Sicht ist das Phänomen recht einfach zu erklären: Die zunehmende Gewöhnung an eine vorhandene Funktionalität führt dazu, dass diese als gegeben angenommen wird mit der Folge, dass das Anspruchsdenken zunimmt. Da die Software diesen erhöhten Ansprüchen nicht mehr genügen kann, wird sie als qualitativ schlechter wahrgenommen.

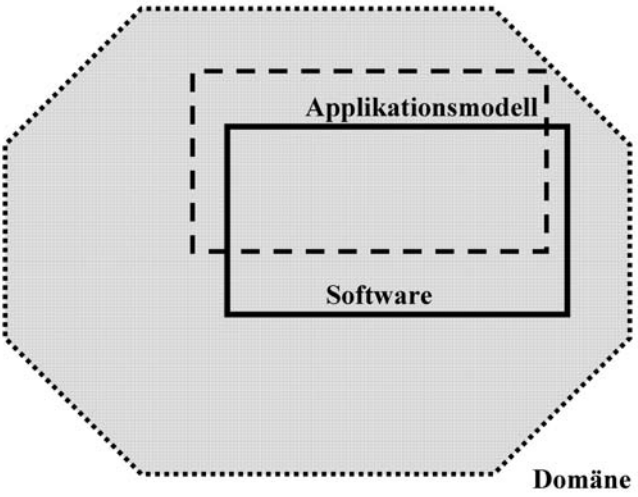

Abb. 4.8: Die verschiedenen Systemgrenzen: Domäne, Applikationsmodell und tatsächliche Software

4.11 Volatilität

Der Volatilitätsindex, als Maß für die Anfälligkeit eines Softwaresystems, zeigt eine Reihe von interessanten Eigenschaften. Generell lässt sich konstatieren, dass, wenn ein bestimmter Wert des Volatilitätsindexes erreicht wird, ein Austausch der Software dringend notwendig ist, da die Software jenseits dieses Punktes schon so sehr gealtert bzw. komplexer geworden ist, dass die Kosten der Maintenance exorbitant werden.

Doch wie sieht die Entwicklung der Volatilität während der Softwareevolution aus? Zunächst sei darauf hingewiesen, dass die gewählte Implementierungssprache einen hohen Einfluss auf den Volatilitätsindex als solchen besitzt, s. Abb. 4.9.

Programmiersprachen mit einem niedrigen Grad an semantischer Information, beispielsweise COBOL, haben einen höheren Volatilitätsindex als Sprachen mit einem hohen Grad an Semantik, beispielsweise relationale Datenbanken oder Java, da diese Systeme stets Hilfsmittel enthalten, um Änderungen in der Benutzerschnittstelle wie auch der physischen Struktur sowie „deep structural changes“ vorzunehmen. Die Folge der Gleichzeitigkeit der Änderung führt in Gleichung 2.36 zwar zu einem größeren Zähler, dieser wird jedoch durch einen größeren Nenner kompensiert. Die Folge ist, dass der Volatilitätsindex länger klein bleibt.

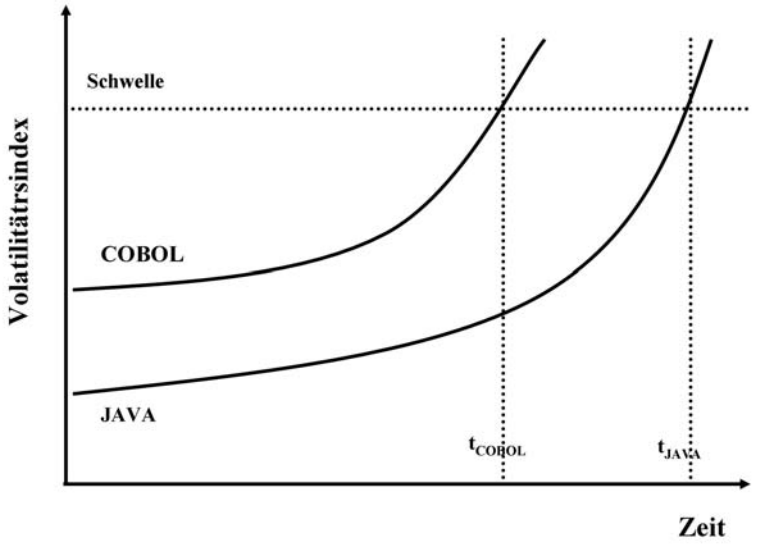

Abb. 4.9: Volatilität für verschiedene Sprachen

Die allgemeine Volatilitätskurve eines Softwaresystems zeigt eine Badewannenkurve auf, s. Abb. 4.10. Der erste Bereich ist der initiale Zustand, welcher häufig von größeren strukturellen Veränderungen begleitet wird, daher der hohe Volatilitätsindex. Nach der Einführung beginnt die klassische Evolutionsphase, in der die Software altert. Am Ende der Evolutionsphase wird die Software obsolet und muss letztendlich in der Revolutionsphase abgelöst werden.

Detaillierte Untersuchungen über das zeitliche Verhalten der Volatilität nutzen dabei nicht nur den Volatilitätsindex, Gl. 2.36, an sich, sondern auch sein zeitliches Verhalten.

Ein einfacher Amplitudenvergleich des Volatilitätsindexes alleine würde bei unterschiedlicher Software notwendigerweise zu sehr unterschiedlichen Ergebnissen führen. Zwei weitere Größen sind für den Vergleich verschiedener Softwaresysteme interessant. Das eine Maß ist die Periodizität, sie bestimmt, in welchem Rhythmus der Volatilitätsindex schwankt. Die Periodizität ist definiert durch:

$$\tau(t) = \frac{1}{t}\frac{1}{N}\sum_{j=1}^{N}\Delta_j t, \tag{4.9}$$

wobei hier die verstrichene Zeit $\Delta_j t$ seit der letzten Änderung gemessen wird und N die Menge aller Änderungen in einem Zeitintervall angibt. Die Division durch das Alter der Software normalisiert die Periodizität.

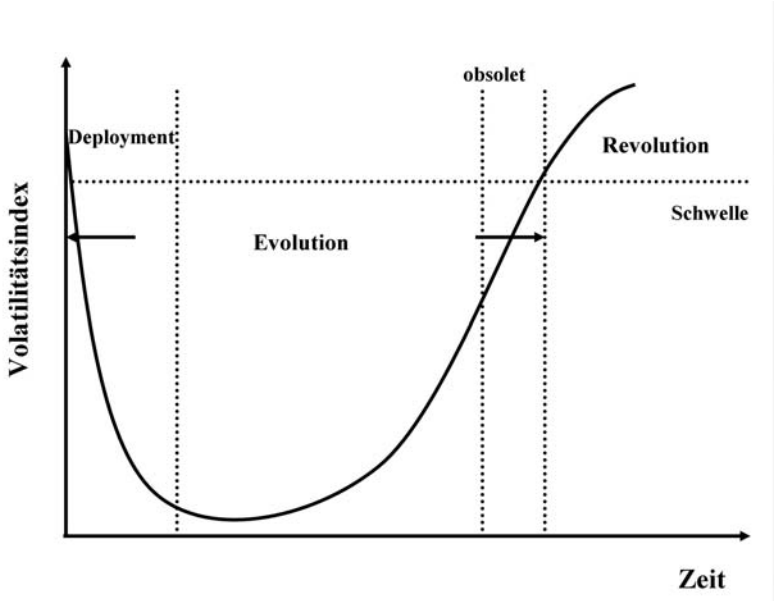

Abb. 4.10: Volatilitätsentwicklung und die Phasen Revolution und Evolution

Das andere Maß ist die Dispersion. Unter Dispersion versteht man die Zerstreuung eines Wellenpaketes in kleinere Pakete, das „Auseinanderlaufen“:

$$\xi = \frac{1}{t^2} \left\langle (\Delta t)^2 \right\rangle - \left\langle \Delta t \right\rangle^2 . \tag{4.10}$$

Die so definierte Dispersion misst die Varianz der Zeit, mit der die Veränderungsereignisse eintreten. Die Dispersion misst somit die Abweichung von einer vorgegebenen Periodizität.

Empirische Untersuchungen mit Hilfe der Volatilität, ihrer Periodizität und der Dispersion haben zu der Beobachtung geführt, dass sich Software in vier Kategorien einteilen lässt:

- Gruppe 1: Die Volatilität in dieser Gruppe ist sehr lange niedrig, ca. 90% der Lebensdauer, und steigt am Ende drastisch an.
- Gruppe 2: Für ca. 40-50% der Lebensdauer ist die Volatilität quasi konstant, die restliche Zeit wird von einer höheren Volatilität charakterisiert.
- Gruppe 3: Nach einer kurzen Anfangsstabilität, ca. 10% der Lebensdauer, folgt eine sehr viel höhere Volatilität für den Rest der Zeit.
- Gruppe 4: Diese Gruppe hat die gesamte Zeit eine hohe Volatilität.

Diese unterschiedlichen Muster der Volatilitätsentwicklung deuten auf unterschiedliche Kräfte hin, welche zu diesen Mustern führen. Obwohl sich die meisten Softwaresysteme nach einem der vier Gruppenmuster entwickeln, ist das zugrunde liegende Modell dieser empirischen Beobachtung noch unklar.

4.12 Konsequenzen aus den Evolutionsgesetzen

Bisher wurden die Evolutionsgesetze in ihrer Form als Gesetze rein beschreibend betrachtet, ihre Existenz hat jedoch eine Reihe von praktischen Implikationen, welche durchaus direkte Auswirkungen auf die weitere Entwicklung von Software und speziell von Legacysoftware haben:

1 Kontinuierliche Veränderung:
- Eine verständliche und gut strukturierte Dokumentation muss geschaffen und während der Maintenance auch gepflegt werden, damit der Verfall an Wissen und Architektur möglichst lange hinausgezögert werden kann. Durch ein gutes mentales Modell bei den Mitarbeitern der Maintenance kann der strukturell negative Einfluss von konsekutiven Änderungen einige Zeit im Griff gehalten werden. Dies zögert den Verfall der Software zwar hinaus, kann ihn jedoch letztendlich nicht verhindern.
- Bei einer Änderung müssen alle Aspekte, auch die verworfenen Designentscheidungen, mit ihrer Begründung dokumentiert werden, da nur so, langfristig gesehen, die vorhandene Architektur sinngemäß erhalten bleiben kann.
- Es müssen bewusste Anstrengungen unternommen werden, ein Refaktoring anzugehen, um so die Komplexität zu reduzieren.
- Design über Interfaces tendiert dazu, Teile der Software von anderen zu isolieren und so, zumindest kurzfristig, das Komplexitätswachstum unter Kontrolle zu halten.
- Die Summe der Annahmen, welche über die Domäne durch die Software impliziert werden, müssen bei jedem neuen Release explizit auf ihre Angemessenheit überprüft werden.
- Die Zahl der Änderungen in einem Release sollte klein gehalten werden, da sonst die Komplexität stark nichtlinear anwächst.
- Eine große Zahl von Änderungen in einem Release hat direkte negative Konsequenzen für das Folgerelease, da jetzt von einer sehr viel höheren Komplexität aus gestartet wird.
- Es scheint am günstigsten zu sein, innerhalb der Releases solche Releases, welche sich mit der korrektiven Maintenance befassen, von anderen Releases zu entkoppeln, welche adaptiv ausgelegt sind. Diese Praxis wird heute schon von einer Reihe von COTS-Software-Herstellern aktiv eingesetzt.
- Die Nutzung von metrischer Information zur Planung von Releases.

2 Wachsende Komplexität:
- Die aktive Kontrolle der Komplexität und der Einsatz von Refaktoring müssen ein integraler Bestandteil des Maintenanceprozesses werden. Der sofortige Effekt dieser Aktivitäten wird sehr klein sein, der langfristige jedoch immens. Einer der Gründe für eine Transformation

Tab. 4.3: Risiko eines Releases als Funktion der Metrik M

Risiko	**Wachstum der Metrik**
sehr hoch	$M' \geq \overline{M'} + 2\sigma$
hoch	$\overline{M'} < M' < \overline{M'} + 2\sigma$
niedrig	$M' < \overline{M'}$

einer Legacysoftware liegt in der Reduktion der Komplexität, um dann, in nachfolgender Zeit, eine evolutionsfähige Legacysoftware zu besitzen; daher wird ein solches Vorgehen auch als Lifecycle-Enablement bezeichnet.

- Eine Kontrolle der Komplexität ist nur möglich, wenn diese auch explizit via Metriken gemessen wird; besonders bieten sich hier der Maintainability Index und die Entropie an.
- Die Nutzung von Refaktoring führt zu einem Dilemma, da in den meisten Fällen ein konstantes Budget für die Maintenance eines Legacysystems zur Verfügung steht. Die Folge hinter der Konzentration auf das Refaktoring ist, dass weniger Ressourcen für eine fachliche Weiterentwicklung zur Verfügung stehen, mit der Konsequenz, dass die Unschärfe gegenüber der Domäne stärker zunimmt. Insofern kann Refaktoring kurzfristig betrachtet kontraproduktiv sein.

3 Selbstregulierung:

- Die metrischen Daten über die Software, speziell die Größe und Komplexität der Software, können als Feedbackmechanismen für die Selbstregulierung genutzt werden.
- Solche Metriken machen als steuernde Elemente nur dann Sinn, wenn sie gegenüber einem Eichpunkt gemessen werden können. Von daher ist ein sinnvolles Feedback nur durch die Festlegung von eindeutig definierten „Baselines“ möglich.
- Jedes neue Release sollte zur Kalibrierung der Parameter genutzt werden.
- Mit Hilfe der metrischen Größen kann ein Release geplant oder auch besser gesichert werden:
  Sei $\overline{M'}$ das durchschnittliche Wachstum der Metrik M, d.h.

$$\overline{M'} = \frac{1}{N} \sum_{i=1}^{N} M(t_i) - M(t_{i-1}),$$

  wobei i die Releases durchzählt und σ die Standardabweichung von $\overline{M'}$ ist, so lässt sich das Risiko eines Releases wie in Tabelle 4.3 klassifizieren.
  Ist das Risiko sehr hoch, so ist in der Regel zu empfehlen, zuerst eine Maßnahme einzuleiten, welche die Metrik absenkt, gefolgt von der

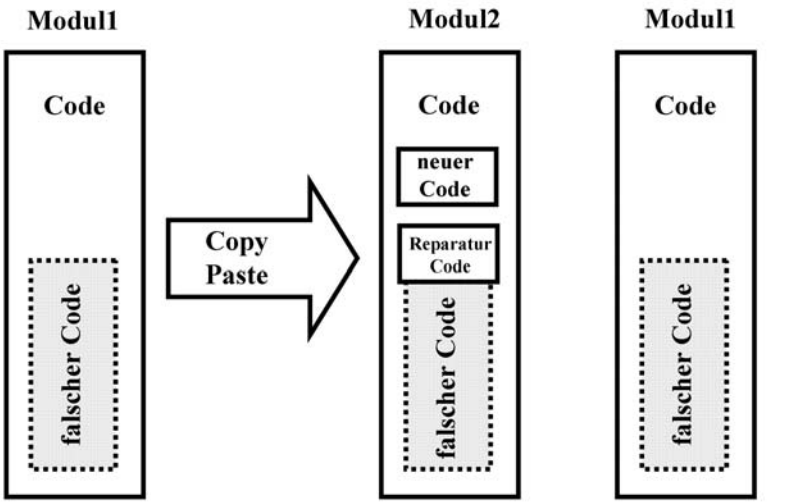

Abb. 4.11: Das Bloating durch Copy & Paste

Änderung auf nun etwas niedrigerem Niveau. Diese Empfehlung findet sich auch bei der Betrachtung der Entropie wieder, s. Abb. 4.15.

- Bei sehr hohen Risiken sollte immer versucht werden, die zu implementierende Funktionalität über mehrere Releases zu verteilen.

4.13 Bloating

Die direkt wahrnehmbare Folge und bis zu einem gewissen Grad auch Ursache der stetig wachsenden Komplexität ist das Bloating oder auch Aufblähen des Sourcecodes, manchmal auch als Lava Flow Pattern bezeichnet. So wie Lava beim Erkalten unveränderbare starre Muster bildet, so werden durch Copy & Paste diese eingefrorenen Strukturen weiterverteilt. Nach heutigen Schätzungen enthalten die meisten Legacysysteme etwa zehnmal soviel Sourcecode wie eigentlich notwendig ist, um damit die fachliche Aufgabe lösen zu können. Die typischen Ursachen für das Bloating sind:

- zu viel Funktionalität
- mangelnde Funktionalität
- persönliche Vorlieben der jeweiligen Softwareentwickler
- generisches Design

Während die ersten drei Ursachen in ihren Auswirkungen offensichtlich negativ sind, ist die letzte, das generische Design, zunächst überraschend. Aber

generisches Design muss ja für alle Eventualitäten gerüstet sein mit der Folge, dass es jede Menge IF-THEN-ELSE, Konfigurationsteile und willkürliche Defaults enthält. Ein weiteres Charakteristikum für generisches Design ist, dass das aufgerufene Programm häufig Überprüfungen über die Aufruffolge oder Ähnliches machen muss, da a priori nicht klar ist, in welchem Kontext das Programm aufgerufen wurde. Um sich gegen Fehler aus der Aufrufreihenfolge zu sichern, programmiert der Softwareentwickler so genannten Guard-Code, dies ist Sourcecode, welcher explizit die Übergabe überprüft, mit der Konsequenz von noch mehr „unnützem“ Sourcecode. Obwohl dieser Guard-Code zu einem gegebenen Zeitpunkt sinnvoll und notwendig war, verliert er oft über den Lebenszyklus eines Unterprogramms hinweg an Sinnhaftigkeit und wird im Endeffekt obsolet. Diese Mechanismen haben ein hohes Maß an Bloating zur Folge.

Unglücklicherweise erzeugt Bloating wiederum Bloating, s. Gl. 4.3 und Abb. 4.11, dies wird durch das Verhalten der Softwareentwickler begünstigt. Die gängigste Form der „Wiederverwendung“ ist das Copy&Paste. Hierbei wird eine alte Codesequenz dupliziert und an die neue Funktionalität angepasst, mit der Folge, dass falscher Sourcecode mitdupliziert wird. Da dieser „fehlerhafte“ Code als störend empfunden wird, wird er partiell umgangen, faktisch lahmgelegt, und es werden neue Zeilen mit neuer Funktionalität im neuen Modul hinzugefügt. Die Konsequenz dieses systematischen Einsatzes von Work-Arounds ist eine Aufblähung des Sourcecodes. Je mehr Zeilen er hat, desto schwieriger ist er im Copy&Paste-Mechanismus zu beherrschen und desto mehr nachfolgender Sourcecode entsteht. Dieses selbstverstärkende Phänomen führt zu einem exponentiellen Wachstum an Sourcecodezeilen:

$$n_{LOC} \sim \mathrm{e}^{\alpha t}$$

4.14 Taxonomie der Änderung

Damit die Evolution einer Software besser beurteilt werden kann, empfiehlt es sich, eine Taxonomie der Softwareänderungen zu erstellen. Softwareänderungen können unterschiedlich klassifiziert werden. Am einsichtigsten erscheint es, eine Klassifikation nach vier verschiedenen Klassen zu wählen, welche sich am sinnvollsten an folgenden Basisfragen orientieren:

- Zeit, d.h. die Frage nach dem Wann.
- Ort, d.h. die Frage nach dem Wo.
- Eigenschaft, d.h. die Frage nach dem Was.
- Art und Weise oder auch Support, d.h. die Frage nach dem Wie.

Obwohl es möglich ist, auch die frühen Phasen eines gesamten Lebenszyklus einer Software zu betrachten, wollen wir uns hier auf Änderungen einer Legacysoftware und damit auf die späten Phasen beschränken. Die Übertragung der Taxonomie auf die frühen Phasen eines Softwarelebenszyklus ist jedoch recht einfach und direkt machbar.

Tab. 4.4: Die Änderungsklassen und ihre Dimensionen

Zeit	**Ort**	**Eigenschaft**	**Support**
Änderungszeitpunkt	Artefakt	Verfügbarkeit	Automatisierung
Historie	Granularität	Aktivität	Formalität
Frequenz	Impact	Offenheit	Typ
Antizipation	Propagation	Sicherheit	

Diese verschiedenen Klassen lassen sich noch einmal näher unterteilen, wobei sich die Veränderung der Software durch diese Unterteilung immer stärker präzisiert. Die verschiedenen Klassen haben unterschiedliche Dimensionen, s. Tab. 4.4. Die so gewählte Einteilung ist eine von vielen möglichen, sie erscheint jedoch als die sinnvollste.

4.14.1 Zeit

Die erste Klasse der Änderungen in der Software im Laufe der Evolution beschäftigt sich mit den zeitlichen, oder auch temporalen, Eigenschaften von Änderungen. Eine solche Unterteilung ist wichtig, da sie die grundlegende Dimension, die Zeit, der Evolutions- und Komplexitätsentwicklung direkt zugänglich macht.

Änderungszeitpunkt

Abhängig von der Programmiersprache lässt sich, neben der Feststellung, wo sich die Software gerade im Rahmen ihres Lebenszyklus befindet, der Änderungszeitpunkt als Funktion des Maintenanceprozesses betrachten. Dort lassen sich, in Bezug auf ein aktives Softwaresystem, drei Phasen unterscheiden, wann die Änderung stattfindet:

- Statisch – Wenn die Veränderung den Sourcecode des Systems betrifft. Dann muss das System neu compiliert werden, damit die Veränderung wirksam wird.
- Load-Time – Bei dieser Form der Veränderung tauchen die Änderungen erst zur Ladezeit der jeweiligen Software auf. Ein typischer Vertreter dieser Art wäre Smalltalk als Sprache oder der Einsatz einer DLL eines COTS-Software-Herstellers.
- Dynamisch – Hier tauchen die Änderungen erst zum Ausführungszeitpunkt, Execution-Time, auf. Webservices sind hierfür ein Beispiel.

Der traditionelle Ansatz ist einer der statischen Evolution, oft auch als Compile-Time-Evolution bezeichnet. Hierbei ändert der Softwareentwickler den Sourcecode, compiliert und testet[13] ihn und setzt ihn dann in der Laufzeitumgebung der Legacysoftware frei. Dies kann in großen Systemen über

[13] Hoffentlich ...

mehrere Stufen, so genannte Staging-Areas, geschehen, dabei wird auf jeder Stufe das System ausführlich getestet. In der gängigen Praxis wird jedoch häufig auf das Staging verzichtet und die Änderungen werden direkt in die Produktion freigesetzt. Die Motivation hinter diesem Vorgehen ist ein subjektiv empfundener Geschwindigkeitsvorteil von Seiten des Softwareentwicklers. Ob dieses Quick-Fix-Verfahren, s. Abschn. 7.5, tatsächlich schneller ist, bleibt sehr fragwürdig. Allerdings hat sich diese Form der Bequemlichkeit[14] soweit verselbstständigt, dass dieses Verhaltensmuster in der Maintenance von Legacysystemen nur noch sehr schwer korrigierbar ist. Trotz seinem häufigen Vorkommen ist die Bezeichnung Compile-Time-Evolution irreführend, da der tatsächlich verändernde Bestandteil nur das freigesetzte Binärcompilat ist.

Im Gegensatz dazu steht die dynamische Evolution. Hier werden die Veränderungen an der Legacysoftware direkt zur Laufzeit produziert, genauer gesagt dann, wenn die entsprechende Komponente geladen wird, ohne dass das Gesamtsystem gestoppt werden muss. In den meisten Fällen geschieht die Evolution der neuen Komponente vor der Änderung des Laufzeitsystems, die nachfolgende Integration der Änderung wird als Run-Time-Activation bezeichnet. Im Fall einer direkten Veränderung der Laufzeitumgebung der Legacysoftware spricht man von einer Runtime-Evolution. Diese Form der Evolution setzt eine Reflexivität[15] des Laufzeitsystems voraus, eine solche kann beispielsweise in einer Smalltalkumgebung gefunden werden.

Statische und dynamische Evolution schließen sich weder gegenseitig aus, noch müssen sie, nota bene, abhängig sein, da statisch veränderte Compilate die Voraussetzung für Run-Time-Activation sein können. Es ist mehr die Frage der Programmiersprache und des generellen Aufbaus der Legacysoftware, welche eine solche Unterscheidung impliziert.

Die Load-Time-Evolution ist eine Art Mittelweg zwischen den beiden Extremen. Ein typischer Vertreter dieser Art ist der Mechanismus des Java-Classloaders, welcher die Klassen lädt, aber eine einmal geladene Klasse nicht mehr verändert, solange sie im Hauptspeicher resident ist. Ein weiteres Beispiel für eine solche Load-Time-Evolution ist eine Legacysoftware, bestehend aus mehreren IMS-Transaktionen. Die einzelne IMS-Transaktion würde einer Compile-Time-Evolution unterliegen, aber nachdem sie einmal geladen wurde, verbleibt sie resident im Hauptspeicher. Sowohl Load-Time- als auch dynamische Evolutionssysteme verlangen spezielle Mechanismen, mit denen die Evolution im Laufzeitsystem kontrolliert werden kann. Neben einem solchen Kontrollmechanismus müssen solche Systeme auch ein hohes Maß an Reflexivität besitzen; denn ohne dass das System in der Lage ist Auskunft über sich selbst zu geben, ist eine gesteuerte Evolution schwerlich möglich.

[14] Einzelne Unternehmen gehen sogar so weit, diese Form des Chaos als „eXtreme Programming“, s. Abschn. 11.4.1, zu bezeichnen.

[15] Reflexivität ist die Fähigkeit eines Systems, Auskunft über sich selbst zu geben.

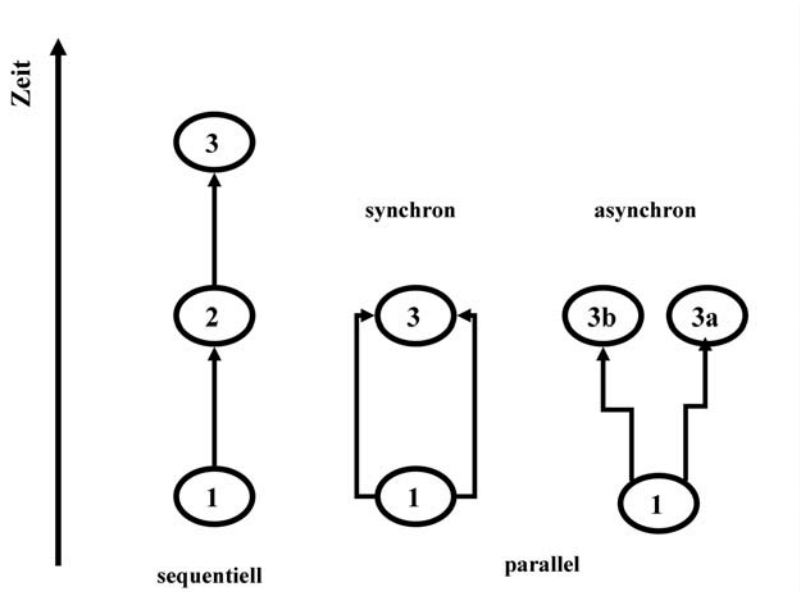

Abb. 4.12: Die verschiedenen Formen der Änderungen: sequentiell, synchron parallel, asynchron parallel

Historie

Die Änderungshistorie einer Legacysoftware bezieht sich auf die Geschichte aller Veränderungen, gleichgültig ob sequentiell oder parallel, welche an der Legacysoftware vorgenommen worden sind. Werkzeuge, die diese Veränderungen aufzeichnen und gegebenenfalls visualisieren, nennt man Versionskontrollsysteme.

In vollständig unversionierten Systemen – ein Zustand, der bei Legacysystemen manchmal anzutreffen ist – wirken Veränderungen destruktiv auf das System, da jetzt der alte Zustand durch einen neuen ersetzt wird, ohne dass sich der alte Zustand nachträglich vollständig rekonstruieren lässt. Dies ist, wenn man die weite Verbreitung von Versionskontrollsystemen auf fast jeder Plattform betrachtet, kein Resultat der Systemeigenschaften der Legacysoftware, sondern der Unfähigkeit der Maintenanceorganisation, die entsprechenden Mechanismen einzusetzen.

Die Systeme, bei denen die Versionierung dazu führt, dass zur Compilezeit mehrere Versionen vorliegen können, aber zur Laufzeit stets nur eine spezifische Version vorhanden sein kann, nennt man Systeme mit statischer Versionierung. Im Gegensatz hierzu stehen die dynamisch versionierbaren Systeme, welche unterschiedliche Laufzeitsysteme simultan[16] im gleichen Softwaresys-

[16] Nach dieser Klassifikation ist Java statisch versionierend, da in einem Namespace nur eine Klasse mit einem Namen vorkommen kann.

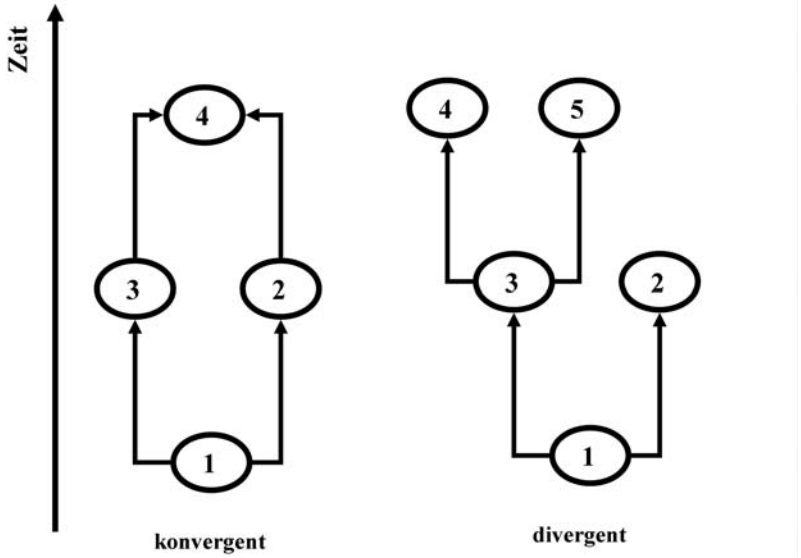

Abb. 4.13: Konvergente und divergente Änderungen

tem erlauben. Ein Beispiel für dynamische Versionierung ist CORBA, da einzelne CORBA-Interfaces die Versionsnummer explizit im Aufruf mitführen. Dynamische Versionierung hat den sehr großen Vorteil, eine echte Koexistenz zwischen unterschiedlichen Versionen der Legacysoftware zu ermöglichen. Dies ist insofern wichtig, da dann das Unternehmen bei seinen internen Abläufen eine gewisse Zeit hat, sich auf das geänderte Legacysystem einzustellen. In einem solchen System existieren beide Versionen so lange parallel, bis die alte Version obsolet ist, d.h. es existieren keine Nutzer der alten Version mehr, und diese kann sicher aus dem Gesamtsystem entfernt werden.

In Bezug auf die Historie der Veränderung kann zwischen, s. Abb. 4.12:

- sequentiell
- parallel
  - synchron parallel
  - asynchron parallel

unterschieden werden.

Im Fall der parallelen Änderungen geschehen die Veränderungen am System durch unterschiedliche Personen zur gleichen Zeit. Im anderen Falle, dem sequentiellen Fall, werden die Veränderungen nacheinander durchgeführt und ermöglichen eine Reihe von sukzessiven Versionen, daher auch der Name sequentielle Änderungen. Die parallelen Veränderungen, d.h. simultane Änderungen durch unterschiedliche Personen, lassen sich in synchron und asyn-

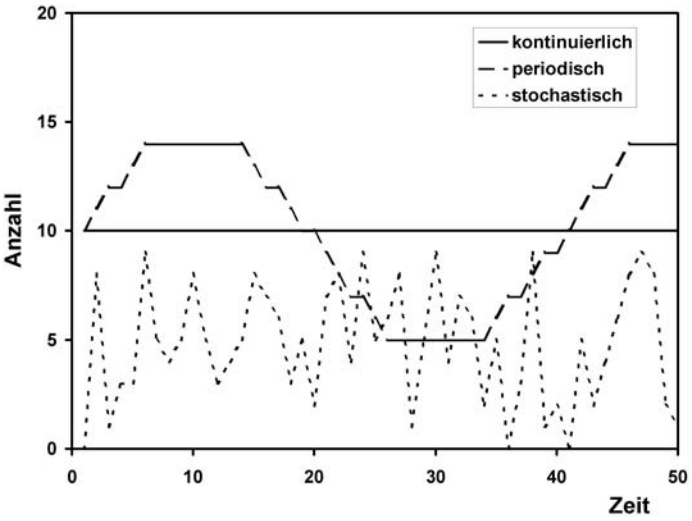

Abb. 4.14: Die verschiedenen Formen der Änderungsfrequenzen: kontinuierlich, periodisch, stochastisch

chron klassifizieren, je nachdem, ob nur eine Folgeversion oder mehrere Folgeversionen entstehen.

Die Existenz von Parallelität führt zur Schaffung von konvergenten und divergenten Versionen, s. Abb. 4.13. Im konvergenten Fall werden unterschiedliche Versionen zu einer neuen Version zusammengefasst. Diese Zusammenführung der einzelnen Versionen bezeichnet man als Merging. Bei divergenten Versionen existieren diese ab dem Zeitpunkt der Entstehung bis zum Lebensende des Systems; solche divergenten Versionen werden als Branches[17] bezeichnet.

Frequenz

Neben der reinen Form der Änderung ist auch die Frequenz[18], d.h. die Zahl der Änderungen pro Zeiteinheit, wichtig. Eine Fourieranalyse der Änderungen zeigt das Verhalten der Änderungen als Funktion der Zeit an oder, anders ausgedrückt, die zeitlichen Schwankungen der Änderungshäufigkeiten liefern ein klassifizierendes Abbild.

[17] Branches auf Teilen eines Systems sind eine notwendige Voraussetzung für Produktlinien, s. Kap. 9.

[18] Die wissenschaftlich korrekte Maßeinheit für die Frequenz ist Hertz (Hz). Leider hat sich diese Größenordnung nicht durchgesetzt, so dass man sehr oft Angaben wie 5 Änderungen pro Monat, d.h. 1.93 μHz, vorfindet.

Grundsätzlich kann die Änderungsfrequenz unterschieden werden in:

- kontinuierlich
- periodisch
- stochastisch

Diese verschiedenen Änderungshäufigkeiten sind in Abb. 4.14 exemplarisch dargestellt.

Die Ursachen für diese zeitlichen Schwankungen sind durchaus unterschiedlich. Zum einen sind die Schwankungen durch die interne Organisation der Maintenance einer Legacysoftware geprägt, zum anderen oft auch durch äußere Einflüsse. Die Softwareunternehmen, welche Softwaresysteme aus dem Sozialversicherungsumfeld pflegen, beobachten meist eine vierteljährige Periodizität in den Änderungen der Legacysoftware, da hier die Sozialversicherungsträger faktisch im Quartalsrhythmus Veränderungen produzieren, welchen die Software sehr zeitnah folgen muss. Eine andere Quelle für ein periodisches Verhalten kann die interne Urlaubsplanung eines Unternehmens sein oder ein internes Qualitätssicherungsverfahren, das ein *Change Advisory Board* enthält; dieses gibt dann die entsprechenden Änderungen blockweise frei.

Die Frequenz der Änderung spielt für die Planung der Maintenance wie auch für die Werkzeugnutzung im Rahmen einer Versionskontrolle eine maßgebliche Rolle. Bei einer geringen Anzahl von Änderungen ist die geforderte Werkzeugunterstützung meist recht einfach, im Gegensatz zu hohen Änderungsraten – hier ist eine ausgefeilte Versionskontrolle[19] unabdingbar; ohne diese sind Rollback-Mechanismen nur schwer vorstellbar.

Antizipation

Veränderungen an einem bestehenden Softwaresystem können entweder schon bei der Ersterstellung antizipiert worden sein, oder sie tauchen als nichtantizipierte, quasi spontan unvorhergesehene, Änderungen auf. Vorhergesehene Änderungen an Softwaresystemen entstehen in der Regel auf Grund eines Releaseplanes, bei dem ausgelieferte Software bewusst vorzeitig ausgeliefert wurde, um die Time-to-Market-Zeit möglichst kurz zu halten.

Speziell bei den COTS-Software-Herstellern ist dies bei neuen Produktpaletten oft der Fall, da sich ja hier die Softwareentwicklung direkt refinanzieren muss bzw. die Konkurrenz der anderen COTS-Software-Hersteller de facto oft Releasetermine erzwingt. In diesen Fällen muss die Software antizipativ verändert werden, damit sie irgendwann einmal ihren vollen Leistungsumfang erreichen kann.

Alle Legacysysteme sind faktisch so alt, dass jede Form der antizipativen Änderung entweder nicht mehr aktuell oder geplant bzw. schon durchgeführt

[19] Besonders die Produkte CVS, Open Source, und ClearCase, IBM, erfreuen sich zunehmender Beliebtheit.

wurde. Von daher sind alle Änderungen an Legacysystemen stets nichtantizipativ. Dieser Klassifikation steht nicht die Tatsache entgegen, dass, z.B. bei der Euroumstellung, Änderungen lange geplant sein können; wichtig ist hier die Tatsache, ob diese spezielle Änderung auch schon bei der Ersterstellung geplant war oder nicht.

4.14.2 Ort

Das zweite, praktisch orthogonale Klassifikationsschema beschäftigt sich mit dem Ort der Änderung und den Mechanismen, welche für die Änderungen der Objekte[20] verantwortlich sind.

Artefakt

Im Laufe einer Entwicklung entstehen viele verschiedene Artefakte, diese rangieren in einem sehr weiten Bereich. Einige Beispielartefakte sind:

- Architekturdokumente
- Anforderungsanalyse
- Designstudie
- Sourcecode
- Dokumentation
- Online-Hilfe
- Testsuites

Offensichtlich brauchen die verschiedenen Artefakte unterschiedliche Unterstützungsmechanismen im Rahmen aller Softwareprozesse, gleichgültig ob Erstentwicklung oder Maintenance.

Am einfachsten lassen sich bei einer statischen Evolution die Artefakte wiederum selbst versionieren. Bei dynamischer Evolution ist dies ungleich schwieriger, da hier meist das Gesamtsystem, oder große Teile von diesem, ein Artefakt darstellt.

Granularität

Ein weiterer beeinflussender Faktor für die Evolution einer Legacysoftware ist die Granularität der Änderung. Granularität gibt die Größe des Artefakts an, welcher verändert wird. Rangieren kann er von sehr grob-granular bis hin zu feingranular. Eine feingranulare Änderung wäre beispielsweise die Änderung eines einzelnen Statements im gesamten Legacysystem.

Traditionell liegt die Trennlinie auf Dateiebene, d.h. üblicherweise werden Änderungen, die nur eine Datei betreffen und in dieser Datei dann auch

[20] Objekt in diesem Sinne ist der Gegenstand, welcher verändert wird, es wird damit keinerlei Form der Objektorientierung impliziert.

durchgeführt werden, als feingranular bezeichnet. Diese Trennung hat zwei Ursachen, zum einen lassen sich dateiübergreifende Veränderungen sinnvoll nur durch den Einsatz eines Versionskontrollsystems verfolgen und zum anderen sind die meisten Einzeldateiänderungen nur geringfügige, für den Softwareentwickler intellektuell problemlos überschaubare, Veränderungen.

Impact

Der Impact, auch als Auswirkung einer Änderung bezeichnet, kann von lokal bis hin zu systemweit rangieren. Üblicherweise wird im Rahmen der Maintenance der Versuch unternommen den Impact möglichst lokal zu halten, da jeder Impact, welcher nicht lokal ist, zu einer sehr großen Komplexität und damit zu sehr hohen Kosten führt. Ein typisches Beispiel für eine solche impactminimierende Strategie ist der Einsatz von Wrappern oder dem Façade Pattern, s. Kap. 13. Diese Patterns versuchen, durch die Erhaltung oder Schaffung eines Interfaces, Änderungen in ihrem Impact auf einzelne Teile zu begrenzen; die Stabilität der Schnittstelle „sperrt“ die Änderung der Komponente ein.

Die gesamte Komponententechnologie ist letztendlich der Versuch, den Impact einer Änderung auf die jeweilige Komponente zu beschränken und damit eine Risikovermeidung vorzunehmen.

Neben der „räumlichen“ Ausdehnung des Impacts in einem Legacysystem kann der Impact auch angeben, welche verschiedenen Artefakte bzw. welche unterschiedlichen Abstraktionsebenen von den Veränderungen betroffen sind. Je mehr Artefakte und je verschiedener die Abstraktionsebenen sind, welche von einer Veränderung betroffen sind, desto höher ist das implizierte Risiko.

Propagation

Veränderungen mit einem nichtlokalen Impact ziehen per definitionem Änderungen in anderen Artefakten nach sich. Die Durchführung dieser nachfolgenden Änderungen nennt man Change Propagation. Die Vorstellung dahinter ist es, die Artefakte als einen Wald von Dominosteinen zu sehen; das Kippen eines Dominosteines, d.h. die Änderung eines Artefakts, zieht eine Reihe von anderen „Umfallern“ nach sich.

In Legacysystemen kann die Propagation meistens erst ex post gefunden werden. In modernen Entwicklungsumgebungen, speziell in Java-Umgebungen, wird ein Teil der Propagation automatisch durchgeführt, bzw. es wird darauf hingewiesen.

Da die Propagation sehr schnell zu Änderungen mit einem systemweiten Impact führen kann, wurde schon frühzeitig versucht, die Propagation zumindest passiv beobachtbar zu machen. Vorgehensweisen und Werkzeuge, welche dies bewerkstelligen, erlauben es, eine so genannte „Traceability Analysis“ durchzuführen. Bei der Traceability wird zwischen der vertikalen und der

Tab. 4.5: Eigenschaften und Evolutionstypen

Subklassifikation	**Ausprägung**	**statische Evolution**	**dynamische Evolution**
Verfügbarkeit	permanent	nein	ja
	nicht permanent	ja	ja
Aktivität	reaktiv	nein	ja
	proaktiv	nein	ja
Offenheit	offen	ja	ja
	geschlossen	ja	nein

horizontalen Traceability unterschieden. Es wird von einer horizontalen Traceability gesprochen, wenn die Verknüpfung der verschiedenen Artefakte auf der gleichen Abstraktionsebene stattfindet. Vertikale Traceability impliziert, dass die Verfolgung der Änderung über unterschiedliche Abstraktionsebenen hinweg geht. Horizontale Traceability wird oft schon durch die Entwicklungsumgebung oder einfache Werkzeuge gewährleistet. Mit dem Aufkommen des Interesses an Architekturgovernance wird die vertikale Traceability immer wichtiger, da sich bei einem Architekturwechsel die Frage nach dem Impact der Änderung in einem großen System stellt.

Die meisten traditionellen Verfahren sind Ex-post-Verfahren und funktionieren, indem jeder Änderung eine Bugnummer oder Change-Request-Nummer zugeordnet und diese Nummer und damit alle geänderten Artefakte in dem spezifischen Versionskontrollsystem rückverfolgbar gemacht werden.

In den meisten Fällen sind Veränderungen mit einem hohen Impact auch Veränderungen, welche hohe Kosten verursachen, von daher sind Impact und Propagationsbetrachtungen sehr wichtig. Allerdings gibt es auch Gegenbeispiele: Das Abändern z.B. von Datenbanknamen zieht sich zwar durch das ganze Legacysystem, daher ein systemweiter Impact, und durch viele Sourcecodedateien, daher eine starke Propagation, allerdings kann die Abänderung meistens werkzeuggestützt durchgeführt werden, was den Aufwand eher gering hält.

4.14.3 Eigenschaften

Diese Klassifikation nach den Eigenschaften ist wiederum orthogonal zu den beiden vorhergehenden, Zeit und Ort, und beschäftigt sich mit den Eigenschaften des Legacysystems, die durch eine Änderung verändert werden. An dieser Stelle werden ausschließlich die Systemeigenschaften betrachtet, nicht jedoch funktionale Eigenschaftsveränderungen. Funktionale Eigenschaftsveränderungen sind eine Form der fachlichen Evolution und resultieren aus der Anforderungsevolution, s. Abschn. 4.15.

Der Zusammenhang zwischen den einzelnen Systemeigenschaften und den beiden extremalen Evolutionstypen ist in Tab. 4.5 dargestellt.

Verfügbarkeit

Ein Legacysystem muss stets auch verfügbar sein, da es in der Regel das Kernsystem eines Unternehmens darstellt. Aber die Verfügbarkeit hat durchaus auch Abstufungen. So müssen erst seit dem Aufkommen von Internetanbindungen die Legacysysteme eine 7×24-Verfügbarkeit[21] haben. Vor dieser Zeit und auch heute noch für einen Großteil der Legacysysteme reicht eine 8-18-Uhr-Verfügbarkeit völlig aus. Es gibt jedoch auch Legacysysteme, welche schon immer eine permanente Verfügbarkeit benötigten, so z.B. Telefonvermittlungssysteme. Viele solcher Legacysysteme können aus wirtschaftlichen Gründen nicht gestoppt werden und benötigen daher einen dynamischen Evolutionsmechanismus, um Teile eines solchen Legacysystems verändern zu können.

Aktivität

Die Änderungsaktivitäten, die in einem allgemeinen Softwaresystem auftauchen können, fallen in zwei unterschiedliche Kategorien:

- reaktiv
- proaktiv

Bei den reaktiven Softwaresystemen sind alle Änderungen extern getrieben, d.h. der Impuls zur Veränderung findet außerhalb der Software statt. Im Gegensatz dazu stehen proaktive Systeme, bei denen das System die Veränderungen an sich selbst[22] vorantreibt.

Ein proaktives Softwaresystem muss stets seine Umgebung wie auch sich selbst beobachten und außerdem Mechanismen enthalten, um aus diesen Beobachtungen Schlüsse zu ziehen und die somit „berechneten“ Veränderungen umsetzen zu können. Softwaresysteme mit künstlicher Intelligenz und einem hohen Maß an Reflexivität fallen in eine solche Kategorie. Allerdings ist anzumerken, dass es kein ernstzunehmendes Legacysystem gibt, welches proaktiv konzipiert wurde. Proaktive Softwaresysteme haben bisher die akademische Welt noch nicht verlassen.

Offenheit

Softwaresysteme werden als offen bezeichnet, wenn sie spezifische Mechanismen zur Erweiterung des Systems beinhalten. Offene Systeme enthalten meist eine Art Framework, welches die Ausdehnung des Systems erlaubt. Zwar geben solche Frameworks ein gewisses Maß an Unterstützung, um damit antizipierte oder nicht antizipierte Änderungen zu ermöglichen, jedoch tragen solche

[21] 7×24 bedeutet Montag bis Sonntag und jeden Tag volle 24 Stunden.

[22] Der Computer HAL-9000 aus Stanley Kubricks *2001 A Space Odyssey* ist ein Beispiel für ein solches proaktives System.

Frameworks immer ein implizites Modell über die Form und Implementierung einer Software in sich.

Im Gegensatz zu den offenen Systemen stehen die geschlossenen Systeme. Solche müssen schon die gesamte fachliche Funktionalität zum ursprünglichen Bauzeitpunkt enthalten. Dies bedeutet nicht, dass geschlossene Systeme nicht erweiterbar wären, sondern nur, dass geschlossene Systeme keine spezifischen Mechanismen enthalten, um sich erweitern zu lassen.

Ein Paradebeispiel für offene Systeme sind kernel- oder microkernelbasierte Betriebssysteme, wie beispielsweise Linux oder auch Windows NT. Solche Betriebssysteme enthalten eindeutige Regeln und Mechanismen, welche ihre Erweiterung steuern und ermöglichen. Ein anderes Beispiel für ein offenes System sind Sprachen, bei denen das erstellte Programm selbst ein für das Programm veränderbares Objekt ist, hierzu gehören Smalltalk und Lisp.

Geschlossene Systeme verfügen nicht über explizite Mechanismen zur Erweiterung. Hierzu zählen Standardsprachen wie COBOL oder Fortran. Diese sind, auf Grund eines fixierten Sprachumfanges, nicht mehr erweiterbar, bzw. es fehlt diesen Sprachen die Ebene der Metaprogrammierung.

Die meisten Legacysysteme fallen in eine Art Zwischenkategorie. Ursprünglich waren sie als geschlossene Systeme konzipiert worden. Mit zunehmender Lebensdauer der Legacysoftware zeigte sich jedoch die Notwendigkeit, dieses System möglichst effizient erweitern zu können. Dieser Forderung wird in der Praxis auf zwei Arten Rechnung getragen:

- Einbau eines spezifischen Frameworks, um die Erweiterungen zu ermöglichen. Hierbei gibt es durchaus unterschiedliche Erweiterungsmechanismen. Dazu zählen:
  - Dynamische Aufrufe auf Programmiersprachenebene, d.h. ausführbare Objekte werden erst zur Laufzeit eingebunden. Implizit werden hiermit Unterprogrammbibliotheken aufgebaut.
  - Dynamische Aufrufe durch Transaktionstabellen. Hierbei werden die Transaktionsaufrufe an Systeme weitergegeben, wobei das konkret aufgerufene System erst zur Laufzeit ermittelt wird.
  - Einsatz höherwertiger Kommunikationsprotokolle, so z.B.:
    - CORBA
    - EAI-Systeme
    - Webservices
  - Einsatz einer XML-Schnittstelle zur Entkoppelung.
- Alternativ dazu ist eine Optimierung des Softwareentwicklungsprozesses ohne Nutzung eines Erweiterungsframeworks möglich. Wenn aber der Softwareentwicklungsprozess schnell genug für die meisten Veränderungen durchgeführt werden kann, ist die Evolution auch ohne ein Framework möglich. Die Model Driven Architecture, MDA, ist ein Ansatz in dieser Richtung.

Sicherheit

Die Sicherheit einer Änderung hat verschiedene Aspekte, rangierend von der Verhaltenssicherheit über die Kompatibilität bis hin zu Konzepten wie strukturelle und semantische Sicherheit einer Änderung. Im Kontext der Evolution eines Systems wird zwischen zwei Formen der Sicherheit unterschieden:

- statische Sicherheit
- dynamische Sicherheit

Ein System wird als statisch sicher bezeichnet, wenn zum Compilezeitpunkt nachgewiesen werden kann, dass alle sicherheitsrelevanten Aspekte überprüft werden können. Ein System, welches Mechanismen für diese Überprüfung zur Laufzeit enthält, wird dynamisch[23] sicher genannt.

4.14.4 Support

Während der evolutionären Veränderung einer Software können verschiedene Arten von Support für diese Änderungen geleistet werden. Bei dem Support handelt es sich wiederum um eine orthogonale Kategorie zu den vorhergehenden drei Kategorien.

Automatisierung

Eines der einfachsten Unterscheidungsmerkmale des Supports ist die Werkzeugunterstützung für eine Änderung. Diese rangiert in den Bereichen:

- vollautomatisch
- halbautomatisch
- manuell

Eines der Ziele hinter dem Softwareengineering ist es, einen möglichst hohen Grad an Automatisierung zu erreichen. Meist wird dies für die Problematik der Ersterstellung stärker untersucht, allerdings gibt es auch, hauptsächlich angetrieben durch das Reengineering, Ansätze zur Unterstützung der Automation bei Softwareänderungen.

Vollautomatische Softwareproduktion in Reinform wird bei dem MDA-Ansatz verfolgt. Für ein rein statisch evolvierendes System könnte dieser Ansatz zusammen mit einer vollautomatischen Werkzeuglandschaft dazu führen, dass die Softwareevolution nicht die Veränderung eines bestehenden Systems ist, sondern als eine Abfolge disjunkter, jeweils funktional neuer Systeme verstanden werden kann.

Trotz allen technischen Fortschritts, am weitesten verbreitet in den Legacysystemen ist immer noch die Form der manuellen Änderung. Einer der

[23] Der Java-Classloader ist ein Beispiel für ein dynamisch agierendes Sicherungssystem.

Gründe liegt darin, dass automatisierende Werkzeuge entweder keine oder nur geringe Unterstützung bei der Maintenance liefern, bzw. so konzipiert sind, dass ihr Einsatz schon die Ersterstellung der Software durch das Werkzeug voraussetzt.

Ironischerweise wurde ein Teil der Legacysysteme durch den Einsatz von CASE-Werkzeugen[24] ersterstellt. Im Laufe der Zeit stellten sich jedoch die Defizite, speziell bei dem Support von Änderungen durch diese Werkzeuge heraus, so dass Ende der neunziger Jahre Anstrengungen unternommen wurden, von diesen Werkzeugen wegzukommen.

Formalität

Ein Änderungssupportmechanismus kann entweder ein Ad-hoc-Verfahren sein, oder es unterliegt ihm ein Formalismus. Solche Formalismen müssen auf einem entsprechenden mathematischen Hintergrund basieren, um effektiv zu sein. Mathematische Formalismen, meistens basierend auf der Graphentheorie, ermöglichen es, Impact und Propagation genauer zu betrachten und vorauszusagen. Obwohl kommerzielle Werkzeuge oft solche Formalismen einsetzen, um einen gewissen Grad an Automatisierung zu erreichen, sind diese Formalismen in den Werkzeugen doch nicht direkt sichtbar und unterliegen in den meisten Fällen einem gewissen Maß an Heuristik.

Zwar existieren Formalismen für spezifische Probleme, so beispielsweise Formalismen bei der Programmverifikation, allerdings ist ihre Praxisreife zur Zeit eher als dürftig zu betrachten.

Typ

Die Art und Weise, wie die Änderung durchgeführt wird, kann die Änderung selbst beeinflussen. Die Änderungen lassen sich in zwei große Gruppen einteilen:

- Strukturelle Änderungen – Dies sind Veränderungen, die die Struktur der Legacysoftware abändern. Typische Kandidaten sind Architekturänderungen, da diese die Struktur sehr weitreichend verändern.
  Oft werden die strukturellen Änderungen noch weiter unterschieden in:
  - Additive strukturelle Veränderungen – Die additiven Veränderungen fügen neue Teile zum Legacysystem hinzu und erhöhen so die Entropie bzw. die Komplexität.
  - Subtraktive strukturelle Veränderungen – Subtraktive Veränderungen entfernen strukturelle Teile der Legacysoftware. Sie sind eine Möglichkeit die Entropie bzw. die Komplexität eines Systems zu senken. In den meisten Fällen sind Subtraktionen durch den Wegfall von funktionalen Anforderungen oder den Einsatz einer COTS-Software für vorhandene Systemteile getrieben.

[24] In der ersten Hälfte der neunziger Jahren waren besonders IEW und ADW beliebt.

  - Normale[25] strukturelle Veränderungen – Diese sind meistens Folgen eines Refaktorings.
- Semantische Aspekte – Semantische Veränderungen sind orthogonal zu den strukturellen Veränderungen, d.h. die Bedeutung bestimmter Daten und Funktionen verändert sich. Funktionale Veränderungen, durch Fachbereiche initiiert, fallen in ein solches Schema. Die semantischen Aspekte von Softwareevolution lassen sich in zwei Kategorien einteilen:
  - Semantikerhaltende Operationen – Bei einer semantikerhaltenden Änderung wird an der Bedeutung der Objekte innerhalb der Legacysoftware nichts verändert. Typisch für diese Form der Evolution ist eine Restrukturierung des Systems ohne eine Abänderung der Funktionalität.
  - Semantikändernde Operationen – Dies sind Änderungen, welche funktionale Aspekte der Legacysoftware abändern.

Die meisten praktischen Änderungen an Legacysystemen fallen in beide Gruppen, jedoch ist es nützlich, sich die Unterschiede explizit klar zu machen. Häufig ist es erstrebenswert, in beiden Kategorien getrennt vorzugehen, d.h. eine geplante Änderung des Systems in zwei Schritten durchzuführen.

1 Zuerst die strukturelle Veränderung bei Beibehaltung der Semantik,
2 gefolgt von einer semantikändernden Operation, um das gewünschte Endresultat zu erzielen.

Durch dieses Vorgehen lässt sich das Risiko insgesamt minimieren, d.h. wenn ein System $\mathcal{S}$ durch eine Evolution, hier mit dem Operator $\widehat{O}$ gekennzeichnet, abgeändert werden soll, so gilt für das Risiko R:

$$R\left(\widehat{O}_{gesamt}(\mathcal{S})\right) > R\left(\widehat{O}^{semantikerhaltend}_{strukturell}(\mathcal{S})\right) \cup R\left(\widehat{O}^{strukturerhaltend}_{semantikändernd}(\mathcal{S})\right).$$

Beide Operationen simultan auszuführen ist in der Regel zu komplex und fehleranfällig, daher senkt eine sequentielle Veränderung in zwei Schritten das Risiko, da jetzt jeder einzelne Schritt eine geringere Fehleranfälligkeit besitzt.

Die Erfahrung zeigt außerdem, dass in den meisten Fällen gilt[26]:

$$\left[\widehat{O}^{semantikerhaltend}_{strukturell}, \widehat{O}^{strukturerhaltend}_{semantikändernd}\right] \neq 0. \tag{4.11}$$

Folglich unterscheidet sich die Änderung, je nachdem, in welcher Reihenfolge die Operationen zur Gesamtveränderung durchgeführt werden. Der Hinter-

[25] Der englische Ausdruck ist *alteration*.

[26] Die Darstellung in Kommutatorform hat sich in der Quantenphysik

$$\left[\hat{P}, \hat{Q}\right] = \hat{P}\hat{Q} - \hat{Q}\hat{P}$$

für Operationen bewährt, bei denen die Reihenfolge ihrer Durchführung eine Rolle spielt bzw. Operationen die einer Unschärferelation unterliegen.

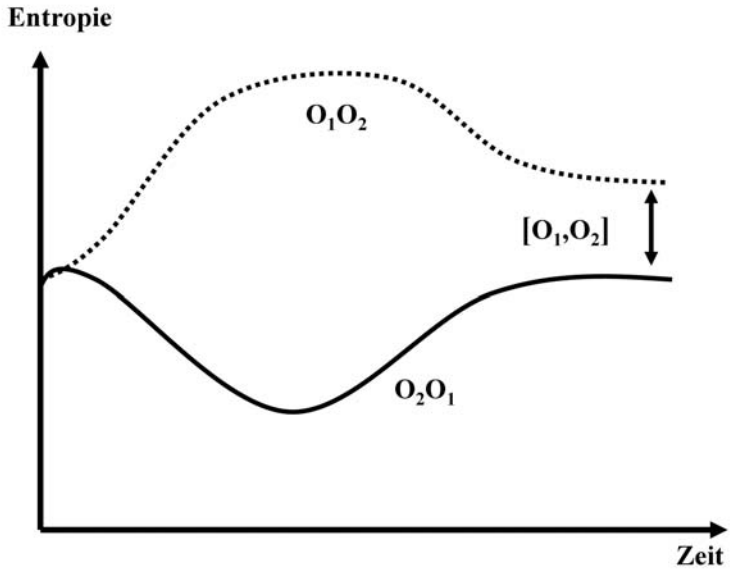

Abb. 4.15: Die Nichtvertauschbarkeit der Änderungen im Entropie-Zeit-Diagramm

grund für diese Beobachtung ist, dass wenn zuerst restrukturiert wird, Entropie und Komplexität leicht absinken, was den nachfolgenden semantikändernden Schritt einfacher macht. Im umgekehrten Fall wird versucht, die Semantik auf einem höheren Wert der Entropie und Komplexität abzuändern, was meist zu höheren Aufwänden, sowie höherem Risiko führt. In Abb. 4.15, mit:

$$O_1 = \widehat{O}^{strukturerhaltend}_{semantikändernd}$$
$$O_2 = \widehat{O}^{semantikerhaltend}_{strukturell}$$

sind beide Wege durch das Entropie-Zeit-Diagramm schematisch dargestellt. Die unterschiedliche Endentropie ergibt sich aus der Nichtvertauschbarkeit der beiden Operationen.

Zu einem gewissen Teil beruht diese Beobachtung auf der „Zerfallsrate“ von Wissen in den Köpfen der Beteiligten. Wird nämlich in einem nicht gut verstandenen System eine neue Funktionalität eingeführt, d.h. eine semantische Änderung vorgenommen, so wird sie strukturell „irgendwie“ integriert, da ein Teil der Softwareentwickler die Struktur der Legacysoftware nicht genau kennt. Eine nachfolgende strukturelle Änderung trifft nun auf unterschiedliche stilistische Teile, den alten Teil der Legacysoftware und auf die neu implementierte Funktionalität, meist verknüpft mit einem gehörigen Maß an Bloating. Diese Tatsache erhöht die Wahrscheinlichkeit, dass das entstehende System nach einer strukturellen Änderung eine höhere Entropie besitzt. Im

umgekehrten Fall setzen sich die Softwareentwickler auf einer kleineren Sourcecodebasis[27] mit der Struktur auseinander und können daher viel effektiver arbeiten. Die neue Struktur ist bei der nachfolgenden semantischen Änderung den Softwareentwicklern noch präsent, so dass die Wahrscheinlichkeit einer strukturell sauberen Implementation mit einer niedrigeren Entropie ansteigt. Die Beobachtung des Verhaltens der Entropie zusammen mit dem Grundsatz, dass stets eine möglichst niedrige Entropie anzustreben ist, lässt sich folgende Faustregel für geplante große Veränderungen von Legacysystemen formulieren: *Die strukturellen Veränderungen sind immer vor den semantischen Veränderungen auszuführen!*

4.15 Anforderungsevolution

Die ältesten kontinuierlich genutzten Legacysysteme sind die Telefonsysteme, da sie auf ein Lebensalter von über 50 Jahren zurückblicken können. Statistische Untersuchungen über solche sehr alten Legacysysteme führen zu der empirischen Beobachtung, dass die Phasen der langsamen Veränderung von Anforderungen gefolgt werden von kurzen Phasen, in denen sich die Zahl der Anforderungen sprunghaft ändert. Erstaunlicherweise nimmt die Zahl der Anforderungen nach einer rapiden Wachstumsphase oft leicht ab. Der Hintergrund für diese Beobachtung lässt sich leicht deuten: Das starke Wachstum ist auf Fusions- oder Übernahmeprozesse und das Abflachen danach auf die Konsolidierungsphasen zurückzuführen.

Tab. 4.6: Monatliche Änderungsrate nach Erstellung der Spezifikation

Softwarekategorie	**monatl. Rate**
Informationssystem	1.5 %
Systemsoftware	2 %
Militärsoftware	2 %
Öffentl. Hand	2.5 %
Betriebsw. Software	3.5 %

Mit Blick auf die Evolutionsräume, s. Abb. 4.1, lässt sich feststellen, dass sprunghafte Anforderungsevolution eine Konsequenz aus Evolutionen im Bereich der Organisation ist, während umgekehrt Veränderungen des Legacysystems zu langsamerer Evolution der Anforderungen führen. Die organisatorischen Veränderungen führen zu den Co-Evolutionskaskaden, s. S. 185, wobei die Effekte hier enorm sein können, da das Problem sich mit der Zahl der Evolutionsräume potenziert.

[27] In der Regel erhöhen neue Funktionalitäten die Menge an Sourcecode.

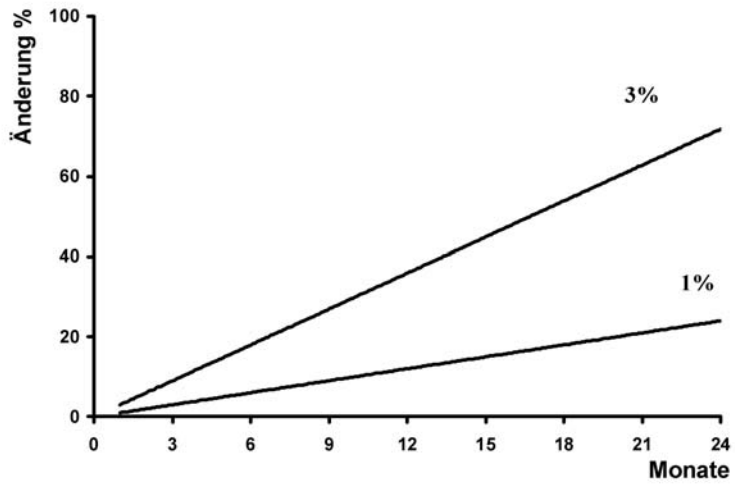

Abb. 4.16: Die Änderungen der Spezifikation in Prozent

Gemessen wird eine solche Entwicklung anhand des „Maturity Index“. Dieser ist definiert als der Quotient der Änderungen pro Release:

$$\mathcal{M}_{Maturity} = 1 - \frac{n_{changed}\,(\text{Anforderung})}{n_{total}\,(\text{Anforderung})}, \tag{4.12}$$

wobei $n_{total}\,(\text{Anforderung})$ die Zahl der Anforderungen im betrachteten Release und $n_{changed}\,(\text{Anforderung})$ die Zahl der geänderten Anforderungen ist. Typischerweise gilt für den Maturity Index eines Legacysystems:

$$\frac{4}{5} \leq \mathcal{M}_{Maturity} \leq 1.$$

Die Legacysysteme zeigen in Bezug auf die Anforderungen, die in ihnen schon realisiert worden sind, eine erstaunliche Stabilität. Hintergrund dafür ist, dass die bestehende implementierte Funktionalität meistens ausreicht, sonst hätte die Legacysoftware überhaupt nicht das jetzige Alter erreicht. Die statistischen Zahlen dazu sind:

- Neuerung: 59%,
- Löschung: 35%,
- Änderung: 5%.

Die geringe Anzahl von „echten“ Änderungen zeugt von einem hohen Maß an Effektivität im Legacysystem.

Die Anforderungsevolution taucht auch im Rahmen jedes Reengineering-projekts auf. Bei der Herstellung neuer Software wird die Anforderungsevolution oft auch als „Requirements Creep“ bezeichnet und zwar dann, wenn diese Evolution nach der Fertigstellung der Spezifikation stattfindet. Diese Form der Anforderungsevolution ist, für die jeweilige fachliche Domäne, s. Tab. 4.6, bzw. Abb. 4.16, sehr unterschiedlich. Bei betriebswirtschaftlicher Software, so beispielsweise bei einem ERP-System, ergibt sich nach einem Jahr eine Änderungsrate von $m^{12} \approx 1.035^{12} \approx 51\%$. Aber nicht nur fachliche Inhalte beeinflussen die Software; einer der verbreitetsten Faktoren für „Feature Creep“ ist Technologie, nämlich die vorzeitige Verwendung von unreifer Technologie. Solche neuen „Silver Bullets“, s. Abschn. 13.15, werden im Zwei-Jahres-Rhythmus von den Herstellern oder, bei Methoden, von den Consultingunternehmen auf den Markt geworfen.[28] Dies kann dazu führen, dass eine Software nie sinnvoll zum Einsatz kommt, da kurz vor Projektende eine neue Technologie oder Methodik[29], ein neues „Silver Bullet“, benutzt werden muss.

4.16 Wertentwicklung

Jedes Legacysystem ist das Resultat einer Investition, welche in der Vergangenheit getätigt wurde. Insofern hat eine Applikation innerhalb eines Legacysystems einen gewissen Wert, welcher sich naturgemäß im Laufe der Evolution verändert, s. Abb. 4.17. Am Anfang des Lebenszyklus ist die Applikation noch eng mit den Geschäftsprozessen verknüpft und hat daher einen hohen Wert, mit zunehmendem Alter werden Anpassungen vorgenommen, um sich weiter an den Geschäftsprozessen orientieren zu können, aber irgendwann einmal kommt der Punkt, wo dies nicht mehr geht. Der Wert der einmal getätigten Investition ist zu diesem Zeitpunkt in Gefahr.

An dieser Stelle ergibt sich die Frage nach dem weiteren Vorgehen: Wird die Applikation abgeschrieben und durch eine neue ersetzt? In diesem Falle würde der Wert des ehemaligen Investments gegen Null gehen. Oder wird eine Transformation, s. Kap. 6, eingeleitet mit dem Versuch, einen großen Teil des einmal getätigten Investments zu erhalten?

4.17 Komplexitätskosten

Wie schon im Abschnitt 4.4 zu sehen war, steigt die Komplexität der Software stets an. Daraus lässt sich sofort schließen, dass die Aufwände zur Evolution

[28] Einige Zeitschriften unterstützen dieses Verhalten in dem sie „Buzzwords“ kreieren.

[29] Diese Methodik wird selbstverständlich vom initiierenden Consultingunternehmen perfekt beherrscht wird.

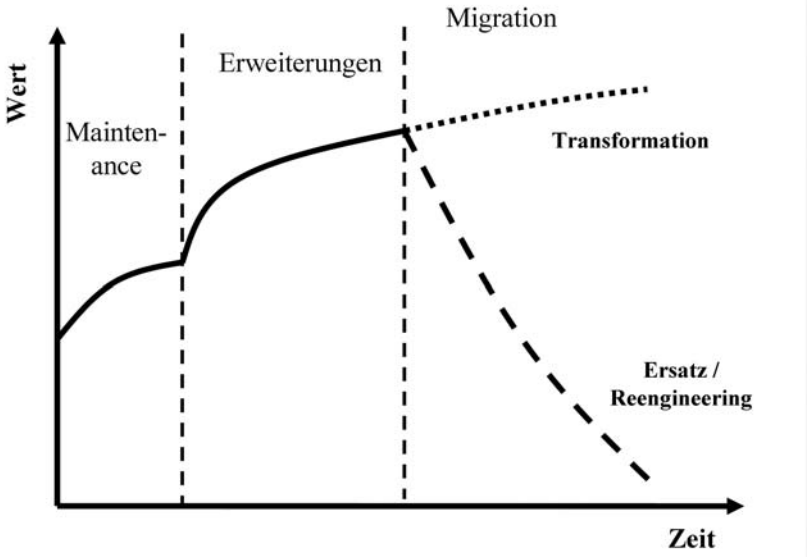

Abb. 4.17: Die Wertentwicklung einer Applikation

der Legacysoftware auch stark ansteigen müssten. Das ist aber, nach empirischen Studien zu urteilen, nicht der Fall. Warum?

Dieser scheinbare Widerspruch löst sich auf, wenn wir betrachten, wie Maintenancebudgets vergeben werden. Sie werden primär als fixe Kosten mit einem festen Zeitbudget über viele Jahre vergeben. Diese Vergabepraxis spiegelt aber nicht die realen Anforderungen bzw. das Wachstum an der zu pflegenden Legacysoftware wider. Die Folge ist, dass die Maintenancemannschaft ausweicht und die erbrachte Leistung reduziert, um in der Lage zu sein, im nachfolgenden Zeitraum das Budget halten zu können. Von daher sieht es nach außen hin so aus, als ob der Maintenanceaufwand unabhängig vom Alter der Legacysoftware sei, allerdings wächst das Maintenancebacklog stetig an.

Die Konstanz hat klare organisatorische Wurzeln, in den meisten Unternehmen wird die Maintenance als operatives und nicht als Projektgeschäft verstanden. Dies hat zur Folge, dass für ein gegebenes Legacysystem die jeweiligen Zuständigen die Wahl haben aus den Parametern:

- Erweiterung der Funktionalität
- Dauer der einzelnen Maintenanceaufgaben
- Zahl der Mitarbeiter für die Legacysoftware
- Zahl der gelösten Probleme
- Budget

Meistens suchen sie sich willkürlich Parameter aus und optimieren danach das Legacysystem.

Da im Rahmen der Planung das mögliche Budget bzw. die Kostenstellen schon im Voraus festgelegt werden, bleibt nur der Spielraum in den anderen Parametern. Dieses Vorgehen erzeugt unternehmensübergreifend den Anschein, als ob der Aufwand für die Maintenance eines Legacysystems unabhängig vom Alter sei.

Zu dieser empirischen Beobachtung trägt noch Folgendes bei: Je jünger, gerechnet ab der Ersterstellung, eine Legacysoftware ist, desto rapider wird die Erhöhung des Volatilitätsindexes sein. Dies liegt darin begründet, dass „jüngere“ Legacysysteme eine ab initio höhere Komplexität besitzen. Dieses Komplexitätsmaß kann von „alten“ Legacysystemen erst nach einer gewissen Zeit erreicht werden. Wir beobachten jedoch eine Momentaufnahme aus heutiger Sicht mit der Folge, dass die Ursprungskomplexität und die aktuelle Volatilität selten in die empirische Kostenbetrachtung eingeht und damit das eigentliche Bild verfälscht.

4.18 Datenqualität

Es gibt faktisch keine Legacysysteme, bei denen sich die Benutzer nicht über die Qualität der enthaltenen Daten beschweren bzw. bei denen diese Qualität fast gar nicht oder nur rudimentär bekannt ist. Typische Eigenschaften im Bereich der schlechten Datenqualität sind:

- Dubletten, d.h. doppelte Entitäten.
- Unmögliche Werte, d.h. Verletzung von Domänen, beispielsweise für das Attribut Geschlecht einer Person:
  1 ist weiblich
  2 ist männlich
  3 nicht bekannt
  4 ??
- Fehlende Werte, in den meisten Fällen wird dann das Feld mit Blanks oder Low-Value besetzt.
- Nichteindeutige Schlüssel, obwohl diese eindeutig sein sollten.
- Verletzte Geschäftsregeln, beispielsweise ein sechsjähriger Angestellter mit 28 Kindern, der seit 25 Jahren im Unternehmen ist.
- Falsche Datentypen, so beispielsweise Charakter in numerischen Feldern.

Die IT-Abteilungen haben über Jahrzehnte hinweg den Benutzern bei der Erzeugung der schlechten Qualität „geholfen“. In den meisten Fällen ist schlechte Datenqualität nicht auf bewusste Aktionen der Benutzer zurückzuführen, sondern auf die Interaktion der Softwareentwicklung mit den jeweiligen Fachabteilungen.

Die schon immer existente Unfähigkeit von Softwareentwicklungsabteilungen, auf Veränderungen und neue Anforderungen aus den Fachbereichen zu reagieren, hat quasi als „Notstandsmaßnahme“ bei den Benutzern dazu geführt, einige Felder anders als im ursprünglichen Sinne zu nutzen oder sie

mehrfach zu interpretieren. Diese spezielle Veränderung der Nutzung wird als „Work-Around“ bezeichnet. Besonders beliebt sind hierbei kryptische Schlüsselschemata, bei denen entweder die Daten verschlüsselt werden, weil der Platz nicht ausreicht, oder ein Attribute in logische Subtypen unterteilt wird, welche zum Teil völlig domänenfremd[30] sind.

Interessanterweise wurde durch dieses Vorgehen ein Faktum von Seiten der Fachbereiche geschaffen, welches die Softwareentwicklung der Unternehmen nicht mehr umgehen konnte. Folglich wurden sehr ausgeklügelte Mechanismen und Programme geschrieben, um bei Änderungen am Legacysystem mit den veränderten Daten leben zu können. Diese Anstrengung war und ist in den meisten Fällen den Fachbereichen nicht bewusst, mit der Folge, dass sich das Verhaltensmuster auf beiden Seiten heute genauso fortsetzt.

Häufig wird dieses Problem von der Unternehmensleitung unterschätzt, da die Kosten nicht transparent sind, bzw. es ja schon immer trotz der schlechten Qualität funktioniert hat. Zwar ist diese Überlegung an sich korrekt, allerdings werden die durch schlechte Datenqualität nicht realisierbaren Umsätze nicht berücksichtigt, von eventuellen Imageverlusten und steigender Mitarbeiterfrustration ganz zu schweigen.

4.19 Architekturevolution

Nicht nur die Anforderungen und der Sourcecode, auch die Architektur bildet einen Evolutionsraum, s. Abb. 4.1. Im Gegensatz zu anderen Evolutionsräumen wird der Architekturraum in der Praxis meist nicht durch die Organisationsevolution bestimmt, sondern ist in der Regel ein Ausdruck für Modeströmungen oder bestimmte technische Limitierungen. Theoretisch sollten Unternehmen eine Architekturgovernance beherrschen und diese auch explizit auf vorhandene Legacysysteme ausüben, doch leider findet dies de facto nicht oder nur sehr selten statt.

Die zunehmende Abweichung einer Legacysoftware von einer einmal definierten Architektur bezeichnet man als Architekturerosion, bzw. die Abweichung von einer in den meisten Fällen hypothetischen Gesamtarchitektur als Architekturvarianz. Beide Erscheinungen haben eine hohe Entropie innerhalb

[30] Bei einem deutschen Touristikunternehmen war es Anfang der neunziger Jahre bei Pauschalreisen üblich, Kinderbetten als Teilnehmer mit dem Namen „Kinderzustellbett“ zu buchen. Die Reisebüros buchten das „Kinderbett“, aber gaben ein Alter unter 2 Jahren an, so dass dieser „Teilnehmer“ nicht bepreist wurde. Dies war die einzige Möglichkeit, die entsprechende Information an die Hoteliers im Zielgebiet weiterzuleiten. Übersehen wurde allerdings, dass im Noncharterbereich nun für „Mr. Kinderzustellbett“ auch Flugtickets für reguläre Airlines bestellt und gedruckt wurden. Die Reaktion des Unternehmens auf die Aufdeckung dieses Problems war es, eine Liste der Kinderzustellbettverträge zu drucken und einen Mitarbeiter damit zu beauftragen, die gedruckten Flugtickets vor der Kuvertierung aus der Druckstraße zu entfernen!

der Legacysoftware zur Folge. Die hohe Entropie taucht nicht sofort, sondern verzögert auf. Sie resultiert dann, wenn die Änderungen an der Legacysoftware im Stile der neuen Architektur durchgeführt werden, die restlichen Teile aber in der alten Architektur verharren.

4.20 Mitose

Fast alle Softwaresysteme starten recht klein und wachsen dann zu großen Systemen heran, wenn sie erfolgreich eingesetzt werden. Der Erfolg ist gerade das, was das Legacysystem erzeugt, wenn die Systeme in der Vergangenheit nicht erfolgreich gewesen wären, gäbe es sie heute gar nicht mehr. Allerdings hat der Erfolg des Legacysystems eine andere Konsequenz, neben der evolutionären Entwicklung setzt auch eine „Software-Mitose[31]“ ein. Die Software wird an verschiedenen Stellen des Unternehmens eingesetzt, häufig sogar in völlig unterschiedlichen Ländern. Wenn aber Software an verschiedenen Stellen eingesetzt wird, kommt es unausweichlich zu unterschiedlichen lokalen Änderungen. Eine solche lokale Änderung wird aber an Ort und Stelle oder durch eine Art von Customizing durchgeführt. Auf Dauer dupliziert sich der zentrale Kern und erzeugt zusammen mit den jeweiligen lokalen Veränderungen eine Anzahl von Varianten. Diese immer stärker entkoppelten Varianten lassen das Gesamtsystem als sehr schwerfällig und änderungsresistent erscheinen.

Diese „Software-Mitose“ ist zunächst einmal kein negatives Phänomen, ist sie doch ein Beweis für den Verbreitungsgrad und die Nutzung der Software, also ein Nachweis für ihre Angemessenheit bezüglich der Geschäftswelt. Nach den ersten Erfolgen vermehrt sich der Ruf nach mehr und mehr lokalen Veränderungen oder Veränderungen am Kern, diese werden erfüllt und über kurz oder lang entsteht ein System mit sehr hoher Entropie und niedriger Maintainability. Die Software ist unpflegbar geworden. Der Prozess kann durch bestimmte Maßnahmen etwas hinausgezögert werden, aber nach einiger Zeit wird der Ruf nach einem generischeren System von Seiten der Softwareentwickler laut. Die Softwareentwickler beginnen ein Reengineeringprojekt mit der Zielsetzung, alle Varianten in den Kern zu integrieren. Am Ende dieses Projektes wird der nun anstehende Veränderungsstau durch die lokalen Organisationen eingefordert. Zu diesem Zeitpunkt stellen die Softwareentwickler fest, dass das so entwickelte generische System sehr schwer an die gesamten lokalen Änderungen anzupassen ist und erzeugen wieder lokale Clones. Der Zyklus beginnt von vorne. Ein Ausweg aus diesem circulus vitiosus ist die Einführung von Produktlinien, s. Kap. 9.

[31] Als Mitose bezeichnet man den Vorgang der Kernteilung bei Zellen eines Lebewesens. Im Anschluss an diese Kernteilung erfolgt die Durchschnürung des Zellleibs, so dass aus einer Zelle zwei Tochterzellen entstehen.

5

Migration

...
And they would go and kiss dead Caesar's wounds
And dip their napkins in his sacred blood,
Yea, beg a hair of him for memory,
And, dying, mention it within their wills,
Bequeathing it as a rich legacy
Unto their issue.

Julius Caesar,
William Shakespeare

Im Kontext von Legacysystemen ist die Migration der Vorgang des Übergangs von einer Implementierungsform in eine andere Implementierungsform. Die Migration betrifft die Umstellung von:

- der Einstellung der Menschen, in der Regel der Mitarbeiter, doch kann man in Spezialfällen auch von einer Kundenmigration sprechen,
- Prozessen,
- Technologien,
- Softwaresystemen.

Im Gegensatz zur Maintenance, s. Kap. 7, muss sich jedoch am System einer der grundlegenden Bestandteile, beispielsweise die Hardware oder die Architektur verändern, damit der Prozess als Migration bezeichnet werden kann. Im Sinne der Entwicklungsbetrachtung sind Migrationen immer revolutionäre Veränderungen, welche im Rahmen einer regulären Evolution nicht darstellbar sind. Ein zweites Charakteristikum für Migration ist der hohe Impact, s. Abschn. 7.8, auf einen großen Teil der Legacysoftware, s. Abb. 5.1.

Für jede Umstellung eines Legacysystems, sei es der Ersatz durch ein COTS-Software-System oder eine vollständige Neuentwicklung, wird immer eine Migration benötigt, sogar das Abschalten eines bestehenden Legacysystems verlangt nach einer Migration.

Vor der Betrachtung von Migrationsstrategien ist es hilfreich, sich den Aufbau eines Unternehmens zu verdeutlichen. Ein aus technischer Sicht recht gutes Modell des Unternehmens ist in dem so genannten Enterprise-Stack wiedergegeben, s. Abb. 5.2.

Auf der obersten Ebene des „Stacks“ sind die organisatorischen und geschäftsprozessuralen Themen angesiedelt. Die darunter liegenden Legacy-

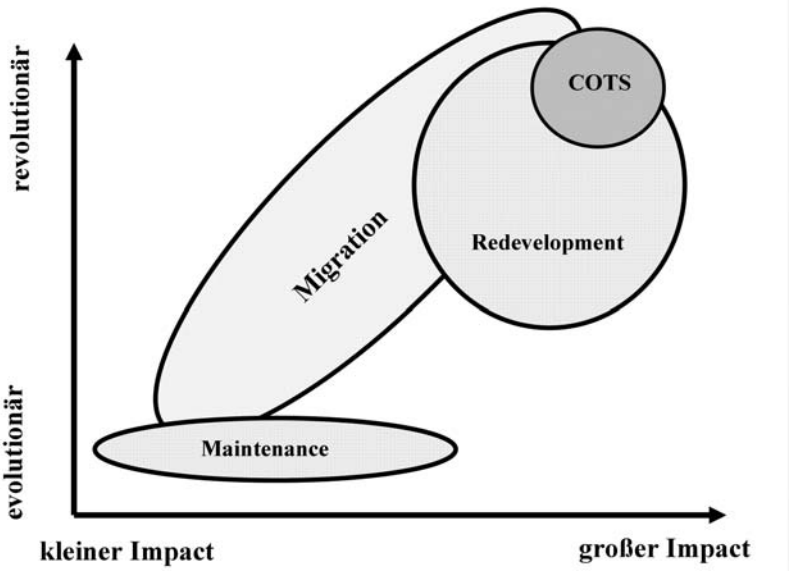

Abb. 5.1: Die verschiedenen Formen der Legacyentwicklung in einer Evolutions- und Impactdarstellung

und COTS-Software-Systeme dienen zur Verwirklichung der Geschäftsprozesse. Aus technischer Sicht besteht der Enterprise-Stack aus folgenden Elementen:

- Die Legacysoftware- und COTS-Software-Systeme – Ihre Aufgabe ist es, die Geschäftsprozesse direkt zu unterstützen. Es sind genau diese Systeme, welche den Fachbereichen direkt zugänglich und damit konkret erfahrbar sind. Änderungen in den Geschäftsprozessen werden mit hoher Wahrscheinlichkeit Änderungen in diesem Teil des Enterprise-Stacks zur Folge haben. Die Konsequenz aus dieser Tatsache ist, dass sich schnell verändernde Geschäftsbereiche in der Lage sein müssen, durch sehr schnelle und flexible Veränderungen der Legacy- und COTS-Software-Systeme zu reagieren. Geschäftsbereiche mit einer niedrigeren Änderungsrate implizieren, auf die Zeit gesehen, stabilere Legacy- und COTS-Software-Systeme. Systeme in diesem Segment werden oft auch als operative Systeme bezeichnet. Operativ, weil die Operationen zur Erreichung des eigentlichen Geschäftsziels durch diese Systeme erst ermöglicht werden.
- Die Applikationsinfrastruktur – Moderne Systeme enthalten eine Reihe von sehr abstrakten Softwaresystemen, welche den Fachbereichen nur indirekt zugänglich sind und die im Grunde hauptsächlich zur Unterstützung der operativen Systeme dienen, ohne dass sie selber Fachlichkeit besitzen. Zu diesem Bereich der Applikationsinfrastruktur zählen:
  - Webserver, wie beispielsweise Apache oder IBM-WebSphere

  - Applikationsserver, wie beispielsweise JBoss, IBM-WebSphere, BEA-Weblogic
  - Middleware, häufig auch Enterprise Application Integration Systeme, aber auch einfachere Kommunikationssoftware, so beispielsweise CORBA, MQ-Series, aber auch CICS, IMS und Tuxedo
  - Datenbanken, rangierend von CODASYL über ISAM, VSAM, IMS bis hin zu relationalen Datenbanken: Oracle, DB2 oder SQL-Server
  - Betriebssysteme, rangierend von MVS, OS/400, z/OS, etliche UNIX-Derivate, aber mittlerweile auch Linux-Derivate und Windows-Systeme. Je älter eine Legacysoftware ist, desto größer ist die Wahrscheinlichkeit, dass sie nur das Betriebssystem als Applikationsinfrastruktur benötigt.
- Computing und Datenspeicherplattformen – Diese Hardwarekomponenten ermöglichen erst die Existenz und Unterstützung der operativen Systeme durch die Applikationsinfrastruktur. Diese Ebene ist üblicherweise ein Sammelsurium an heterogener Hardware von völlig unterschiedlichen Herstellern, mit dem Risiko, dass die darüber liegenden Schichten von Änderungen stark betroffen sein können.[1]
- Netzwerkinfrastruktur – Die Netzwerkinfrastruktur ermöglicht erst die Kommunikation zwischen den einzelnen Systemen. Sie rangiert von langsamen analogen Telefonverbindungen bis hin zu Gigabit-Verbindungen auf Glasfaserbasis. Viele Legacysysteme entstanden vor dem Durchbruch von Ethernet als der De-facto-Netzwerkstandard und benutzen daher intern oft antiquierte Verbindungstechnologien.
- Facilities – Diese Komponente wird oft übersehen, aber ohne die Gebäude und die gesamte Infrastruktur, welche damit verknüpft ist, sind die anderen Ebenen nicht denkbar. Vermutlich wird diese Komponente auf Grund ihrer extremen Stabilität als permanent gegeben angesehen. Projektmanager, welche einmal einen Umzug eines großen IT-lastigen Unternehmens organisieren mussten, wissen, wie aufwändig eine Änderung der Facilities sein kann.

Die Anforderungen von Performanz, Sicherheit, Skalierbarkeit und Verfügbarkeit müssen über alle mittleren Ebenen hinweg sichergestellt sein. Der untere Abschnitt des Enterprise-Stacks besteht aus den Teilen, welche die Infrastruktur verwalten und erstellen. Werkzeuge sind nötig, um die Forderungen nach Performanz, Sicherheit, Skalierbarkeit und Verfügbarkeit mess- und steuerbar zu machen, die Prozesse und Mitarbeiter, um das gesamte Gebilde veränder- und kontrollierbar zu erhalten.

Bei jeder Migration gibt es die Wahl zwischen einer inkrementellen Form oder einer Big-Bang-Migration, s. Tab. 5.1. Generell sind auf Grund der besseren Risikokontrolle inkrementelle Strategien zu bevorzugen. Die einzige Ausnahme von dieser Regel ist der Fall, dass es sich um ein nichtzerlegbares

[1] Ganze Hardwareplattformen, so beispielsweise Honeywell-Bull oder auch CP/M-Maschinen, aber sogar BS-2000-Maschinen von Siemens, sind obsolet oder in der Gefahr, obsolet zu werden.

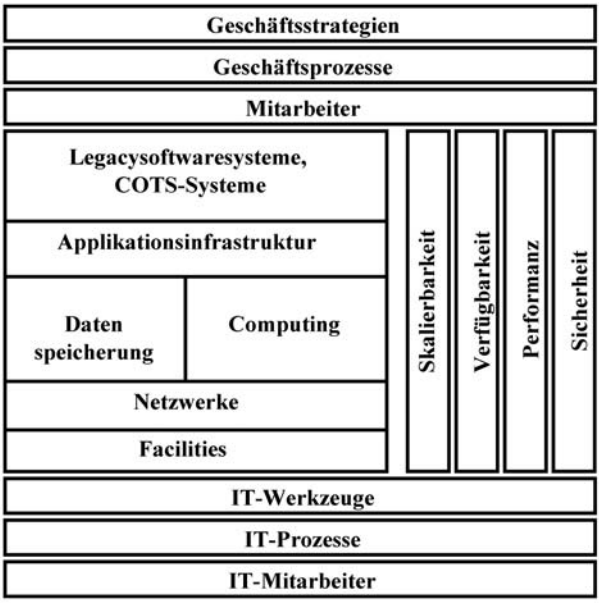

Abb. 5.2: Der Enterprise-Stack, ein Modell für die technische Infrastruktur eines Unternehmens

Tab. 5.1: Vergleich zwischen Big-Bang und inkrementeller Migration

	Big-Bang	**inkrementell**
geeignet	nichtzerlegbar	zerlegbar
Risiko	sehr hoch	kontrollierbar
Risikofall	Gesamtprojekt	Teilprojekt
Ziel	sofort	inkrementell
Planbarkeit	Termin	optimistisch
Gesamtdauer	niedriger	höher
Kosten	niedriger	höher

Problem handelt; dann ist eine inkrementelle Lösung nicht mehr machbar. Typisch für große nichtzerlegbare Probleme sind Deadlineprobleme, beispielsweise das Jahr-2000-Problem, oder die Abschaltung einer ganzen Hardwareplattform. Aber selbst in diesen Fällen lässt sich durch eine gewisse Koexistenzphase das Risiko besser kontrollieren als mit einer Big-Bang-Migration in Reinform.

Obwohl es, speziell aus dem Blickwinkel der Risikovermeidung, gute Gründe für eine inkrementelle Migration gibt, wird oft der Big-Bang-Strategie in der Praxis der Vorzug gegeben. Hierbei spielt die Psychologie eine wichtige Rolle; Big-Bang-Migrationen erreichen recht früh einen Punkt, an dem es für das Projekt keinen Weg zurück mehr gibt. Wenn dieser Punkt erreicht ist, hat

das Migrationsprojekt eine Eigendynamik erhalten, welche sich nicht mehr aufhalten lässt. Bei zögerlich orientierten Organisationen kann eine solche Schaffung von Fakten vorteilhaft sein. Wird die Big-Bang-Migration primär aus organisatorischen Gründen eingesetzt, wird sie in der Literatur als „Burn the Ships"-Strategie[2] bezeichnet.

5.1 Enterprisemigration

Unter dem Begriff Enterprisemigration wird der Prozess der Migration des gesamten Enterprise-Stacks verstanden, nicht nur die Migration der operativen Systeme. Eine solche eingeschränkte Sicht spiegelt die tatsächliche Komplexität, welche in den heutigen Unternehmen herrscht, nicht mehr wider. Die Enterprisemigration hat Auswirkungen auf alle Bereiche des Unternehmens. Zu diesen Auswirkungen zählen:

- Geschäftspraktiken und -politiken, welche im Unternehmen vor der Enterprisemigration angesiedelt waren, müssen überdacht und geändert werden.
- Werkzeuge, welche zur Kontrolle und Verfolgung der operativen Umgebung und Infrastruktur gedient haben, müssen neu bewertet, abgeändert oder ergänzt werden.
- Software, welche primär die operativen Systeme unterstützt hat, sei es COTS-Software oder auch Individualsoftware, muss in die Postmigrationsumgebung überführt werden. In der Regel ist sie Teil der Migration. Es gibt jedoch auch Migrationsprojekte, welche diesen Teil bewusst abspalten, um damit die Komplexität der Enterprisemigration zu reduzieren.
- Die Mitarbeiter müssen auf die neue Umgebung trainiert und geschult werden.
- Die Einführung von Webtechnologien macht bestimmte Terminals oder Emulationssoftware obsolet.
- Änderungen in der Hardware verlangen Änderungen an den Gebäuden, weniger oder mehr Platz, andere Form der Kühlung etc.
- Neue technische Möglichkeiten machen bestehende Prozesse und Verfahren obsolet.[3]

[2] Die Bezeichung „Burn the Ships" stammt aus der Iliade von Homer. Odysseus, der Listenreiche, ließ die Schiffe der Griechen am Strand von Troja verbrennen, um damit ihren Rückzug zu verhindern und sie somit zum Kampf gegen die Trojaner zu zwingen.

[3] Die Hartnäckigkeit von althergebrachten Prozessen und Verfahren ist enorm:
• Es gibt noch heute eine US-Cavalry, obwohl diese zur Zeit sehr wenig Pferde besitzt.
• Eine ganze Zeit lang fuhren auf englischen Elektroloks Heizer mit.
• Bis in das Jahr 1976 gab es in Großbritannien ein Gesetz, welches verlangte, dass die Londoner Taxen einen Ballen Heu dabei haben mussten. Daher rührt die eckige Form des Kofferraums der Londoner Taxen.

5.2 Organisatorische Aspekte

Jede Organisation hat eine lange Erfahrung als Gemeinschaft und wie sie auf Veränderungen reagiert. Diese Geschichte gibt den Hintergrund für die Einstellung eines gesamten Unternehmens und der individuellen Mitarbeiter in Bezug auf jede Veränderung an. In Organisationen, in denen in der Vergangenheit Veränderungen erfolgreich waren und mit dieser Veränderung das Leben jedes Einzelnen in der Organisation vereinfacht wurde, sind allen Veränderungen gegenüber positiv eingestellt; wobei die „Vereinfachungen“ sehr divers sein können, sie rangieren von größerem Erfolg auf dem Markt, mehr Einkommen, mehr Freizeit, erhöhter Motivation bis hin zu sehr individuellen Zielen. Umgekehrt führt jede als negativ empfundene Veränderung zu einem intensiven Misstrauen gegenüber zukünftigen Veränderungen. Die Mitarbeiter in Organisationen haben ein sehr langes kollektives Gedächtnis bezüglich positiver und negativer Veränderung in ihrer Organisation. Dieses kollektive Gedächtnis ist einer der Schlüsselfaktoren für eine erfolgreiche Migration.

Eine interessante Konsequenz aus dieser Beobachtung ist die Feststellung, dass sich erfolgreich verändernde Organisationen auch in Zukunft mit einer höheren Wahrscheinlichkeit verändert werden können[4], während verfehlte Veränderungen eine Stagnation nach sich ziehen.

Wenn die Bereitschaft zur Veränderung nicht sehr verbreitet ist, meist auf Grund von kollektiver negativer Erinnerung, muss besonders viel in Motivation, Marketing und Überzeugungsarbeit bezüglich der anstehenden Veränderung investiert werden.

5.3 Technische Migration

Der Begriff der technischen Migration suggeriert Machbarkeit und Präzision, quasi ingenieurmäßiges Vorgehen. Daher eine Warnung vorweg:

Es ist unmöglich, ohne Verständnis der Domäne zu migrieren!

Damit überhaupt eine technische Migration möglich ist, muss die Domäne verstanden werden.[5] Aber auch die Untermenge der technischen Migrationen kann durchaus unterschiedliche Formen annehmen. Es hat sich als sinnvoll herausgestellt, diese in folgende Formen zu unterteilen:

- Reengineering – Beim Reengineering wird die vorhandene Legacysoftware unter vollständiger Beibehaltung der bestehenden fachlichen Funktionalität neuentwickelt. Das Reengineering kann vollständig oder auch nur partiell sein. Beim partiellen Reengineering wird nur ein Teil des vollständigen

[4] *Nicht die Großen werden die Kleinen fressen, sondern die Schnellen die Langsamen.*

[5] Diese einfache Erkenntnis limitiert den Einsatz von Off- und Nearshoring, s. Kap. 8, für Migrationsprojekte drastisch.

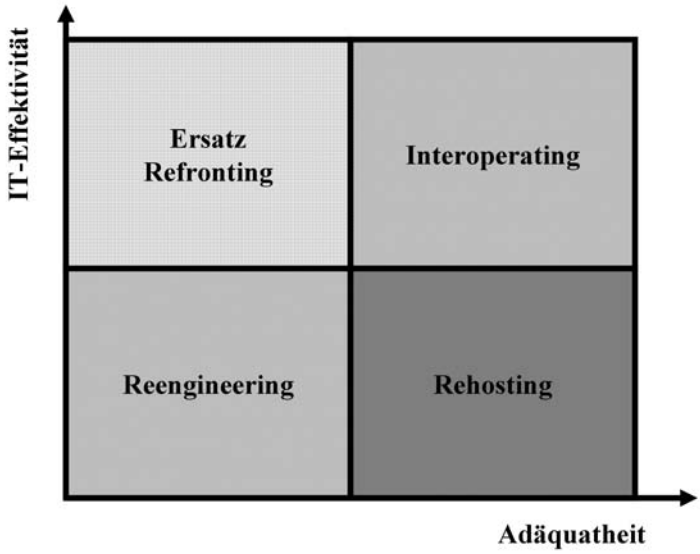

Abb. 5.3: Die verschiedenen technischen Migrationsstrategien in Abhängigkeit der Adäquatheit für die Geschäftsprozesse und der Effektivität der IT-Unterstützung dieser Prozesse

Legacysystems durch etwas Neues ersetzt, was praktisch gesehen die Regel ist. Die Übergänge zum Refronting sind in gewissen Bereichen fließend, allerdings beinhaltet das Reengineering immer einen großen Teil an Softwareentwicklung.

- Rehosting – Der Begriff Rehosting ist faktisch eine Art inverses Reengineering. Die Legacysoftware bleibt größtenteils unverändert, dafür wird aber die Umgebung komplett verändert, so beispielsweise bei einer Migration eines COBOL-basierten Legacysystems von einer obsoleten Hardwareplattform auf eine neue Hardwarelandschaft.
- Interoperation – Unter der Interoperation wird eine Strategie verstanden, bei welcher parallel zum Legacysystem ein neues System aufgebaut wird. Die Legacysoftware wird nicht mehr funktional erweitert, sondern lebt nur noch, bis der entsprechende Geschäftsbereich erfolgreich migriert wurde.
- Ersatz-Refronting – Unter diesem Begriff wird entweder der komplette Ersatz, sei es durch eine COTS-Software oder eine völlig neue Individualsoftware, verstanden. Refronting bezeichnet eine spezielle Technik, bei der nur die bestehende Benutzerschnittstelle ersetzt wird.[6]

[6] Dies wird von Softwareentwicklern oft als „*den Zombie schminken*“ tituliert.

- Transformation – Die Legacytransformation ist die inkrementelle Umwandlung eines bestehenden Systems in eine neue Architektur mit der Maßgabe, möglichst risikoarm zu agieren.
- Freeze – Es besteht auch immer die Möglichkeit nichts zu tun, was als Freeze bezeichnet wird. Der einzige Vorteil hinter einer Freezestrategie ist, dass sie keine direkten Kosten produziert, da das Legacysystem sich nicht verändert. Allerdings können die indirekten Kosten, welche durch fehlende oder falsche Funktionalität entstehen, immens sein.

Doch es bleibt die Frage: Wann sollte welche Migrationsstrategie gewählt werden? Diese Frage lässt sich, in aller Allgemeinheit, leider nur qualitativ beantworten, s. Abb. 5.3. Hier sind die beiden Dimensionen Adäquatheit bezüglich des Geschäftsprozesses und die IT-Effektivität gegeneinander aufgetragen. Beides sind künstliche Dimensionen, welche als Ersatz für schwer messbare Eigenschaften stehen. Die beiden Dimensionen zergliedern sich wie folgt:

- Messgrößen für die Adäquatheit können sein:
  - Die Menge an Zeit, welche gebraucht wird, um neue Funktionen einzuführen oder bestehende zu verändern.
  - Die Einfachheit der Benutzung der Legacysoftware.
  - Die Fähigkeit, die funktionalen Anforderungen zu erfüllen.
  - Die Fähigkeit, ein zukünftiges Wachstum zu unterstützen.
- Als Messgrößen für die IT-Effektivität können dienen:
  - „Total Cost of Ownership“, TCO
  - Technologische Stabilität
  - Qualität der Services
  - Implementationstechnologie

In Abb. 5.3 wird aufgezeigt, welche Migrationsstrategie bei welcher Konstellation am günstigsten ist. Die Qualität der Legacysoftware ist für jede mögliche Form der Wiederverwendung entscheidend, denn eine qualitativ minderwertige Legacysoftware sollte nie wiederverwendet werden. Da die Qualität der Legacysoftware an sich nicht messbar ist, können nur Größen wie:

- Veränderbarkeit
- Flexibilität
- architektonische Kompaktheit
- Maintainability

als Grundlage für die Beurteilung der Qualität der Legacysoftware herangezogen werden. Die Einzigartigkeit der Legacysoftware lässt sich aus dem Grad der Anpassung an die unternehmensspezifischen Geschäftsprozesse ableiten. Generell gilt: Je generischer eine Legacysoftware ist, desto leichter kann und wird sie durch eine COTS-Software ersetzt. Die Legacysysteme, welche in den rechten oberen Quadranten fallen, erfüllen ihren Zweck im Unternehmen und sind gleichzeitig sehr effektiv. Sie sollten im Rahmen der regulären Maintenance, s. Kap. 7, gewartet und erhalten bleiben. Die Systeme, die in den

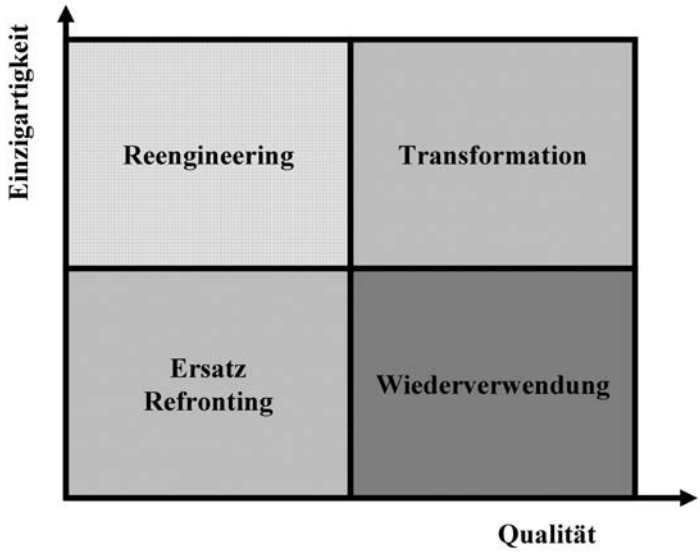

Abb. 5.4: Die verschiedenen technischen Migrationsstrategien in Abhängigkeit von der Qualität der Legacysoftware und der Standardisierbarkeit des Geschäftsprozesses

linken oberen Quadranten fallen, sind zwar sehr IT-effektiv, unterstützen die Geschäftsprozesse jedoch nicht besonders gut. Sie sollten technologisch verbessert werden. Die letzte Kategorie, die beiden unteren Quadranten, sind die besten Kandidaten für eine Migration. Je nach der fachlichen Adäquatheit sollte ein Rehosting oder ein Reengineering vorgenommen werden.

Mit der gewählten Definition von Migration, S. 87, ist man in der Lage, eine breite Palette von Kontexten für die Migration zu untersuchen. Dazu zählen:

- Die Migration eines COTS-Software-Systems, s. Kap. 10, von einer Betriebsplattform auf eine andere Plattform.
- Die Migration von einer Datenbank auf eine andere Datenbank, zum Teil auch von Datenbanken mit drastisch unterschiedlicher Technologie, beispielsweise von einem hierarchischen Datenbanksystem zu einem relationalen Datenbanksystem.
- Die Migration der Legacysoftware von einer Plattform auf eine andere, oder von einem Legacysystem zu einem neuen Softwaresystem, sei es ein verändertes Legacysystem oder ein COTS-Software-System oder ein neues System bestehend aus Individualsoftware.

5.4 Softwareentwicklungsstrategien

Damit eine Migration überhaupt möglich ist, muss sich die Software entwickeln. Die verschiedenen Konzepte rangieren von einfachem Wrapping über Refronting, auch als Screenscraping bezeichnet, bis hin zum kompletten Reengineering. Auch der Ersatz durch ein COTS-Software-System ist im Grunde eine Softwareentwicklungsstrategie. In den meisten Fällen ist ein komplettes Reengineering sowie ein COTS-Ersatz sehr risikoreich und kostspielig.

5.4.1 Wrapping

Der Begriff Wrapping[7] bedeutet, ein bestehendes Legacysystem mit einer Hülle zu umgeben und es damit an neue Operationsformen oder an neue Oberflächen anzupassen. Wird die Oberfläche ersetzt, so spricht man von einem Refronting. Ein Wrapper lässt immer den „gewrappten“ Teil des Systems unangetastet. Folglich bleibt dieser in dem vorhandenen Zustand zurück.

Der Wrapper stellt nun einem externen System oder Benutzer einen Service zur Verfügung, bei dem die Implementierungsform für den Nutzer gleichgültig ist. Insofern hat das Wrapping den Vorteil, dass eine sehr gut getestete Komponente weiterhin genutzt werden kann. Da die meisten funktionalen Wrapper relativ dünn sind – Ausnahmen sind hier die Refronting-Wrapper – erhöht sich die Entropie der Legacysoftware nur minimal. Auch der Maintainability Index bleibt in den meisten Fällen relativ konstant.

Üblicherweise werden, je nach ihrem Verwendungsgebiet, vier unterschiedliche Wrappertypen klassifiziert:

- Datenbankwrapper – Sie haben eine „Gatewayfunktion“ zu den alten Legacydaten hin. Allerdings existiert auch der umgekehrte Fall, ein Legacysystem, welches eine hierarchische Datenbank nutzt. In manchen Fällen kann diesem Legacysystem ein spezieller Wrapper zur Verfügung gestellt werden, der die hierarchischen Anfragen auf relationale Abfragen in einer relationalen Datenbank abbildet und so eine neuere Technologie in Gestalt einer älteren kapselt.
- Systemservices-Wrapper – Mit dieser Form des Wrappers werden Systemdienste, meist sehr betriebssystemnah, wie Drucken, Sortieren und ähnliche, der Legacysoftware in ihrer neuen Umgebung zur Verfügung gestellt.
- Applikationswrapper – Die Applikationswrapper umhüllen ganze Applikationen oder Transaktionen, s. Abb. 5.5. Damit können neue Systeme oder neue Oberflächen die bestehende Legacyfunktionalität weiterhin nutzen, ohne die Implementierung zu kennen.
- Funktionswrapper – Die Funktionswrapper bieten eine einzelne Funktion an, so beispielsweise ein Scoring oder eine Kreditberechnung. Damit werden nur Teile eines Systems gewrappt.

[7] Zu deutsch: Umhüllen.

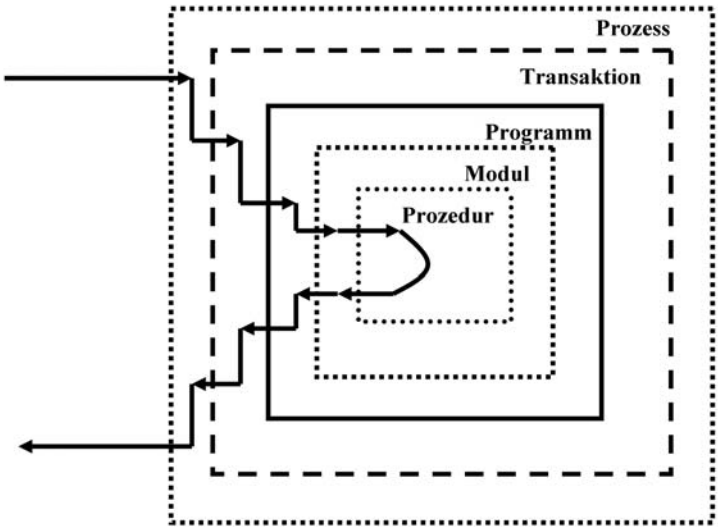

Abb. 5.5: Die verschiedenen Ebenen des Wrappereinsatzes

Neben dieser mehr funktional orientierten Klassifizierung lassen sich Wrapper auch dem Niveau der Granularität quasi orthogonal, s. Abb. 5.5, ihres Einsatzgebietes einteilen:

- Prozess – Wenn ein ganzer Prozess gekapselt wird, so simuliert der Wrapper dem neuen System gegenüber die Verfügbarkeit. Intern nimmt der Wrapper die Daten der externen Schnittstelle entgegen und baut sie als eine Art Job-Input für die interne Schnittstelle zusammen, um anschließend für die Rückrichtung das Gleiche in umgekehrter Reihenfolge vorzunehmen. Dies kann so geschickt gemacht werden, dass das neue System den Aufruf als eine Online-Verbindung empfindet.
- Transaktion – Ein Wrapper auf der Transaktionsebene spricht eine Legacytransaktion an, aber anstelle der üblichen Maskenaus- und -eingabe wird nur der Datenstrom genommen und dem neuen System als eine Art Schnittstelle zur Verfügung gestellt. Die 3270-Wrapper sind eine Abart hiervon, da in diesem Falle der 3270-Datenstrom nicht angezeigt, sondern dieser direkt von einem Programm interpretiert wird. Diese Form der Wrappertechnik ist nicht auf Legacysysteme beschränkt; so existieren heute Wrapper, die HTML-Seiten dynamisch auslesen und den Inhalt der HTML-Seite anderen Systemen als Funktion zur Verfügung stellen.
- Programm – Beim Wrappen eines Programms bekommt dieses eine neue Input/Output-Schnittstelle, d.h. das Programm kann auch von anderen als dem originären Legacysystem aufgerufen werden.

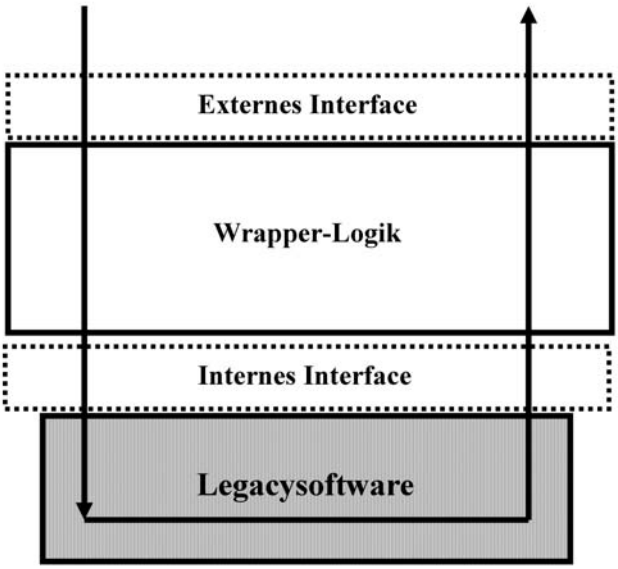

Abb. 5.6: Die Anatomie eines Wrappers

- Modul – Bei einem Modulwrapper bleibt die Unterprogrammschnittstelle erhalten, das neue System kann das Legacyprogramm einfach mit-„linken“. Die meisten der reversen Wrapper, d.h. Wrapper, welche der Legacysoftware Funktionen eines neuen Systems zur Verfügung stellen, fallen in diese Kategorie.
- Prozedur – Das Wrappen auf der Prozedurebene ist wohl das Schwierigste, da jetzt eine einzelne Prozedur gegen eine neue Funktion ausgetauscht wird. Da in der Regel Prozeduren sehr viel häufiger und vor allem mit einem globalen Kontext gerufen werden – in diesem Fall werden die Daten nicht direkt übergeben, sondern stehen in einem globalen Speicherbereich zur Verfügung – stellt sich dies als recht aufwändig und komplex dar.

Die Anatomie eines Wrappers in allgemeiner Form, s. Abb. 5.6, ist recht einfach. Er besteht aus einem externen und einem internen Interface mit einer Art Wrapperlogik zwischen diesen beiden, welche der externen Seite eine Sicht und der internen Seite eine andere Sicht simuliert; dadurch werden beide Systeme voneinander isoliert. Welche konkrete Form für die Wrapperlogik gewählt wird, hängt von der Granularität des Kontextes, s. Abb. 5.5, sowie dem Aufruf bzw. den Kommunikationsmöglichkeiten des Legacysoftware- bzw. des Neusystems ab.

Refronting

Viele Legacysysteme besitzen eine exzellente Funktionalität, jedoch eine für heutige Maßstäbe schlechte Benutzerschnittstelle. Dateneingaben durchlaufen eine strenge Sequenz von Masken und enthalten oft recht kryptische Kürzel, welche durch die limitierte Oberfläche[8] hervorgerufen wurden. Die Benutzerschnittstelle ist vor ca. 20-30 Jahren als charakter-orientierte Oberfläche entstanden.

Ob diese Oberfläche tatsächlich „schlecht“ ist, mag bezweifelt werden, da erfahrene Benutzer mit einer solchen Oberfläche eine sehr hohe Eingabegeschwindigkeit erreichen können. Allerdings ist die Lernkurve relativ flach, und solche Legacysysteme sind für Anfänger nur schwer verständlich. Umgekehrt wird eine mausorientierte Oberfläche bei den langjährigen Benutzern der Legacysoftware zu großen Frustrationen führen, da nun die subjektive Performanz drastisch nachgelassen hat. In den letzten Jahren ist eine Renaissance der 3270-Oberflächen zu beobachten, einige wenige Fachbereiche fordern mittlerweile wieder Software mit den „alten“ Oberflächen.

Unabhängig von der Diskussion, ob die Benutzerschnittstelle gut oder schlecht ist, werden meistens Forderungen nach einer windows-ähnlichen[9] Oberfläche laut. In diesem Fall kann es eine gute Strategie sein, die fachliche Funktionalität vollständig beizubehalten und nur die Benutzerschnittstelle auszutauschen; dies wird als Refronting bezeichnet. Unter dem Begriff „screen scraper“ oder „GUI enabler“ existieren eine Reihe von COTS-Software-Komponenten, die es ermöglichen, klassische charakter-orientierte Bildschirme windows-ähnlich zu gestalten. Dies geht heute so weit, dass diese „GUI-Enabler“ HTML-Code für Webserver generieren, so dass auch modernere Architekturformen unterstützt werden können. Allerdings wird immer eine Architekturform benötigt, die es erlaubt, Teile des alten Legacysystems als Backend zu benutzen. Die Folgen des Einsatzes von Refronting sind nicht zu unterschätzen, denn die zusätzliche Komplexität macht sich durchaus bemerkbar. So gilt für die entstehende Entropie des Gesamtsystems:

$$S_0 \leq S_0' + S_{Screenscraper} \leq S'.$$

Auch der Maintainability Index, s. Gl. 2.40, sinkt ab, da sich das Halsteadvolumen Θ_V, wie auch die anderen metrischen Größen, speziell γ, erhöhen. Dies hat zur Folge, dass die so entstandene Software schwerer zu pflegen ist.

Das Refronting ist zwar eine Reaktion auf veränderte Gewohnheiten der Benutzerschnittstelle und wird von den entsprechenden Werkzeugherstellern als „lebensverlängernd“ verkauft, die drastische Absenkung des Maintainability Index, sowie die starke Erhöhung der Entropie lassen jedoch nur den

[8] Die Oberfläche besteht aus 32×80 Zeichen.

[9] Ob sich hinter dem Begriff „windows-ähnlich“ Windows 95, Windows 98, NT, Windows 2000, Windows XP, ..., verbirgt, ist in der Regel unklar; meist ist eine Bedienung mit der Maus und einer ereignisorientierten Benutzerführung gemeint.

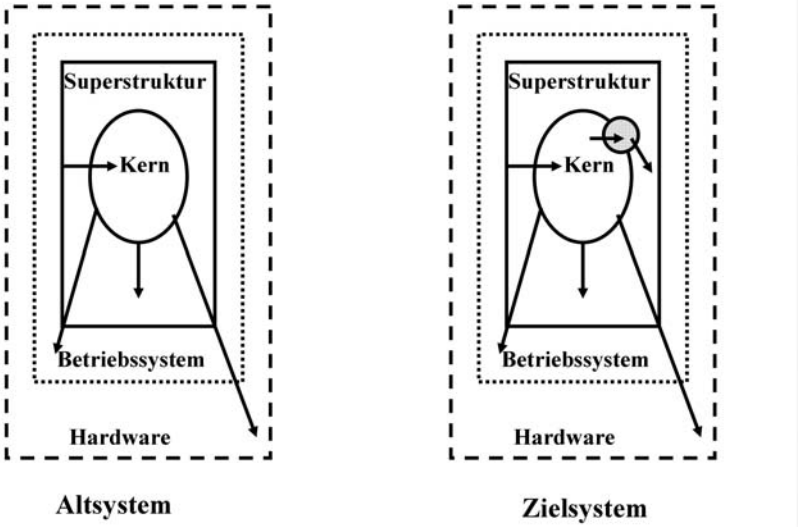

Abb. 5.7: Das Unwrapping als Loslösung vom Altsystem und Migration in ein Zielsystem. Die „warts“ sind schattiert dargestellt

Schluss zu, dass die Volatilität des so veränderten Legacysystems drastisch zugenommen hat, mit der Folge, dass die Restlebensdauer sich verkürzt! Obwohl diese Verkürzung auf den ersten Blick kontraproduktiv ist, kann das Refronting eine gute Strategie sein, wenn langfristig ein komplettes Reengineering oder ein COTS-Ersatz angestrebt wird.

5.4.2 Unwrapping

Der Begriff des Unwrappings steht für die exakt gegenteilige Vorgehensweise wie beim Wrapping. Korrekterweise wird ein Unwrapping fast immer von einem Wrapping gefolgt. Die Basisidee hinter dem Unwrapping ist, sich ein bestehendes Modul aus Legacycode in zwei Teilen aufgebaut vorzustellen, s. Abb. 5.7:

- Kerncode,
- Superstrukturcode.

Hinter dem Kerncode verbirgt sich die Implementierung einer fachlichen Funktionalität. Der Kerncode enthält praktisch den „computational core“ des Moduls. Die Superstruktur, welche den Kerncode enthält, ist die Adaption der fachlichen Funktion an die konkrete Technologie, in welcher die Legacysoftware entstanden ist. Häufig ist es viel zu teuer oder risikoreich, die alte Umgebung beizubehalten oder das fachliche Modul völlig neu zu entwickeln. In

diesem speziellen Falle – dieser Fall ist der Portierung sehr verwandt – lohnt sich das Unwrapping. Die Superstruktur muss von dem Kern isoliert werden und wird anschließend im Zielsystem nachgebildet. Dies ist in der Praxis ein recht schwieriges Unterfangen, da es sehr schwer ist, den hypothetischen Kern von der Superstruktur zu unterscheiden. Daher wird in schwierigen Fällen oft auch ein Teil der Superstruktur in die neue Umgebung mitgenommen. Die Teile der alten Superstruktur, welche sich in der neuen Superstruktur wiederfinden, nennt man Warzen, „warts“. Solche Warzen erhöhen systematisch die Entropie und Komplexität bei gleichzeitiger Absenkung des Maintainability Index.

5.4.3 COTS-Ersatz

Die Refrontingstrategie ist eine Variation der Ersatzstrategie. Bei der Ersatzstrategie kann die Legacysoftware durch eine Reihe von COTS-Software-Systemen ersetzt werden, s. Kap. 10. Allerdings setzt dies voraus, dass das neue System auch auf der vorhandenen Infrastruktur lauffähig ist.

Eine andere Möglichkeit ist es, anstelle des COTS-Ersatzes eigene Software als Ersatz zu entwickeln. Eine solche Vorgehensweise ist, wie die Erfahrung zeigt, s. S. 105, mit einem sehr hohen Maß an Risiko verbunden. Es gibt in der Literatur unzählige Bücher über diese Form der klassischen Softwareentwicklung.

Eine dritte Möglichkeit ist der Teilersatz. Dieser ist am effektivsten, wenn Bereiche der Applikationsinfrastruktur ersetzt werden. Hierbei wird die Sedimentation, s. S. 259, von COTS-Software genutzt. Dabei wandert immer mehr Supportfunktionalität in immer tiefere Schichten. Der Ersatz als Strategie erzeugt das höchste Einsparungspotential bei gleichzeitig höchstem Risiko, aber Ersatz als Mittel zum Zweck bei einzelnen Komponenten ist eine risikoarme, kostengünstige Strategie.

5.4.4 Rehosting

Beim Rehosting bewegt sich das vollständige Legacysystem von einer Umgebung in eine andere, ohne eine Veränderung der Funktionalität. Für die Legacysysteme kann dies auf drei unterschiedliche Arten, je nach Umgebungstyp und Legacysystem, durchgeführt werden:

- Recompilation – Eine Applikation oder auch ein komplettes Legacysystem kann in eine neue Umgebung portiert werden. Da die neue Umgebung eine Binärkompatibilität verlangt, müssen neue Binärobjekte erzeugt werden[10]. Anpassung an die neue Umgebung reduziert sich auf die Anpassung an das neue Betriebssystem, da ja der Sourcecode der Legacysoftware in der

[10] In einer .NET-Umgebung oder einer Java-Umgebung ist das nur bedingt erforderlich.

Regel vorhanden ist und es in den meisten Fällen auch die entsprechenden Compiler im neuen System gibt. Falls dies nicht zutreffen sollte, kann Emulation ein Ausweg sein.

Die Betriebssystemabhängigkeit zur alten Umgebung kann auf zwei Arten umgangen werden:

- Einsatz einer Kompatibilitätsbibliothek – Hierbei wird eine Softwarebibliothek erworben, welche die „alten“ Aufrufe als Funktionen mit den bekannten Schnittstellen enthält und in der neuen Umgebung zur Verfügung stellt. Diese Softwarebibliothek kann so mächtig sein, dass eigene Laufzeitumgebungen für die Legacysysteme eingesetzt werden.[11] Oft werden solche Bibliotheken von Herstellern angeboten, so bietet beispielsweise Sun eine HP/UX-Kompatibilitätsbibliothek an.
- Code Transformation – Es gibt eine ganze Reihe von Werkzeugen, welche den Sourcecode auf die neue Umgebung transformieren können. Im Gegensatz zur Kompatibilitätsbibliothek ist hier ein Fallback meist nur schwer möglich.

Die neu übersetzten Programme werden dann „gelinkt“ und in der neuen Umgebung ausgeführt.

- Emulation – Unter Emulation wird die Simulation der alten Umgebung innerhalb der neuen verstanden. Obwohl hierdurch eine neue Softwareschicht eingebracht wird, welche theoretisch die Performanz verschlechtert, kann der Gesamteffekt, bedingt durch eine sehr viel schnellere Zielhardware, durchaus ein Performanzgewinn sein.[12] Ein Vorteil ist, dass keinerlei Recompilation nötig ist. Manchmal sind bei Legacysystemen zwar noch die Quellen, aber nicht mehr alle Bestandteile der Entwicklungsumgebung vorhanden. In diesem Fall ist Emulation eine echte Alternative. Ein anderer Fall kann sein, dass Entwicklungsumgebungen und Compiler auf der neuen Plattform einfach zu teuer sind und daher ein emulativer Weg einzuschlagen ist. Emulationen tauchen oft auch in Testumgebungen auf, so beispielsweise als VMWare, wo auf einer Linux-Umgebung diverse Betriebssysteme emuliert werden können.[13]
- Technologieportierung – Bei der Technologieportierung hat das Zielsystem die Fähigkeit, den Sourcecode der Legacysoftware auszuführen. Typisch für eine solche Technologieportierung ist der Austausch der relationalen Datenhaltung.

[11] Die Firma Acucorp, ein Hersteller von COBOL-Compilern, bietet Kunden an, sich eigene Laufzeitumgebungen für COBOL-Objekte zu bauen.

[12] Ironischerweise müssen DOS-Emulatoren auf neueren PCs eine sehr viel niedrigere CPU-Taktrate für alte Spiele simulieren. Space-Invaders lässt sich ohne dieses „Downgrade“ nicht auf einem modernen PC spielen. Die Spielgegner waren für eine 8-MHz-Umgebung konzipiert, auf einer 3-GHz-Umgebung sind sie 400mal so schnell ...

[13] Korrekterweise wird ein virtueller Rechner simuliert.

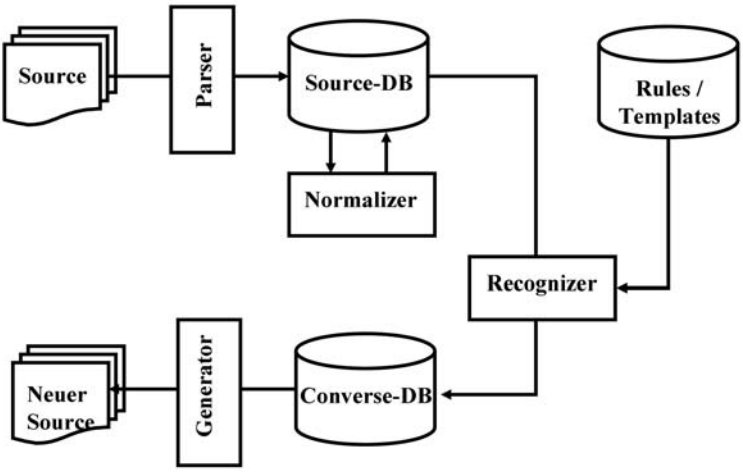

Abb. 5.8: Schematischer Ablauf einer automatischen Portierung

Rehosting liefert schnelle Ergebnisse, ohne ein hohes Risiko zu erzeugen, da die bestehende Funktionalität nicht angetastet wird. Der Haupteffekt sind die niedrigeren Kosten oder die Zukunftssicherheit der neuen Plattform. Rehosting gehört zu den schnellen Lösungen, es ist dann eine sinnvolle Alternative, wenn entweder die Zeit oder die Ressourcen für ein Reengineering nicht ausreichen oder die Legacysoftware nur noch eine endliche Lebenserwartung besitzt.

Allerdings gibt es beim Rehosting auch Dinge zu beachten: So kann sich beispielsweise bei einer Umstellung vom EBCDIC[14]- auf einen ASCII[15]-Zeichensatz, wie es bei einer Umstellung von einer IBM-Mainframe auf einen Unixrechner üblich wäre, die Sortierreihenfolge innerhalb der Programme verändern.[16] Ein viel größeres Problem ist die Portierung der Umgebung, speziell die der Batch-Umgebung. Auf einer Mainframe existiert eine eigenständige Batchsprache, JCL, welche den Batchbetrieb sicherstellt. In das Betriebssystem integrierte Subsysteme mit ähnlicher Mächtigkeit sind innerhalb der Windowswelt praktisch unbekannt und auf den Unixplattformen nur bedingt verfügbar. Da die meisten Legacysysteme sehr batchlastig sind, stellt dies ei-

[14] Abkürzung für Extended Binary Coded Decimal Interchange Code. EBCDIC-Codierung ist typisch für eine IBM-Mainframe.

[15] Abkürzung für American Standard Code for Information Interchange. ASCII ist die Zeichencodierung auf Unix und PCs.

[16] Für EBCDIC gilt 'A9' < '1G' während für ASCII '1G' < 'A9' ist!

ne hohe operative Hürde jenseits der Code-Kompatibilität, s. Abb. 5.8, dar. Ein weiteres Problem einer Portierung ist, dass die Mitarbeiter meistens die Ursprungsplattform besser und intensiver kennen als die Zielplattform. Dies führt dazu, dass korrigierende Maßnahmen auf Seiten der Zielplattform deutlich mehr Kosten[17] verursachen als meistens antizipiert wird. Von daher ist es sinnvoller eine Art „Scaffolding“ im Ursprungssystem zu betreiben, d.h. so viel wie möglich dort für die Portierung vorzubereiten, um zum einen die automatische Portierung zu vereinfachen und zum anderen möglichst wenige Änderungen in der Zielumgebung nach der Umsetzung vornehmen zu müssen. Dies kann sogar so weit gehen, dass Sourcecode entsteht, der nur dazu dient die Portierung zu vereinfachen.

5.4.5 Interoperation

In manchen Fällen ist es ratsam eine Legacysoftware überhaupt nicht anzutasten; dann muss die Interoperabilität sichergestellt werden. Ein solches Vorgehen kann sinnvoll sein, wenn:

- die Legacysoftware sehr effizient und adäquat ist und gleichzeitig mit der neuen Umgebung wechselwirken kann.[18] In einem solchen Fall sind die möglichen Risiken einer anderen Strategie einfach viel zu hoch.
- vertragliche Situationen, so beispielsweise Leasing oder Outsourcing zu diesem Schritt zwingen. Dies ist eines der Risiken des Outsourcings! Auf lange Sicht werden „outgesourcte“ Legacysysteme die „Parias[19]“ innerhalb der eigenen IT-Infrastruktur. Sie sind weder vollständig integriert noch haben sie eine adäquate Maintenancemannschaft und sie laufen auf einer veralteten Hardware.

5.5 Maintenanceende

Die Software erreicht nach langer Maintenance irgendwann einmal den Zeitpunkt, wo es sich nicht mehr lohnt die Software weiter zu warten. Die Software hat das Ende ihres Lebenszyklus erreicht, s. Kap. 4. Wenn dieser Zeitpunkt erreicht ist, stellt sich die Frage: Was tun?

---

[17] Consultingunternehmen, die sich auf Portierungen spezialisiert haben, schlagen gerne eine 1:1-Umsetzung vor, da sie genau wissen, dass der Kunde nicht die nötigen erfahrenen Ressourcen hat um in der Zielumgebung effektiv arbeiten zu können. Folglich muss das portierende Unternehmen Aufträge an das Consultingunternehmen vergeben.

[18] Unter den Softwareentwicklern gibt es das geflügelte Wort: „Never touch a running system . . . “

[19] Ein treffender Begriff, wenn man an die vielen Outsourcingversuche mit indischen Unternehmen denkt.

Die prinzipielle Entscheidung, welche getroffen werden muss, ist die, ob die Software komplett neu geschrieben werden soll, ob sie durch eine COTS-Software, s. Kap. 10, ersetzt werden kann oder ob sich ein Reengineering lohnt. In fast allen Fällen stellt das Komplett-neu-Bauen[20] ein Vorgehen mit einem sehr hohen Risiko dar. Neben den mangelnden Spezifikationen und fehlenden Dokumentationen[21] ist der Ressourcen- und Zeitverbrauch enorm. Im Umfeld kommerzieller Softwarehersteller kann dieser hohe Verbrauch dazu führen, dass das Softwareunternehmen von Konkurrenten aus dem Markt gedrängt wird. Ein gutes Beispiel für diesen Fall ist das amerikanische Unternehmen Netscape, welches versuchte, seinen Netscape-Web-Browser mit der Version 5.0 auf den Markt zu bringen. Die Firma Netscape versagte jedoch dabei, und so gelang es Microsoft, ihrem Microsoft Internet Explorer eine Marktdominanz zu verschaffen. Ursprünglich für Ende 1998 angekündigt, kam die Version 5.0 nie richtig zum Zuge und erst im November 2000, d.h. zwei Jahre zu spät, wurde die Version 6.0 auf den Markt geworfen; erst die Version 7.0, welche Mitte 2002 offiziell freigegeben wurde, hatte die Qualität, wie sie vier Jahre früher erwartet wurde. Ein ähnliches Schicksal war dem Produkt Lotus-123 beschieden. Aber auch Microsoft ging es nicht besser: Der Versuch, das Produkt Word von Grund auf neu zu bauen – das Projekt hatte den internen Namen Pyramid – wurde gestoppt, so dass das heutige Word im Kern noch sehr alte Teile enthält. Im Gegensatz zu Netscape konnte sich Microsoft die Parallelität des Projektes Pyramid und eines anderen Reengineeringprojekts leisten.

5.6 Reengineering

Ironischerweise war das Software Reengineering in der Vergangenheit das „hässliche Entlein“ im Vergleich zum viel dominierenderen Business Process Reengineering. Im Folgenden werden wir unter Reengineering stets das Software Reengineering verstehen. Obwohl beide in der heutigen Welt eng verzahnt sind, hat das Business Process Reengineering in der Vergangenheit stets größere Aufmerksamkeit produziert. Die stärkere Betonung des Business Process Reengineerings hat diverse Ursachen, welche im Management wie auch im Consultingsektor verankert sind.

Neben den unbestreitbaren Vorzügen des Business Process Reengineerings, als Reaktion auf den zunehmenden Wettbewerbsdruck, hat Business Process Reengineering aber auch einige Auswüchse produziert, deren Hauptursachen in der Psychologie bzw. Soziologie von Wirtschaftsunternehmen liegen:

[20] Die Softwareentwickler nennen dies: „Auf der grünen Wiese bauen ...“, bzw. im Englischen: „... to build from scratch ...“

[21] Oft fehlt die Dokumentation ganz bewusst. Eine nicht vorhandene Dokumentation lässt bestehendes Know-how einzelner Mitarbeiter als sehr wichtig erscheinen. Nicht zu dokumentieren ist einer der einfachsten Wege, zu einem „Guru“ für die Applikation zu werden.

- Business Process Reengineering ist beim Management beliebt, da es Aktivität signalisiert. Diese Aktivität wird allen, vor allem dem Aufsichtsrat wie auch den Mitarbeitern und sogar den Kunden offensichtlich als eine aktive Beeinflussung des Unternehmens präsentiert. Die Frage nach der Zweckmäßigkeit dieser Maßnahmen steht oft hinten an. Speziell in Zeiten einer Konjunkturflaute kanalisieren viele Manager ihre Hilflosigkeit und auch ihr Unvermögen die Situation zu bewältigen[22], mit dieser Form des Aktionismus. Was liegt näher, als diesen Aktionismus wissenschaftlich zu untermauern und ihn Business Process Reengineering zu nennen.
- Business Process Reengineering hat immer Managementattention, d.h. Personen, welche über den Einsatz von Geld und anderen Ressourcen entscheiden, sehen Business Process Reengineering als ihr persönliches Anliegen an.
- Viele Consultingunternehmen haben sich auf „organisatorische Beratung" spezialisiert. Für die Consultingunternehmen ist dies ein idealer Sektor, da die Ergebnisse nur sehr schwer vergleichbar sind und eine Verifikation des Erfolges eines Business Process Reengineerings häufig nicht möglich ist; bzw. wenn sich ein Markterfolg herausstellt, so war dies eine Konsequenz des Business Process Reengineerings und im Falle eines Misserfolges war entweder die Umsetzung schlecht oder unvorhersehbare Marktereignisse haben den Erfolg verhindert. Die beteiligten Unternehmensberater sind oft auf Ebene der Vorstände und des höheren Managements tätig und liefern dann die notwendigen „Impulse" in Form von Powerpointfolien[23].

Im Gegensatz zu diesem sehr wichtigen, visionären und impulsgebenden Business Process Reengineering beschäftigt sich das Software Reengineering mit Legacysystemen, welche a priori schon als veralteter Ballast gelten. Die wichtigsten Gründe für diese Einstellung zum Software Reengineering sind verknüpft mit dem Status der Legacysysteme in Unternehmen, bzw. dem Ansehen der Softwareentwicklungsabteilungen:

- Software Reengineering ist eine technische Disziplin, sie gilt als inhärent schwierig und langweilig.
- Software Reengineering beschäftigt sich mit dem „Alten", dem Langsamen, der Dinosauriersoftware. Diese Dinosauriersoftware ist zu langsam für unsere moderne Welt, zu langsam im Hinblick auf Performanz und Flexibilität.

[22] Häufig ist auch das Phänomen der kognitiven Dissonanz, d.h. der verzerrten Wahrnehmung bei Managern in solchen Stresssituationen zu beobachten. Dem Autor wurde von einem Vorstand während einer Besprechung über Vertriebsmaßnahmen einmal gesagt: *„. . . bitte bringen Sie mich nicht mit Tatsachen durcheinander!"*

[23] Die häufige Verwendung von Powerpointfolien hat Unternehmensberatern den Spitznamen Powerpointwarriors eingebracht. Im klassischen Consultinggeschäft der „organisatorischen Beratung" ist der Preis direkt mit der Anzahl der Powerpointfolien korreliert. Es gilt die Faustregel: 500 Euro pro Folie.

- Ein Manager, der etwas Neues erschafft, hat ein höheres Ansehen als einer, der in dem Ruf steht, nur etwas zu erhalten.
- Software Reengineering besitzt nicht den Reiz des neuen Spielzeugs, den Reiz des „höher – schneller – weiter“. Bei unternehmensübergreifenden Treffen wird der Manager, der seine IT völlig neu bestückt, von anderen bewundert. Diese Form des „peer pressure“ führt oft zu Entscheidungen[24], die ohne jede ökonomische Vernunft gefällt werden.

Alle diese Mechanismen und Vorurteile tragen zur allgemeinen Einschätzung der Wertigkeiten von Business Process Reengineering und Software Reengineering bei. Jenseits dieser Vorurteile ist anzumerken, dass es de facto kein Business Process Reengineering in einem Unternehmen ohne Software Reengineering gibt. Dies ist in dem hohen Durchdringungsgrad der Unternehmen durch softwaregestützte Lösungen begründet, so dass heute das eine ohne das andere nicht mehr denkbar ist. Für die meisten Kenner von Unternehmensorganisationen ist es einleuchtend, dass ein Business Process Reengineering häufig in der Veränderung der bestehenden Softwarelandschaft resultiert. Allerdings ist in zunehmendem Maße zu beobachten, dass auch aus einem Software Reengineering der Zwang zu einem Business Process Reengineering resultieren kann. Am eindeutigsten kann dies bei der Ersetzung des vorhandenen Legacysystems, siehe hierzu auch Kap. 10, durch eine COTS-Software gesehen werden. Der Einsatz von großen Enterprise Ressource Planning Systemen wie SAP, Navision oder auch Oracle Finance kann nicht ohne eine Reorganisation[25] der Geschäftsprozessabwicklung vonstatten gehen, was das Ziel eines Business Process Reengineerings ist.

5.7 Business Process Reengineering

Neben der Sichtweise der Struktur- und Ablauforganisation eines Unternehmens hat sich das Prozessdenken in den neunziger Jahren immer stärker entwickelt. Mit der Einführung von Prozessmodellen in den betrieblichen Kontext konnte auch über eine softwaretechnische Unterstützung dieser Prozessmodelle nachgedacht werden.

Eine der Konsequenzen aus der verstärkten Prozessmodellierung ist der noch heute anhaltende Boom der Workflow-Management-Systeme. Obwohl der Prozessgedanke im betriebswirtschaftlichen Umfeld schon relativ alt ist, erzielte er seine durchschlagende Wirkung erst durch die ganzheitliche Analyse mit nachfolgender softwaretechnischer Unterstützung.

[24] Besonders Softwareunternehmen, die vom Verkauf ihrer Lizenzen leben, versuchen den Besitz ihrer Software zu einem Statussymbol für den IT-Manager zu machen.

[25] Der Vertrieb und das Marketing der Hersteller werden nicht müde zu beteuern, dass sich ihre Software an das Unternehmen anpassen könnte. De facto ist das aber kein sinnvolles Vorgehen, da die einzugehenden Kompromisse sowohl für die Software als auch für die Organisation schnell untragbar werden.

Unter dem Begriff des Business Process Reengineerings wird nun das fundamentale Überdenken und das radikale Redesign von wichtigen Unternehmensprozessen verstanden. Idealerweise sollten die Erfolge in Form von Qualität oder Zeit- oder Kostenersparnis messbar sein. Trotz gegenteiliger Beteuerungen sind fast alle Unternehmen reaktiv, d.h. sie starten ein Business Process Reengineering erst dann, wenn der äußere Druck in Form von

- Kundenbeschwerden,
- fallenden Marktanteilen,
- massiver Konkurrenz,
- usw.

so stark zugenommen hat, dass andere Methoden, wie beispielsweise Druck auf die Lieferanten auszuüben, keine spürbaren Resultate mehr liefern. Sobald der Leidensdruck hoch genug ist, wird mit einem Business Process Reengineering[26] begonnen. In den meisten Fällen wird eine Prozessverbesserung durch Vereinfachung und Standardisierung von Prozesselementen erreicht. Eng verwandte Verfahren zum Business Process Reengineering sind:

- Lean Management
- Total Quality Management
- Six Sigma
- Kaizen

Eine weitere Verbesserung der Effizienz kann durch den Einsatz von Software zur Unterstützung oder kompletten Realisierung von Prozesselementen erreicht werden. Genauer betrachtet ist hier auch die Legacysoftware anzusiedeln, da ihr Ursprung in den meisten Fällen der Versuch war, einen „manuellen“ Prozess durch einen Softwareprozess zu unterstützen. Allerdings pflegte man dies zum Zeitpunkt der Erstellung der Legacysoftware nicht Business Process Reengineering zu nennen!

Kritisch sollte allerdings angemerkt werden, dass der Einsatz von Software zur Ersetzung von manuellen Arbeitsschritten nur einen geringen Effizienzschub liefert. Die Software kann auch völlig neue Prozesselemente erstellen, die „manuell“ überhaupt nicht denkbar sind. Erst ein solcher Einsatz, völlig neue Prozesse zu verwirklichen, liefert sehr hohe Effizienzgewinne. Solche „neuen“ Prozesse können im B2C- oder B2B-Sektor liegen. Besonders das starke Aufkommen von Internetshoplösungen im B2C-Sektor zeigt, dass es durchaus möglich ist neue Prozesse zu schaffen.

[26] Im weiteren Sinne kann auch die Aufgabe eines Geschäftsfeldes oder das Outsourcing eines solchen als eine Form des Business Process Reengineerings betrachtet werden.

5.8 Replacement

Eine spezielle Form des Reengineerings ist das Replacement, auch als „Cold Turkey[27]“ bezeichnet. Legacysysteme werden oft mit faktisch identischen Kopien der ursprünglichen Software ersetzt, mit dem Unterschied, dass das Replacement etwas erweiterte Funktionalität besitzt. Durch eine Wiederholung dieses Replacements wird die organisatorische Struktur, welche in der Software wiederzufinden ist, noch verstärkt, s. Abschn. 13.14.

Der Grund für diese Tendenz, einen Ersatz der Software durch ein strukturell gleiches System mit nur geringer funktionaler Erweiterung zu finden, liegt in der Schwierigkeit von organisatorischen Restrukturierungen sowie in der Erfassung der Anforderungen an zukünftige Systeme. Während des Replacements dient das abzulösende System als ein Prototyp für das neue System, die noch nicht durchgeführten Änderungen werden als Change Requests in das neue System aufgenommen, ohne die Spezifikationen oder die Geschäftsprozesse als solche zu hinterfragen! Dieses Vorgehen wird deswegen eingeschlagen, weil es innerhalb der Entwicklung einfacher ist und weil es die Kommunikation durch ein greifbares „Spezifikationsartefakt“, das abzulösende System, sehr einfach macht. Diese Form der Bequemlichkeit macht jede Form von Business Process Reengineering, s. Abschn. 5.7, unmöglich, denn bei diesem Replacement wird in den meisten Fällen übersehen, dass die Endbenutzer des Systems dieses oft anders gebrauchen als ursprünglich intendiert[28] war. Oft wird das Wissen über die nichtintendierte Nutzung von den Anwendern nicht weitergegeben, was bei einem zukünftigen Reengineering zu noch größeren Problemen führt.

In fast allen Fällen ist der „Cold Turkey“-Ansatz viel zu risikoreich. Besondere Risiken im Zusammenhang mit Legacysoftware und dem „Cold Turkey“-Ansatz sind:

- Ein besseres System muss versprochen werden. Kein Fachbereich investiert Geld, wenn er nicht einen subjektiven Gewinn an Funktionalität erhält. Daher werden im Rahmen des neuen Systems oft zusätzliche Funktionen versprochen, welche die Komplexität des Vorgehens wie auch die Entropie der entstehenden Software erhöht.
- Undokumentierte Abhängigkeiten zu anderen Systemen und Prozessen sind bei Legacysoftware die Regel und werden sehr oft übersehen.
- Die Datenmenge des aktiven Legacysystems kann zu groß sein, um überhaupt genügend Zeit für einen „Cold Turkey“-Ansatz zu haben.
- Verzögerungen im Projektablauf werden faktisch nie toleriert.
- Die schiere Größe eines solchen Projektes macht es vermutlich schon an sich risikoreich.

[27] Im Amerikanischen bedeutet *Cold Turkey* ursprünglich, einen Drogenentzug durch sofortiges Absetzen der Droge ohne jede Form von Substituten durchzuführen.

[28] Siehe auch Fußnote, S. 85

5.9 Software Reengineering

Sehr eng mit der Maintenance und einer möglichen Neuentwicklung ist der Begriff des Reengineerings verknüpft. Das Reengineering steht quasi auf halbem Wege zwischen einer vollständigen Neuentwicklung, via einer völlig neuen Spezifikation, und der regulären Maintenance. Unglücklicherweise sind hier die Begrifflichkeiten nicht ganz eindeutig und zum Teil auch fließend. Daher wird in diesem Buch der Begriff des Reengineerings stets im umfassenden Sinn genutzt:

> *Reengineering im weiteren Sinne ist der Prozess der Umstellung von bestehender, in der Regel aktiver, Software auf neue Gegebenheiten.*

Im Gegensatz zur reinen Maintenance beinhaltet das Reengineering stets eine größere Menge von Umstellungen, sei es eine Portierung auf eine neue Hard- oder Softwareplattform oder neue Funktionalitäten. In allen Fällen findet jedoch beim Reengineering im weiteren Sinne eine strukturelle Veränderung der Software statt. Dies steht im Gegensatz zur Maintenance, welche die vorhandene Legacysoftware nicht strukturell verändert.

> *Reengineering im engeren Sinne ist der Prozess der Veränderung der einer Software zugrunde liegenden Architektur.*

Reengineering im engeren Sinne lässt sich somit klar von der klassischen Maintenance unterscheiden. Im Gegensatz zur Neuentwicklung, welche in den meisten Fällen auch eine neue Softwarearchitektur beinhaltet, wird beim Reengineering ein Teil des alten Systems erhalten.

5.9.1 Taxonomie des Reengineerings

Das bekannteste Reengineeringmodell stammt von Byrne, s. Abb. 5.9, und beinhaltet die Blöcke:

- Reverse Engineering
- Restructuring
- Forward Engineering

Mit dem Reverse Engineering wird anhand der vorhandenen Artefakte versucht, ein Modell des Systems zu entwickeln, wobei das Ziel ist, auf einer höheren Abstraktionsebene agieren zu können. Beim Restructuring werden quasi gleiche Abstraktionsebenen ausgehend vom Ursprungssystem in das Zielsystem überführt. Das Forward Engineering stellt wiederum ein Verfahren dar, das analog einer Neuentwicklung ist.

Beim Reengineeringmodell, Abb. 5.9, entspricht der erste Schritt, das Reverse Engineering, dem systematischen Aufbau einer Dokumentation über das bestehende Legacysystem. Dieser Schritt ist einer der schwersten, da, in der

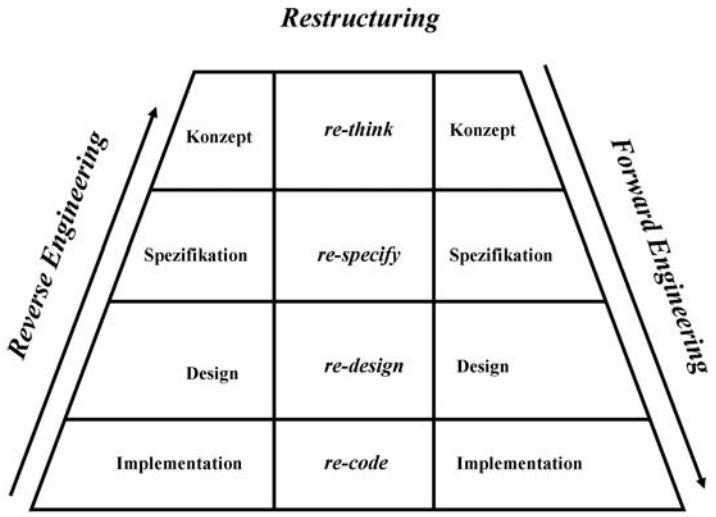

Abb. 5.9: Die Reengineeringpyramide

Regel, aus dem vorhandenen Sourcecode[29] ein Modell über die Applikationen erstellt werden muss. Üblicherweise existieren für das bestehende Legacysystem keine sinnvollen Dokumentationen mit der Folge, dass diese komplett neu erstellt werden müssen. Die Techniken der Modellbildung für das Verständnis des Legacysystems sind dieselben wie bei der Maintenance, s. Kap. 7.

Für das Restructuring sind zwei Modelle notwendig, zum einen ein Modell des Legacysystems und zum anderen ein Modell des Zielsystems, des „geplanten“ neuen Systems. Auf höchster Ebene sind diese Modelle die jeweiligen Architekturen der beiden Systeme.

5.9.2 Systemsicht

Aus systemtechnischer Sicht ist das Reengineering eine spezielle Form der Migration, s. Abb. 5.10. Das Problem ist: Wie kann man von dem gegebenen Zustand, dem Altsystem, zu einem neuen Zustand, dem Zielsystem, unter gegebenen Randbedingungen gelangen?

Altsystem

Das Reengineering wird eingeleitet, wenn das bestehende System sich nicht mehr weiter entwickeln kann, vor allen Dingen nicht mehr im Rahmen von

[29] Eine alte alte Programmiererweisheit lautet:*Im Code steckt die Wahrheit.*

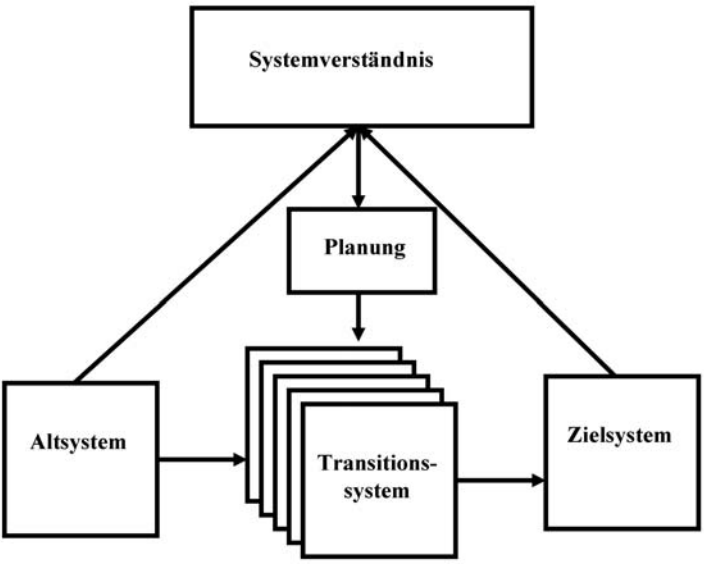

Abb. 5.10: Reengineering als systematischer Übergang

vertretbaren Kosten. Eine Kostenexplosion für die Maintenance eines bestehenden Legacysystems ist einer der häufigsten Auslöser für ein Reengineering neben einer Reihe von externen Auslösern, wie Technologiesprünge oder dem Verschwinden von Produkten im COTS-Software-Sektor. Aber nicht nur softwareintrinsische Vorgänge lösen das Reengineering aus. Andere oft primäre Quellen sind Veränderungen in den Geschäftsprozessen. Solche Veränderungen des Geschäftsumfeldes üben Druck auf das bestehende Legacysystem aus, sich an die neuen Gegebenheiten zu adaptieren. Falls eine Adaption nicht möglich ist oder zu lange dauert, wird das Legacysystem schnell obsolet.

Einer der Schlüsselvoraussetzungen für das Reengineering ist, dass das Legacysystem Teile besitzen muss, welche es lohnt zu erhalten, ansonsten würde ein Ersatz die bessere Alternative sein. Das vorliegende Altsystem hat meist schon eine sehr lange Historie von Änderungen und Tuningmaßnahmen hinter sich gebracht und ist nun in einem Zustand, welcher die Maintenance massiv erschwert. Unglücklicherweise fehlt jede Form der Dokumentation, wie beispielsweise die Architektur oder das Design, wenn sie je existiert haben, oder die Dokumentation ist drastisch veraltet. Dieses Fehlen der Dokumentation ist eines der Risiken bei dem Übergang zum Zielsystem. Das einzig verlässliche Dokument des Legacysystems ist der Sourcecode selbst.

Zielsystem

Das zukünftige Zielsystem ist eine Mischung aus Teilen des bestehenden Legacysystems, welches weiterhin erhalten bleiben muss, Eigenschaften eines neuen „state-of-the-art“ Systems und Eigenschaften, welche aus einer geänderten Umgebung wie auch der Systemhistorie abgeleitet werden können, ohne dass sie dem Endbenutzer explizit bekannt sind.

Typische Beispiele für erhaltenswerte Größen aus dem bestehenden Legacysystem sind Funktionalität, Performanz und Qualität der Ergebnisse. Typische Eigenschaften eines „State-of-the-art“-Systems sind Portabilität, Modularität, Datenunabhängigkeit, Offenheit, Interoperabilität, nahtlose Integration und Flexibilität. Das Zielsystem kann durchaus völlig neue Technologie nutzen, dabei sind dem Endbenutzer meist nicht alle neuen Möglichkeiten a priori einsichtig.

Systemverständnis

Das bestehende Legacysystem und das Zielsystem stellen eine Repräsentation des Systemverständnisses dar; die Zahl der insgesamt möglichen Implementierungen ist sehr groß, aber das Alt- sowie das Zielsystem stellen zumindest zwei mögliche Repräsentationen dar. Das Verständnis über das System basiert auf Artefakten, dazu zählen, neben dem Sourcecode, die Modelle des Systems. Typische Modelle sind Domänenmodell, Designmodell und Architektur.

5.9.3 Vorgehen

Das Vorgehen zum Übergang in eine neue Lösung oder in eines der Transitionssysteme, s. Abb. 5.10, unterscheidet sich nicht prinzipiell voneinander. Im tatsächlichen Entwicklungsteil des Reengineerings, s. Abb. 5.11, wird zunächst das System analysiert und dann werden alternativ COTS-Software, s. Kap. 10, oder auch eine Wiederverwendung bzw. Neuentwicklung bestehender Teile in Betracht gezogen.

Eine erfolgreiche Beendigung dieses Prozessabschnittes kann an den folgenden Faktoren gemessen werden:

- Das reengineerte Legacysystem erfüllt die gewünschten Anforderungen.
- Es ist ein getestetes und ausführbares System entstanden.
- Das System kann über eine Transition, s. Abb. 5.12, in Produktion gehen.

Eines der größten Hindernisse in der Produktionsübernahme, s. Abb. 5.12, ist eine unerfüllte oder falsche Endbenutzererwartung. Von daher ist eine explizite Beteiligung der Endbenutzer immer ein „sine qua non“.[30] Der zweite Aspekt ist die organisatorische oder korrekterweise die geschäftsprozessurale

[30] Die Beteiligung des Kunden ist bei den agilen Vorgehensmodellen, speziell beim eXtreme Programming, einer der zentralen Punkte.

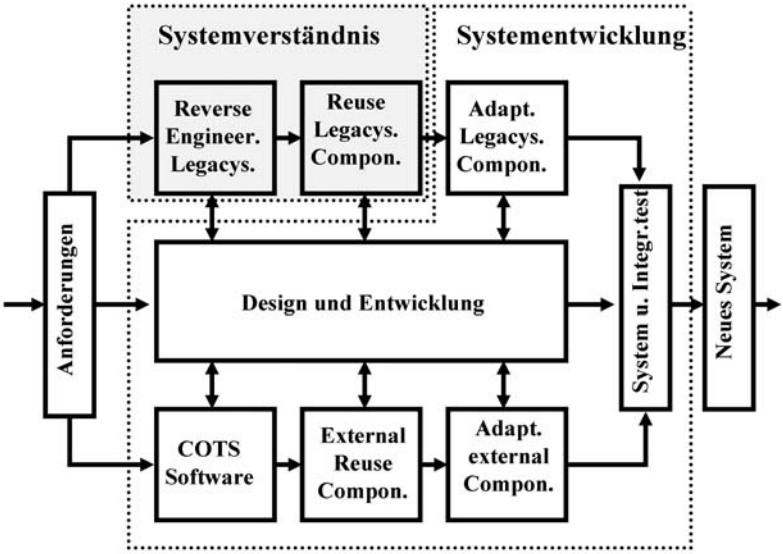

Abb. 5.11: Reengineering als Entwicklungsvorhaben

Einbettung des neuen Systems in die gesamte Organisation. Neue Systeme ermöglichen oft eine Veränderung der bestehenden Geschäftsprozesse.

Der Glaube, das Training und die Schulungen des zukünftigen Systems für die Endbenutzer auf ein Minimum reduzieren zu können und damit Zeit zu gewinnen, ist einer der häufigsten Fehler in der Transitionsphase.

5.10 Reverse Engineering

Die heutigen Softwareentwickler werden mit einer großen Menge an Legacysystemen konfrontiert, welche die Eigenschaften haben, schwer verständlich zu sein, hauptsächlich auf Grund ihrer Größe, Komplexität und Historie.

Das größte Problem des Reverse Engineerings scheint das Verstehen des vorliegenden Legacysystems zu sein, Schätzungen belaufen sich darauf, dass ein Anteil von 50-90% des Aufwandes eines Reengineeringprojekts in das Verstehen des Systems investiert werden. Die Fähigkeiten und Fertigkeiten zur Modellbildung spielen hierbei eine zentrale Rolle. Diese Prozesse sind sehr komplex und werden beeinflusst durch eine Vielzahl von externen Faktoren, so z.B.: Ressourcen, Pläne, Betriebspolitik, Trends und Moden, Marketing und manchmal auch „Flurfunk“. Die Komplexität eines solchen Unterfangens kann eine Führungskraft durchaus überwältigen. Im Vergleich hierzu ist Business Process Reengineering in der Praxis meistens einfacher, da hier der Schwerpunkt sehr oft auf der Automatisierung eines linearen Prozesses liegt.

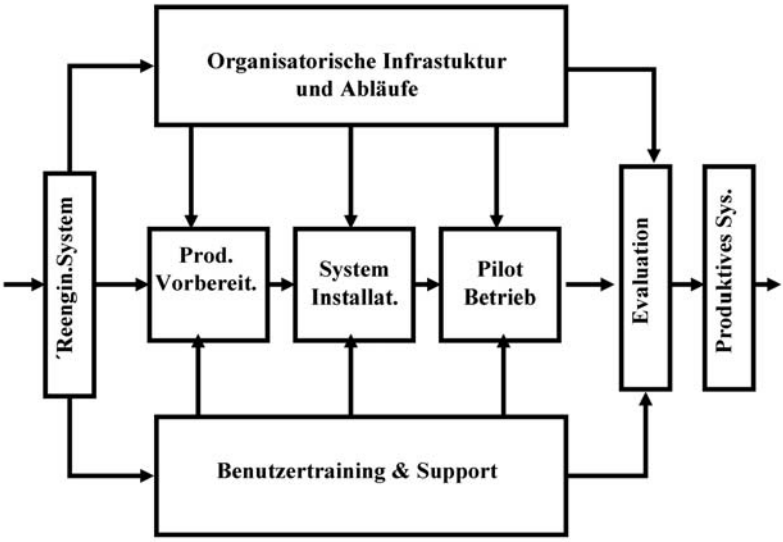

Abb. 5.12: Die Transition in die Produktion

Das Reverse Engineering wiederum enthält unter anderem zwei wichtige Teildisziplinen:

- Redokumentation – Unter dem Begriff Redokumentation werden alle Anstrengungen verstanden, ein für den Menschen verständliches Modell der Legacysoftware zu bauen. Dieses Modell entsteht aus der Implementierung[31] und wird zunehmend abstrahiert.
- Design Recovery – Hierbei wird versucht, explizit das mögliche Designmodell zu entwickeln.

Sowohl Redokumentation als auch Design Recovery sind ein beliebtes Tummelfeld für Werkzeughersteller, welche, in der Regel, versprechen, dass ihr jeweiliges Werkzeug alles enthält, um genau die obigen Aufgaben[32] bewerkstelligen zu können. Ein entsprechender Werkzeugeinsatz kann nur die bestehende Dokumentation bzw. den bestehenden Code transformieren. Die Fähigkeit zur Abstraktion ist aber eine primär intellektuelle Leistung und damit dem Menschen vorbehalten. Allerdings haben bestimmte Werkzeuge, wie beispielsweise Profiler, einen durchaus zweckmäßigen Einsatz. Mit Hilfe eines Profilers lässt sich der dynamische Aufbau der Legacysoftware beobachten und subsequent

[31] In der Praxis wird häufig auch „alte“ Dokumentation mit berücksichtigt.

[32] Häufig sind die Hersteller bereit, diese Fähigkeiten ihrer Tools auch an kleinen akademischen Problemstellungen zu demonstrieren. Diese so genannten „clean room problems“ sind meist extrem weit von der Realität entfernt und enthalten nur einfache Codesequenzen.

auch beschreiben. Je nach Architektur kann es möglich sein, dass an Hand des statisch vorliegenden Codes nicht auf die zur Laufzeit vorhandene Architektur geschlossen werden kann; in diesem Fall lohnt sich ein Profiler.

Obwohl es sehr viele Versuche gegeben hat, aus dem vorhandenen Sourcecode mit Werkzeugunterstützung eine Dokumentation zu erzeugen, muss man sagen, dass alle diese Versuche wenig dazu beigetragen haben, die Legacysoftware semantisch zu verstehen. Diese Werkzeuge können die Menge an zu lesendem Code durchaus reduzieren, sind aber kein Ersatz für die intellektuelle Leistungsfähigkeit eines Menschen. Genau wie der Sourcecode muss auch die Implementierung des inhärenten Datenmodells verstanden werden. Hier existieren unterschiedliche Datenbanktechnologien. Am schwierigsten ist die Analyse eines Legacysystems, welches hauptsächlich dateibasiert agiert, da solche Systeme keine Mechanismen haben, um die entstehenden Daten in einem Metamodell – bei relationalen Datenbanken ist stets ein Implementierungsmodell hinterlegt – zu beschreiben. Die dateibasierten Legacysysteme haben ihr Datenmodell de facto in der Programmlogik „versteckt“.

5.10.1 Redokumentation

Ein Kriterium für unzureichende Dokumentation ist die Frage, ob die Menge und Qualität der Dokumentation ausreicht, um tatsächlich ein Reengineering zu starten.

Für die Redokumentation werden verschiedenste Werkzeuge angeboten. Allen diesen Werkzeugen liegt die Idee zugrunde, den Sourcecode komprimiert darzustellen. Interessanterweise gehört auch Javadoc, ein Bestandteil des Java-SDK's, zu dieser Klasse von Werkzeugen. Die heutigen Redokumentationsgeneratoren lassen sich in die Klassen:

- Documentation Generator,
- Reverse Literate Programming

einteilen.

Documentation Generator

Die Basisidee hinter jedem Documentation Generator ist, dass ein Programm in verschiedene Abstraktionsschichten zerlegt werden kann. Diese Abstraktionsschichten sind sprachspezifisch und werden üblicherweise als Levels bezeichnet. So beispielsweise beim COBOL, s. Tab.5.2:

Alle Werkzeuge, welche in diese Klasse fallen, müssen eine Island Grammar beherrschen. Unter Island Grammar versteht man die Fähigkeit, nur syntaktische Elemente eines Sourcecodes aufzuzeichnen und zu interpretieren, die aktiv erkannt werden. So kann sichergestellt werden, dass nur die wichtigen Teile[33] interpretiert werden. Jede Island Grammar besteht aus

[33] Ob die so gefundenen Teile wirklich wichtig sind, unterliegt der Interpretation des Anwenders.

Tab. 5.2: Die COBOL Systemhierarchie

Level	Dokumentation
system	Aufgabenstellung, Idee des Gesamtsystems
subsystem	Sinn und Zweck, batch jobs, Transaktioncodes
batch job	Datenbanken, Häufigkeiten, Entitäten
programm	Parameter, Programmverhalten, Zugriffe
section	Funktionalität, externe Calls, Variablen

1 Produktionsregeln, welche zur Redokumentation relevant sind.
2 Produktionsregeln für alle restlichen Konstrukte, z.B., sie als Kommentare zu behandeln oder einfach zu ignorieren.[34]
3 Definitionen für die gesamte Struktur des Programms.

In gewissem Sinne können solche Documentation Generatoren wie Präprozessoren betrachtet werden und häufig werden sie auch, projektspezifisch, aus Parsersprachen wie REXX, Python oder Perl aufgebaut.

Reverse Literate Programming

Beim Reverse Literate Programming wird ein Programm als eine Enzyklopädie verstanden. Die Basisidee geht auf Donald E. Knuth zurück. Literate Programming beinhaltet den Ansatz, Dokumentation und Konstruktion eines Programms aus der gleichen Quelle zu konstruieren. Im Rahmen des Reverse Literate Programmings wurde diese Idee mit der Fähigkeit von Hypertext verknüpft.

Neuere Sprachen, speziell alle GML-, **G**eneralized **M**arkup **L**anguage-Abkömmlinge, besitzen diese Fähigkeiten. Dazu zählen neben HTML auch XML und alle daraus abgeleiteten Sprachen.

5.10.2 Architekturrekonstruktion

Der Begriff der Softwarestruktur bezeichnet alle Artefakte, welche von Softwareentwicklern genutzt werden, um ein mentales Modell der Software bilden zu können. Ohne ein mentales Modell der Software sind weder Design noch Implementierung möglich. Die Struktur der Software ist die Zerlegung der Legacysoftware in Komponenten, bzw. auch Schichten, und die Wechselwirkungen zwischen diesen Komponenten und Schichten untereinander.

Oft wird die Architekturrekonstruktion als eine erweiterte Form der Redokumentation gesehen, was leider der falsche Ansatz ist. Eine Architektur ist sehr stark an Abstraktion und Modellbildung interessiert, weniger daran, jede einzelne fachliche Funktion detailliert zu beschreiben. Das Ziel hinter

[34] Javadoc ignoriert alle Javasprachelemente, welche nicht mit einem @ beginnen oder Bestandteil der Signatur sind.

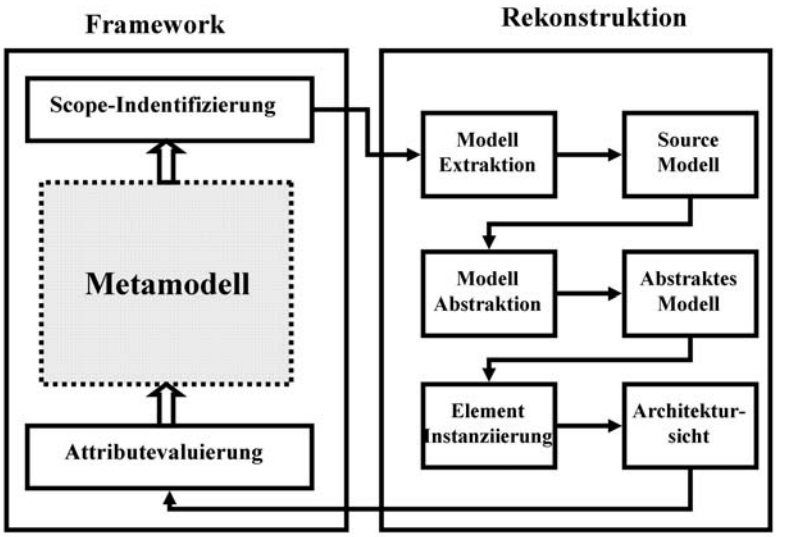

Abb. 5.13: Die QADSAR-Methodik zur Architekturrekonstruktion

dem Prozess der Architekturrekonstruktion ist es, die Architektur von der Implementierung ausgehend abzuleiten. Von daher stellt sich bei der Architekturrekonstruktion sofort die Frage nach der Vollständigkeit, d.h. die Frage: Wann ist es genug? Ziel muss es sein, gerade genug Information über die Architektur zu modellieren, um in der Lage zu sein, sie zu verstehen. Ein Zuviel an Information, ein zu hoher Grad an Detailwissen, führt zu einem „Verwaschen" der Architektur.

Damit diese spezielle Anforderung an die Architekturrekonstruktion gelöst werden kann, ist hier ein Redokumentationsmodell mit einem Feedbackloop sinnvoll, das QADSAR-Vorgehen, **Q**uality **A**ttribute **D**riven **S**oftware **A**rchitecture **R**econstruction. Die QADSAR-Methodik optimiert den Einsatz eines QAD-Frameworks, **Q**uality **A**ttribute **D**riven Analysis Framework, als Metamodell, um eine Architekturrekonstruktion gezielt zu ermöglichen. Die Basisidee hinter dem Quality Attribute Driven Analysis Framework ist die Feststellung, dass ein technisches System sehr viele Attribute besitzt, von denen aber nur einige wenige wirklich „wichtig" sind. Diese wichtigen Attribute beeinflussen das System maßgeblich in allen seinen Erscheinungsformen, d.h. sie haben bei Veränderungen den größten Einfluss. Ein solches Attribut wird dann als Qualitätsattribut bezeichnet. In der Regel werden die Qualitätsattribute von außen, d.h. von der Domäne, vorgegeben. In diesem Fall sind die so genannten Qualitätsattributsszenarien auch die Testfälle für das System. Das System verhält sich in dieser Betrachtungsweise also rein reaktiv, die Veränderung des Qualitätsattributs erzeugt einen Stimulus und die Reaktion des

Systems auf diesen Stimulus wird gemessen. Falls ein Attribut gewählt wird, bei dem trotz eines starken Stimulus nur eine schwache Reaktion einsetzt, so ist dies kein entscheidendes Attribut für eine Architekturrekonstruktion und sollte auch nicht verwendet werden. Im originalen QAD-Modell kann jedes Attribut als Qualitätsattribut eingesetzt werden; für eine Architekturrekonstruktion ist es jedoch sinnlos, ein Attribut zu nehmen, auf das nur schwach reagiert wird.

Das Vorgehen durchläuft dabei folgende Schritte, s. Abb. 5.13:

- Scope-Identifizierung – Die Scope-Identifizierung dient dazu, den Untersuchungsgegenstand sowie die zu modellierende Architektursicht zu definieren. Der tatsächliche Scope hängt von den zu betrachtenden Qualitätsattributen und dem jeweils zu beschreibenden System ab.
- Modellextraktion – Bei der Modellextraktion wird eine Untermenge von Quellen aus den vorhandenen Quellen ausgewählt. In der Regel handelt es sich hierbei um Sourcecodedateien, aber es können auch Funktionen, Klassen, Schnittstellen und so weiter sein. Neben den reinen Quellen spielen hier auch die statischen Relationen zwischen den Quellen, so beispielsweise zwischen einer COBOL-Datei und einer Menge entsprechender COBOL-Copybooks, eine große Rolle und werden, üblicherweise werkzeuggestützt, beschrieben. Dynamische Relationen werden durch die entsprechenden Profiling- und Tracingwerkzeuge dargestellt.
- Modellabstraktion – Die im vorherigen Schritt erzeugten Elemente sind fast immer viel zu detailliert, um als ein Modell benutzt zu werden. Aus diesen Gründen wird ein aggregiertes Modell, welches von den konkreten Ausprägungen abstrahiert, erstellt.
- Elementinstanziierung – Mit Hilfe des aggregierten Modells wird die entsprechende Architektursicht erzeugt. Die typischen Elemente der Architektursichten sind Schichten, Komponenten und viele andere abstrakte Objekte.
- Qualitätsattributevaluierung – In diesem Schritt wird untersucht, ob die Architektursicht tatsächlich auf das verwendete Attribut reagiert und gegebenenfalls das entsprechende Feedback genutzt.

Zwar gibt die QADSAR-Methodik ein allgemeines Framework für die Rekonstruktion einer Architektur wieder, sie beantwortet jedoch nicht die Frage, wie konkret vorgegangen werden soll. Die Focus-Vorgehensweise beantwortet diese Fragestellung. Im Rahmen von Focus werden sechs Schritte zur Architekturrekonstruktion inkrementell rekursiv durchlaufen. Die einzelnen Schritte sind:

1 Identifizierung der Komponenten – Im Rahmen der Implementierung werden die konkret vorhandenen Komponenten identifiziert, sie sind somit Teile der physischen Architektur. Im Fall von Legacysystemen lassen sich die Komponenten in zwei unterschiedliche Typen einteilen:
   - Prozesskomponenten,

- Datenkomponenten.

Beide Typen zusammen geben ein Abbild der Legacysoftware auf Komponentenebene wieder.

2 Vorschlag einer idealisierten Architektur – Die erste, hypothetische Architektur wird zunächst postuliert. Dabei wird man in der Regel von seiner allgemeinen Erfahrung über die Bauweise von anderen Legacysystemen geleitet. Diese Architektur muss nicht notwendigerweise zu Beginn vollständig korrekt sein. Bei der nächsten Iteration kann sie wieder korrigiert werden.

3 Abbildung der identifizierten Komponenten auf die idealisierte Architektur – Die Komponenten müssen innerhalb der Architektur angeordnet werden, d.h. sie müssen in der Architektur erkennbar sein. Falls dies nicht der Fall ist, kann entweder die gefundene Komponente falsch sein oder die Architektur wurde noch nicht gut genug identifiziert.

4 Identifizierung der Schlüssel-Use-Cases – Diese Use-Cases dienen zur zukünftigen Identifizierung von Änderungen innerhalb der Legacysoftware, außerdem muss es eine Abbildung zwischen den Use-Cases und den Komponenten, quasi orthogonal zu der physischen Architektur, geben.

5 Analyse der Komponentenwechselwirkungen – Die einzelnen Komponenten müssen zu einem Ganzen integriert werden; speziell die Connectoren zwischen den Daten- und Prozesskomponenten werden hier beschrieben.

6 Erzeugung der verfeinerten Architektur – Aus diesem Vorgehen entsteht wiederum eine neue Architektur, die als Eingabe für den ersten Schritt zur Identifizierung der Komponenten genutzt werden kann.

Die Focus-Vorgehensweise hat den Vorteil, dass die Architektur nicht a priori korrekt und vollständig sein muss, sondern sich inkrementell entwickelt und somit auch Fehler kontinuierlich korrigiert werden können.

5.11 Datenstrategien

Neben den strukturellen Anstrengungen bei einer Migration spielt die Migration der Daten eine sehr große Rolle. Allerdings lässt sich eine solche nur in die Gesamtmigrationsstrategie einbinden und macht auch nur in dem entsprechenden Kontext Sinn. Die Daten der Legacysoftware spielen jedoch eine immense Rolle, sie stellen in den meisten Fällen das größte Kapital[35] des Unternehmens dar.

Eine Datenmigration findet jedoch in den meisten Fällen im Rahmen einer größeren Migration statt. Solche Migrationen zerfallen, zumindest aus softwaretechnischer Sicht, in die Software- und die Datenmigration. Beide Migrationsschritte können in unterschiedlicher Abfolge durchgeführt werden, was durchaus seine Konsequenzen hat:

[35] Eine Tatsache, die den meisten Unternehmen nicht bewusst ist.

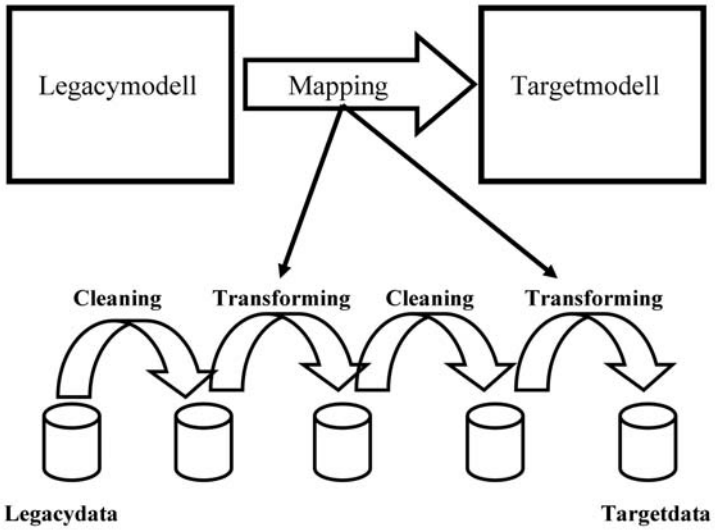

Abb. 5.14: Das Datenmigrationsschema

- Datenmigration zuerst – Der Vorteil hiervon wäre, dass die Software dann auf einem definierten Stand aufsetzen kann, quasi ein festes Datenmodell. Dies hat entwicklungstechnisch gesehen seine Vorzüge, da die Wahrscheinlichkeit in einer „alten“ Architektur zu verharren niedriger wird, wenn schon eine neue Datenbank vorhanden ist. Der Nachteil ist, dass nun das bestehende Legacysystem im ersten Schritt an die neue Datenhaltung angepasst werden muss und sich erst danach verändern kann. Dies erhöht die Komplexität bei gleicher Absenkung des Maintainability Index. Aber die schlechte Wartbarkeit war ja in der Regel einer der Hauptgründe für die Migration! Durch die hohen Investitionsleistungen bei der Vorwegnahme der Datenmigration tendieren Unternehmen dazu, die Ergebnisse dieses Schrittes unangetastet zu lassen.
- Simultane Migration – Dieses Vorgehen ist zwar theoretisch möglich, es hat aber den höchsten Personalbedarf, da jetzt beide Schritte simultan durchgeführt werden müssen. Aus Kostengründen wird diese Strategie fast nie eingeschlagen. Außerdem wird durch die Gleichzeitigkeit die Komplexität des Migrationsprozesses deutlich erhöht, im Rahmen eines Big-Bang-Ansatzes ist diese Strategie aber immer notwendig.
- Datenmigration zuletzt – Die übliche Strategie ist es, die Daten am Schluss zu migrieren. Speziell der Einsatz einer Persistenzschicht, welche Abbildungen zwischen dem Alt- und Neusystem vornehmen kann, hilft hier deutlich.

Allerdings ist bei fast allen Legacysystemen die Datenqualität, s. Abschn. 4.18, als schlecht anzusehen. Der Prozess der Umwandlung zerfällt schematisch gesehen in folgende Schritte, s. Abb. 5.14:

- Cleaning
- Transformation
- Cleaning
- Transformation
- ...

Das Ziel ist es, ausgehend von den Legacydaten das neue postmigrierte System mit den Targetdaten zu bestücken. Hierbei sind einige Aktivitäten durchzuführen.

- Das Legacymodell muss auf das Targetmodell abgebildet werden. Aus dieser logischen Abbildung der jeweiligen Implementierungsformen entsteht das Datamapping.
- Das Datacleaning ist der Schritt zur Bereinigung der Daten. Da Legacydaten notorisch für ihre schlechte Datenqualität bekannt sind, ist dies unabdingbar.
- Es hat sich bewährt, Zwischendatensysteme anzulegen, um den Gesamtprozess einfacher und kontrollierbarer zu machen.

Diese einzelnen Schritte sind in ihrem Aufwand und ihrer Komplexität nicht zu unterschätzen.

Zwar ist es denkbar, die Legacysoftware während der Migration stillzulegen, dies ist in der Praxis jedoch nicht machbar, da Legacysysteme das Kerngeschäft des Unternehmens unterstützen und dieses nicht stillstehen darf. Bei Banken wird die maximale Auszeit – die Zeit, welche sie nicht operativ sein können – ohne ihre Existenz zu gefährden, auf etwa 2-5 Arbeitstage geschätzt. Nach dieser Zeit ist die Bank so stark angeschlagen, dass sie langfristig gegen ihre Konkurrenten auf dem Markt verliert. Dieser Verlust tritt nicht sofort, sondern erst nach einigen Monaten oder wenigen Jahren ein. Folglich muss jede sichere Datenstrategie die Idee der Koexistenz beider Systeme beinhalten, da Datenmigrationen in der Regel sehr zeitintensiv sind.

Die Abbildung zwischen den beiden Modellen, das Datamapping, ist der Versuch, zunächst alle möglichen Zustände des Legacysystems abzubilden, d.h. das Mapping $\widehat{O}$ versucht zu gewährleisten, dass gilt:

$$\phi = \widehat{O}\psi \qquad \forall \psi \in \Psi_{Legacydata}, \tag{5.1}$$

wobei ϕ die Zustände des neuen Systems und $\Psi_{Legacydata}$ die Menge aller möglichen Zustände des Legacysystems darstellt. Der einzelne Zustand ψ des Legacysystems setzt sich zusammen aus den Zuständen der einzelnen fachlichen Objekte durch:

$$\psi = \prod_{i \in alle fachlichen Objekte} \psi_i.$$

Die möglichen Einzelzustände ψ_i sind durch ihre fachlichen Eigenschaften bzw. die gewählte Datenimplementierung limitiert. Die Objektzustände ψ_i ergeben sich aus den möglichen Ausprägungen für die Attribute durch

$$\psi_i = \prod_{j \in Attribute} \psi_i^j.$$

Somit ist $\Psi_{Legacydata}$ die Menge aller möglichen Ausprägungen aller Attribute in allen Objekten.

In der Praxis reicht jedoch eine kleinere Menge aus, da nur nach der Transformation einer Untermenge $\Psi' \subset \Psi_{Legacydata}$ gesucht wird. Es handelt sich deswegen um eine echte Untermenge, weil die Datenmigration zu einem festen Zeitpunkt t_0 startet und zunächst versucht, den zu diesem Zeitpunkt vorliegenden Zustand des Gesamtsystems zu migrieren. Es reicht also aus, in der Lage zu sein, eine Menge von Zuständen Ψ' zu migrieren, welche die Eigenschaft haben, in der Nähe von $\psi(t_0)$ zu liegen, d.h. der Unterschied ist eine kleine zeitliche Weiterentwicklung. Mit $\Delta t = t - t_0$ gilt:

$$\psi(t) = \psi(t_0) + \delta(\Delta t). \tag{5.2}$$

Die zu betrachtenden Migrationszustände sind jetzt:

$$\Psi' = \bigcup_{\Delta t} (\psi(t_0) + \delta(\Delta t))$$

mit

$$\Psi' \subset \Psi.$$

Bei allen Legacysystemen ist der zweite Summand in Gl. 5.2 klein gegenüber dem ersten. Für jede Metrik, s. Kap. 2, gilt somit:

$$M(\psi(t_0)) \gg M(\delta(t - t_0)). \tag{5.3}$$

Eine Datenmigration $\widehat{O}$ sollte die Eigenschaft haben:

$$\begin{aligned} \phi &= \widehat{O}\psi(t) \\ &= \widehat{O}(\psi(t_0) + \delta(\Delta t)) \\ &\approx \widehat{O}\psi(t_0) + \widehat{O}\delta(\Delta t). \end{aligned} \tag{5.4}$$

Die letzte Näherung, Gl. 5.4, setzt voraus, dass sich die Transformation auf der Untermenge Ψ' quasi linear verhält. Unter der Maßgabe, dass eine Durchführung der Datenmigration eine gewisse Zeit Δt in Anspruch nimmt, liegt es nahe, die Datenmigration inkrementell vorzunehmen, mit der Folge, dass jede Iteration nur noch einen geringen Änderungsaufwand hat, $\Delta^{(k)}t = t_k - t_{k-1}$:

$$\begin{aligned}
\phi &= \widehat{O}\psi(t_0) \\
\Delta^{(1)}\phi &= \widehat{O}\delta(\Delta t) \\
\Delta^{(2)}\phi &= \widehat{O}\delta(\Delta^{(2)}t) \\
&\dots \\
\Delta^{(k)}\phi &= \widehat{O}\delta(\Delta^{(k)}t).
\end{aligned}$$

Unter der Annahme, dass jede Folgeänderung im Zeitintervall $\Delta^{(k)}t = t_k - t_{k-1}$ immer weniger Durchführungszeit braucht, konvergiert das Verfahren unter der Annahme Gl. 5.4. In der Praxis sind oft kleinere Ausfallzeiten im Bereich von wenigen Stunden, beispielsweise in den Nächten oder über Weihnachten, akzeptabel, so dass wenige Iterationen völlig ausreichen.

Die viel größere Schwierigkeit ist durch die zeitliche Veränderung des Zielsystems gegeben. Wenn dieses Zielsystem sich während der inkrementellen Datenmigration in einem operativen Betrieb befindet, so verändert es sich, bevor die nächste Teilmigration einsetzen kann. Aus dem Blickwinkel des Zielsystems gibt es für eine Veränderung zwei treibende Kräfte, die Datenmigration und das operative Geschäft. Folglich stellt sich die Entwicklung des Zielsystems durch folgende Gleichung dar:

$$\begin{aligned}
\phi(t_0) &= \widehat{O}\psi(t_0) \\
\phi(t_1) &= \phi(t_0) + \widehat{O}\delta(\Delta t) + \widehat{P}(\Delta t)\phi(t_0) \\
&\dots \\
\phi(t_k) &= \phi(t_{k-1}) + \widehat{O}\delta(\Delta^{(k)}t) + \widehat{P}(\Delta^{(k)}t)\phi(t_{k-1}).
\end{aligned}$$

In dieser Situation existieren drei grundsätzliche Lösungsalternativen, um die Schwierigkeiten in der Wechselwirkung zwischen der zeitlichen Weiterentwicklung $\widehat{P}$ und der nächsten inkrementellen Datenmigration zu beherrschen:

- Die Datenmigration $\widehat{O}$ berücksichtigt, dass sich Zustände mittlerweile verändert haben mit der Konsequenz, dass sich die Datenmigration sehr komplex ausbildet:

  $$\widehat{O}(t) = \widehat{O}(t_0, \phi(t))$$

  Ein Vorteil dieses Vorgehens ist, dass es beliebig oft wiederholt werden kann. Falls sich ein Fehler eingeschlichen hat, oder der Einsatz der Legacysoftware in seinem alten Zustand sich für bestimmte Gebiete verlängert, so bleibt der Datenmigrationsoperator $\widehat{O}$ doch stabil. Außerdem erweist er sich auch als „idempotent“, d.h.

  $$\begin{aligned}
  \widehat{O}^n &= \widehat{O}\widehat{O}\widehat{O}\widehat{O}\widehat{O}\widehat{O}\dots\widehat{O}\widehat{O} \\
  &= \widehat{O}.
  \end{aligned}$$

  Obige Formel zeigt, dass die Datenmigration beliebig oft wiederholt werden kann. Da normalerweise die Datenmigration nur für eine gewisse Un-

termenge Ψ' der möglichen Zustände konstruiert wird, ist die Idempotenz nur für eine endliche Zeit[36] möglich.

- Die zweite Option ist die Reduktion auf genau einen primären Datenmigrationsschritt. Ein solcher ist technisch sehr viel einfacher, da jetzt sofort nur der Schritt

$$\begin{aligned}\phi(t_0) &= \widehat{O}\psi(t_0)\\ \phi(t) &= \widehat{P}(\Delta t)\phi(t_0)\end{aligned}$$

notwendig ist. Eine solche Form der Datenmigration wird als Big-Bang-Datenmigration bezeichnet. Selbstverständlich wird sie im Vorfeld an alten Zuständen $\psi(t < t_0)$ getestet. Aber die konkrete Durchführung findet zu genau einem Zeitpunkt t_0 statt. Leider ist dieses einfache Verfahren aus Gründen der Verfügbarkeit oft nicht wirklich praktikabel.

- Die dritte Option verlangt eine organisatorische Regelung. Da die Datenmigration die Übertragung von neuen Sätzen immer gewährleisten muss, sind nur die schon bisher übertragenen Sätze problematisch. Wenn diese von den nächsten Schritten der inkrementellen Datenmigration ausgeschlossen werden, entspricht das Vorgehen einer Big-Bang-Datenmigration. Allerdings müssen alle in den Zeitinkrementen im Legacysystem geänderten Daten auch noch übertragen werden. Dies wird, wenn es sich nicht um zu viele handelt, durch eine organisatorische Regelung der Doppelerfassung in beiden Systemen vorgenommen. Oft ist es kostengünstiger, diese Form der Übertragung zu wählen als ein besonders aufwändiges Schema zu implementieren.

Die eigentliche Datenmigration $\widehat{O}$ ist recht komplex; es hilft jedoch, sie in die einzelnen Schritte Datacleaning und Datamapping, s. Abb. 5.14, zu zerlegen. Hierbei wechseln sich die Schritte Datacleaning $\widehat{O}_C$ und Datamapping $\widehat{O}_M$ ab, mit dem Resultat, dass sich die allgemeine Datentransformation darstellen lässt durch:

$$\widehat{O} = \prod_{i=1}^{N} \left(\widehat{O}_M^{(i)} \widehat{O}_C^{(i)} \right). \tag{5.5}$$

Mit Gl. 5.5 ist die Datenmigration eine Abfolge von Cleaning- und Mappingschritten. Wichtig ist es, die Cleaningschritte immer zuerst durchzuführen. Das Cleaning hat den Vorteil, dass der Zustandsraum, den die Daten annehmen können, verkleinert wird:

$$\begin{aligned}\Psi'' &= \widehat{O}_C \Psi'\\ \Psi'' &\subset \Psi'\\ \Psi' &\subset \Psi.\end{aligned}$$

[36] Der Idealfall ist natürlich, dass die Bedingung für die Menge aller möglichen Zustände Ψ gilt, dies ist aber in der Regel zu teuer.

Durch diese Projektion wird die Menge der relevanten Datenmappings eingeschränkt, mit der Folge, dass $\widehat{O}_M$ einfacher zu konzipieren ist.

Es hat prozesstechnische Vorteile, eine Transformationskette, s. Gl. 5.5, zu nutzen. So kann die Komplexität jedes einzelnen Schrittes weiter reduziert werden. Die im Allgemeinen günstigste Vorgehensweise ist es, die einzelnen Schritte der Transformationskette auf jeweils orthogonale Teilräume wirken zu lassen, so beispielsweise die Adressen getrennt von den Konten zu transformieren. In einer Matrixdarstellung würden dann die einzelnen Transformationen von Gl. 5.5 zu Blockmatrizen werden. In sehr komplexen Fällen ist es jedoch nicht mehr möglich, die einzelnen Schritte als Matrix darzustellen.

5.11.1 Koexistenz

Werden nun die zeitlichen Inkremente Δt sehr klein gewählt, d.h. $\lim \Delta t \to 0$, so degeneriert die Datenmigration zu einer Koexistenz beider Systeme. Aus Gründen der Risikovermeidung kann dies oft sinnvoll sein. Trotz dieser Koexistenz ist zumindest der erste Datenmigrationsschritt $\phi(t_0) = \widehat{O}\psi(t_0)$ notwendig, um eine initiale Füllung des neuen Systems zu erreichen. Im weiteren Sinn muss dann ein Mechanismus existieren, welcher die zeitliche Entwicklung beider Systeme simultan sicherstellt, d.h. es muss eine Datenmigration $\widehat{S}(\Delta t)$ geben, so dass gilt:

$$\begin{pmatrix} \psi(t) \\ \phi(t) \end{pmatrix} = \begin{pmatrix} \widehat{S}(\Delta t) & \psi(t_0) \\ \widehat{O}\psi(t_0) \end{pmatrix}.$$

Beide Systeme sind zwar miteinander gekoppelt, entwickeln sich aber parallel. Üblicherweise durchlaufen koexistente Migrationen 4 Phasen, mit, je nach Bedarf, unterschiedlicher Länge:

- Initialisierungsphase – Hierbei wird das neue System erstmalig gefüllt. Dies gilt für die Zeiten $t < t_0$, und insofern hat der Zustandsvektor die Form:

$$\begin{pmatrix} \psi(t_0) \\ \phi(t_0) \end{pmatrix} = \begin{pmatrix} \psi(t_0) \\ \widehat{O}\psi(t_0) \end{pmatrix}.$$

- Nach dieser initialen Füllung ist die Legacysoftware das führende System, d.h. alle Änderungen werden im Legacysystem vorgenommen und dann auf das neue System übertragen und dort eventuell auch geprüft.

$$\begin{pmatrix} \psi(t) \\ \phi(t) \end{pmatrix} = \begin{pmatrix} \widehat{P}(\Delta t)\psi(t_0) \\ \phi(t_0) + \widehat{O}\delta(\Delta t) \end{pmatrix}.$$

  Das neue System spielt permanent die Änderungen der Legacysoftware nach.

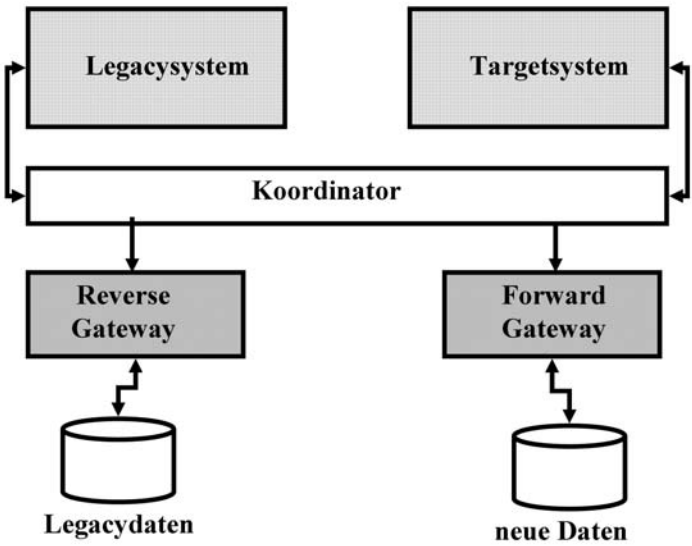

Abb. 5.15: Die Chicken-Little-Strategie

- Das neue System wird zum Zeitpunkt τ das führende System, d.h. die Änderungen finden nur noch im neuen System statt. Aus Gründen der Risikovermeidung, oder weil noch Teile des Legacysystems den „alten“ Zustand benötigen, werden die Daten jetzt faktisch remigriert und eventuell auch erneut geprüft.

$$\begin{pmatrix} \psi(t > \tau) \\ \phi(t > \tau) \end{pmatrix} = \begin{pmatrix} \psi(\tau) + \widehat{O}^{-1}\phi(\tau) \\ \widehat{P}(\Delta t)\phi(\tau) \end{pmatrix}.$$

  Die Daten werden faktisch in die Legacysoftware zurückgespielt.
- Der letzte Schritt ist das Abschalten der Koexistenz. Ab diesem Zeitpunkt T existiert nur noch der Datenhaushalt des neuen Systems. Damit gilt für die Veränderung des Zustandes:

$$\begin{pmatrix} \psi(t > T) \\ \phi(t > T) \end{pmatrix} = \begin{pmatrix} \psi(T) \\ \widehat{P}(\Delta t)\phi(T) \end{pmatrix}.$$

  Dieser Übergang wird auch „Cut-Over“ genannt.

Die beiden bekanntesten Vorgehensweisen zur Implementierung einer Koexistenz sind:

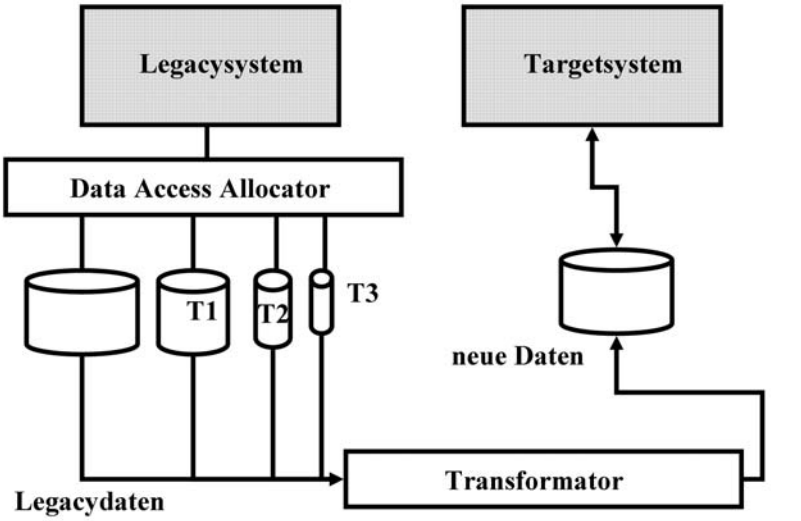

Abb. 5.16: Die Butterfly-Strategie

- „Chicken-Little-Strategie“, s. Abb. 5.15 – Die Chicken-Little-Strategie[37] beruht auf dem Einsatz zweier Gateways: Einem Forward Gateway, es entspricht der Abbildung $\phi = \widehat{O}\psi$, und einem Reverse Gateway mit $\psi = \widehat{O}^{-1}\phi$. Insgesamt gesehen ist die Chicken-Little-Strategie eine aus 11 Schritten bestehende Strategie, mit den Schritten:
  1 Inkrementelle Analyse der Legacysoftware
  2 Inkrementelle Dekomposition der Legacysoftware
  3 Inkrementelles Design des Zielinterfaces[38]
  4 Inkrementelles Design der Zielapplikation[38]
  5 Inkrementelles Design der Zieldatenbank[38]
  6 Inkrementelle Installation der Zielumgebung
  7 Inkrementelle Implementierung der Gateways
  8 Inkrementelle Migration der Legacydatenbank[39]
  9 Inkrementelle Migration der Legacysoftware
  10 Inkrementelle Migration der Legacyinterfaces
  11 Inkrementelles Abstellen der Legacysoftware
  Typischerweise ist die Last auf dem Gesamtsystem relativ hoch, da neben den eigentlichen Applikationen auch noch die Gateways aktiv sein müssen.

[37] Der Name geht auf den amerikanischen Ausdruck „to chicken out“, kneifen oder feige sein, zurück.
[38] Im Fall eines COTS-Software-Systems vorgegeben.
[39] Der eigentliche Migrationsprozess

Diese werden in den seltensten Fällen in Bezug auf Performanz entworfen, mit der Folge, dass die entstehenden Chicken-Little-Systeme zwar als recht risikoarm, dafür aber als sehr langsam gelten. Eine subjektiv empfundene schlechte Performanz kann aber ein ganzes Migrationsprojekt gefährden, s. Abschn. 5.2.

- „Butterfly-Strategie“, s. Abb. 5.16 – Die zweite Strategie sieht ein inkrementelles Laden vor. Hierbei werden Schnitte gebildet, welche dann in das neue System migriert werden. Alle schon migrierten Bestände werden eingefroren. Der Data Access Allocator schreibt nur noch die Unterschiede gegenüber den schon migrierten Beständen weg, mit dem Ziel, die Menge an zu migrierenden Daten konsekutiv zu verkleinern. Der Vorteil dieses Verfahrens ist es, dass die Migration auch während der Nacht[40] stattfinden kann, dann, wenn die Performanzanforderungen nicht so streng sind wie während des operativen Tages. Als Prozessmodell besteht die Butterfly-Strategie aus sechs aufeinander folgenden Phasen:
  1 Migrationspräparation: Den Umfang, das Zielsystem, die Architektur und die Anforderungen sowie das Messverfahren für den möglichen Erfolg festlegen.
  2 Die Legacysoftware semantisch verstehen und das Zieldatenmodell entwickeln.
  3 Aufbau einer Testdatenhaltung für das Zielsystem auf Grund eines Testbestandes aus der Legacysoftware. Hier werden die Transformationsregeln falsifiziert.
  4 Migration aller Komponenten, außer den Daten vom Legacysystem, in das neue Zielsystem.
  5 Inkrementelle Migration des Datenbestandes vom Legacydatenbestand in den Zieldatenbestand.
  6 Übergabe des neuen Zielsystems an die Produktion, der Cut-Over.

  Wichtig bei dieser Strategie ist es, dass erst dann, wenn alle Daten migriert wurden, der produktive Einsatz des neuen Systems beginnt. Das Ziel beider Strategien ist es, die sukzessiven Datenmigrationen in ihrem zeitlichen Aufwand zu reduzieren:

$$T_{i+1}^{Datenmigration} < T_i^{Datenmigration}.$$

  Wird der Zustand erreicht, dass die Zeit für ein i so klein ist, dass das Unternehmen mit der produktiven Auszeit leben kann, beispielsweise 6 Stunden am Wochenende, so wird die Legacysoftware endgültig abgeschaltet. Die Zahl der temporären Datenbanken, welche kreiert werden müssen, lässt sich recht einfach berechnen. Angenommen, die Ursprungsgröße der Legacydatenbank sei Ξ_0 und die Geschwindigkeit, mit der die Daten aus der Legacydatenbank in das neue System migriert werden können, sei μ. Die Geschwindigkeit, mit der eine temporäre Datenbank erzeugt werden kann, sei ν, dann ergibt sich für die Größe G nach der i-ten Iteration:

[40] Sinnigerweise oft als Batchnacht oder Batchfenster bezeichnet.

$$G_i = \Xi_0 \left(\frac{\nu}{\mu}\right)^i.$$

Daraus folgt sofort, dass für $\frac{\nu}{\mu} < 1$ gilt:

$$\lim_{i \to \infty} G_i = 0.$$

Wenn die Größe G_i nur klein genug ist, d.h. $G_i < \varepsilon$, geht die Transformation in einem Schritt. Dieser Punkt ist nach der Iteration N erreicht mit:

$$N = \Gamma \left(\frac{\log \Xi_0 - \log \varepsilon}{\log \mu - \log \nu}\right)$$

Entscheidend für N ist das Verhältnis zwischen dem Cut-Off ε und der Gesamtmenge Ξ_0 sowie den beiden Geschwindigkeiten. In der Regel sollten 5-20 Iteration ausreichen.

5.11.2 Datamapping

Das Datamapping gehört zu den intellektuell herausforderndsten Tätigkeiten im Rahmen einer Migration. Es erfordert sehr tiefe Kenntnisse der Datenmodelle beider Systeme, des ursprünglichen Legacysystems und des neuen, postmigrierten, Systems. Neben der reinen syntaktischen Abbildung der unterschiedlichen Datenmodelle ist hier ein hohes Maß an semantischem Wissen erforderlich.

Diese Formen der Abbildungen brauchen viel Erfahrung über die Auswirkungen, die dieses Mapping auf die Daten hat. Am einfachsten sind noch Themenkreise wie Kennziffern oder Schlüssel. Solche Schlüsselsysteme lassen sich im Rahmen der Extraktion aus der Legacysoftware recht gut lösen. Hier empfiehlt es sich, die Schlüssel durch ihren echten semantischen Kontext zu beschreiben. Die strukturellen Abbildungen sind meistens über einfache Fallunterscheidungen zu gewährleisten.

Neben dieser rein statischen Abbildung, welche in der Regel relativ einfach ist, existiert die Abbildung der Zustände. Legacysysteme sind dafür bekannt, dass sie einen großen Teil der Zustände, welche die fachlichen Objekte annehmen können, nicht explizit modelliert haben. Einer der Gründe hierfür liegt darin, dass die Zustandsübergänge nicht explizit überprüft werden, sondern impliziert durch die Logik innerhalb der Legacysoftware gegeben sind. Dies hat die Folge, dass die Menge aller möglichen Zustände wie auch die Menge der möglichen Übergänge zwischen diesen einzelnen Zuständen oft nicht hinreichend bekannt sind. Erzwingt das neue System nun ein explizit zustandsbehaftetes Arbeiten mit den fachlichen Objekten, so taucht die Situation auf, dass es widersprüchliche Zustände gibt. Diese Widersprüche müssen im Datacleaning beseitigt werden.

Viel tückischer sind aber andere Formen: So kann beispielsweise durch den Prozess des Mappings ein fachliches Objekt produziert werden, das weder dargestellt noch bearbeitet werden kann, da es keine validen Zustandsübergänge

durchführen kann. Noch drastischer ist die Situation, wenn fachliche Objekte im neuen System unauffindbar werden.[41] Diese Situation lässt sich nur durch explizite Suchprogramme erkennen. Zwar könnte man auf die Idee verfallen, hier Mechanismen aus dem Datawarehousebereich zu nutzen, so beispielsweise die Tatsache, dass die Summe der Umsätze vor und nach der Migration identisch sein muss. Leider sagt dies nichts über die Auffindbarkeit des fachlichen Objektes aus, da die Summe normalerweise direkt aus den Umsätzen in einer Datenbank gebildet wird. Ein explizites Prüfprogramm, das jedes einzelne fachliche Objekt instanziiert, ist hier vonnöten. Durchaus hilfreich ist es, Prüfsummenvergleiche auf den zu migrierenden Datenbeständen im Legacysystem und im Zielsystem durchführen zu können. Immerhin lässt sich mit dieser Maßnahme überprüfen, ob ein definierter Datenbestand vollständig migriert wurde.

5.11.3 Datacleaning

Beim Datacleaning ist die Zielsetzung, die Daten so zu bereinigen, dass sie für das Zielsystem auch verwendbar sind. Der aufwändigste Teil des Datacleanings ist die Isolation und Lösung der Probleme, welche mit der schon bekannten schlechten Datenqualität einhergehen. Typischerweise sind in den Legacydatenbeständen folgende Problemfälle häufig anzutreffen:

- Redefines – Zusätzliche Informationen werden in einem vorliegenden Datenelement gespeichert. Typisch hierfür sind Notizfelder, in denen alles Mögliche abgelegt sein kann. Bei der Migration muss ein Algorithmus gefunden werden, dies automatisch abzubilden. Allerdings kann oft auch mit einer Beibehaltung dieser Praxis des Überdefinierens gelebt werden, da die Endanwender damit auch vertraut sind. Eine beliebte Variante davon sind Überdefinitionen in Namen oder Nummernkreisen: Wenn der letzte Buchstabe des Kundennamens groß geschrieben wird, so ist das ein wichtiger Kunde ...
- Fremdbestimmung – Eine solche Fremdbestimmung liegt vor, wenn die Interpretation eines Attributes von dem eines anderen Attributes abhängt. Wenn beispielsweise ein Datum als Heirats- oder Sterbedatum interpretiert werden muss, je nachdem, welche Schlüsselziffer in einem anderen Feld gesetzt wurde. Diese Abhängigkeit muss in der Migration vollständig abgebildet werden.
- Falscher Inhalt – Falsche Inhalte lassen sich oft überhaupt nicht finden. Es gibt allerdings zwei Ausnahmen:
  - Typverletzung – In dem Fall, dass anhand des Namens der Datentyp klar ist, lässt sich ein falscher Inhalt finden, beispielsweise im Feld Alter steht „Gx“.

[41] Bei den meisten COTS-Einführungsprojekten wird die Qualität der Datenmigration nur durch Stichproben überprüft. Eine sträfliche Nachlässigkeit.

  - Abhängiges Feld – wenn das betrachtete Attribut aus anderen abgeleitet werden kann, beispielsweise, wenn das Alter aus dem Geburtsdatum berechnet werden kann.[42] Solche Redundanzen sind immer anfällig für falsche Inhalte, da unterschiedliche Prozesse unterschiedliche Attribute abändern bzw. nutzen.

  Solche falschen Inhalte lassen sich nur durch intensive Prüfungen auffinden und nur durch menschlichen Eingriff beseitigen, da a priori nicht klar ist, was ein entsprechender Korrekturwert sein könnte. Beispiele für solche falschen Inhalte sind auch:
  - alte Postleitzahlen
  - 29ter Februar in Nichtschaltjahren
- Fehlende Werte – Fehlende Werte entstehen dadurch, dass die entsprechenden Felder überhaupt nicht gefüllt werden. Sie stellen einen Spezialfall des falschen Inhaltes dar.
- Formatverletzung – Solche Verletzungen tauchen bei der Zusammenfassung von Attributen oder der bewussten Formatierung auf. Datumsformate oder Reihenfolgen von Vor- und Nachname sind beliebte Kandidaten für Formatverletzungen. Einzig ein Parsen der Dateninhalte und eine Plausibilitätsprüfung auf diesen hilft hier weiter.
- Zuviel Daten – Nach einer langen Evolutionsperiode, was de facto typisch für alle Legacysysteme ist, kann es passieren, dass Attribute zwar noch vorhanden sind, aber nicht mehr benötigt werden. Relationale Datenbanken tun sich notorisch schwer damit, einmal vorhandene Attribute wieder zu entfernen. Der SQL-Standard sieht zwar einen

  ```
  ALTER TABLE ... ADD COLUMN...
  ```

  Befehl vor, hat aber keinen

  ```
  DROP COLUMN ....
  ```

  Befehl. In der Regel können diese Daten einfach ignoriert werden.
- Multiple Quellen – Oft sind in einem Legacysystem die Datenhaushalte der einzelnen Teilanwendungen separat voneinander realisiert, mit der Folge, dass dieselbe Information redundant und zum Teil widersprüchlich in verschiedenen Datenhaushalten auftaucht. Dies kann sogar innerhalb derselben Datenbank geschehen. Die einfachste Möglichkeit, dieses Problem zu lösen ist es, eines der beiden Systeme als führendes System zu definieren, dessen Daten die Werte bestimmen.

[42] Bei einem großen Reiseunternehmen nutzten Kunden diese Tatsache aus, indem sie schon lange im Voraus buchten. Da zu diesem Zeitpunkt ihre Kinder in eine billige Alterstufe fielen, aber die Reise durchgeführt wurde, als die Kinder schon älter waren, sparten die Kunden Geld.

- Untypisierte Daten – Häufig sind Textfelder anzutreffen, die einen komplexen Inhalt enthalten, der aktiv interpretiert werden muss. Die Strategie ist hier, analog zum Redefines-Fall vorzugehen.
- Verlorene Relation – In manchen Fällen werden Beziehungen nicht explizit gespeichert. Typisch ist das bei Adressen, wo es plötzlich für den Kunden mehrere Adressen gibt oder auch bei fehlgeleiteten Historisierungen. Solche Fälle lassen sich nur durch einen manuellen Eingriff lösen.
- Sonderzeichen – Bedingt durch die Gutmütigkeit von COBOL können in Feldern alle mögliche Werte stehen, auch wenn es sich nicht um Buchstaben oder Ziffern handelt. Solche Probleme lassen sich in COBOL am einfachsten durch ein „INSPECT – REPLACE“ lösen. In anderen Sprachen muss das Feld Byte für Byte interpretiert werden. Speziell „C“ interpretiert einen Wert, der Hexadezimal 0, „x00“, ist, als das Ende einer Zeichenkette, mit dem Erfolg, dass nicht weitergelesen wird; so wird, wenn „x00“ irgendwo auftaucht, die Zeichenkette abrupt beendet.
- Multiple Repräsentation – Beliebt ist es auch, Abkürzungen oder Schlüssel unterschiedlich festzulegen, beispielsweise kann für die Angabe „weiblich“ stehen:
  - w
  - 1
  - wb.
  - weibl.
  - x
  - XX
  - A
  - true
  - ...

5.12 Organisatorische Migrationsprobleme

Bisher lag der Schwerpunkt auf den typischen technischen Problemen der Migration und der Legacysoftware. Es gibt jedoch eine Reihe von weiteren Problemen, die stark in der Organisation und den eingesetzten Vorgehensweisen im Rahmen der Migration verankert sind. Dazu zählen:

- Falsche Strategien – Basisentscheidungen bezüglich der Migrationsstrategie, so beispielsweise Big-Bang oder inkrementelle Migration, können jedes Migrationsprojekt gefährden. Ein typisches Beispiel ist der Ersatz eines bestehenden Legacysystems durch ein COTS-Software-System und den simultanen Abbau von Mitarbeiterwissen[43] über die bestehende Legacysoftware. Diese Kombination führt zu einem immens hohen geschäftlichen Risiko.

[43] Auch als „Burn the Ships“ bekannt, s. Fußnote S.91.

- Berater – Expertenwissen von außen kann, muss aber nicht immer hilfreich sein. Oft werden Berater von Seiten der Unternehmensleitung nur eingesetzt, um Unruhe zu produzieren oder den Anschein von Aktivität zu erwecken, oder der „Experte“ wird anhand seines Preises ausgewählt. In bestimmten Situationen sollte die Migration auf Berater zurückgreifen, aber wenn, dann auf die richtigen!
- Intellektuelle Trägheit – Fehlende Ausbildung und mangelndes Wissen über die Zielplattform einer Migration führen zu massiven Problemen. Vor allem ein Wechsel in der zugrunde liegenden Entwicklungsphilosophie, dem Softwareentwicklungsparadigma, ist die größte intellektuelle Herausforderung bei den Beteiligten. Die Umstellung der Denkmuster ist für jeden Menschen schwer und dauert relativ lange, in der Größenordnung von Jahren. Je nach Unternehmenskultur und der intrinsischen Motivation der Beteiligten schwankt die Zahl derer, die einen Wechsel überhaupt nicht schaffen[44]; oft wechseln in einem solchen Fall die „erfahrenen“ Entwickler in die Revision[45]. Eine weit verbreitete Erscheinung ist die, dass das mittlere Management die Umstellung nicht vornimmt, während die Softwareentwickler sich umstellen. Dies führt langfristig zu einer Lähmung des Unternehmens. Ein anderes Problem taucht auf, wenn das Unternehmen massiv neue Kräfte, in der Regel jüngere Mitarbeiter, einstellt, welche die neue Plattform beherrschen. Neben den Ängsten bei den Mitarbeitern, die auf der Plattform des Legacysystems weiter arbeiten, entsteht dann eine ungewollte Zweiklassengesellschaft, die um Ressourcen wie Budget und Zeit kämpft. Viele der „alten Hasen“ sind versucht, den „Neulingen“ zu zeigen, dass sie besser sind, und grenzen diese von jeder Kommunikation und Information aus. Die „Neulinge“ vermitteln den „alten Hasen“ im Gegenzug das Gefühl, obsolet zu sein. Eine solche Kombination ist auf Dauer nicht haltbar. Einzige Ausnahme ist, wenn die Mitarbeiter für das Legacysystem von ihrer Alterstruktur[46] her so sind, dass sie mit dem Abschalten des Legacysystems in Rente gehen.
- Kontrollverlust – Wenn die Informationen über das Legacysystem nicht vorhanden sind oder das Unternehmen keine organisatorische Kontrolle über die Legacysoftware hat, ist eine Migration sehr problematisch, da der Startpunkt unklar bleibt.
- Fehlende Anforderungen – Oft werden die Anforderungen nicht klar und deutlich, vor allen Dingen nicht vollständig genug, an das neue System festgelegt. Aussagen wie: „Genauso wie das Altsystem, nur Windows-basiert!“ sind für eine Migration nicht besonders hilfreich.

[44] Die Erfahrung des Autors legt die Daumenregel nahe, dass im Mittel ein Drittel der Beteiligten den Wechsel nicht vollziehen kann.

[45] Dies geschieht mehr oder minder freiwillig und wird spöttisch als „... *wandelt sich vom Paulus zum Saulus*“ bezeichnet.

[46] Das Team, welches die Legacysoftware bis zu ihrem Ende pflegt, wird sinnigerweise auch als das „Titanic-Orchester“ bezeichnet.

- Superprozess – Viele Unternehmen haben sehr ausgefeilte Vorgehensmodelle. Leider sind diese jedoch oft sehr generisch oder stark auf die Neuentwicklung von Software zugeschnitten. Eine kritiklose Übernahme solcher inadäquaten Modelle produziert nur Misserfolge und Frustrationen.
- Fehlendes Rückgrat – Das fehlende Durchhaltevermögen von Seiten des Managements ist ein Problem, welches in allen größeren und vor allen Dingen zeitlich lange andauernden Projekten zu Schwierigkeiten führt. Keine Migration kann ohne die explizite und andauernde Rückendeckung des Managements funktionieren.

6

Legacytransformation

...
Are you a god? would you create me new?
Transform me then, and to your power I'll yield.
But if that I am I, then well I know ...

The Comedy of Errors,
William Shakespeare

Neben den mehr oder minder „drastischen" Maßnahmen zur Veränderung eines Legacysystems, beispielsweise durch einen vollständigen Ersatz oder ein komplettes Reengineering, gibt es noch eine Zwischenstrategie, welche sanfter ist, das Transformieren. Hierbei lassen sich die extrem hohen Risiken abmildern und besser kontrollieren als bei den Radikalmaßnahmen eines Ersatzes.

Genauer betrachtet ist die Transformation eine Abfolge von kleinen, beinahe kontinuierlichen, Reengineeringschritten mit dem Ziel, die Entropie langfristig abzusenken oder zu stabilisieren, bzw. eine neue Technologie einzuführen. Aus Sichtweise der Migration handelt es sich bei der Transformation um eine Form der Permanentmigration. Auf der anderen Seite können die einzelnen Transformationen so klein gewählt werden, dass die Unterscheidung zur Maintenance, vor allen Dingen zur adaptiven Maintenance, s. Kap. 7, nur noch schwer möglich ist. Das primäre Unterscheidungsmerkmal zwischen der Transformation und der Maintenance ist der Charakter des Prozesses. Die Transformationen sind als Projekte eingebunden, während die adaptive Maintenance eine permanente Tätigkeit ist.

Das Ziel hinter der Transformationsstrategie ist es, nur einen Teil des Legacysystems abzuändern und den größten Teil des verbleibenden Systems intakt zu lassen. Die Szenarien sind hier durchaus unterschiedlich, sie rangieren von

- einem einfachen Rehosting inklusive der Portierung
- über ein Web-Enabling
- bis hin zum Aufbau einer Service Oriented Architecture, SOA, s. Abschn. 12.8.

Ein gewisses Maß an Reengineering ist dabei immer notwendig, es betrifft aber stets nur einen Ausschnitt aus dem gesamten Legacysystem. Aus heutiger Sicht bietet sich eine Service Oriented Architecture, eingebettet in eine J2EE-

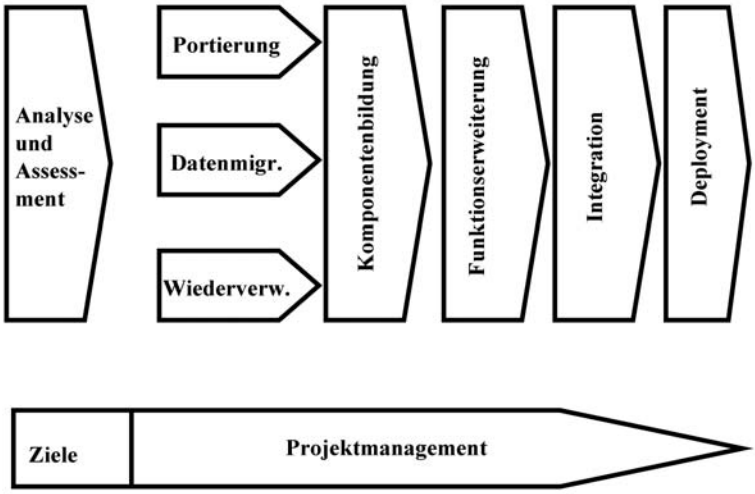

Abb. 6.1: Der Transformationsprozess

oder .NET-Architektur, als langfristiges Ziel einer solchen Migration an. Aber auch andere Architekturformen sind denkbar.

Aus dem Blickwinkel der Investition, welche in ein Legacysystem eingebracht wurde, s. Abschn. 4.16, ist die Wahl einer Transformation ein möglicher Weg das finanzielle Risiko zu minimieren. Nicht nur finanzielle Risiken tauchen auf; eine Transformation im Vergleich zum Ersatz oder dem kompletten Reengineering ist alleine deswegen eine interessante Alternative, weil 48 % aller Projekte mit mehr als 10.000 Funktionspunkten und über 65 % aller Projekte mit über 100.000 Funktionspunkten vor der Fertigstellung gestoppt werden.[1]

6.1 Transformationsprozess

Schematisch betrachtet besteht der Transformationsprozess zur Veränderung einer bestehenden Legacysoftware aus einer Reihe von Aktivitäten, s. Abb. 6.1. Die einzelnen Aktivitäten sind:

[1] Aus Sicht der Softwareevolution, s. Kap. 4, gilt für diese Projekte:

Die Revolution frisst ihre Kinder!

Dantons Tod
Georg Büchner

- Analyse und Assessment – Bevor überhaupt ein vernünftiges Vorgehen möglich ist, muss die bestehende Applikation genau untersucht werden. Eigentlich muss vor dem Starten des Prozesses eine Vision oder eine langfristige Strategie für das Legacysystem vorhanden sein.
- Portierung[2] – Das Ziel hinter einer möglichen Portierung ist zum einen eine automatische Sourcecodeübersetzung und zum anderen ein Refaktoring, d.h. eine Verbesserung der Sourcecodequalität unter der Beibehaltung aller fachlichen Eigenschaften. Das Ziel dieser Portierung kann auch der Übergang auf eine andere oder geänderte Hard- oder Betriebssystemplattform sein. Durch die Portierung bleiben im Grunde alle bestehenden Probleme erhalten und es werden durch den Sprachwechsel nur noch neue hinzugefügt.
- Datenmigration – Die Datenmigration ist ein selbstständiges, komplexes Gebiet mit eigenen Regeln und Vorgehensweisen, s. Abschn. 5.11.
- Wiederverwendung – Das Ziel ist hierbei, einen möglichst großen Teil der bestehenden Applikationen wiederverwenden zu können. Falls dies durch Portierung nicht möglich ist, kann versucht werden, einen Teil der Geschäftslogik zu extrahieren und diesen Teil völlig neu zu implementieren, s. Abschn. 5.9. Hier sind die Übergänge zwischen einem großen Reengineeringsprojekt und einem Transformationsprojekt fließend, allerdings ist der Fokus eines Transformationsprojekts stets darauf, die Änderungen zu minimieren und einen möglichst hohen Anteil an Investitionen zu erhalten.
- Komponentisierung – Eines der Hauptprobleme, warum eine Legacysoftware für viele Endbenutzer unattraktiv ist, liegt in der Architektur begründet. Meist sind Legacysysteme monolithisch aufgebaut und haben nicht mehr die Fähigkeit, sich an neue oder anders strukturierte Geschäftsprozesse anzupassen. Von daher ist eine Zerlegung in Komponenten zwecks besserer Wiederverwendung sinnvoll. Die Komponentisierung impliziert fast immer eine neue Architektur der entstehenden Software.
- Funktionale Erweiterungen – Neben der reinen technischen Umsetzung einer Legacysoftware empfiehlt es sich immer, die Gelegenheit wahrzunehmen und die Funktionalität des Systems zu erweitern.[3]
- Integration – Die umgestaltete Software muss in die neue Umgebung, beispielsweise Web- oder Applikationsserver, integriert werden.
- Deployment – Das Ende des Transformationsprozesses mit der Übergabe der Software an den Betrieb und den Beginn der Maintenancephase, s. Kap. 7.

[2] Auch unter dem Begriff der Transliteration bekannt.

[3] Oft ist dies die einzige Möglichkeit, Fachbereiche davon zu überzeugen, in eine Transformation zu investieren, da diese primär an der möglichen neuen Funktionalität interessiert sind. Da die Fachbereiche die IT finanzieren, müssen die Entscheider der Fachbereiche das Gefühl haben, einen Gegenwert für ihr Geld zu erhalten.

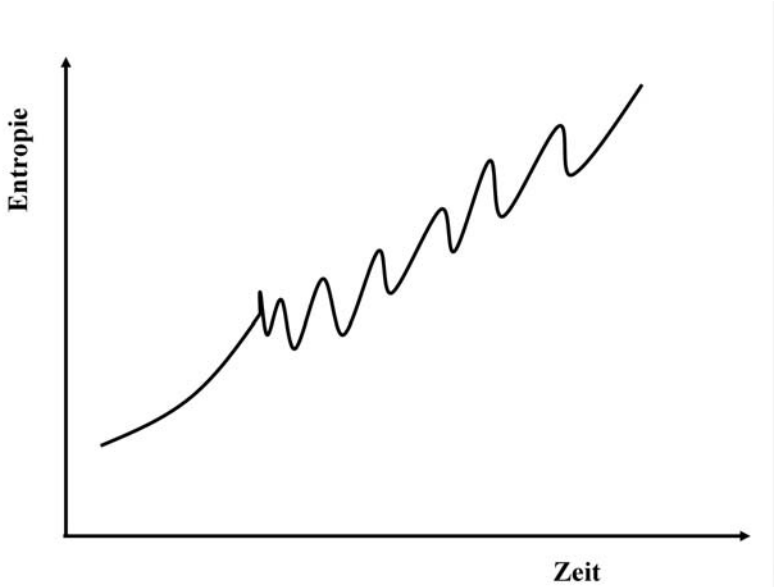

Abb. 6.2: Der Transformationsprozess mit kleinen Entropieschüben

Für diesen Transformationsprozess existieren eine Reihe mehr oder minder typische Probleme, welche in vielen Projekten zu beobachten sind. Neben den klassischen Problemen jedes Projektes:

- unklare Zielsetzungen
- inkompetentes Management
- überoptimistische Zeitplanung
- persönliche Konflikte und Egoismen

existieren typische Probleme, welche sich auf die Softwareartefakte und auf die Architektur zurückführen lassen:

- Architekturprobleme und ihre Lösungen:
  1 Monolithische Monster – Je älter eine Legacysoftware ist, desto größer die Wahrscheinlichkeit, dass sie einen stark monolithischen Charakter besitzt. Der große Nachteil dieser Architektur ist die extrem starke Abhängigkeit von Darstellung, Logik und Datenhaltung in einem großen, meist undurchsichtigen Konglomerat. Eine Zerlegung in eine Schichtenarchitektur kann nur durch eine konsequente Modellierung gewährleistet werden. Dabei sollte das bestehende System neben der fachlichen Komponentisierung auch ein Refaktoring durchlaufen, um überhaupt in der Lage zu sein, eine saubere Schichtenbildung vornehmen zu können.
  2 Betriebssystemabhängigkeiten – Die meisten Werkzeuge zur automatischen Übersetzung von Sourcecode auf eine neue Plattform enthalten

Mechanismen zur Umsetzung auf die neue Umgebung. Allerdings sollte stets berücksichtigt werden, dass die grundsätzlichen Zugriffsmuster, s. Abschn. 12.1, nicht automatisch verändert werden und einen solchen Einsatz drastisch limitieren können, wenn nicht vorher ein Refaktoring mit dem Ziel einer Schichtenbildung vorgenommen wurde.

3 Nichtintegrative Stove-Pipe-Architekturen – Es ist geradezu ein Ziel des Transformationsprozesses, diese nichtintegrativen Architekturen wieder integrativ zu machen.

4 Unklare Systemgrenzen – Speziell wenn die Legacysoftware von 4GL-Werkzeugen, welche Ende der achtziger bis Mitte der neunziger Jahre beliebt waren, so beispielsweise CSP, ADW oder IEF, erstellt wurde, ist die Gefahr von unklaren Systemgrenzen gegeben. Häufig lassen diese 4GL-Systeme den Unterschied zwischen ihren eigenen Laufzeitumgebungen und dem Betriebssystem verwaschen.[4] 4GL-Systeme sind in der Regel proprietär, bedingt durch die Tatsache, dass die Sprachen und Werkzeuge in der jeweiligen Entwicklungsumgebung solcher 4GL-Systeme sich typischerweise um ein bestimmtes Datenbankmanagementsystem gruppieren. Sie sind dann sehr eng mit den entsprechenden Datenbanken verwoben. So beispielsweise:
   - ADS/Online für IDMS – Computer Associates,
   - Natural für Adabas – Software AG,
   - PL/SQL für Oracle.

   Bei 4GL-Systemen ist eine Schichtung nur schwer zu realisieren. Da viele dieser 4GL-Werkzeuge die Fähigkeit haben, so genannten „native Code“, in aller Regel COBOL, zu generieren, ist dieser generierte „native Code“ ein guter Ausgangspunkt für ein Refaktoring. Erstaunlicherweise ist generierter Code in der Regel sehr gut lesbar. Zwar erscheint der generierte Code beim ersten Kontakt etwas ungewöhnlich – Softwareentwickler sprechen gerne von suboptimal[5] –, aber auf Dauer ist die immer wiederkehrende identische Programmstruktur sehr leicht zu erlernen. Folglicherweise läuft die mentale Modellbildung bei generiertem Code recht schnell ab. Die Tatsache, dass die meisten 4GL-Sprachen nur sehr wenige unterschiedliche syntaktische Elemente besitzen, vereinfacht die Modellbildung zum Verständnis des Sourcecodes um ein weiteres.

- Probleme aus Softwareartefakten und Lösungsansätze:
  1 Undokumentierte Funktionalität – Im Rahmen eines Application Minings muss der Sourcecode in Interfaces, Ablaufstruktur, I/O zerlegt werden. Bei diesem Prozess bleibt ein kleiner Rest zurück, welcher

[4] Pikanterweise macht Java mit seiner virtuellen Maschine genau dasselbe!

[5] Hier wiederholt sich die Geschichte. Zu Beginn der Hochsprachen wie Fortran, COBOL oder C erzeugten viele Compiler einen Assemblerzwischencode. Dieser Zwischencode wurde oft von Assemblerprogrammierern als umständlich und inperformant belächelt ...

die tatsächliche Geschäftslogik isoliert darstellt. Eine andere Möglichkeit besteht darin, die Veränderung von Daten und damit implizit der Geschäftsobjekte zu verfolgen und aus diesem Tracing die Lokation sowie das Vorhandensein der Geschäftslogik zu schließen.

2 Sourcecoderedundanzen – In Legacysystemen ist ein nicht zu vernachlässigender Teil des Sourcecodes als Folge der Evolution, speziell des Bloatings, redundant. Eines der Ziele hinter einem Refaktoring ist es, diese Redundanzen auf ein Minimum zu reduzieren.

6.2 Refaktoring

Ist einmal erkannt worden, dass ein Legacysystem zu komplex geworden ist, um sich noch länger evolutionär entwickeln zu können, so müssen Schritte zur Reduktion der Komplexität vorgenommen werden. Neben einer Reduktion der Redundanz – Redundanz führt zu Komplexität – ist das Refaktoring die zweite Methodik zur Komplexitätsreduktion. Im Gegensatz zum vollständigen Reengineering, das bei der Entropie einen revolutionären Charakter hat, produziert die Transformation eine „Sägezahnkurve“, s. Abb. 6.2. Zwar sind auch das komplette Reengineering wie auch der COTS-Ersatz in der Lage, die Entropie des Legacysystems abzusenken, beide Verfahren führen jedoch zu dem bekannten Erstanstieg bei der Volatilität, s. Abb. 4.10. Eine Transformation hingegen ändert die Volatilität in der Regel nur geringfügig ab.

Ziel des Refaktorings ist es, eine Zerlegung des Legacysystems in lose gekoppelte oder vollständig entkoppelte Subsysteme oder einzelne Module zu erreichen. Wie kann aber ein solches Subsystem gefunden werden? Folgende Daumenregeln, in absteigender Wichtigkeit, helfen dabei:

1 Der Code muss sehr ähnlich sein, beispielsweise in einer Sourcecodedatei liegen. Spürbar wird dies, wenn bei Änderungen immer dieselben Teile abgeändert werden. Dann besitzen diese Teile eine starke Koppelung, was ein praktisches Indiz für ein Subsystem ist.

2 Wenn er verwendet wird, wird er meistens als ein Block verwendet. Die Verwendung kann andere Sourcen implizieren, so können sich speziell Datendefinitionen durch ganze Legacysysteme ziehen. Charakteristisch für eine starke Koppelung ist auch, dass für eine Änderung eines Teils der Legacysoftware immer Wissen über den internen Aufbau eines anderen Teils des Legacysystems notwendig ist.

3 Das Subsystem enthält gemeinsame Funktionalitäten.

Die typischen Hilfsmittel zur Entkoppelung der Subsysteme sind Patterns, s. Kap. 13. Dazu zählen vor allen Dingen:

- Gateway
- Adaptor
- Façade

Die Entkoppelung hat als Ziel, das gesamte Legacysystem zu stabilisieren; siehe auch S. 199, wo dieser Mechanismus aus Sicht eines Small-Worlds-Netzwerk betrachtet wird.

6.3 Zielplattformen

Die möglichen Zielplattformen für eine Transformation können durchaus sehr unterschiedlich sein. Es bieten sich für ein Legacysystem zur Zeit folgende Alternativen[6], s. Kap. 12:

- J2EE
- .NET
- COBOL
- SOA

Zwar ist man nicht aus technischen Gründen auf diese Plattformen beschränkt, jedoch resultieren einige Vorteile aus der Wahl zwischen diesen Alternativen.

6.3.1 SOA

Bei der Wahl einer Service Oriented Architecture, s. Abschn. 12.8, wird von der bestehenden Legacysoftware primär die Präsentationsschicht bzw. der Steuerungsteil herausgelöst und aus dem verbleibenden Rest ein System aufgebaut, welches auf direkte Aufrufe reagieren kann. Der Vorteil einer solchen Architektur liegt in der möglichen zukünftigen Wiederverwendung der bestehenden feingranularen Funktionalitäten.

Üblicherweise wird der „Backendteil[7]“ einer SOA auf einer Mainframe verbleiben. Dies hat zur Folge, dass die bestehenden COBOL- und eventuellen Assemblerprogramme in der neuen Architektur weiterhin genutzt werden können. Als Kommunikationsprotokolle bieten sich MQ-Series, IMS/DC und CORBA[8] an. Die entsprechenden Applikationsserver können dann in einer beliebigen Architektur existieren, sei es J2EE oder .NET. In einer SOA ist die gesamte fachliche Funktionalität innerhalb der Services auf der Serviceschicht angesiedelt.

Bei der Frage, welche Services in dieser SOA enthalten sein sollten, besteht die Herausforderung darin, das Legacysystem in solche Blöcke zu zerlegen, welche später als „Objekte“ im Sinne von Geschäftsobjekten dienen können.

---

[6] Keine dieser „Alternative“ kommt in der Praxis in Reinkultur vor. Die meisten realen Systeme mischen verschiedene Zielplattformen.

[7] Eine 3-Tier-Architektur wird oft als eine Struktur mit „Front Tier “, „Middle Tier“ und „Backend“, s. S. 12.3.3, dargestellt.

[8] CORBA ist im Grunde genommen kein echtes Protokoll, aber für die Diskussion in diesem Buch ist dies irrelevant.

Dieser Umschwung hin zu einer objektorientierten Darstellung gefolgt von einer Neugestaltung der Geschäftsprozessimplementierung kann drei verschiedene Alternativen zur Bestimmung von Services nutzen:

- Datengetrieben: Mit dieser Strategie werden die Datenstrukturen zur Auffindung der Services genutzt. Sinnvoll ist dieses Vorgehen, wenn das Datenmodell stabil und gut entwickelt ist, oder eine Datenmigration, s. Abschn. 5.11, zum gegebenen Zeitpunkt gescheut wird.
- Funktionsgetrieben: In diesem Fall werden die vorhandenen Funktionen der Legacysoftware gekapselt und als Services zur Verfügung gestellt. Dies ist eine Strategie, welche recht schnell die ersten Ergebnisse zeigt, aber auf Dauer die Komplexität nicht absenkt.
- Objektgetrieben: Die anspruchsvollste Strategie ist die objektgetriebene Strategie. Hierbei wird das neue System aus Funktionen und Datenstrukturen simultan aufgebaut.

6.3.2 COBOL

Trotz aller Vorurteile, COBOL gehört zu den am besten portierbaren Sprachen überhaupt. COBOL-Compiler und Laufzeitumgebungen sind für praktisch alle wichtigen Plattformen erhältlich, ja es gibt sogar ein .NET-COBOL von Fujitsu-Siemens.

Viele Compilerhersteller im COBOL-Umfeld bieten moderne Oberflächenbibliotheken sowie eine gute Integration relationaler Datenbanken, in Form von embedded SQL, an. Nach geeignetem Refaktoring ist COBOL eine sehr gute Wahl als Zielplattform. Neben diesem mehr technisch fundierten Grund existiert noch ein weiterer Vorteil eines ausgedehnten COBOL-Einsatzes: Alle Unternehmen, die große Bestände an Legacysoftware in der Maintenance haben, besitzen in der Regel auch einen hohen Prozentsatz von Mitarbeitern mit exzellentem COBOL-Know-how. Für die Umstellung dieser Mitarbeiter[9] auf C# oder Java wäre eine Mindestzeit von neun Monaten zu veranschlagen. Im Gegensatz zu den Versprechungen von Schulungsanbietern braucht man in diesen Sprachen sehr viel Erfahrung, um in der Lage zu sein, selbstständig zu programmieren.

Ein weiterer Grund für COBOL als mögliche Zielplattform ist die Tatsache, dass es Codegeneratoren für unterschiedlichste Sprachen gibt, die von einer anderen Sprache nach „native“ COBOL generieren können. So wird heute auf dem Markt angeboten:

- CSP nach COBOL
- PL/I nach COBOL
- Easytrieve nach COBOL

Wenn man den bestehenden Einsatz von COBOL betrachtet, so zeigt sich Erstaunliches. Laut Gartner Group gilt für COBOL:

[9] Siehe auch Fußnote, S. 134.

- 70% der Daten in der Geschäftswelt werden mit COBOL verarbeitet.
- 90% aller Transaktionen an Bankautomaten werden mit COBOL durchgeführt.
- 30 Milliarden COBOL-Transaktionen finden täglich statt.
- Die Gesamtinvestitionen in COBOL-Software beträgt mehr als 3 Trillionen US$.
- Alle Fortune-100-Unternehmen nutzen COBOL.
- 492 der Fortune-500-Unternehmen nutzen COBOL.

Trotz des großen öffentlichen Interesses für andere Sprachen, ist COBOL der Standard in den betriebswirtschaftlichen Sprachen.

6.3.3 .NET und J2EE

Die Wahl zwischen J2EE und .NET ist eine oft hitzig geführte Debatte. Grundsätzlich sind beide Plattformen geeignet, das Transformationsergebnis von Legacysoftware aufzunehmen, und beide stammen aus dem Bereich der Applikationsserverplattformen.

Allerdings sollte berücksichtigt werden, dass sich durch die Wahl von .NET ein Unternehmen auf Windowsplattformen konzentriert, was bei der hohen Marktdurchdringung von Microsoft im Client-Markt, mehr als 90%, sowieso schon gegeben ist. Der Plattform J2EE ist zugute zu halten, dass die Integration in „klassische“ Teleprocessingmonitore wie IMS/DC und CICS, genau wie nach MQ-Series, via JMS, etwas einfacher ist.

Auf der Sourcecodeebene ist zu beachten, dass die Sprachen, C# und Java, nur sehr mühsam eine Festkommaarithmetik beherrschen, etwas, was in der Sprache COBOL zum Sprachstandard gehört.

6.4 Projektmanagement

Das Projektmanagement eines Transformationsprozesses, s. Abb. 6.3, hat ähnlich wie ein echtes Migrationsprojekt einige Besonderheiten, welche in einer generischen Form des Projektmanagements so nicht gegeben sind. Speziell der kontinuierliche Aufbau von Wissen ist eine der Besonderheiten dieser Form von Projekten. Die betrachtete Legacysoftware ist typischerweise wenig dokumentiert und die Fachbereiche wissen meistens wenig über die konkreten Ausprägungen der Geschäftsprozesse, welche im Legacysystem implementiert wurden. Daneben fehlt unter den Softwareentwicklern oft auch die technische Expertise große Teile der Legacysoftware zu verstehen.

Das wichtigste bei einem Transformationsprozess ist das Projektmarketing, denn innerhalb des Unternehmens wird die Weiterverwendung eines Legacysystems nicht als Fortschritt, sondern höchstens als Status Quo, in den meisten Fällen jedoch als Rückschritt, empfunden. Hier ist es unabdingbar, eine klare Informationspolitik über die Vorteile und Risiken einer solchen Strategie zu vermitteln. Die meisten Consultingunternehmen versuchen ihre Kunden

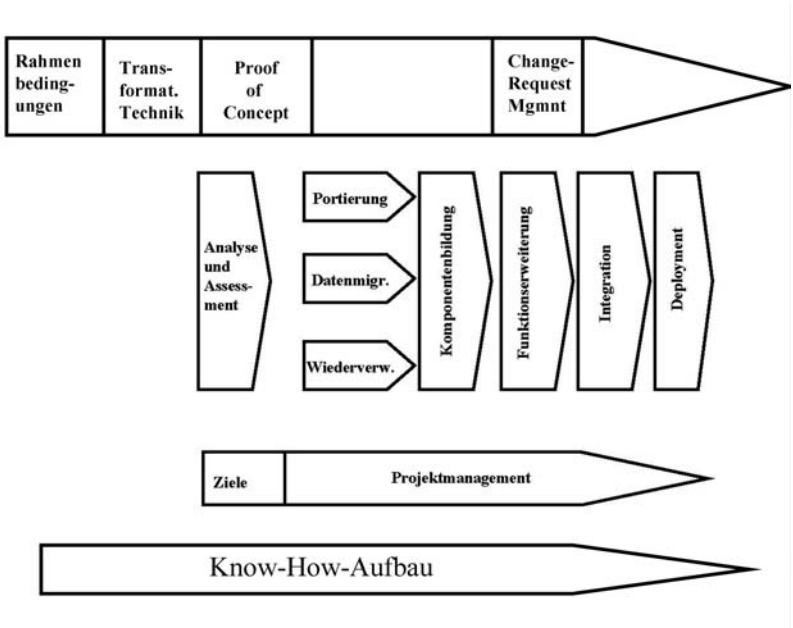

Abb. 6.3: Das Transformationsprozess-Projektmanagement

zu einem kompletten Reengineering zu überreden, dies schafft hohe Umsätze und, bedingt durch das niedrige Durchschnittsalter innerhalb der Consultingunternehmen, haben diese kaum echtes COBOL- und Mainframe-Know-how, weshalb eine solche Empfehlung für die Consultingunternehmen auch Sinn macht.

6.5 Transformationsbeispiel

Bisher wurden hauptsächlich einzelne Schritte in einer lang andauernden Migration betrachtet, nun sollen diese einzelnen Teile anhand zweier Beispiele in einen mehr strategischen Kontext gebracht werden. Die Beispiele sind zwei verschiedene Szenarien: Zum einen der Aufbau eines neuen Systems aus einem Legacysystem via User Interface Integration und zum anderen der Aufbau einer „neuen“ Software via Message-Driven-Integration.

Der Ausgangspunkt für beide Betrachtungen ist ein Legacysystem, welches aus einer Reihe von eng gekoppelten Legacysoftwareteilen besteht, welche aber untereinander „siloartig“ abgeschottet sind, die klassischen „Stove-Pipe[10]-Legacysysteme“.

[10] Nach dem englische Begriff für Schlot. Die einzelnen Applikationen bilden parallel stehende, isolierte, Schlote für die Daten.

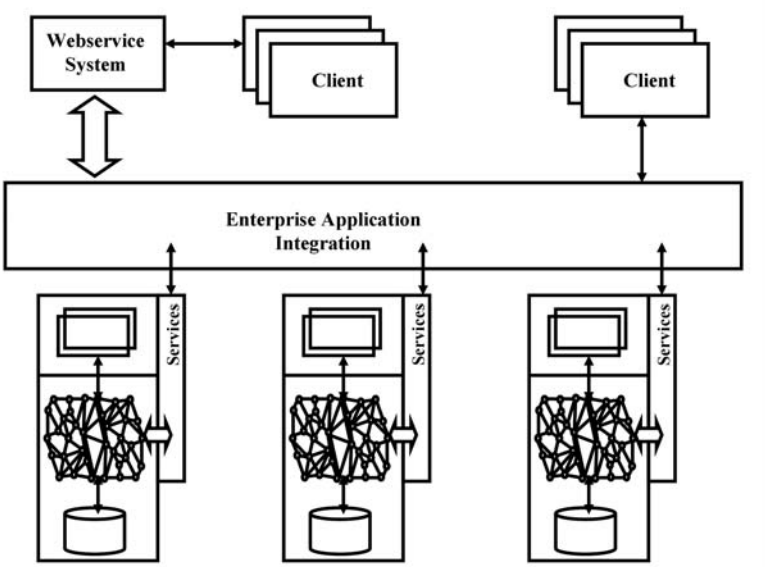

Abb. 6.4: Das User Interface Enablement

6.5.1 User Interface Integration

Unter dem Begriff User Interface Enablement wird die Bereitstellung bzw. Umwandlung von Legacysystemen hin zu einer stärkeren Service Oriented Architecture auf Basis der Benutzerschnittstelle verstanden. Aufgrund dieses Vorgehens hat sich der Begriff User Interface Integration dafür eingebürgert. Diese Integrationsform ist keine Enterprise Application Integration im eigentlichen Sinne, da die Applikationen nicht in beliebiger Art und Weise im Rahmen einer für EAI üblichen Busarchitektur, s Abschn. 12.6, integriert werden. Im Idealfall erzeugt das User Interface Enablement mit einem webservicebasierten User Interface eine einfache Koppelungsmöglichkeit innerhalb der Präsentationsschicht, so z.B. via einem Portal.[11] Bei einer echten Enterprise Application Integration würde die Verknüpfung nicht in der Präsentation, sondern innerhalb der Geschäftslogik oder Geschäftsobjektschicht stattfinden. Dieses User Interface Enablement läuft in drei Schritten ab:

1 Empowerment – Der erste Schritt ist die Nutzung der vorhandenen Schnittstellen der diversen Applikationen. Diese werden in einer webbasierten Oberfläche, beispielsweise innerhalb eines Portals, zur Verfügung gestellt. Die breite Streuung der „webifizierten“ Funktionalität erzeugt eine dramatische Steigerung der Nutzung dieser Applikationen, da nun die

[11] Der Begriff Präsentationsschicht wird hier im Sinne der Client-Server-Architektur genutzt. Innerhalb eines J2EE-Application-Servers residiert diese Präsentationsschicht in einem serverseitigen Web-Container.

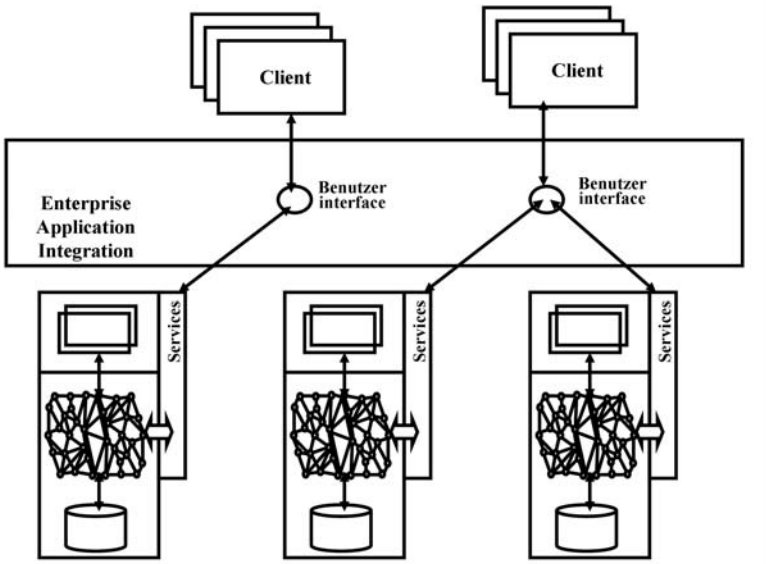

Abb. 6.5: Das User Interface Empowerment

Zahl der potentiellen Nutzer drastisch ansteigt, s. Abb. 6.5. Allerdings steigt auch die Komplexität an, da die zusätzlichen Teile die Komplexität erhöhen.

2 Vereinfachung – Der nächste Schritt ist, die aus dem Empowerment resultierende Funktionalität zu nutzen, um neue vereinfachte Benutzerschnittstellen zu bauen, welche die Informationen zielgruppengerecht liefern können. Die während des Empowerments produzierten Benutzerschnittstellen bilden in der Regel nur das interne Rollenkonzept der jeweiligen Applikation ab, nicht jedoch das Konzept, welches durch die Nutzung eines Portals oder einer Oberfläche entsteht, die einer breiteren Masse zugänglich ist. Typisch für einen solchen Schritt sind so genannte Self-Service-Programme[12] im Bereich Personalverwaltung, siehe auch Abb. 6.6.

3 Aggregation – Der dritte Schritt in der Integrationsstrategie bedeutet die Aggregation von Daten aus verschiedenen Applikationen oder auch Organisationen, um damit neue Funktionalität zu schaffen. Diese Aggregation ist sowohl durch die vereinfachten Interfaces, als auch den Einsatz von neu geschaffenen Webservices auf Basis des Legacysystems möglich geworden. Nun können im Gegensatz zu den beiden vorherigen Schritten, welche sich auf die Effizienzsteigerung konzentrierten, völlig neue Informationen geschaffen bzw. neue Prozesse entwickelt werden, siehe auch Abb. 6.7.

[12] Auch als Kioskfunktionalität bekannt.

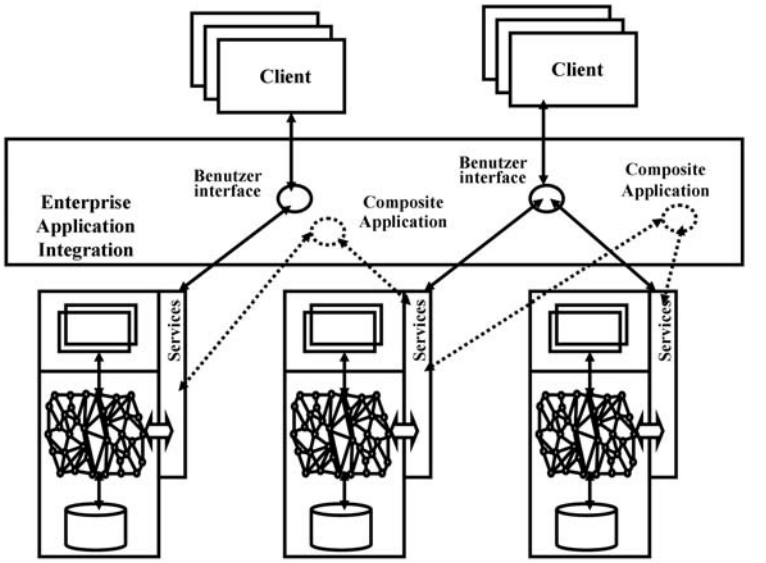

Abb. 6.6: Auflösung der Koppelung und anschließende Vereinfachung

Die Legacysoftwareteile des Stove-Pipe-Legacysystems bilden den Ausgangspunkt für die Migration. Im ersten Schritt bleibt diese enge Koppelung erhalten, s. Abb. 6.5, da zunächst nur ein User Interface Enablement vorgenommen wurde. Damit wurden eine Reihe von Benutzerschnittstellen geschaffen, welche schon einen Fortschritt an sich gegenüber der alten Legacysoftwarearchitektur darstellen.

Im nächsten Schritt, s. Abb. 6.6, wird die enge Koppelung der Legacysoftwareteile untereinander aufgelöst und mit Hilfe eines Enterprise-Application-Integration-Systems abgebildet. Dieses System wurde zwar schon für den ersten Schritt zum Enablement eingesetzt, entfaltet jetzt aber erst seine vollständige Wirkung. Ein solches erstelltes System hat nun den Vorteil, dass Portale recht einfach unterstützt werden können.

Der finale Schritt ist es, völlig neue, aggregierte Applikationen[13] produzieren zu können, s. Abb. 6.7. Auf diese Art und Weise ist es möglich aus einem mehr oder minder starren Legacysystem ein neues System zu schaffen, welches sich flexibel verhält. Allerdings kann das nur der Anfang sein, da das entstehende System zwar nicht mehr das Problem einer typischen „Stove Pipe“ Architektur hat und verteilter ist, dafür aber die Entropie erhöht ist, bei gleichzeitiger Absenkung des Maintainability Index. Dies ist eine Situation, welche langfristig betrachtet nicht gut tragbar ist. Von daher muss ein echtes

[13] Korrekterweise können Informationen neu verknüpft werden, um damit neue Aussagen zu treffen.

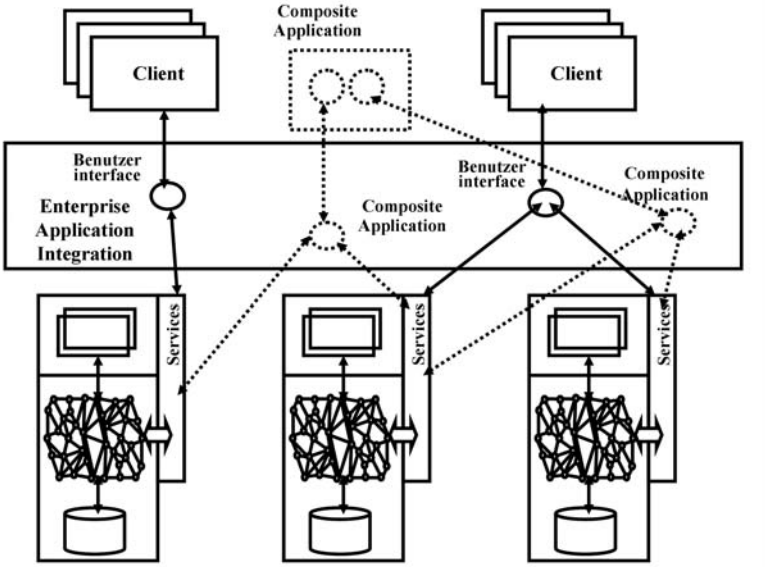

Abb. 6.7: Schaffung aggregierter Applikationen

Refaktoring innerhalb der wiederverwendeten Legacyservices einsetzen, um das System in eine Evolutionsphase des Lebenszyklus zu bewegen.

6.5.2 Message Driven Integration Roadmap

Im Fall eines Message-Driven-Ansatzes ist das Vorgehen etwas anders. Eine Message-Driven-Integration ist dann sinnvoll, wenn die Koppelung der Legacysoftwareteile untereinander über Transaktionsprotokolle, beispielsweise IMS oder CICS, abläuft oder dateibasiert ist. In diesem Falle sind die Legacysoftwareteile schon a priori isolierbar. Trotzdem sind sie eng miteinander gekoppelt, s. Abb. 6.5.

Der erste Schritt zu einem neuen System ist es, alle Legacysoftwareteile durch das entsprechende Message-Protokoll miteinander zu verbinden, s. Abb. 6.8, so dass nach dieser Maßnahme eine Enterprise Application Integration auf der Ebene der Legacysoftwareteile vorhanden ist.

Auf dieser Basis werden nun sofort, ohne den Umweg des Enablements, neue aggregierte Applikationen geschaffen, s. Abb. 6.7, da sich der darunter liegende Integration Broker hervorragend für eine solche Aufgabe eignet. Integration Broker auf einer Message Basis, gleichgültig von der konkreten Implementierungsform, eignen sich hervorragend dazu Legacysysteme geschickt miteinander zu koppeln; einziger Nachteil ist ein oft merkbare Latenzzeit[14].

[14] Unter der Latenzzeit versteht man im Allgemeinen das Zeitintervall vom Ende eines Ereignisses bis zum Beginn der Reaktion auf dieses Ereignis.

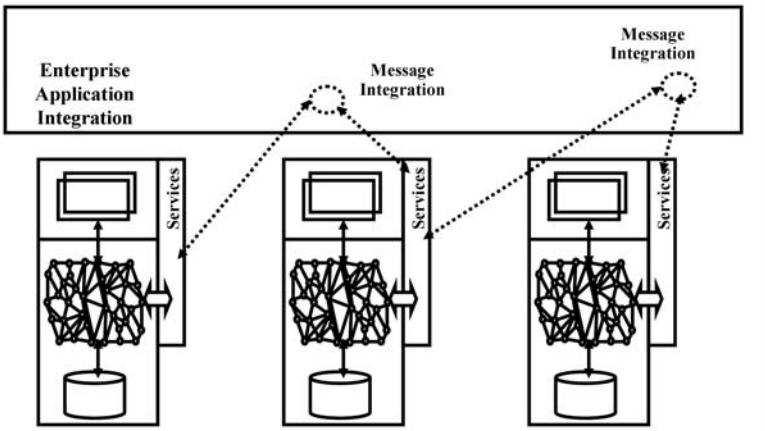

Abb. 6.8: Message Driven Integration

Ein weiterer großer Vorteil eines Message-Driven-Ansatzes ist die Fähigkeit asynchrone Koppelungen vornehmen zu können. Diese Asynchronität hat bei heterogenen oder verteilten Systemen den immensen Vorteil einer sehr viel höheren Ausfallsicherheit im Vergleich zu einer synchronen Koppelung.

7

Maintenance

... At the time of my death it is my intention that the then current versions of TEX ... be forever left unchanged, except that the final version numbers to be reported in the "banner" lines of the programs should become TEX, Version π ...
From that moment on, all "bugs" will be permanent "features".

Donald E. Knuth

Die Softwaremaintenance[1] ist die wohl limitierendste Größe in der gesamten Applikationsentwicklung. Die Maintenance spielt sowohl im Rahmen der Forschung als auch der Lehre und Ausbildung leider nur eine untergeordnete Rolle, s. S. 1. Die Hauptziele hinter der Maintenance sind wie folgt:

- Die Kontrolle über die täglichen Funktionen der Legacysoftware zu erhalten und sie somit auch den Endanwendern weiter zur Verfügung stellen zu können.
- Die Kontrolle über die Veränderung der Legacysoftware zu behalten.
- Die aktuell benutzte fachliche und technische Funktionalität zu erhalten.
- Eine zukünftige Verschlechterung des Systems zu verhindern. Die Verschlechterung zu verhindern ist eines der schwierigsten Unterfangen, da sich der Entropiesteigerung entgegengestellt werden muss und fast jede Veränderung die Entropie erhöht.

Die Softwaremaintenance wird offiziell nach dem IEEE Standard 1219 definiert durch:

> *The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.*

Folglich beginnt die Maintenance erst, nachdem die Software ausgeliefert wurde. Für die Legacysysteme ist eine solche Diskussion, wann Maintenance beginnt, müßig, da ein aktives Legacysystem per definitionem stets ausgelie-

[1] *Maintenance* aus dem Lateinischen: manu tenere – in der Hand halten.

fert ist. Es unterliegt, auf Grund des Alters der Software[2], auch immer einer Maintenance. Die Legacysoftware befindet sich stets im Zustand der Evolution bzw. des Servicing, s. Kap. 3. Ein weiterer Aspekt hinter der Maintenance ist, dass sie in der Regel für die Endbenutzer und nicht für die Software durchgeführt wird, was den hohen Anteil an Umgebungseinflüssen in der Maintenance erklärt.

Aber Maintenance existiert nicht im luftleeren Raum. Die Legacysoftware wird durch drei Basisfaktoren beeinflusst:

- Umgebung – Die Software lebt in einer technischen und organisatorischen Umgebung, welche sich permanent verändert und damit auch Druck auf eine Änderung der Software auslöst.
- Endbenutzer – Damit eine Software weiterhin eingesetzt wird, muss sie die sich ändernden Bedürfnisse der Endbenutzer erfüllen. Diese sind nicht unbedingt immer funktionaler Natur, auch in den Erwartungen der Endbenutzer gibt es Modeströmungen.
- Maintenancepersonal – Die Softwareentwickler, welche die Software verändern, sind auch beeinflussende Faktoren für die Maintenance.

Ein wichtiger Aspekt der Maintenance ist die Unzufriedenheit der Endbenutzer der Legacysoftware mit der Maintenance. Die Endbenutzer haben sehr oft das Gefühl, für ihr Geld nichts Adäquates zu erhalten, da die Ergebnisse der Maintenance meist nicht aktiv kommuniziert werden. Im Fall der Fehlerbeseitigung sehen die Endbenutzer es als eine Selbstverständlichkeit[3] an, dass Fehler beseitigt werden.

Konkrete Zahlen für Softwaremaintenance zu nennen ist relativ schwierig, da viele Unternehmen keinerlei Interesse an der Publikation solcher Zahlen haben. Die verlässlichsten Angaben über Softwaremaintenance – zumindest lassen sie sich in diesen Fällen einfach verifizieren – stammen aus dem Bereich der Open-Source-Software. Selbst konservative Schätzungen gehen davon aus, dass über 80% der Aufwände, welche während der Lebensdauer eines Softwareproduktes entstehen, auf die Softwaremaintenance entfallen.

Im Rahmen der COTS-Software-Entwicklung gibt es wenig Interesse daran, diese Situation drastisch zu verbessern, da die meisten Softwareunternehmen einen großen Teil ihres Umsatzes aus so genannten Wartungsverträgen generieren. Diese Verträge liegen üblicherweise zwischen 10-25% der Lizenzsumme als Wartungsvertrag pro Jahr. Wird eine durchschnittliche Lebenserwartung von 5-8 Jahren pro lizenzierter Software in Betracht gezogen, so erzeugt die Wartung über die Hälfte des jährlichen Umsatzes für diese Unternehmen. Eine solche Lebensdauer lässt sich bei der COTS-Software für so genannte Horizontalsoftware, s. S. 242, so z.B. Buchhaltungssysteme oder

[2] Oft wird hier anstelle des Begriffs *Alter der Software* der Euphemismus *Reife* angewendet. Besonders Softwarehersteller nutzen gerne Euphemismen für ihre Software.

[3] Hier übertragen die meisten Endbenutzer ihre Lebenserfahrung im Umgang mit Handwerkern auf die Maintenance der Legacysoftware.

Lohnabrechnungssoftware, durchaus erzielen. Trotz dieser hohen Zahlen wird Softwaremaintenance generell als ein notwendiges Übel betrachtet, welches man erst dann, wenn es nötig wird, angehen sollte, am liebsten überhaupt nicht. Es sind primär psychologische und projekttechnische Ursachen, welche zu diesem offensichtlich betriebswirtschaftlich unsinnigen Verhalten in Bezug auf die Softwaremaintenance führen.

Die Fehleranfälligkeit der Maintenance ist auf den zeitlich sehr langen Lebenszyklus von Legacysystemen zusammen mit der traditionell hohen Mitarbeiterfluktuation in Maintenancemannschaften zurückzuführen. Diese beiden Faktoren resultieren in einer geringen Wahrscheinlichkeit, dass der ursprüngliche Designer der Legacysoftware noch zur Verfügung steht, aber gerade diese Person ist in der Regel die einzige, welche noch über ein kontinuierliches Wissen verfügt. Zwar gibt es in einigen Legacysystemen eine Menge von Designdokumentation, aber meist keine über die Designmotivation. Die Motivation hinter dem gewählten Design ist aber extrem wichtig, da nur so Entscheidungen auf eine solide Basis gestellt werden können. Die Art und Weise, wie heute dokumentiert wird, bzw. früher dokumentiert wurde, ist der Maintenancephase nicht zuträglich.

Alle Vorgehensmodelle angefangen vom Wasserfallmodell über das V-Modell bis hin zum Rational Unified Process, s. Abschn. 11.2, vernachlässigen in ihrer Grundform die Maintenance bzw. auch das Reengineering, s. Abschn. 5.9. Alle klassischen Vorgehensmodelle konzentrieren sich auf die Neuentwicklung! Aber da Maintenance ca. 80% der Gesamtkosten ausmacht, ist es ungemein interessant, Vorgehensmodelle für die Maintenance genauer zu betrachten.

Die Maintenance ist sehr eng mit der Softwareevolution verknüpft, s. Kap. 4. Die Softwaresysteme im Allgemeinen werden einem kontinuierlichen Wandel, ähnlich dem von Lebewesen, unterzogen und genau wie diese gibt es Überlebende und Fossilien als Resultat des Selektionsdruckes. Zwar gibt es noch kein geschlossenes, allgemein anerkanntes theoretisches Modell der Maintenance, aber einige Fakten sind offensichtlich:

- Eine reine Sourcecodebetrachtung ist nicht ausreichend. Auch die anderen Artefakte des gesamten Lebenszyklus, die Artefakte aus praktisch allen Evolutionsräumen, sind entscheidend.
- Legacysysteme sind keine homogenen Systeme in dem Sinne, dass exakt eine einzige Version eines einzelnen Programms das jeweilige Legacysystem darstellt. Die Legacysysteme bestehen immer aus einer Vielzahl von unterschiedlichen Programmen, welche mit den verschiedensten Werkzeugen, von CASE-Werkzeugen über unterschiedliche Programmiersprachen bis hin zu verschiedenen Versionen von Compilern derselben Programmiersprache, erzeugt wurden. Oft ist auch eine Heterogenität in der Datenhaltung zu beobachten. Diese Heterogenität hat auch ihre Auswirkungen auf die Designmaintenance sowie auf das Vorgehen während der Maintenance einer Legacysoftware.

- Die Legacysoftware besteht nicht aus einzelnen, quasi isolierten Objekten. Alle Teile des Legacysystems sind sowohl statisch als auch dynamisch miteinander verknüpft.
- Ein Großteil des Wissens über das Design geht im Laufe der Zeit verloren. Es ist primär das Wissen über die Designentscheidungen, welches verloren geht, da das finale Produkt ja direkt zu beobachten ist.
- Die Alterung der Legacysoftware kann nicht ohne Metriken, s. Kap. 2, bestimmt werden.

Interessanterweise führt eine „bessere“ Softwareentwicklung zu mehr Maintenance und nicht zu weniger! Dies erscheint zunächst etwas widersprüchlich. Bei näherer Betrachtung zeigt sich jedoch, dass sich Software, welche „besser“ gebaut worden ist, auch leichter verändern lässt. Dies hat zur Folge, dass tatsächlich mehr Veränderungen vorgenommen werden, was zu einer Erhöhung der Maintenance führt. In den meisten Organisationen gibt es für jede Legacysoftware sehr große Warteschlangen an Änderungen für diese Software, die Maintenance Backlogs; besser gebaute Software kann einfach das Backlog schneller abbauen und so mehr Raum für neue Anforderungen freimachen. Ein gut gepflegtes Legacysystem mit einem hohen Maintainability Index wird nicht durch eine risikoreiche Neuentwicklung gefährdet, von daher ist es bei solchen Systemen effektiver, die Ressourcen für die Maintenance, als für eine Neuentwicklung zu verwenden. In einer solchen Umgebung werden dann alle freien Ressourcen zur Maintenance bewusst und gesteuert eingesetzt, was zu einem hohen Maß an Zufriedenheit und Akzeptanz bei den Endanwendern aus den Fachbereichen führt.

7.1 Softwarequalität

Dem Problem der Qualität in der Software allgemein, und speziell dem Fall von Legacysoftware, kann man sich aus zwei faktisch orthogonalen Richtungen nähern. Zum einen aus der Produktsicht, wobei es manchmal fraglich ist, ob die Legacysoftware tatsächlich Produktcharakter[4] besitzt, zum anderen aus der Prozesssicht, s. Tab. 7.1.

Da sich eine Legacysoftware als solche schon etabliert hat, bleibt die Frage, wie solche Verfahren, Tab. 7.1, überhaupt anwendbar sind. Obwohl diese sich primär auf den Produktionsprozess, sprich die Ersterstellung von Software konzentrieren, lassen sie sich auch auf die Themen Migration und Maintenance anwenden.

[4] Zusätzlich zum Produkt wird im Marketing der Produktcharakter durch die klassischen „vier P's“ definiert: **P**reis, **P**latzierung, **P**rodukt, **P**romotion. Obwohl die meiste Legacysoftware diese Anforderungen nicht erfüllt, kann man bezüglich der Qualitätsfrage die Software wie ein Produkt behandeln.

[5] Die agilen Vorgehensmodelle werden als Lightweight-Prozesse bezeichnet.

[6] Dokumentenlastige Vorgehensmodelle wie RUP, EUP und V-Modell werden als Heavyweight-Prozesse bezeichnet.

Tab. 7.1: Die Ansätze für Softwarequalität

	Konformität	Verbesserung	Vorgehensmodell
Produkt	ISO 9126	„Best Practices“	lightweight[5]
Prozess	ISO 9000	CMM	heavyweight[6]
Prozess	TQM	SPICE (ISO 15504)	heavyweight[6]

Das Basisvorgehen zum Einsatz der entsprechenden Methodiken der Qualitätssicherung geschieht entweder top-down oder bottom-up:

- Top-down – In dieser Vorgehensweise wird primär der Prozess bzw. die Reife des Unternehmens und seiner Prozesse beurteilt. Die grundsätzliche Vorgehensweise ist die einer Bewertung der „fehlenden“ Prozessteile und ihre nachfolgende Einführung, bzw. die Verbesserung der vorhandenen Prozesse. Die Idee ist: Je besser und kontrollierbarer der Erstellungsprozess des Produktes, desto besser das Produkt. Da ein Vergleich aber nur schwer ohne eine Referenz möglich ist, bieten sich Vorgehensweisen wie CMM, das Capability Maturity Model, als ein Referenzsystem an, an dem das einzelne Unternehmen sich messen kann.
- Bottom-up – Bei den Bottom-up-Verfahren ist das Ziel die Messbarkeit, einmal die des Produktes, zum anderen die des Prozesses, um anhand der Messungen Differenzen und Verbesserungen vorschlagen zu können. Die Basis eines solchen Verfahren ist stets die Einführung von Metriken, s. Kap. 2.

Die beiden Reifegradverfahren CMM und SPICE unterscheiden sich primär in der unterschiedlichen Granularität von Reifegrad. Das CMM ist ein „Staged“-Modell und versucht den Reifegrad des gesamten Unternehmens zu klassifizieren. Die Klassen reichen von 1, d.h. unvollständig, bis zu 5, d.h. einem optimierenden Unternehmen:

1 Initial – Der Prozess ist ad hoc und im Grunde chaotisch. Der Erfolg hängt alleine von der Anstrengung und dem Glück einzelner Mitarbeiter ab. Organisationen, welche sich primär ihrer Softwareproduktqualität verschrieben haben, setzen heute oft agile Methoden ein. Aus Sicht von CMM oder SPICE sind diese Organisationen völlig chaotisch.
2 Wiederholbar – Es existiert ein Basisprojektmanagement. Die notwendigen Managementdisziplinen erlauben eine Wiederholbarkeit von Projekterfolgen in ähnlichem Umfeld. Ob dies überhaupt für eine Maintenance möglich ist, erscheint fraglich, für eine detaillierte Diskussion im Falle von Maintenance s. Abschn. 7.11.
3 Definiert – Der Softwareprozess ist beschrieben und definiert. Alle Aktivitäten sind dokumentiert und es existiert ein akzeptiertes Vorgehensmodell. Primär prozessorientierte Organisationen, welche erfolgreich standardisierte Vorgehensmodelle einsetzen, fallen in diese Kategorie.

4 Managed – Messungen bezüglich der Produkte sowie der Prozesse werden vorgenommen und zur Steuerung eingesetzt. Beim Erreichen dieser Stufe ist der Prozess nicht nur qualitativ, sondern auch quantitativ erfasst worden. Damit ein Outsourcingunternehmen, s. Kap. 8, überhaupt sinnvoll arbeiten kann, sollte dieses Niveau die Voraussetzung sein.
5 Optimierend – Ein kontinuierlicher Verbesserungsprozess, welcher über ein Feedback-System, analog Abb. 7.4, implementiert worden ist, muss innerhalb des Unternehmens vorhanden sein um diese Stufe zu erreichen. Erst sehr wenige große Offshoring-Unternehmen, fast nur solche mit einem Geschäftsschwerpunkt[7] in den USA, haben diese Stufe erreicht.

Das SPICE-Modell beinhaltet die gleiche Klassifikation, allerdings auf Prozessebene. Innerhalb von SPICE ist es möglich, dass sich einige Prozesse auf Stufe 4 und andere auf Stufe 2 befinden.

Das CMM ist in den USA und in dem klassischen Outsourcingland Indien sehr beliebt, während sich europäische Unternehmen öfters dem SPICE-Modell zuwenden. Die beiden Hauptprobleme für CMM liegen darin, dass das Modell keine theoretische Basis besitzt. Zum einen wurde es einfach vom SEI[8] als Modell postuliert und dient dem amerikanischen Verteidigungsministerium als Hilfsmittel zur Beurteilung von Softwarelieferanten. Zum anderen ist eine rein quantitative Nutzung von Metriken, ohne die entsprechende Technologie zu berücksichtigen, sehr fragwürdig.

Das Capability Maturity Model als Verbesserungsmodell hat das Defizit, dass nicht beschrieben wird, wie „Reifegrad“ gemessen werden kann. So wird beispielsweise von CMM gefordert, dass in allen Schlüsselaktivitäten Messungen vorgenommen werden sollen, um den aktuellen Zustand der Aktivitäten zu bestimmen. Allerdings wird erst ab der Stufe 4, Managed, ein quantitatives Prozessmanagement explizit erwähnt. CMM, wie alle formalisierten Prozessmodelle, führt langfristig dazu, dass die Unternehmen immer risikoärmere Schritte unternehmen, was zur Folge hat, dass sie technologischen Fortschritt erst sehr spät wahrnehmen. Diese Grundhaltung führt in einer sich rapide ändernden Technologielandschaft zu einem De-facto-Stillstand. Dieser implizit entstehende Stillstand führt bei den Unternehmen zu einem langsamen, aber stetigen Zurückbleiben gegenüber der aktuellen Entwicklungstechnologie.

[7] Alle Unternehmen, die an die amerikanischen Streitkräfte liefern, müssen CMM Stufe 4 erreicht haben, aber nicht nur die Unternehmen selber, sondern auch deren Subunternehmen.

[8] SEI, das Software Engineering Institute an der Carnegie Mellon Universität, Pittsburgh, USA. Das SEI wird durch das U.S. Department of Defense über das Office of the Under Secretary of Defense for Acquisition, Technology and Logistics gesponsort.

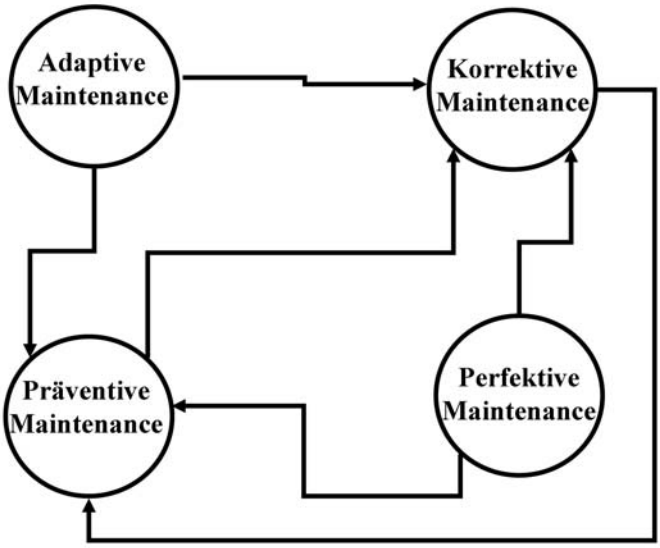

Abb. 7.1: Die gegenseitige Beeinflussung der verschiedenen Maintenanceformen

7.2 Taxonomie

Die Begrifflichkeiten innerhalb der Maintenance wechseln, abhängig von den Beteiligten. Die Mitarbeiter, die an der Ersterstellung der Legacysoftware beteiligt waren, tendieren dazu, alle Maintenancetätigkeiten als Modifikationen oder „Change Requests“[9] zu bezeichnen. Andere Softwareentwickler, welche später hinzukommen, bevorzugen meist den Ausdruck Wartung für jede Form der ändernden Tätigkeit. In Bezug auf das Bugfixing stimmt die Nomenklatur beider Gruppen wieder überein. Diese unterschiedliche Begrifflichkeit führt oft zu Missverständnissen.

Die mangelnde Begrifflichkeit führt dazu, dass Entscheidungen nur schwer vermittelbar sind bzw. falsch getroffen werden. Auf dem Gebiet der Maintenance entstand Anfang der siebziger Jahre der Begriff des „Maintenance Icebergs“. Hinter diesem „Iceberg“ verbargen sich ganz unterschiedliche Ansichten, was Maintenance bedeuten kann, rangierend vom üblichen Bugfixing bis hin zur systematischen Erweiterung der Funktionalität einer Legacysoftware[10].

[9] Für diese Mitarbeiter ist das geschaffene Legacysystem mehr eine Lebensaufgabe denn ein Projekt, d.h. sie sehen es als eine nie enden wollende Tätigkeit an.

[10] Die betroffenen Systeme wurden damals noch nicht als Legacysysteme bezeichnet, aber sie unterlagen schon der Softwareevolution.

Traditionell wird die Maintenance in vier verschiedene Formen[11] unterteilt, s. Abb. 7.1:

- korrektive Maintenance
- adaptive Maintenance
- perfektionierende Maintenance
- präventive Maintenance

Obwohl diese Einteilung willkürlich ist, wird sie in der Literatur konsequent in der vorliegenden Form zitiert. Die Einteilung richtet sich primär nach dem Auslöser für die jeweiligen Maintenancetätigkeiten.

7.2.1 Korrektive Maintenance

Die korrektive Maintenance befasst sich mit den Defekten[12] im bestehenden Legacysystem, wobei ein Defekt einen Unterschied zwischen der Benutzererwartung oder der Spezifikation und dem tatsächlichen Systemverhalten darstellt. Ein solcher Defekt muss nicht notwendigerweise softwaretechnischer Natur sein. Es gibt Defekte, welche auf Grund von fehlgeleiteten Schulungen, mangelnder Unterstützung bei der Einführung etc. entstehen. Wir wollen hier nur die „echten“ softwaretechnischen Defekte betrachten. In der Literatur findet sich für die Beseitigung dieser Defekte meist der Begriff des Bugfixing, wobei Bug den einzelnen Defekt bezeichnet.[13]

Ein solcher Defekt in einem Legacysystem kann das Resultat sein von:

- Designdefekten – Solche Designdefekte entstehen in geringer Zahl originär, d.h. schon das ursprüngliche Design hatte den entsprechenden Defekt. In den meisten Fällen sind Designdefekte Folgedefekte, welche durch unzulängliche Erweiterungen der Funktionalitäten im Sinne von adaptiver, perfektionierender oder, in seltenen Fällen, präventiver Maintenance ausgelöst werden. Die häufigste Ursache sind falsch verstandene Change Requests. Eine seltenere Möglichkeit ist, dass ein Designdefekt als Folge der Korrektur eines anderen Designdefekts entsteht; obwohl der Anteil dieser

[11] Einige Autoren erweitern diese Einteilung noch durch die Definition einer „predelivery maintenance“. Dies beinhaltet alle Tätigkeiten während des Designs um ein zukünftige Maintenance zu erleichtern.

[12] Der Begriff Defekt wurde bewusst anstelle des Begriffes Fehler gewählt. Unglücklicherweise hat der Ausdruck Fehler im deutschen Sprachraum die Konnotation eines moralischen Versagens.

[13] Im Allgemeinen wird der Begriff Bug auf Grace Hopper und das Auffinden eines Insektes in einem der ersten Computer zurückgeführt. Hierbei handelt es sich vermutlich um eine „urban legend“, da der Begriff Bug für einen Defekt schon zum Ende des 19. Jahrhunderts in der elektrotechnischen Literatur auftaucht. Admiral Grace Hopper war die erste, die den Begriff Debugging in Zusammenhang mit einem Computer benutzte, nachdem sie eine Motte in den Relais des *Harvard Mark II* fand, aber die Benutzung des Wortes Bug als Bezeichnung für einen Fehler in elektrischen Anlagen war schon zu Lebzeiten Thomas A. Edisons üblich.

Konstellation an der Gesamtmenge der Defekte mit zunehmendem Alter des Legacysystems zunimmt, stellt diese Konstellation die Ausnahme dar. Die Designdefekte als Folge von Logik- oder Codierdefekten kommen praktisch nicht vor, da solche Defekte meist sehr lokal sind. Designdefekte gehören bei den korrektiven Defekten zu den am schwierigsten identifizierbaren. Sie werden meist nach einem Wechsel des Maintenanceteams entdeckt, da nun andere Softwareentwickler mit einem anderen subjektiven Systemmodell dasselbe Legacysystem pflegen. Beim Prozess des Abgleichens des Systemmodells im Kopf des Softwareentwicklers mit der Realität in Form des Codes der Legacysoftware werden solche Defekte oft freigelegt.

- Logikdefekte – Typische Logikdefekte sind:
  - Off-by-one-Error – Speziell in der Sprache C starten Arrays mit 0 und nicht mit 1 wie in Fortran oder COBOL. Die Folge ist, dass sehr oft das erste Element eines Arrays nicht sauber behandelt wird.
  - Sortierfehler – Mangelnde Erfahrung bei den Softwareentwicklern führt zu inkorrekten Annahmen über die Sortierreihenfolge. Typisch sind hier Unterschiede bei der Sortierreihenfolge zwischen EBCDIC und ASCII.[14]
  - Gruppenwechselfehler – Eine besonders beliebte Fehlerquelle ist das Wiederaufsetzen[15] nach einem Gruppenwechsel.
- Codierdefekte – Die meisten Codierdefekte werden durch die modernen Compiler schon gemeldet und müssen dann entsprechend bearbeitet werden. In COBOL sind unterschiedliche Größen, der in den Operationen beteiligten Datenstrukturen beliebte Codierdefekte. In anderen Sprachen, so z.B. C oder Smalltalk, kann die schwache Typisierung von Variablen zu Codierdefekten führen. Solche Defekte lassen sich oft nur durch sehr spezielle Datenkonstellationen reproduzieren.

Die meisten Defekte dieses Typus werden in einem Legacysystem von Benutzern gemeldet. Dies unterscheidet die Maintenance von der Neuentwicklung; bei dieser werden die meisten korrektiven Defekte von den Betatestern aufgedeckt. Die Zahl der Defekte in Software wird meistens auf etwa 2-5 Defekte pro 1000 Zeilen Code geschätzt, eine Tatsache, welche Sprachen mit kompakter Notation bevorzugt. Allerdings tendieren Sprachen mit kompakter Notation wie APL oder C++ zu schwerwiegenden Designdefekten auf Grund ihrer Mächtigkeit.[16]

[14] Dieser Unterschied wird häufig bei Portierungen übersehen.

[15] Bei der Verwendung einer relationalen Datenhaltung und konkatentierten Suchbegriffen führt ein solcher Wechsel zu einem recht komplexen Konstrukt aus AND- und OR-Prädikaten innerhalb der SQL-WHERE-Klausel.

[16] Am deutlichsten wurde dies für C++ formuliert: *C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do succeed, you will blow away your whole leg.*

Bjarne Stroustrup
Erfinder von C++

7.2.2 Adaptive Maintenance

Die adaptive Maintenance befasst sich mit der Anpassung der Software an die Veränderungen in der jeweiligen Umgebung. Bei der adaptiven Maintenance handelt es sich meistens um Veränderungen auf den folgenden Gebieten:

- Deploymentumgebung
- Entwicklungsumgebung
- Geschäftsumfeld

Deploymentumgebung

Die Veränderungen der Laufzeitumgebung sind meistens auf die Veränderungen im Bereich von Hard- bzw. Softwareumgebungen zurückzuführen. Traditionell wird eine solche Form der Maintenance oft mit dem Begriff der Portierung belegt. In den Anfängen der IT war die Hardware- und Betriebssystementwicklung eng miteinander gekoppelt, d.h. neue Hardware wurde stets mit einem neuen Betriebssystem verbunden, so dass eine Simultanumstellung nötig war. Moderne Betriebssysteme isolieren die Hardware so stark von der applikativen Software, dass eine Maintenance auf Grund einer Hardwareveränderung selten ist. Falls auf der neuen Hardware doch ein Defekt auftaucht, wird dieser meistens durch Einsatz eines neuen Treibers behoben! Assemblerbasierte Legacysysteme bilden hier eine Ausnahme, da sie oft implizite Annahmen über die Hardware treffen.

Die Aufwände für die Umstellung ganzer Legacysysteme auf neuere Betriebssysteme wie z/OS oder Windows-XP sind für die IT-Abteilungen immens. Angetrieben werden diese Veränderungen durch das Produktmanagement der großen Betriebssystemhersteller, welche mittlerweile Produktzyklen[17] von etwa 2 Jahren für ihre Betriebssysteme propagieren. Ein Ausweichen auf Open-Source-Produkte bei den Betriebssystemen wie z.B. Linux ist nur ein scheinbarer Ausweg, da hier die Produktzyklen, je nach Derivat, unter einem Jahr liegen können. Der Hebel der Betriebssystemhersteller ist hier die Einstellung des Supports für das Produkt. Auf diese Weise wird für die Legacysoftware eine adaptive Maintenance erzwungen.

Immer häufiger wird adaptive Maintenance durch Veränderungen ausgelöst, welche nicht im Bereich Hardware oder Betriebssysteme anzusiedeln sind, da die Veränderungen COTS-Software wie beispielsweise Datenbanken betreffen und die Veränderung der Datenbanksoftware eine adaptive Maintenance im Legacysystem nach sich zieht. Je stärker und starrer ein System in

[17] Die Zykluszeit eines Produktes ist die Zeit von der ersten Auslieferung des Produktes bis zur ersten Auslieferung des Nachfolgerproduktes. Die Lebensdauer eines Produktes ist selbstverständlich länger, ca. 2-4 Zykluszeiten. Da aber ein ökonomisch denkender Hersteller nur eine begrenzte Anzahl von miteinander konkurrierenden eigenen Produkten im Markt tolerieren wird, werden „alte“ Produkte schnell vom Markt genommen.

seine Umgebung angekoppelt ist, desto höher ist die Wahrscheinlichkeit für das Aufkommen von adaptiver Maintenance.

Entwicklungsumgebung

Auch die Veränderungen in der Entwicklungsumgebung selbst können adaptive Maintenance auslösen. Hier sind die Auslöser ähnlich denen der Deploymentumgebung. Die COTS-Software-Lieferanten haben auch für den Compiler, die CASE-Werkzeuge oder die Klassenbibliotheken eigene Releasezyklen und stellen dementsprechend auch den Support für diese Produkte ein und lösen damit, im Zusammenhang mit Veränderungen in der Deploymentumgebung, adaptive Maintenance aus. Dieses Problem ist nicht auf ältere Sprachen, wie z.B. COBOL, Assembler, PL/I oder Fortran, beschränkt, auch Unternehmen, welche Software mit Java erstellten, haben leidvoll die Umstellung von JDK 1.1 auf die Version 1.2 erfahren. Dasselbe gilt für die Windowsentwicklungsplattformen, welche über die Varianten COM, DCOM, COM+ schließlich beim heutigen .NET endeten.

Geschäftsumfeld

Neben dem soft- oder hardwaretechnisch ausgelösten adaptiven Maintenanceaufkommen existiert auch eine Maintenance, welche durch die Veränderung der betriebswirtschaftlichen oder sozioökonomischen Basis ausgelöst wird. In der Regel werden diese Auslöser der adaptiven Maintenance als „Change Requests“ bezeichnet. Ein besonders markantes Beispiel für einen solchen Maintenanceauslöser war die Euroumstellung zu Beginn des Jahrtausends.

Ein weiterer Auslöser für adaptive Maintenance kann auch der Softwaremarkt sein. Es gibt durchaus Konstellationen, bei denen die Legacysoftware noch alle funktionalen und ergonometrischen Anforderungen erfüllt, aber auf Grund von Modeströmungen oder Konkurrenzprodukten mit abweichender oder höherer Funktionalität wird das bestehende System als „veraltet“ empfunden.

7.2.3 Perfektionierende Maintenance

Diese Form der Maintenance versucht die geänderten Benutzeranforderungen abzudecken. Auch diese werden als „Change Requests“ bezeichnet.[18] Interessanterweise tendiert Software dazu, kurz nach der initialen Einführung einen hohen Anstieg an perfektionierender Maintenance zu produzieren. Dieser Anstieg ist ein Zeichen für die verstärkte Nutzung der Software, da erst durch die Nutzung das Bedürfnis nach Veränderung produziert wird. Andere Quellen sind interne Veränderungen innerhalb der Organisation, welche „Change Requests“ in der Software produzieren.

[18] Aus Sicht der Anforderungsanalyse wird nicht zwischen adaptiver und perfektionierender Maintenance unterschieden, mit der Folge, dass der Begriff „Change Requests“ in beiden Fällen benutzt wird.

Tab. 7.2: Maintenancekategorien

	korrigierend	**erweiternd**
proaktiv	präventiv	perfektionierend
reaktiv	korrektiv	adaptive

7.2.4 Präventive Maintenance

Hinter dem Begriff der präventiven Maintenance verstecken sich Anstrengungen, die Maintainability, d.h. die Fähigkeit, Maintenance ausführen zu können, zu erhöhen. Diese Maintenanceform beinhaltet die Vervollständigung der Dokumentation, Kommentierung des Sourcecodes, sowie die Verbesserung der Struktur des Legacysystems. Die präventive Maintenance ist der Versuch, der Wirkung der Entropie, s. Kap. 4, zu widerstehen. Organisatorisch betrachtet wird diese Form von Maintenance von dem Maintenanceteam selbst ausgelöst. In objektorientierter Software, welche auf Vererbung von Klassen aufbaut, existiert die Spezialform des Refaktoring, bei dem neuere Strukturen im Klassenmodell realisiert werden. In der Literatur findet sich oft auch der Begriff „reductive maintenance“; hierrunter wird das Phänomen der lokalen Entropiesenkung verstanden, d.h. durch entsprechende Maßnahmen werden Teile der Legacysoftware restrukturiert mit dem Ergebnis, dass die resultierende Entropie niedriger ist als die Anfangsentropie vor der Restrukturierung. Diese Spezialform der Maintenance lässt sich aber unter dem Oberbegriff präventive Maintenance subsumieren.

Von diesen vier Formen der Maintenance ist die korrektive Maintenance die traditionelle Form der Wartung. Die anderen drei Formen werden öfters auch als Softwareevolution bezeichnet. Eine andere Einteilung kann nach der ISO-14764-Norm gewählt werden, s. Tab. 7.2. Obwohl sich die Maintenance typischerweise mit selbsterstellten Legacysystemen befasst, werden in den letzten Jahren die Fragen in Bezug auf COTS-Software immer wichtiger, s. Kap. 10. Die stärkere Marktdurchdringung von COTS-Software verlangt eine andere Form der Maintenance als eine, welche für klassische Legacysysteme geeignet ist, s. Abschn. 10.4.

7.3 Kostenverteilung

Die Gesamtkosten für die Maintenance sind sehr viel höher als es den meisten IT-Unternehmen bewusst ist. Eine Daumenregel besagt, dass die Software sich etwa 80% der Zeit im Zustand der Maintenance befindet, Legacysoftware natürlich zu fast 100%. Diese Tatsache lässt sich auch durchaus auf die Kosten umlegen; so rangieren bei Untersuchungen die Kosten für die Maintenance zwischen 49 bis hin zu 75% der gesamten Lebenszykluskosten für ein Softwarepaket. Diese immensen Kosten werden häufig ignoriert!

Tab. 7.3: Verteilung der Maintenanceanteile im Linux-Kernel

Phase	**Changelog**	**Aufwand**
korrektiv	ca. 57 %	ca. 87 %
adaptiv	ca. 2 %	ca. 0 %
perfektionierend	ca. 39 %	ca. 12 %
präventiv	ca. 2 %	ca. 1 %

Tab. 7.4: Verteilung der Maintenanceanteile nach Lienz et al. mit vermutlich großen systematischen Fehlern, da sie auf einer reinen Befragung mit unklarer Kategorisierung beruht. Neuere Untersuchungen durch Purushothaman&Perry zeigen ein anderes Bild.

Phase	**Anteil(Lienz)**	**Anteil(P&P)**
korrektiv	ca. 20 %	ca. 33 %
adaptiv	ca. 25 %	ca. 48 %
perfektionierend	ca. 50 %	ca. 8 %
präventiv	ca. 5 %	ca. 0 %
nicht klass.	0 %	ca. 10 %

Innerhalb der Maintenancephase zeigt sich bei großen Systemen eine Verteilung für die einzelnen Typen der Maintenance wie in Tabelle 7.3 dargestellt. Die vorgefundenen Werte wurden aus dem Changelog und den Aufwandsdaten für 20 verschiedene Releases des Linux-Kernels gewonnen, der niedrige Anteil an adaptiver Maintenance ist darauf zurückzuführen, dass es sich um eine Betriebssystemsoftware handelt. Für andere Legacysysteme mögen die Zahlen variieren, jedoch werden sie qualitativ ähnlich sein. Die Menge an korrektiver Maintenance, die geleistet wird, ist einfach enorm hoch.

Das in den Tabellen 7.3 und 7.4 dargestellte Zahlenwerk zeigt den relativen Anteil der jeweiligen Maintenanceform. Die Größe des Gesamtproblems wird sichtbar, wenn man zusätzlich noch die Schätzungen aus dem Jahr 2000 zur Rate zieht, bei denen die Gesamtmenge an Legacysourcecode auf 100 Milliarden Zeilen, in der Regel COBOL, geschätzt wurde, von denen 80% wiederum entweder als völlig unstrukturiert, undokumentiert oder gepatcht[19] gelten.

In der Literatur wird leider immer wieder eine alte Untersuchung aus dem Jahre 1978 von Lienz et al. zitiert, s. Tab. 7.4, welche stärker die subjektive Einschätzung der Firmen bezüglich der Maintenance darstellt. Neuere Untersuchungen zeigen deutliche Abweichungen. Der hohe Anteil an perfektionierender Maintenance bei Lienz ist wohl darauf zurückzuführen, dass bei

[19] Unter „patched“-Code verstehen Softwareentwickler Sourcecode, der an einer Stelle verändert wurde, wobei diese Veränderung nur symptomatisch war, ohne dass strukturelle Überlegungen oder systematischr Veränderungen vorgenommen wurden.

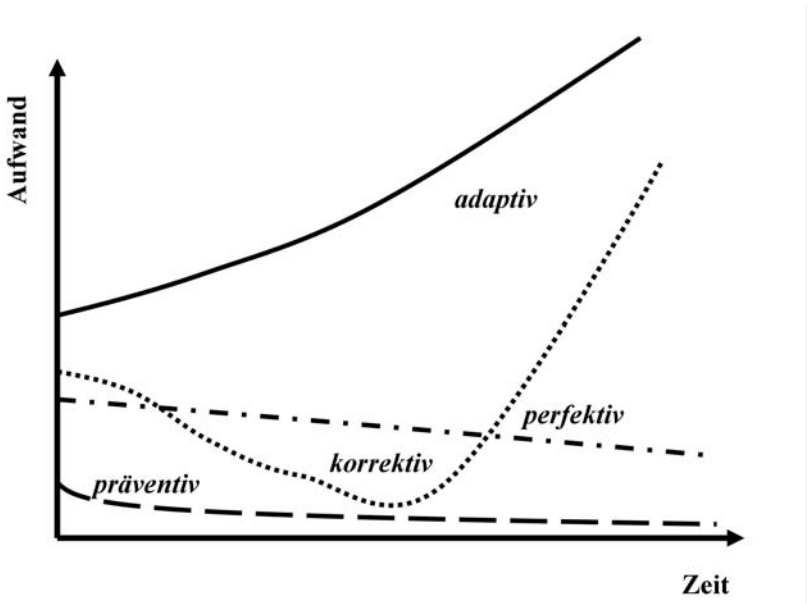

Abb. 7.2: Schematische zeitliche Entwicklung der verschiedenen Arten von Maintenance in einem Legacysystem

Befragungen Softwaremanager dazu neigen, die Maintenance an ihren Produkten mit dem Euphemismus „perfektionierend“, zu beschönigen.

Unabhängig von der Art der Maintenance lässt sich folgende generelle Tendenz feststellen: Je größer, s. Abschn.2.6 über Volatilität, und älter, s. Kap. 4 über die zeitliche Evolution der Komplexität, der Sourcecode einer Legacysoftware ist, desto aufwändiger wird die Maintenance. Hinzu kommt, dass sehr alte Legacysysteme oft nur schlecht strukturiert[20] sind bzw. die ehemalige Entwicklungssystematik nicht mehr bekannt ist.

Generell ist festzustellen, s. Abb. 7.2, dass mit zunehmendem Alter die Menge an korrektiver Maintenance abnimmt. Allerdings steigt sie mit der Realisierung der adaptiven Maintenance wieder stark an. Der Grund für dieses Verhalten ist, dass auf Dauer die adaptive Maintenance wiederum Fehler produziert.

Obwohl Restrukturierung die Maintenancekosten senkt, ist es sehr unwahrscheinlich, dass die Endbenutzer die entsprechenden Aufwände finanzieren, auch wenn ihnen die Langzeiteffekte der Restrukturierung klar gemacht werden können. Für die meisten Unternehmen sind die Gewinne im Vergleich zu den Kosten und Risiken eines solchen Projektes einfach zu niedrig. Endbenutzer sind primär an Funktionalität und Performanz und sehr wenig an Wartbarkeit interessiert.

[20] Schlecht im Sinne eines nicht nachvollziehbaren Gesamtbauplanes der Software.

Tab. 7.5: Maintenance und Produktentwicklung

Charakteristikum	**Maintenance**	**Produktentwicklung**
Ziel	Befriedigung von Anforderung	fertiges Produkt
Verhalten	reaktiv	aktiv
Planungsgröße	Ressourcenauslastung	Leistungsumfang
Kenngröße	Reaktionszeit	Termintreue

7.4 Maintenanceservices

Jede Form von Maintenance sollte, im Gegensatz zur traditionellen Neuentwicklung, welche einen starken Gewerkcharakter hat, als eine Art der Dienstleistung betrachtet werden, s. Tab. 7.5.

Eine Dienstleistung[21] unterscheidet sich von der klassischen Produkterstellung durch folgende Eigenschaften:

- Dienstleistungen sind nicht direkt greifbar[22]. Folglich können sie weder inventarisiert noch patentiert werden und ihr Wert ist nur schwer quantifizierbar.
- Dienstleistungen sind heterogener und vielgestaltiger als Produkte, da sie von Menschen direkt erzeugt werden. So sind Kategorien wie Wiederholbarkeit und Vorhersagbarkeit oft nur schwer anwendbar.
- Dienstleistungen werden fast gleichzeitig produziert und verbraucht, im Gegensatz zu Produkten, welche meistens eine Lagerhaltung haben. Zentralisierung und Massenproduktion sind für Dienstleistungen problematisch.
- Dienstleistungen sind „verderblich“[23], nicht so Produkte, d.h. Dienstleistungen können nicht gespeichert werden.
- Ein Schwerpunkt besteht im direkten Kundenkontakt. Dies hat zur Konsequenz, dass die Qualität der Dienstleistung von der Fähigkeit des Kunden abhängt, sich zu artikulieren.
- Die Qualität einer Dienstleistung hängt von vielen sehr schwer kontrollierbaren Faktoren ab.
- On-demand-Delivery – Die Dienstleistungen werden meistens direkt durch die Nachfrage nach ihnen ausgelöst.
- Im Gegensatz zu Produkten ist die Preisbildung bei Dienstleistungen recht schwer.

Es sollte trotzdem im Auge behalten werden, dass die Übergänge zwischen Gewerkcharakteristika und Dienstleistungen bei der Maintenance nicht hundertprozentig sind, da perfektive Maintenance durchaus auch einen Gewerkcharakter haben kann. Trotzdem ist es besser, Maintenance als eine Dienstleis-

[21] Dienstleistungen werden hier im Sinne der englischen „services“ verstanden.
[22] intangible
[23] perishable

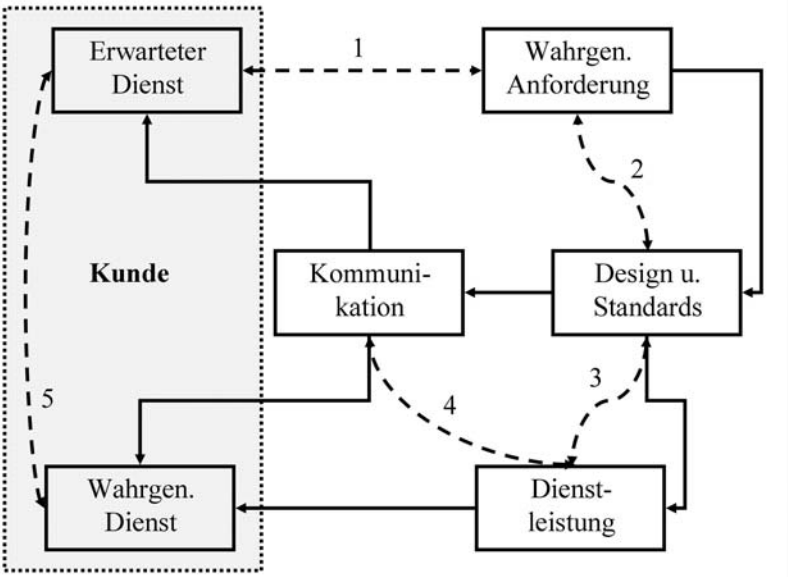

Abb. 7.3: Die Differenz zwischen wahrgenommener und erwarteter Maintenance

tung zu betrachten, speziell da die Kunden bzw. Endanwender häufig Dienstleistungskriterien wie Pünktlichkeit, Aktualität und Qualität für die Beurteilung von Maintenance ansetzen. Ein weiterer Aspekt ist die Tatsache, dass die Steuerung von Maintenance nicht durch Projektmanagementtechniken, sondern durch so genannte Queue-Managementtechniken in Form von Prioritäten etc. geschieht.

Die allgemeine Unzufriedenheit vieler Unternehmen mit der Maintenance einer Legacysoftware hängt eng mit der Differenz zwischen der wahrgenommenen Qualität der Maintenance im Vergleich zu der erwarteten Qualität ab, s. Abb. 7.3. Diese Differenzen sind es, welche die Unzufriedenheit bei den Empfängern der Dienstleistung produzieren. Detailliert betrachtet handelt es sich um fünf Differenzen, s. Abb. 7.3, welche sich wie folgt aufschlüsseln:

1 Die erwartete Dienstleistung aus Sicht des Kunden unterscheidet sich von der erbrachten Leistung. Einer der Hauptgründe liegt in der mangelnden Kommunikation über das, was im Rahmen von Maintenance machbar ist und was nicht. Oft werden hier subjektive Messungen mit objektiven vermischt, z.B.: Der Kunde erwartet, dass der Fehler innerhalb von 2 Stunden behoben ist, die Maintenanceorganisation garantiert, dass auf den Fehler innerhalb von 2 Stunden reagiert wird, wobei für die Maintenanceorganisation Reaktion nicht gleichzeitig auch Lösung des Problems bedeutet. Im Bereich der Performanz wäre ein Beispiel: Maintenance garantiert, dass 95% aller Zugriffe der Legacysoftware innerhalb von 5 Sekunden abgear-

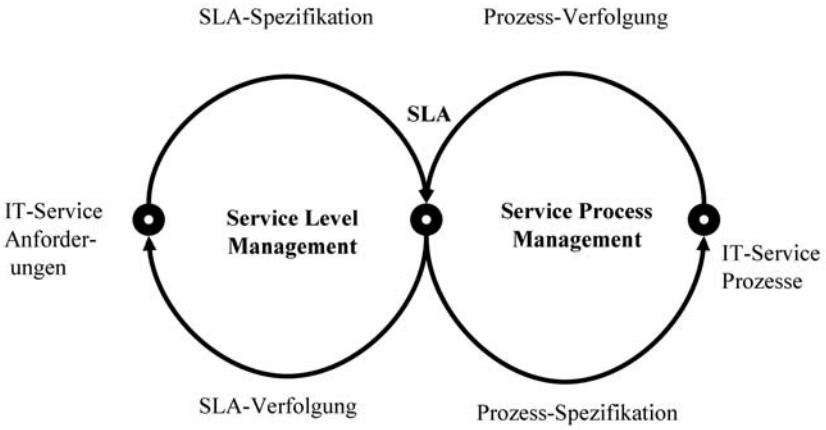

Abb. 7.4: Der Feedbackloop bei Maintenanceservices

beitet werden, der Kunde erwartet jedoch eine „sofortige“[24] Antwort für alle Zugriffe.

2 Die Definition der Maintenanceaufgaben unterscheidet sich von den vom Kunden gewünschten Aufgaben. Diese Differenz ist oft bei „professioneller“ Maintenance in Form eines Outsourcings zu beobachten. Oft sind es die elementaren Interessen der beiden Beteiligten, welche sich widersprechen, beispielsweise möchte der Kunde einen schnellen Neustart des Systems, um wieder arbeiten zu können, das Maintenanceunternehmen jedoch eine möglichst lange Zeit das System im Komplettzugriff – ohne einen Benutzer – haben um den Fehler in Ruhe finden zu können.

3 Auf Grund von organisatorischen oder qualitativen Schwächen unterscheidet sich die definierte von der gelieferten Dienstleistung. Zu diesem Problem tragen auch die Kunden bei, welche beispielsweise vorbei am Helpdesk direkt mit einem Softwareentwickler sprechen.

4 Die Information, die kommuniziert wird, stimmt nicht mit dem überein, was tatsächlich geliefert wird. Dies ist ein sehr weit verbreitetes Problem, da die Kommunikation in der Regel durch Marketing oder Vertrieb vorgenommen wird, welche ein inhärentes Interesse haben, die eigenen Fähigkeiten und Qualitätsstandards möglichst positiv darzustellen. In dieselbe Kategorie fällt auch die Beobachtung, dass bei COTS-Software-Herstellern die einzelnen Fehlerbeseitigungen oft nicht an den Kunden gemeldet wer-

[24] Menschen haben im Sekundenbereich ein wenig ausgeprägtes Zeitgefühl.

den, da sie in den allgemeinen Maintenancepool einfließen, bzw. COTS-Software-Hersteller öfters Defekte einfach abstreiten.[25]
Im Gegensatz zu den meisten COTS-Software-Herstellern gehen Open-Source-Produkte hier oft sehr innovative Wege, indem sie alle ihre Defekte offen legen und für alle im Internet auswertbar machen. Im Fall von Mozilla und Firefox geht das sogar so weit, dass einzelne „Kunden" über das Internet abstimmen können, welcher Fehler als nächster beseitigt werden soll.

5 Die Gesamtdifferenz Δ_5 ist das Resultat der vorhergehenden Differenzen:

$$\Delta_5 = \sum_{i=1}^{4} \Delta_i$$

Wie lassen sich diese Differenzen zwischen dem Anspruch und dem Bedürfnis auf der einen Seite und den Fähigkeiten und Zusicherungen auf der anderen Seite minimieren? Da Punkt 5 das Resultat der vorherigen vier Differenzen ist, reicht es aus, sich auf die ersten vier Differenzen zu konzentrieren.

Management der Commitments

Für jede Form der Maintenance ist es wichtig, die Commitments[26] sauber zu planen und zu dokumentieren. Am besten funktioniert dies in Zusammenarbeit mit dem Kunden der Maintenance; die so entstehenden Commitments bilden dann einen Teil der Spezifikation, welche ein Service Level Agreement, SLA, basierend auf den Anforderungen des Kunden, entstehen lassen. Dieser Prozess ist nicht einfach, da sich viele Teile der Commitments nicht quantifizieren bzw. nicht messen lassen. Neben dem Inhalt des Commitments bezüglich einer Dienstleistung sollten auch stets die Messmethoden festgelegt werden. In der Praxis reicht dies oft nicht aus, da Kunden zum Teil widersprüchliche Forderungen aufstellen. Die so erstellten Service Level Agreements sollten:

- messbar,
- auf den Kunden zugeschnitten sein.

[25] Es ist besonders vertrauenserweckend, bei der Fehlermeldung via Telefon von der Hotline des COTS-Software-Unternehmens die Aussage zu bekommen: „... Sie sind der Allererste, der uns so einen Fehler meldet". Gegen diese Aussage spricht, im Normalfall, eine einfache statistische Überlegung. Die Hotlinemitarbeiter werden angewiesen, Defekte nicht als Fehler zu bezeichnen und abzuwiegeln, obwohl jeder in der Softwareindustrie weiß, in welcher Größenordnung Defekte in komplexen Systemen vorhanden sind.

[26] Die korrekte deutsche Übersetzung wäre „Versprechen". Allerdings hat das Wort Versprechen im Deutschen eine Doppelbedeutung mit einer negativen Konnotation, in der Bedeutung von versehentlicher Falschaussage, von daher wird im weiteren Text der Ausdruck Commitment im Sinne von Zusicherung benutzt.

Das Service Level Agreement dokumentiert als eine Art Vertrag die Übereinkunft zwischen der Maintenanceorganisation und dem jeweiligen Kunden. Jedes Service Level Agreement sollte mindestens enthalten:

- Die Spezifikation der zu liefernden Maintenance
- Welcher messbare Service Level, d.h. wie schnell, wie fehlerfrei etc.
- Die Rahmenbedingung und Beistellpflichten des Kunden, genau diese werden sehr häufig massiv verletzt
- Ein Eskalationsverfahren
- Ein Reportingverfahren
- Die Ausschlüsse für die Maintenance, beispielsweise höhere Gewalt, Krieg, Unruhen, Bundestagwahlen, etc.

Besonders wichtig ist, dass die Service Level Agreements aus Kundensicht nachvollziehbar formuliert werden; Maintenanceorganisationen tendieren oft dazu, ihre inneren Abläufe einem Kunden „überzustülpen“.

Planung der Maintenance

Maintenancetätigkeiten resultieren aus den Service Level Agreements und müssen dementsprechend geplant werden. Hierfür werden die klassischen Planungsinstrumente, wie:

- Aufwandsschätzung,
- Risikoanalyse,
- Einsatzplan

genutzt.

Im Fall einer nichtkritischen Maintenancetätigkeit, d.h. die Maintenance muss nicht sofort beim Kunden ausgeführt werden, lassen sich die Aktivitäten zu einem Release bündeln. Die Releaseplanung folgt in der Regel zwei unterschiedlichen Strategien:

- Kalenderstrategie – Dies ist die verbreitetste Strategie, wobei hier ein Releasetermin angekündigt wird und dann, bei festem Maintenanceteam, die bis zu diesem Zeitpunkt implementierten Veränderungen freigegeben werden. Die typischen Treiber hinter dieser Strategie sind:
  - Wettbewerb unter den COTS-Software-Herstellern – Neue Releases suggerieren zukünftigen Käufern Aktualität, bzw. es ist ein funktioneller Vorsprung eines Konkurrenten einzuholen bzw. der eigene auszubauen. Hier unterliegt die Termingebung meist dem Marketing oder dem Produktmanagement.
  - Adaptive Maintenance – Softwarehersteller, welche sehr stark durch die Gesetzgebung beeinflusst werden, so beispielsweise Lohnbuchhaltungssysteme, welche sich faktisch durch neue Verordnungen quartalsweise ändern, wählen diesen Weg. Hier sind die Hersteller gezwungen, sich mit den Wirkungsterminen der jeweiligen Verordnungen zu synchronisieren, was zu festen kalendarischen Releaseterminen führt.

- Featurestrategie – Bei der Featurestrategie entsteht der nächste Release, wenn alle vorher angekündigten Features realisiert wurden. Vertreter für eine solche Strategie sind oft Open-Source-Produkte.

Eine explizite Planung unter Berücksichtigung der Service Level Agreements hilft, die Differenz zwischen der definierten und der gelieferten Dienstleistung gering zu halten.

Verfolgung der Aktivitäten

Neben der Planung und den Service Level Agreements muss die Ausführung der Dienstleistung überwacht und gemessen werden, da nur so eine Einhaltung oder Verletzung der Service Level Agreements nachgewiesen werden kann.

Auf Grund der Ergebnisse der Messungen können Service Level Agreements neu ausgehandelt werden. Außerdem können hier interne Analysen Optimierungspotentiale aufzeigen.

Management der Ereignisse

Unter Event Management wird die Behandlung von Ereignissen verstanden, welche zu einer Verletzung von Service Level Agreements führen oder Anlass zur Maintenance liefern. Zu diesen Ereignissen gehören:

- Change Requests der Kunden
- Vorkommnisse[27], welche die Operation des Systems beim Kunden behindern oder gefährden – die klassischen „Bugs“ gehören zu diesen Vorkommnissen

Jedes dieser Ereignisse während der gesamten Maintenance muss beachtet und bearbeitet werden. Zur Unterstützung des Eventmanagements dienen üblicherweise folgende Instrumente.

- Ein Bug- und Request-Tracking-System – Ein solches System[28] verfolgt alle Maintenanceereignisse und verzeichnet Zuständigkeiten sowie Historien dieser Ereignisse.
- Die Ereignisse müssen:
  - identifiziert
  - aufgezeichnet
  - begutachtet
  - verfolgt

  werden. Besonders die Verknüpfungen zwischen den Änderungen im Legacysystem und den jeweiligen Maintenanceereignissen sind wichtig. Idealerweise handelt es sich hierbei um eine bidirektionale Verknüpfung, so dass

[27] Vorkommnisse werden hier im Sinne des ITIL-Incidents verstanden.

[28] Großer Beliebtheit erfreut sich hier Bugzilla, ein Open-Source-Bugtracking System.

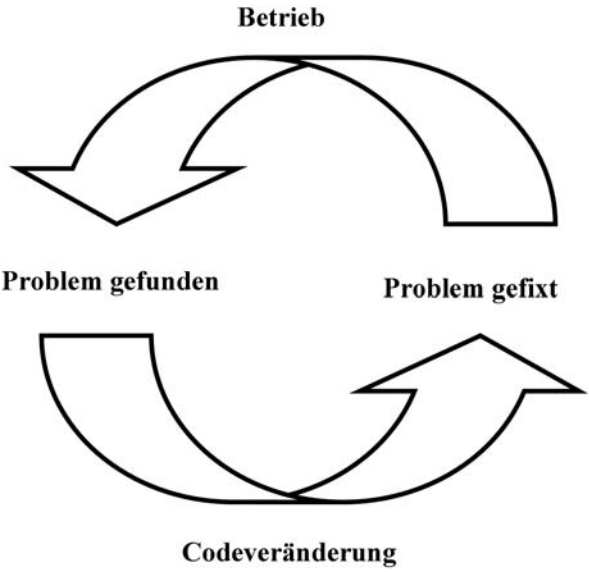

Abb. 7.5: Das Quick-Fix-Modell

die Abhängigkeit zwischen Systemveränderungen und Ereignissen verfolgt werden kann.

- Standardreports – Reports über das zeitliche Verhalten der Ereignisse ermöglichen es, interne Abläufe zu bewerten und damit auch in der Zukunft besser planen zu können.

Langfristig führen diese Dienstleistungsmechanismen zu einer Einordnung der Software Maintenance für jede Form der Software, nicht nur Legacysysteme, als Bestandteil von ITIL, der IT Infrastructure Library.

7.5 Maintenanceprozess

Aus der vorherigen Diskussion wird sehr schnell offensichtlich, dass die Maintenance nicht mit den gleichen Prozessen funktionieren kann wie die Neuentwicklung. Von daher sind traditionelle Prozesse, wie z.B. das Wasserfallmodell oder auch das V-Modell, für die Maintenancephase eines Softwareproduktes und damit für die Evolution einer Legacysoftware ungeeignet. Selbst neuere Vorgehensmodelle wie RUP, s. Abschn. 11.2, oder EUP, s. Abschn. 11.3, sind nur partiell geeignet. Zwar unterstützen beide die größeren Entwicklungsvorhaben im Bereich der adaptiven Maintenance, für die anderen Typen sind sie jedoch ungeeignet. Einzig die agilen Methoden, s. Abschn. 11.4, zeigen vielversprechende Ansätze.

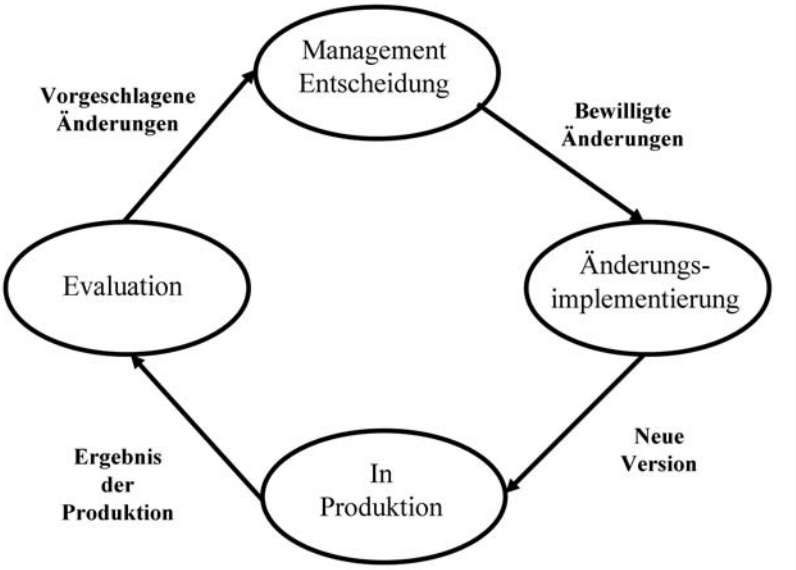

Abb. 7.6: Das Boehm-Modell

Für den Bereich der korrektiven Maintenance bzw. für kleinere Aufgaben aus den anderen Bereichen existieren z.Z. folgende Prozessmodelle, welche von der allgemeinen Diskussion in Abschn. 11.4 abweichen:

- Quick-Fix-Modell, s. Abb. 7.5 – Hierbei wird ein Defekt, sobald er gefunden wurde, sofort behoben. Als Prozessmodell ist dieses Modell für Einzelpersonen oder sehr kleine Teams gut geeignet. Es beinhaltet jedoch keinerlei Planbarkeit und ist rein reaktiv. Als Modell ist es sehr rigide, da es keinerlei Vorkehrung für Änderungen trifft. Der eine Zustand ist das Coding, der andere das Fixing. In diesen beiden Zuständen werden alle Phasen und Tätigkeiten des Lebenszyklus subsumiert. Typisch für dieses Modell ist es, dass Ripple-Effekte erst beobachtet werden, wenn die Legacysoftware wieder produktiv eingesetzt wird.
- Boehm-Modell, s. Abb. 7.6 – Innerhalb des Boehm-Modells ist der Maintenanceprozess ein geschlossener Kreislauf. Das steuernde Element für die Maintenance sind Managemententscheidungen. In diesem Modell ist es die Aufgabe des Managements, eine Balance zwischen den wirtschaftlichen Kriterien wie Budget und Aufwand auf der einen Seite und den Anforderungen des Produktes, bzw. des Produktmanagements, auf der anderen Seite zu erfüllen.
- Reuse-Modell, s. Abb. 7.7 – Im Rahmen des Reuse-Modells wird davon ausgegangen, dass vorhandene Teile, welche im Repository lagern, wiederverwendet werden können. Das Reuse-Modell durchläuft vier Schritte:

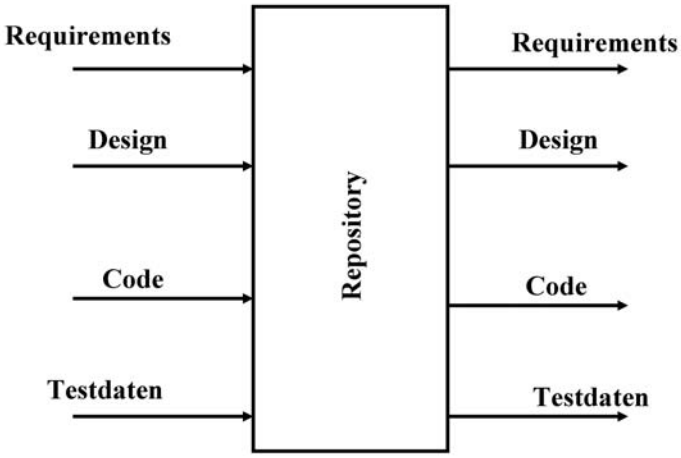

Abb. 7.7: Das Reuse-Modell

1 Identifikation der Teile des Legacysystems, welche wiederverwendet werden sollen
2 Dieses Teil zu verstehen
3 Veränderungen an diesem Teil vorzunehmen, damit die neuen Anforderungen erfüllt werden
4 Integration der veränderten Teile in das Legacysystem

- Osborne-Modell – Innerhalb des Osborne-Modells entstehen technische Probleme während der Maintenance durch schlechte Kommunikation und Missmanagement bzw. mangelnde Kontrolle durch die Führung. Damit diese Probleme in einem beliebigen Vorgehensmodell angegangen werden, enthält das Osborne-Modell vier Strategien:
  1 Permanente Performanzreviews für das Management werden durchgeführt.
  2 Definition bzw. Entwicklung von Metriken zur Messung der Zielerreichung während der Maintenancephase[29].
  3 Ein Qualitätssicherungsprogramm, damit obige Metriken gemessen bzw. das Benchmarking erreicht werden kann.
  4 Die Maintenanceanforderungen werden Teil der Change-Request-Spezifikation.

[29] Solche Metriken zur Messung der Zielerreichung sind Seiteneffekte des Dienstleistungsansatzes, welcher Servicelevels definiert, die selbstverständlich auch überwacht und gemessen werden müssen.

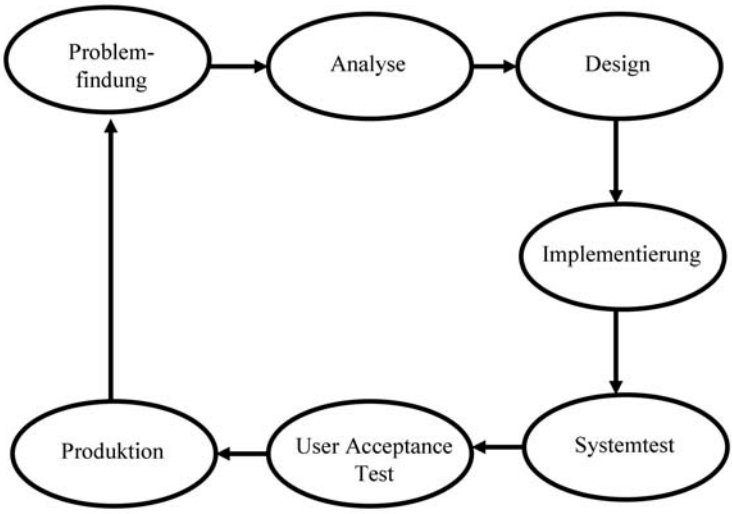

Abb. 7.8: Das IEEE-Modell

- IEEE-Modell, korrekterweise IEEE-1219-98, s. Abb. 7.8,– Dies ist ein zyklisches Modell, in dem sich die Phasen der „traditionellen“ Softwareentwicklung wiederholen. Allerdings ist, im Rahmen der Maintenance, die Dauer der einzelnen Phasen sehr kurz. Die einzelnen Phasen sind:
  1 Problemfindung – Im ersten Schritt des Zyklus wird das Problem oder der Änderungswunsch definiert und bewertet. An dieser Stelle findet eine Klassifikation wie auch eine Priorisierung statt, eventuell kann die Änderung auch verworfen werden.
  2 Analyse – Das gegebene Problem wird analysiert. Zur Analyse zählen die Bestimmung der betroffenen Systemteile, eine Impact-Analyse, aber auch eine Kostenschätzung.
  3 Design – Die Analyseergebnisse werden verwandt um die aktuelle Spezifikation so abzuändern, dass ein neues verändertes System entsteht.
  4 Implementierung – Im Rahmen dieser Phase wird die Veränderung des Sourcecodes durchgeführt.
  5 Systemtest – Übliche Regressionstests stellen in diesem Schritt sicher, dass die neuen Anforderungen erfüllt sind, aber auch dass das System in anderen Bereichen noch die „alte“ Funktionalität beibehalten hat.
  6 User Acceptance Test – Mit einem Abnahmetest bestätigt der Endbenutzer, dass die Änderungen in seinem Sinne durchgeführt wurden.
  7 Produktion – Die so veränderte Software wird in die Produktion übergeben.

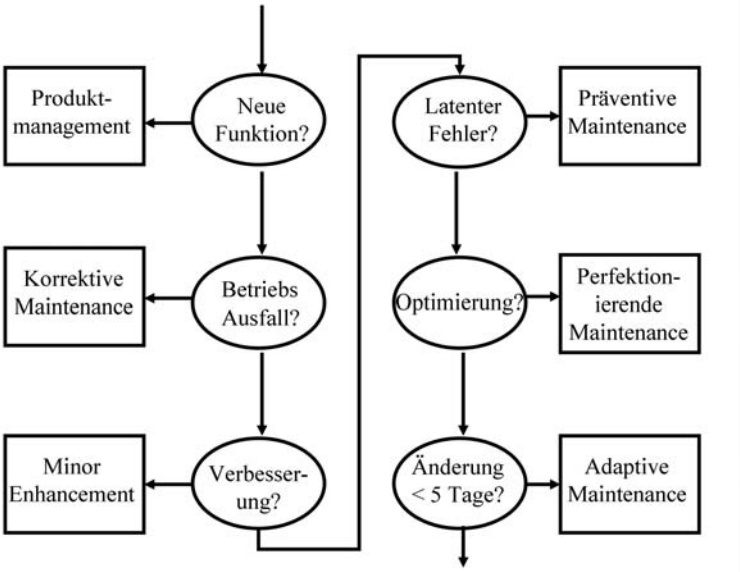

Abb. 7.9: Qualifizierung von Maintenanceaufgaben

Jedes dieser Prozessmodelle bzw. Strategien adressiert verschiedene Aspekte der Maintenance. Im Rahmen eines echten und vollständigen Vorgehensmodells für Maintenance müssen alle diese Aspekte enthalten sein, s. Abschn. 11.5.

Traditionell werden größere Blöcke, d.h. solche mit einem hohen Aufwand, nicht in die reguläre Maintenance genommen, sondern als ein Projekt gesehen; solche Veränderungen werden sehr oft als „Major Enhancement“ bezeichnet. Typische Beispiele für diese Kategorien sind die Euroumstellung und das Jahr-2000-Problem. In beiden Fällen wurden in allen Unternehmen große Projekte gestartet, um dieses adaptive Maintenanceproblem zu lösen.

Eine übliche Trennung zwischen „Major“ und „Minor Enhancement“ ist der Aufwand. Ab etwa 5 Manntagen widmen die meisten Unternehmen eine Maintenanceaufgabe zu einer Projektaufgabe um. Neben den reinen Aufwänden spielt auch die Klassifikation der Maintenanceaufgabe eine Rolle; ein praktikabler Entscheidungsbaum ist in Abb. 7.9 dargestellt.

7.6 Maintenanceprozessverbesserung

Der Prozess der Maintenance lässt sich auch verbessern, s. Abb. 7.4. Jenseits einer einfachen Beurteilung von Prozessen aus Sicht einer Metrik gibt es auch den Ansatz, den Maintenanceprozess aus Sicht des Reifegrades, speziell der CMM-Sicht, s. S. 157, zu betrachten.

Tab. 7.6: Maintenance CMM

CMM-Stufe	Management	Support	Dienstleistung
optimiert	Prozess-Changemanagement	Prozess-Changemanagement	Fehlerprävention
managed	quantitatives Prozessmanagement	quantitatives Prozessmanagement	Service Quality Management
definiert	integriertes Service Management	Prozess, Definition, Training	Dienstleistungsdurchführung
wiederholbar	Service Commitment, Planung, Verfolgung	Konfigurationsmanagement, Event Management	
initial	ad hoc	ad hoc	ad hoc

Für die Maintenance lässt sich dieselbe Stufung wie die des „klassischen“ CMM-Modells nutzen, s. Tab. 7.6. Aus Sicht der Maintenance werden die einzelnen Stufen in Tab. 7.6 definiert durch:

- initial: Der Prozess der Maintenance ist völlig willkürlich ohne jede Struktur oder Planung. Das Ergebnis der Maintenance hängt von den heroischen Anstrengungen einzelner Mitarbeiter ab.
- wiederholbar: Die Basisprozesse der Maintenance als Dienstleistung sind definiert, etabliert und ermöglichen es, Kosten, Performanz und Planung zu bestimmen.
- definiert: Der Maintenanceprozess ist standardisiert und jede konkrete Ausführung ist in einem Standardprozessgerüst integriert.
- managed: Detaillierte Maßzahlen der Maintenance sowie der Maintenancequalität werden erhoben, sind verstanden und kontrollierbar.
- optimiert: Eine kontinuierliche Prozessverbesserung geschieht durch eine Menge von quantitativen Messungen. Außerdem ist die Maintenanceorganisation in der Lage, neueste technische Entwicklungen[30] aufzunehmen.

Im Sinne eines CMM-Modells für Maintenance reift ein Unternehmen beginnend mit der Stufe initial zu den höheren Stufen. Dazu ist es aber notwendig, dass die Maintenanceprozesse auf der jeweiligen Ebene beherrscht werden. Genauer gesagt, es wird auf der Stufe n vorausgesetzt, dass alle vorhergehenden Stufen $n-1$ vollständig beherrscht werden. Hierzu ist auf den unterschiedlichen Stufen die Existenz verschiedenster Prozesse notwendig:

- Initial – Diese Stufe wird durch die bloße Existenz einer Maintenance schon als erreicht angesehen.

[30] Im Gegensatz zu den Vorstellungen des CMM-Prozessmodells tendieren Unternehmen, welche sehr stark prozessorientiert sind dazu technische Neuerungen erst sehr spät oder überhaupt nicht einzuführen.

- Wiederholbar – Prozesse auf dieser Ebene müssen sich darauf konzentrieren, das einmal Beherrschte wiederholbar zu machen. Als erster Schritt werden die Service Level Agreements zwischen Kunden und Maintenanceorganisation vereinbart. Auf dieser Stufe beinhaltet dies die Definition der Maintenance und die Festlegung ihrer Qualität. An beiden Aktivitäten hat der Kunde ein valides Interesse. Um sicherzustellen, dass dies für die Maintenanceorganisation überhaupt realistisch machbar ist, wird intern eine Maintenanceplanung im Sinne einer Ressourcenplanung durchgeführt. Diese muss die vereinbarten Service Level Agreements widerspiegeln.
  In dieselbe Kategorie fällt hier auch das Vertragswerk mit den Subkontraktoren. In der heutigen, stark arbeitsteiligen IT-Welt hat praktisch jedes Unternehmen Subkontraktoren. Bei kleineren Maintenanceunternehmen handelt es sich dabei in den meisten Fällen um Selbstständige, die so genannten „Freelancer“.[31] Gegenüber den Subkontraktoren ist das Maintenanceunternehmen der Kunde und handelt entsprechende Service Level Agreements aus. Dieses Regelwerk wird dann als Subcontract Management bezeichnet. Auf der jetzigen Stufe benötigt das Unternehmen für eine effektive Maintenance noch drei Supportprozesse:
  - Configuration Management – Da die Maintenance nur mit Unterstützung von Soft- und Hardware vonstatten gehen kann, muss auch die Maintenanceumgebung kontrolliert werden. Das Configuration Management stellt sicher, dass zu jedem Zeitpunkt der Zustand der einzelnen Komponenten bekannt und kontrollierbar ist. Daneben kümmert es sich auch um die Umsetzung von Veränderungen und Change Requests der jeweiligen Umgebungen.
  - Event Management – Alle Ereignisse, welche im Laufe eines Maintenanceprozesses auftreten, müssen aufgezeichnet, bewertet und verfolgt werden. Außerdem müssen sie dem Kunden gegenüber berichtet werden. Das größte Aufkommen an Events bilden in der Regel die Defekte sowie – bei Infrastrukturprovidern – die Incidents.
  - Service Quality Assurance – Via Reviews und Audits wird im Rahmen eines Qualitätssicherungsprozesses die Güte des Maintenanceprozesses wie auch der zugesagten Service Level Agreements permanent überprüft.
- Definiert – Auf der dritten Stufe hat das Unternehmen seine Prozesse standardisiert und benutzt die maßgeschneiderten Ableitungen dieser Standardprozesse, um den konkreten Maintenanceprozess durchzuführen. Das Hauptziel dieser Stufe ist es zu realistischeren Einschätzungen der vorhergesagten Performanz der Prozesse zu kommen und damit die Fähigkeit

[31] Der Ausdruck Freelancer geht auf das Italien der Renaissance zurück. Da es weder National- noch größere Territorialstaaten gab, existierten permanente Armeen in unserem heutigen Sinne nicht. Als Freelancer wurden unabhängige, kleine Söldnergruppen bezeichnet, welche ihre Dienste den zahlungskräftigsten Auftraggebern zur Verfügung stellten. Bekanntester Freelancer in Italien war der Engländer Hawkwood.

Tab. 7.7: „Maintenance Impact Ratio“-Faktoren als Funktion der Daten und Sourceänderung

	% Daten	**% Daten**	**% Daten**	**% Daten**	**% Daten**
Code	$\leq 25\%$	$\leq 50\%$	$\leq 75\%$	$\leq 100\%$	$> 100\%$
$\leq 25\%$	0,125	0,25	0,5	0,75	1,2
$\leq 50\%$	0,25	0,5	0,75	1,2	1,2
$\leq 75\%$	0,5	0,75	1,2	1,2	1,2
$\leq 100\%$	0,75	1,2	1,2	1,2	1,2
$> 100\%$	1,2	1,2	1,2	1,2	1,2

zu einer Vorhersage zu erhöhen. Die Schlüsselprozesse fallen in die Kategorien: Management, Support und Dienstleistung. Die Prozesse in der Kategorie Management beschäftigen sich hauptsächlich mit der Anpassung des Standardmaintenanceprozesses sowie der Verbesserung der Service Level Agreements. Üblicherweise werden demselben Kunden nicht nur einige Maintenanceprozesse, sondern auch mehrere andere Dienstleistungsprozesse angeboten. Im Rahmen dieser Stufe wird die Integration dieser diversen Prozesse in ein Gesamtbild vorgenommen.

- Managed – Auf dieser Stufe werden detaillierte quantitative Messungen der Standardprozesse bzw. ihrer adaptierten Versionen vorgenommen. Neben den Prozessen wird auch versucht, die Qualität der Prozesse zu messen und die so entstehenden Daten zu nutzen, um die Qualität des Maintenanceprozesses zu kontrollieren.
- Optimiert – Dienstleister, welche diese Stufe erreicht haben, können ihre Prozesse aktiv verändern, um bessere Prozesse und mehr Qualität in der Maintenance zu produzieren. Die Prozessveränderungen resultieren aus Änderungswünschen, sowie aus dem technologischen Fortschritt.

7.7 Maintenance-Funktionspunkte

Die Metrik der Funktionspunkte, s. Abschn. 2.3 sowie Gl. 2.25 – Gl. 2.27 und Tab. 2.5 – Tab. 2.6, lässt sich um die Frage der Maintenance erweitern. Durch diese Erweiterung ist nun neben der Frage des Implementierungsaufwandes als Ergebnis der Funktionspunktanalyse auch die Maintenance zu bewerten.

Hierbei wird der „Unadjusted Function Point“ V^i_{UFP}, s. Gl. 2.25, um einen „Maintenance Impact Ratio“ m_{MIR} ergänzt. Zwei Fälle sind recht einfach:

- Die einfache Erweiterung der Legacysoftware durch eine Funktion – In diesem Falle gilt für die „Maintenance Impact Ratio“:

$$m_{MIR} = 1, \tag{7.1}$$

da ja die Addition nichts an der Funktionspunktmetrik ändert.

- Die Löschung einer Funktion – Hierfür lässt sich als „Maintenance Impact Ratio“ ansetzen:

$$m_{MIR} = 0.2. \tag{7.2}$$

Für den Fall der Änderung einer bestehenden Funktionalität ergibt sich, je nach der Größe der Änderung, eine unterschiedliche „Maintenance Impact Ratio“, s. Tab. 7.7.

Das Vorgehen zur Berechnung der Funktionspunktmetrik ist nun gegeben durch die adjustierten Maintenancefunktionspunkte V_{AMFP} zu:

$$V_{AMFP} = V_{UMFP} V_{VAF} \tag{7.3}$$

$$= V_{UFP} V_{VAF} V_{MIR} \tag{7.4}$$

Generell muss berücksichtigt werden, dass dieser Ansatz eine Form der linearen Unabhängigkeit der einzelnen Funktionen untereinander annimmt, d.h. der „Ripple-Effekt“ wird als nicht vorhanden angenommen. Dies entspricht auf keinen Fall den Erfahrungen, welche bei der Maintenance von Legacysystemen gemacht werden. Da die Funktionspunktanalyse als Metrik stets linear ist, lässt sich dieser hohe Grad an Nichtlinearität nicht durch eine Justierung der Koeffizienten korrigieren.

Selbst wenn, trotz aller Kritik, eine Maintenance Funktionspunktmetrik als gegeben angenommen wird, so lässt sich der Aufwand E nur schwer als direkt aus der Größe der Änderung des Sourcecodes abgeleitete Größe darstellen. Insbesondere scheint zu gelten:

$$E \nsim \Delta n_{Code}$$
$$E \sim n_{Code}\,(1 + \epsilon \Delta n_{Code})$$

Der Aufwand ist folglich eine Funktion der Größe des Moduls wie auch der Größe der Änderung.

7.8 Impact-Analyse

Die Impact-Analyse beschäftigt sich mit dem Phänomen, welche Seiteneffekte eine geplante Änderung in einem Legacysystem haben kann, siehe auch Taxonomie der Softwareevolution, Abschn. 4.14. Das Hauptproblem bei der Maintenance ist der Ripple-Effekt; darunter wird die unangenehme Situation verstanden, dass eine Änderung in einem System andere Änderungen nach sich zieht. Diese anderen Änderungen werden ungeplant ausgelöst, d.h. zu Beginn waren diese Folgeänderungen noch nicht bekannt, sondern haben sich erst während der primären Änderung als notwendig herausgestellt.

Neben der offensichtlichen Problematik der schlechten Planbarkeit für eine Veränderung der Software, da ja der Ripple-Effekt nicht antizipiert wurde, ist er trotzdem immer ein Indiz dafür, dass das Legacysystem nicht vollständig bekannt und verstandenen worden ist. Solche Ripple-Effekte zeigen auch an,

dass die Legacysoftware eine hohe Volatilität bzw. einen niedrigen Maintainability Index hat.

Historisch gesehen hat die Zunahme an Komplexität in Legacysystemen in den meisten Unternehmen dazu geführt, dass Verfahren zur Schaffung von Traceability, d.h. der Verknüpfung von Anforderungen zu Softwareartefakten, in der Regel Modellen, wie beispielsweise das Datenmodell und Sourcecode, geschaffen wurden. Diese Mechanismen der Traceability sind der Startpunkt für eine Impact-Analyse.

Für Legacysysteme sind drei Arten von Impact-Analysen interessant:

- Anforderungsgetriebene Impact-Analyse – Bei der anforderungsgetriebenen Impact-Analyse stellt sich die Frage, welche Aufwände dabei entstehen, eine neue Anforderung zu implementieren, ohne den Sourcecode zu verändern. Oft hat die Änderung eines Features zur Folge, dass andere, abhängige, Features auch mit verändert werden müssen. Die Fragestellung ist, auch aus einem ökonomischen Gesichtspunkt heraus, interessant, da viele Kunden bereit sind, für neue Features jenseits der vertraglich zugesicherten Maintenance zu bezahlen. Für solche zusätzlichen Features muss eine Kosten-Nutzen-Abwägung oder auch eine Deckungsbeitragsbetrachtung gemacht werden. Außerdem haben die Ripple-Effekte auch Auswirkungen auf ein Produktlinienkonzept, s. Kap. 9. Anforderungsgetriebene Impact-Analyse versucht, alle primären und sekundären Veränderungen zu finden unter der Maßgabe, dass der Sourcecode stabil bleibt.
- Änderungsgetriebene Impact-Analyse – Diese wird bestimmt durch die Fragestellung: Wenn das Softwareartefakt $\mathcal{A}$ abgeändert wird, welche anderen Artefakte müssen zusätzlich verändert werden? Dies kann sowohl horizontal als auch vertikal geschehen.[32] Ziel ist es, alle möglichen Kandidaten für sekundäre Änderungen zu finden und diese dann einzeln zu begutachten. Diese Änderungen können durchaus auch tertiäre und quartäre Änderungen auslösen. In einem solchen Falle spricht man von systemweiter Änderung[33], s. Abschn. 4.14.
- Testgetriebene Impact-Analyse – Diese wird durch die Fragestellung der Testökonomie hervorgerufen. Wenn es keine Ripple-Effekte gibt, müssen sehr viel weniger Module im Rahmen eines Regressionstests überprüft werden.

Bedingt durch diese Treiber besteht die Notwendigkeit, Impact-Analysen betreiben zu können. Die Impact-Analyse muss die Auswirkungen einer Änderung auf Grund einer Abhängigkeitsanalyse vorhersagen können. Für solche Abhängigkeitsanalysen gibt es unterschiedliche Modelle:

[32] Bei einem MDA-Ansatz nur horizontal.

[33] Die Vorstellung gleicht dann mehr einem Gebilde aus Dominosteinen, das durch das Umfallen eines einzelnen Steines zusammenbricht.

- Statistische Modelle – Hierbei wird die Legacysoftware in eine Reihe von Objekten[34] zerlegt, welche eine bestimmte Wahrscheinlichkeit der Verknüpfung haben:

$$p_j = \sum_i \Lambda_{ij}\eta_i. \tag{7.5}$$

Die Größe p_j gibt die Wahrscheinlichkeit an, mit der sich das j-te Objekt verändert, wenn sich Objekt i verändert hat. Die Variable $\boldsymbol{\eta}$ ist der Einheitsvektor, welcher für i den Wert 1 annimmt und sonst 0 ist,

$$\eta_i = \delta_{ij}.$$

Die Verknüpfungsmatrix Λ_{ij} modelliert die Zusammenhänge im System; diese Verknüpfungsmatrix wird in der Regel heuristisch bestimmt. Solche Modelle geben nur die Wahrscheinlichkeit an, mit der sich weitere Objekte verändern. Die Gesamtauswirkung der Veränderung lässt sich durch eine Mehrfachanwendung von Λ aufzeigen, so ist die n-te Änderung gegeben durch:

$$\overrightarrow{p}^{(n)} = \widehat{\Lambda}^n\boldsymbol{\eta}, \tag{7.6}$$

wobei sich die Verknüpfungsmatrix $\widehat{\Lambda}$ am einfachsten durch die Zerlegung:

$$\widehat{\Lambda} = \widehat{1} + \widehat{\Delta},$$

beschreiben lässt. Diese Darstellung hat den Vorteil, dass die primäre Änderung sich als $\widehat{\Delta}$ ergibt. Im günstigsten Falle ist $\widehat{\Delta}$ nilpotent, d.h.

$$\widehat{\Delta}^m = 0$$

für ein $m > 0$. Allerdings reicht es in den meisten Fällen aus, wenn Δ eine Blockstruktur hat. Außerdem muss die Verknüpfungsmatrix nicht unbedingt symmetrisch sein; in der Regel sollte nämlich gelten:

$$\Delta_{ij} \neq \Delta_{ji}.$$

Inhaltlich sagt dies aus: Die Wahrscheinlichkeit, dass sich Objekt i auf Grund einer Änderung von Objekt j ändert, ist nicht identisch mit der Wahrscheinlichkeit, dass sich Objekt j ändert auf Grund einer Änderung von Objekt i.

- Mechanistische Modelle – Solche Modelle implizieren starre Verbindungen zwischen den einzelnen Artefaktobjekten. Die Vorstellung ist die eines starren Gitters, welches die einzelnen Artefakte miteinander verknüpft. Wird nun ein Artefakt verändert, so lässt sich beobachten, wie die so entstandene Störung durch das Gitter läuft. Hierbei müssen jedoch Propagationsregeln im Gitter definiert werden. Wird dies nicht getan, so sind

[34] „Objekt“ wird hier im generischen Sinn als Teil eines Softwareartefaktes gebraucht.

die mechanistischen Modelle ein Spezialfall der statistischen Modelle, da dann die Verknüpfungsmatrix Δ nur aus 0'en und 1'en besteht. Die Propagationsregeln führen dazu, dass sich die mechanistischen Modelle komplizierter als die statischen Modelle verhalten. Die k-te Änderung ergibt sich in solchen Modellen aus:

$$\overrightarrow{p}^{(k)} = = \mathcal{F}\left(\overrightarrow{p}^{(1)}, \overrightarrow{p}^{(2)}, \ldots, \overrightarrow{p}^{(k-1)}\right) \tag{7.7}$$

$$\approx \mathcal{F}'\left(\overrightarrow{p}^{(k-1)}\right) \tag{7.8}$$

$$\approx \Omega_{k-1}^{k}\, \overrightarrow{p}^{(k-1)}. \tag{7.9}$$

Im allgemeinen Fall ist die Propagation der Änderung ein beliebig komplexes Funktional $\mathcal{F}$ aller vorherigen Änderungen, s. Gl. 7.7. Bei den meisten mechanistischen Modellen beschränkt man sich jedoch auf eine direkte Abhängigkeit zwischen der $(k-1)$-ten und der k-ten Änderung, s. Gl. 7.8. In einfachen Fällen lässt sich dieses Funktional $\mathcal{F}'$ durch eine lineare Abhängigkeit modellieren, s. Gl. 7.9. Die statistischen Modelle, s. Gl. 7.6, ergeben sich dann als Spezialfall der Näherung Gl. 7.9 durch:

$$\Omega_{k-1}^{k} = \Lambda \qquad \forall k > 1.$$

- Slicing-Modelle – Ein etwas anderer Ansatz wird bei der Technik des Slicings gewählt. Ein Artefakt wird dabei in zwei logische Teile geteilt, der eine enthält die primären Änderungen, der andere, der so genannte Komplementärslice, bleibt völlig unberührt. Durch das Slicing wird versucht, die Propagation des Ripple-Effekts bewusst einzudämmen, da der Komplementärslice sich per definitionem nicht verändert. Das Slicing wurde zur Beschreibung von Ripple-Effekten in der Dokumentation, welche ja auch ein Artefakt ist, eingesetzt. Hier scheint es gut zu funktionieren, da sich die Dokumentation als ein logischer Baum darstellen lässt. Änderungen in diesem Baum sind dann azyklische Graphen, d.h. Teilbäume. Der Komplementärslice ist eine Menge von Teilbäumen, welche unberührt bleiben. Mit dieser Technik lässt sich für hierarchische Artefakte die Propagation recht gut beschreiben.

Die meisten praktischen Ansätze zur Impact-Analyse bewegen sich, mit Ausnahme der MDA, auf der Ebene der gleichen oder direkt verwandten Artefakte. Sehr viel schwieriger ist es, die Co-Evolution zu beschreiben, s. Abb. 7.10. Die Co-Evolution besteht darin, dass sich ein „höherer" Evolutionsraum, s. Abb. 4.1, verändert, mit der Folge, dass sich der nächst tiefere Evolutionsraum implizit mit verändert. Wenn der Evolutionsraum Ξ_n sich verändert, so folgt diesem auch immer eine Veränderung des niedrigeren Raumes Ξ_{n-1}, d.h. es gilt:

$$\begin{aligned} \Xi_n &\rightarrow \Xi_n^{\dagger} && \textit{Evolution} \\ \Xi_{n-1} &\Rightarrow \Xi_{n-1}^{\dagger} && \textit{Co-Evolution}. \end{aligned}$$

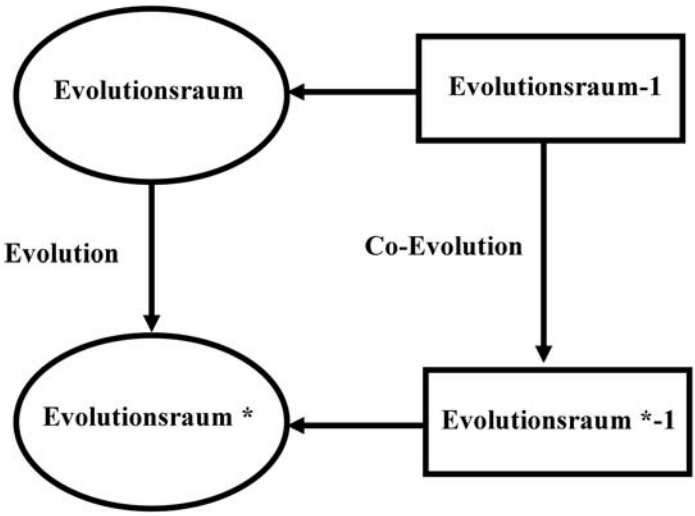

Abb. 7.10: Die Co-Evolution von Evolutionsräumen

Dies hat zu Folge, dass regelrechte Co-Evolutionskaskaden, s. Abschn. 4.15 entstehen, mit der Form:

$$
\begin{array}{rl}
\Xi_n \rightarrow \Xi_n^{\dagger} & \textit{Evolution} \\
\Xi_{n-1} \Rightarrow \Xi_{n-1}^{\dagger} & \textit{Co} - \textit{Evolution}. \\
\Xi_{n-2} \Rightarrow \Xi_{n-2}^{\dagger} & \textit{Co} - \textit{Co} - \textit{Evolution}. \\
\Xi_{n-k} \Rightarrow \Xi_{n-k}^{\dagger} & \ldots
\end{array}
$$

Hierbei muss verfolgt werden, welche Änderungen der Anforderungen Veränderungen bei den Implementierungsartefakten auslösen. Bei generativen Entwicklungssystemen ist das Propagationsmodell für Co-Evolution meist automatisch verfolgbar. Die Betrachtung der Co-Evolution ist immens wichtig, wenn man sich vor Augen führt, dass sich in großen Legacysystemen mehr als 70% der Anforderungen ändern.

Die Co-Evolution hat zusammen mit dem Ripple-Effekt immense Auswirkungen auf den Alterungsprozess eines Legacysystems. Am einfachsten lässt sich dies an der Entwicklung der Entropie der Co-Evolution betrachten. Eine Komponente η_j^n im Evolutionsraum Ξ_n ist mit einer Anzahl von Komponenten η_i^{n-1} im Evolutionsraum Ξ_{n-1} verknüpft; analog zu Gl. 7.5 lässt sich über die Evolutionsräume hinweg definieren:

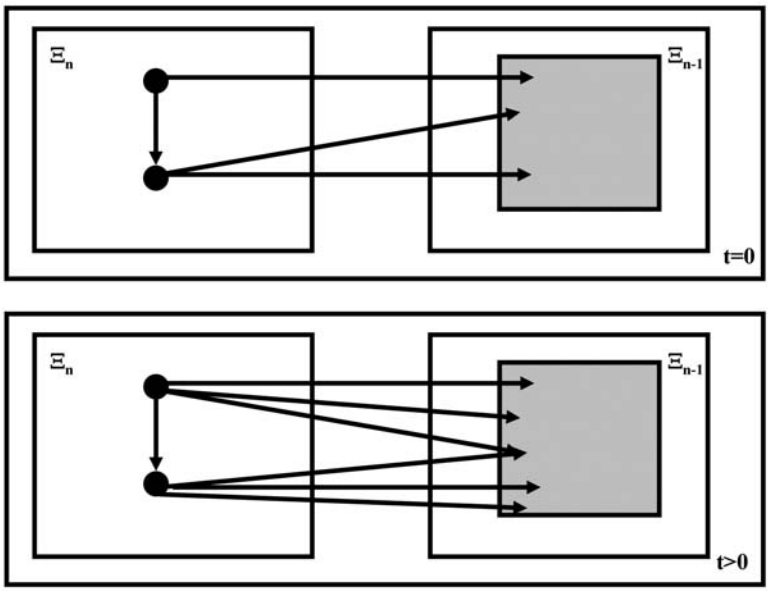

Abb. 7.11: Die Entwicklung der Co-Evolution

$$p_j^{n\mapsto n-1} = \frac{\sum\limits_i \Lambda_{ij}^{n\mapsto n-1}}{\sum\limits_{ji} \Lambda_{ij}^{n\mapsto n-1}}. \tag{7.10}$$

Die in Gl. 7.10 definierte Wahrscheinlichkeit verknüpft die Änderungen der Komponente j einer höheren Abstraktionsebene mit dem Impact auf der niedrigeren Abstraktionsebene. Daraus lässt sich die Co-Evolutionsentropie definieren zu:

$$S = \sum_n S^{n\mapsto n-1} \tag{7.11}$$

$$= -\sum_{jn} p_j^{n\mapsto n-1} \ln p_j^{n\mapsto n-1}. \tag{7.12}$$

Mit zunehmendem Alter steigt die Zahl der Verknüpfungen an, mit der Folge:

$$S(t) > S(0)$$

Diese hier dargestellte Entropieentwicklung lässt sich im Sinne einer Co-Evolution anhand von Beispielen recht gut in die Praxis umsetzen.

Wenn der Evolutionsraum Ξ_n das Analysemodell repräsentiert und parallel dazu das Designmodell Ξ_{n-1} als eine tiefere Stufe betrachtet wird, so bildet p die Verknüpfungen zwischen dem Analyse- und dem Designmodell

ab. Je adäquater nun die Designlösung, d.h. je mehr das Designmodell der Analyse entspricht, desto geringer ist die Entropie. Je ausgefeilter das Design ist, damit ein spezifisches Problem gelöst werden kann, desto stärker unterscheidet sich das Designmodell vom Analysemodell und desto größer ist der Impact bei Änderungen mit der Folge, dass die Entropie drastisch zunimmt.

An dieser Stelle macht sich das Prinzip des Information Hiding bemerkbar: Angenommen eine fachliche Funktionalität ändert sich auf Grund der Veränderung einer Komponente des Evolutionsraumes Ξ_n, dann spielen, wenn das Prinzip des Information Hiding explizit benutzt wird, nur noch die verhaltensändernden Verknüpfungen p_j eine Rolle, da alle anderen wegen des Information Hiding verschwinden. Dies hat zur Folge, dass die Entropie sich nicht erhöht, ja sogar absinken kann. Diese theoretische Betrachtung ist konsistent mit der Beobachtung, dass Information Hiding die Maintenance deutlich einfacher macht.

Einige Entwicklungsumgebungen nähern sich der Impact-Analyse auf konstruktiver Art und Weise, so beispielsweise die Make-Utility oder auch das modernere Ant. In diesen Werkzeugen werden die Abhängigkeiten innerhalb der Entwicklungsumgebung explizit hinterlegt. Während des Bauprozesses werden die einzelnen Artefakte mit Zeitstempel versehen; dies ermöglicht es, unterschiedliche aktuelle Stände automatisch zu vergleichen. In solchen Werkzeugen muss als Regel das abhängige Objekt immer jünger sein als das ursprüngliche Objekt. Werkzeuge wie Make oder Ant benutzen ein Regelwerk, um diese Abhängigkeiten implizit ausführen zu können.[35]

Die Beobachtung des Ripple-Effekts in der Vergangenheit kann auch für zukünftige Prognosen genutzt werden. Wenn alle vergangenen Änderungen einer Legacysoftware über eine lange Zeit beobachtet werden, so zeigt sich, dass es zwischen zwei Sourcedateien[36] i und j eine Korrelation gibt:

$$\xi_{ij}(t_1, t_2) = \begin{cases} 1 \; i = j \\ \delta \; i \neq j \\ 0 \; i \neq j \end{cases}$$

Wenn nun für einen gegebenen Zeitraum, beispielsweise weniger als eine Woche, $\Delta t < 1\,Woche$, für eine Änderung die Korrelation $\delta \neq 0$ nicht verschwindet, so kann angenommen werden, dass die beiden Sourcedateien stets miteinander zu verändern[37] sind. Gilt umgekehrt

$$\xi_{ij}(t_1, t_2) = 0 \qquad \forall \Delta t < x,$$

[35] Da Softwareentwickler oft nicht in der Lage sind, alle Abhängigkeiten zu modellieren, enthalten diese Werkzeuge oft eine „make clean“-Variante, welche alle Objekte zurücksetzt.

[36] Das Verfahren lässt sich auf beliebige versionierbare Teile ausdehnen, allerdings liegen meist nur Informationen über die Änderungszeiten von Dateien vor.

[37] Aus der Existenz einer Korrelation zwischen zwei Dateien wurde eine Kausalität zwischen den Dateien postuliert.

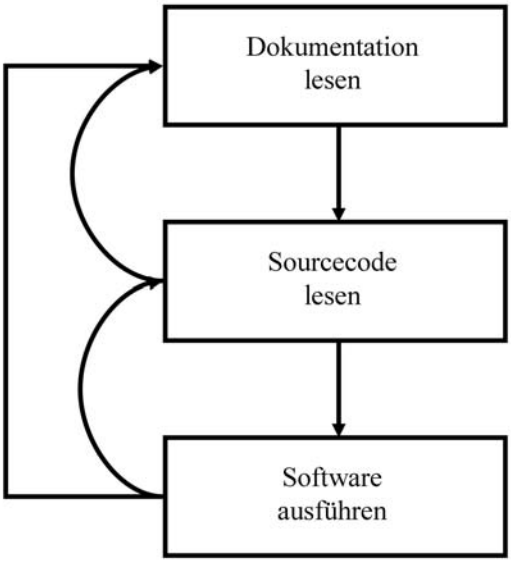

Abb. 7.12: Die Schritte zum Programmverständnis

so haben beide Dateien keine relevante Beziehung zueinander. Auf diese Weise lassen sich Verknüpfungen finden, ohne den Sourcecode semantisch verstehen zu können. Die Information über diese Korrelationen können genutzt werden, um zukünftige Aufwände vorauszusagen.

7.9 Sourcecode

Das Schlüsselartefakt in Legacysystemen ist der Sourcecode. Die meisten Legacysysteme sind ohne die anderen Artefaktformen entstanden, insofern spielt er eine zentrale Rolle. Das Hauptproblem des Sourcecodes an sich ist die Frage seiner Verständlichkeit: Wie kann und wie wird er von einem Softwareentwickler verstanden werden?

Der Sourcecode ist Teil des allgemeinen Verständnisprozesses, s. Abb. 7.12. Nach dem Lesen der Dokumentation über die Software wird der Sourcecode gelesen und anschließend das Programm getestet. Dieser idealtypische Ablauf wird in der Praxis oft massiv durchbrochen.

Werden alle anderen Variablen, wie Architektur und Funktionalität, konstant gehalten, so reduziert sich ein Teil der Problemstellung auf die Frage des typographischen Stils. Unter Berücksichtigung der Tatsache, dass bis zu 90% des Aufwandes für das Verstehen der Legacysoftware investiert werden müssen, s. S. 114, ist es verständlich, dass gerade der Sourcecode in den Mittelpunkt des Interesses rückt.

Doch wie wird überhaupt Sourcecode verstanden? Dazu gibt es zwei Ansätze, zum einen top-down und zum anderen bottom-up. Beim Top-down-Ansatz werden Hypothesen über den Sourcecode explizit oder implizit formuliert, mit dem Versuch der Verifizierung oder der Falsifizierung.[38] Beim Bottom-up-Verfahren wird ein so genanntes „Chunking“ eingesetzt. Diese Chunks sind Teile des Sourcecodes mit einer identifizierbaren Charakteristik, so beispielsweise Algorithmen, Funktionen, Datenstrukturen oder auch Schleifen. Diese Chunks werden durch Aggregation gebildet. Anschließend werden die Chunks verallgemeinert, was zu einer Abstraktion führt, und die so entstehende neue Menge von Chunks wird ihrerseits wieder aggregiert und abstrahiert. Analoge Untersuchungen über das Chunking gibt es im Bereich der Ergonomie. Da ein Mensch in seiner Wahrnehmung limitiert ist – Untersuchungen aus dem Bereich der Ergonomie zeigen, dass ein Mensch nur etwa 7 Chunks gleichzeitig wahrnehmen kann – strukturiert er den Sourcecode in seiner Wahrnehmung entsprechend.[39] Beim Bottom-Up startet der Softwareentwickler mit dem Sourcecode und entwirft ein erstes mentales Modell über den Kontrollfluss des jeweiligen Moduls. Dies liefert die erste Hypothese, welche in der Zukunft stärker modifiziert wird. Die üblichen Aktivitäten für das Verstehen von Sourcecode sind, in zeitlicher Reihenfolge:

1 Lesen der Kommentare und des Sourcecodes
2 Betrachtung des nächsten Elementes des Kontrollflusses
3 Betrachtung der Datenstrukturen
4 Verfolgung einer einzelnen Variablen
5 Chunking der Information für das Langzeitgedächtnis

Während dieser Analyse wird einzig ein Programmmodell entwickelt, welches die Implementierung modelliert, aber nicht, warum der Code in der gegebenen Form erscheint. Der Softwareentwickler entdeckt die Basiskonstrukte, wie beispielsweise einen Gruppenwechsel, aber er weiß nicht, wozu dieser in einer fachlichen Systematik gebraucht wird. Dies hat zur Folge, dass ein neuer Softwareentwickler nur sehr schwer die Auswirkungen von Veränderungen oder die notwendigen Stellen für Veränderungen vorhersagen kann.[40] Je mehr Erfahrung der einzelne Softwareentwickler mit der Legacysoftware hat, desto weniger ist er mit dem Problem des Aufbaus eines Programmmodells konfrontiert.

Im Fall eines Top-down-Vorgehens wird versucht, das Wissen über das Design und die Domäne mit dem vorliegenden Sourcecode in Übereinstimmung zu bringen. Dabei wird die Tatsache genutzt, dass Menschen aktive Problemlöser sind, welche versuchen, neue Informationen in das Gebilde ih-

[38] Das Letztere scheint am schwersten zu fallen.
[39] Die Einführung einer Patternsprache ist der Versuch, die Zahl der Chunks niedrig zu halten, da jetzt in Patterns gedacht werden kann.
[40] Dies ist einer der Gründe für die sehr flache Lernkurve beim Outsourcing, s. Kap. 8.

res bisher erworbenen Wissens einzugliedern.[41] Dieser Problemlösungsprozess besteht aus folgenden Aktivitäten:

1 Überblick über die gesamte Applikation
2 Auswahl des nächsten zu untersuchenden Programmsegmentes
3 Bestimmung der Relevanz des gewählten Codesegmentes für das momentane mentale Programmmodell
4 Überprüfung der Hypothese über den Sourcecode und eventuelle Neuformulierung einer Hypothese

Damit diese Tätigkeit durchgeführt werden kann, ist es sehr hilfreich, Programmpläne, Designmodelle und Domänenwissen zu besitzen. Untersuchungen über Problemlösungen zeigen empirisch, dass fast alle „Experten“ Top-Down-Methodiken einsetzen, im Gegensatz zu Anfängern, welche häufig Bottom-Up starten. Eine weitergehende Beobachtung zeigt, dass „Experten“ nur einen kleinen Teil des Codes betrachten und dann sehr opportunistisch vorgehen. Die „Experten“ scheinen den Code nicht systematisch zu zerlegen. Vermutlich sind sie auf Grund ihrer Erfahrung in der Lage, die Fragmente in ihr vorab erworbenes Wissensskelett einzubetten.

Unter dem Begriff typographischer Stil werden Elemente verstanden wie:

- Anzahl der Kommentare
- Benamung der Variablen
- Größe der einzelnen Sourcedateien
- Einrückung
- Klammersetzung
- Verwendung von Schleifen
- Verwendung von Fallunterscheidungen

Inwieweit diese Dimensionen des typographischen Stils die Verständlichkeit des Sourcecodes beeinflussen, ist noch nicht vollständig erforscht. Einige praktische Erfahrungen, speziell aus dem Vergleich von unterschiedlichen Programmiersprachen und dem aus diesen Sprachen evolvierenden „Best-Practice-Stil“, lassen sich jedoch aufzeigen:

1 Die Verständlichkeit wird durch die Nutzung langer und expliziter Variablennamen.[42] sehr verbessert[43]
2 In nichtobjektorientierten Sprachen hat es sich als vorteilhaft herausgestellt, die Variablen noch zusätzlich mit dem Variablentyp als Namen zu

[41] Gerade diese Tatsache macht einen Paradigmenwechsel so schwer, da das neue Paradigma nicht auf Grund bisheriger Erfahrung abgeleitet werden kann.

[42] Besonders beliebt scheint die Verwendung von weiblichen Namen oder auch Personen des öffentliche Interesses als Grundlage für die Benamung von Variablen zu sein.

[43] Ein Kollege des Autors pflegte seine Variablen in Fortran sequentiell mit X1, X2, X3, ... zu benennen.

versehen[44], bei objektorientierten Sprachen ist dies per se nicht notwendig, aber es scheint auch hier die Verständlichkeit zu erhöhen. Studien aus dem Bereich des „Forward Engineering“, bei denen Softwareentwickler aufgefordert wurden, Objektbezeichnungen zu vergeben, hatten als Resultat, dass die Wahrscheinlichkeit, dass zwei Softwareentwickler ein Objekt gleich benennen, zwischen 7 und 18% liegt, abhängig von der Art des Objekts.

3 Kleine Sourcecodedateien sind meistens einfacher zu lesen. Am günstigsten ist es, wenn eine Funktion auf einer Bildschirmseite darstellbar ist. Der Grund liegt in dem recht limitierten Kurzzeitgedächtnis des Menschen; jedes „Blättern“ erhöht die Wahrscheinlichkeit des Vergessens.

4 Kommentarblöcke, welche Funktionen abstrakt beschreiben, scheinen günstiger als verstreute Kommentare zu sein. Als sehr problematisch erweist sich die Angewohnheit einzelner Programmierer, Sourcecodezeilen auszukommentieren bzw. Kommentare bezüglich Löschungen im Sourcecode zu hinterlassen. Beides ist für das Verstehen des Sourcecodes hinderlich, da hier nur der aktuelle Status Quo entscheidend ist.[45]

7.10 Vorhersagbarkeit

Gleichgültig für welches Legacysystem, aus planerischer Sicht wäre es gut, den Aufwand an Maintenance vorhersagen zu können. So gut es wäre, die entsprechenden Fähigkeiten zu haben, so schwer ist es, exakte Zahlen für die Planung zu erhalten. Ist die Legacysoftware alt und groß genug, so lässt sich anhand der Vergangenheit der Maintenanceaufwand prognostizieren. Je nach Position im Lebenszyklus der Legacysoftware, ist es möglich, eine Art „best fit“ an die Entropiekurve, s. Kap. 4 und Gl. 4.7, zu machen, mit dem Ergebnis, dass der Aufwand E eine Funktion der Entropie S und des Maintainability Index $\mathcal{M}$, s. Gl. 2.40, ist:

$$E = \mathcal{F}\left(\mathcal{M}, S\right). \tag{7.13}$$

Dieses Funktional $\mathcal{F}$ lässt sich durch eine lineare Abhängigkeit annähern zu:

$$E \approx \alpha M + \beta S + \delta. \tag{7.14}$$

Eine Betrachtung des Aufwandes der Vergangenheit, als Funktion der Zeit, überführt die Gleichung 7.14 in eine zeitliche Funktion der Form:

$$E \approx \alpha_1 + \alpha_2 t + \alpha_3 \mathrm{e}^{\alpha_4 t}. \tag{7.15}$$

[44] Die C-Programmierer nennen dies „Hungarian Notation“.

[45] Dies ist bei Designdokumentation genau umgekehrt, hier erweist sich die Aufzeichnung der Designentscheidungen als sehr sinnvoll.

Tab. 7.8: Unterschiedliche Wahrnehmungen von Endbenutzern und Softwareentwicklern

Aspekte	**Endbenutzermeinung**	**Beobachtung des Softwareentwicklers**
Ziele und Werte	Tagesgeschäft ohne Verzögerung ausüben	Benutzer wollen mehr Funktionalität
Auswirkung der Nutzung	Performanzprobleme	Die neue Funktionalität hat die Arbeit vereinfacht
Softwareentwicklerziele und Werte	Softwareentwickler wollten verbessern	Funktionalitätserweiterung
Auswirkung der Entwicklung	längere Antwortzeiten	Funktionalitätserweiterung

Die Koeffizienten α_1 bis α_4 werden anhand der „alten“ Daten bestimmt. Da in den tatsächlich erbrachten Aufwand eine großer Anteil von Unsicherheiten eingeht wie:

- die Qualität der Mitarbeiter,
- die Änderungen der Fachlichkeit,
- die Änderungsgeschwindigkeit des Marktes,

kann eine solche Näherung nur für große Legacysysteme mit einer großen Maintenancemannschaft unter Nichtberücksichtigung massiver äußerer Einflüsse gemacht werden. Zu den massiven äußeren Einflüssen zählen Veränderungen, die große Teile der Legacysoftware betreffen, so z.B.:

- das Jahr-2000-Problem,
- oder die Euroeinführung.

7.11 Menschliche Effekte

Ein besonderes Phänomen mit starken psychologischen Ursachen, welches hauptsächlich in der adaptiven Maintenance auftaucht, ist die „Counterfinality“. Hierunter wird die Beobachtung verstanden, dass eine Reihe von scheinbar richtigen kleinen Schritten zu einem Ergebnis führen, welches der ursprünglichen Absicht widerspricht. Wie kommt es zustande, dass etwas mit den besten Absichten getan wurde und das Ergebnis trotzdem negativ ist?

Die Softwareentwickler und die Endbenutzer leiten ihre Erfahrungen über eine Legacysoftware aus sehr unterschiedlichen Kontexten ab. Eine Person, welche nicht in einem Kontext lebt und versucht, einen anderen Kontext zu verstehen, kann dies nur beschränkt tun; dies wird als „outside understanding“ bezeichnet. Dieses „outside understanding“ wird erlangt durch Beobachtung von Verhalten und eventueller Messung quantitativer Größen. Die Zielsetzungen, Regeln, Bedeutungen und Werteschemata der Endbenutzer kann der

Softwareentwickler[46] nur aus den Beobachtungen ableiten. Im Gegensatz dazu steht das „inside understanding“, welches innerhalb des Kontextes gewonnen wird. Hier wird das Verständnis durch Wechselwirkung untereinander und einem gemeinsamen Wertesystem gewonnen. Ein typisches Beispiel für diese Diskrepanz ist in Tab. 7.8 dargestellt. Beide Seiten nehmen für sich in Anspruch, mit den besten Absichten gehandelt zu haben.

In der Cultural Theory werden menschliche Verhaltensmuster durch Anthropologen untersucht. Danach fallen Menschen in fünf verschiedene Archetypen:

- Hierarchist
- Individualist
- Egalitarist
- Fatalist
- Eremit

Gewonnen werden diese Einteilungen aus der Interpretation und Sichtweise des jeweiligen Menschen auf seine Umgebung und die Welt im Allgemeinen. Die Fachbereichsmitglieder fallen in die Kategorie der Hierarchisten und die Softwareentwickler in die der Eremiten, mit der Folge, dass beide eine völlig unterschiedliche Weltsicht haben, dass sich aber auch die Art und Weise der Informationsgewinnung und Nutzung fundamental unterscheidet. Beobachtungen zeigen, dass zwischen beiden Gruppen wenig echte Dialoge entstehen und sich daher auf formale Prozeduren zurückgezogen wird.

Die Ansicht, dass die Maintenance sehr stark durch die Erfahrung des Maintenancemitarbeiters bestimmt ist, ist naheliegend und wird oft als gegeben hingenommen. Interessanterweise zeigen empirische Studien ein etwas differenzierteres Bild. Wenn man Experten von Legacysystemen mit so genannten Neulingen vergleicht, so zeigt sich, dass die Experten Evolution sehr viel effizienter implementieren als Neulinge, allerdings nur dann, wenn die vorliegende Software einem bekannten Muster folgt. Bezüglich ihrer Fähigkeit, den entstehenden Aufwand einer Maintenanceaufgabe vorauszusagen, gibt es keine Korrelation zwischen Erfahrung und der Vorhersage bei größeren Änderungen. Dies bedeutet, dass die Experten und die Neulinge bei komplexen Maintenanceaufgaben genauso falsch schätzen.

Die Umsetzung der Maintenanceaufgabe geschieht bei den Experten deswegen schneller, weil sie die Information über die Legacysoftware in größeren abstrakten Blöcken gliedern können als Neulinge, und sie lösen die Probleme im Vergleich zu Neulingen auf einer stärker prinzipienbasierten Strategie. Dies führt dazu, dass die Menge an Erfahrungen, sprich das Alter des Experten, eine untergeordnete Rolle spielt. Viel wichtiger als die Dauer der

[46] In Bezug auf Softwareentwicklungsumgebungen oder Compiler ist der Entwickler auch ein Endbenutzer. Interessanterweise beschweren sich Entwickler bezüglich der von ihnen genutzten Software über die gleichen Dinge wie die Mitarbeiter der Fachbereiche: Performanz, Instabilität, Komplexität, Ergonomie ...

Berufserfahrung ist die Art der Erfahrung oder des Trainings , wobei hier zu berücksichtigen ist, dass mit zunehmendem Alter generell mehr Paradigmen erworben wurden, mit der Folge, dass kleinere Probleme effizienter gelöst werden können. Bei größeren Veränderungen sind Neulinge und Experten genauso ratlos! Aus psychologischen Gründen sind Neulinge eher in der Lage, ihre Ratlosigkeit zuzugeben. Experten würden durch ein Eingeständnis der Unfähigkeit, das Problem in den Griff zu bekommen, ihren Expertenstatus und damit eine gehörige Menge sozialer Anerkennung verlieren. Von daher spiegelt das geäußerte Vertrauen nicht die tatsächliche Lage wider.

Die Gründe, warum es die Experten genauso schwer haben wie die Neulinge, liegen in den spezifischen Eigenschaften von größeren Maintenanceaufgaben.[47] Zum einen ist eine Legacysoftware so komplex, dass ein einzelner Softwareentwickler sie nicht mehr vollständig erfassen kann; die Legacysysteme entziehen sich einer detaillierten Modellbildung. Zum anderen ist Maintenance primär eine stochastische Umgebung, d.h. es existieren bestimmte Wahrscheinlichkeiten, dass ein Vorgehen sinnvoll ist, aber selten lassen sich exakte, d.h. deterministische, Strategien bilden.

In einer solchen Umgebung ist der einzelne Softwareentwickler auf seine Fähigkeit angewiesen, Hypothesen zu formulieren und diese zu falsifizieren. Diesem Vorgehen stellen sich aber diverse Hindernisse in den Weg:

- Selektive Wahrnehmung – Die gleichen Ereignisse werden von verschiedenen Personen unterschiedlich wahrgenommen. Folglich werden die in einem Softwareprojekt gewonnenen Erfahrungen auch sehr unterschiedlich interpretiert.
- Das Phänomen der Selbstüberschätzung ist sehr weit verbreitet. Menschen tendieren dazu ihre eigenen Fähigkeit stets besser zu sehen, als diese tatsächlich sind. Eine Umfrage unter zufällig ausgewählten schwedischen Autofahrern ergab, dass 80% der Teilnehmer der Studie sich zu den besten 30% aller Autofahrer rechneten!
- Die Überbewertung der jüngsten Vergangenheit stellt ein weiteres Problem dar. Diese menschliche Einstellung erklärt einen Teil der Aktienhypes.
- Hindsight Bias – Das *„ich-habe-es-ja-gleich-gesagt“.* Kennt man das Ergebnis eines Prozesses, so entwickelt man die Einstellung, dies schon die ganze Zeit gewusst zu haben, es vorhergesagt zu haben.
- Menschen versuchen stets, ihre Vorurteile zu „beweisen“, nicht ihre Hypothesen zu falsifizieren. Folglich wird sich an eine einmal formulierte Hypothese geklammert[48] und alle der Hypothese widersprüchlichen Erfahrungen werden ausgeblendet.

[47] Dies erklärt auch das Blum'sche Paradoxon: Je älter eine Legacysoftware ist, desto mehr Maintenanceerfahrung existiert. Ein Mehr an Erfahrung sollte die Kosten senken. In der Praxis stellt sich jedoch das Gegenteil heraus: Je älter das System, desto aufwändiger die Maintenance.

[48] Die Aufgabe der Hypothese wäre ein subjektiver Verlust, s. auch Fußnote S.224.

- Wenn einmal eine Regel entdeckt wurde, wird sie als gültig angenommen, obwohl sie nicht validiert werden konnte. Die Selbstsicherheit des Experten wächst mit der Fähigkeit, Regeln zu finden, unabhängig von der Tatsache, ob diese Regeln validiert werden konnten oder nicht.
- Häufig wird das Ergebnis durch andere Faktoren, wie beispielsweise „Self-fulfilling Prophesis“, beeinflusst, welche die Softwareentwickler daran hindern, die Falschheit der jeweiligen Einschätzung wahrzunehmen.
- Die meisten Menschen bevorzugen deterministische Abhängigkeiten gegenüber Wahrscheinlichkeiten.[49] Wenn es keine deterministischen Regeln gibt, wird angefangen zu raten! Dies führt bei der Softwareentwicklung dazu, dass vermutet wird, dass bei gleichen Anfangsbedingungen immer dasselbe Ergebnis produziert wird, obwohl eine distanzierte Beobachtung aufzeigen würde, dass eine Entwicklung diverse unterschiedliche Ergebnisse haben kann.
- Übermäßige Selbstsicherheit bei den Experten – Mit der zunehmenden Fähigkeit, Regeln zu „entdecken“, geht eine zunehmende Selbstsicherheit bezüglich der eigenen Vorhersagen einher. In der Praxis tendieren die meisten Maintenancemitarbeiter dazu, ihre eigenen Fähigkeiten zu über- und die Komplexität einer Legacysoftware zu unterschätzen. Die Gründe für diese Differenz sind in der Psychologie zu suchen:
  - Menschen glauben stets, mehr Informationen für ihre Entscheidungen zu nutzen als sie tatsächlich aktiv verwenden, mit der Folge, dass eine Entscheidung um so sicherer erscheint, je mehr Informationen bekannt sind, obwohl sie nur auf einer kleinen Teilmenge der Information beruht.
  - Im Allgemeinen wird niemand nach entkräftender Information bezüglich seiner eigenen Vorhersagen suchen, folglich wird auf den einmal gefassten Hypothesen beharrt.
- Wertlosigkeit der Erfahrung – Vieles an persönlicher Erfahrung, im Sinne von deterministischen Regelwerken in der Maintenance, ja sogar in der gesamten Softwareentwicklung, ist im Grunde wertlos:
  - Die Veränderungen des Kontextes machen die alten Erfahrungen obsolet. Manchmal geht das sogar so weit, dass die Erfahrung den Kontext verändert. Beispielsweise sind Performanzregeln bei hierarchischen Datenbanken auf relationale Datenhaltung nicht übertragbar.
  - Es ist in den meisten Fällen unmöglich zu sagen, was passiert wäre, wenn man anders gehandelt hätte. Von daher lassen sich Ursache und Wirkung oft nicht separieren.
  - Die gewonnene Erfahrung ist meistens sehr kontextabhängig und ein großer Teil des Kontextes ist nicht zu beschreiben und folglich nicht reproduzierbar.

[49] Albert Einstein lehnte lange die Quantenmechanik, welche sehr stark auf Wahrscheinlichkeiten basiert, mit den Worten ab: *... Gott würfelt nicht!*

- Subjektive Heuristiken – Konfrontiert mit dem Versuch, deterministische Regelwerke aufzubauen, benutzen Maintenancemitarbeiter oft Heuristiken, die sie aus ihrer Erfahrung ableiten. Zu diesen Heuristiken zählen:
  - Verfügbarkeitsheuristik – Je einfacher eine Person sich an ein Ereignis erinnern kann, desto besser wird die Person dieses Ereignis reproduzieren. Dies hat zur Folge, dass Wiederholungen oder Patterns einfacher und schneller wiedergefunden werden.[50] Meistens funktioniert diese Heuristik relativ gut. Es gibt allerdings Ausnahmen, wo diese Heuristik völlig unangebracht ist, so beispielsweise, wenn es sich um ein völlig neues Problem handelt.
  - Analogieheuristik – Viele Vorhersagen werden auf Grund von Analogiebetrachtungen durchgeführt. Ein solcher Prozess besteht aus zwei Schritten:
    1 Verankerung – Die „alte“ Erfahrung ist der Ausgangswert.
    2 Adjustierung – Die Verankerung wird auf Grund der Veränderung des Kontextes adjustiert und damit eine neue Vorhersage getroffen.

    Die Schwierigkeit ist hierbei nicht die Verankerung, diese fällt sehr leicht. Die meisten Menschen unterschätzen die Adjustierung drastisch. So kommen die sehr optimistischen Aufwandsschätzungen von Softwareentwicklern zustande. Die gängige Technik, eine „Worst-Case-Schätzung“ durchzuführen, ist der Versuch, eine andere Verankerung zu wählen.
  - Ähnlichkeitsheuristik – Wenn zwei Aufgaben sich in bestimmten Eigenschaften ähnlich sind, überträgt der einzelne Softwareentwickler diese Ähnlichkeit auf alle anderen Eigenschaften der beiden Aufgaben: Je ähnlicher zwei Aufgaben sind, desto wahrscheinlicher ist es, dass sie sich auch gleich verhalten. Diese Heuristik vernachlässigt einen großen Teil der beeinflussenden Faktoren völlig.

Ein Schlüssel zum besseren Verständnis von Maintenance ist es, sie als eine stochastische, von der Wahrscheinlichkeit geprägte, Tätigkeit wahrzunehmen. Aber ein Mensch kann dies nur dann entdecken, wenn er in der Lage ist, die Wahrscheinlichkeitsregeln den deterministischen Regeln gegenüberzustellen. Leider hat die vorherige Erörterung gezeigt, dass der einzelne Softwareentwickler dazu faktisch nur sehr schwer in der Lage ist. Folglich muss diese Feststellung und das dazugehörige stochastische Regelwerk von außen, via Schulung und Training, vermittelt werden.

Zusammenfassend ist zu sagen, dass die Rückkopplung einer Maintenanceaufgabe nur über das Ergebnis stattfindet. Es ist sehr schwer, ein Modell der Aufgabe zu entwickeln, wenn es keinerlei Rückkopplung innerhalb des Modells gibt. Zur Aufwandsschätzung benutzen die „Experten“ die ihnen bekannten Analogien. Diese tendieren bei völlig neuen Aufgabenstellungen zum Versagen. Diese Faktoren beeinflussen die Maintenance drastisch und führen dazu,

[50] Umgekehrt führt daher auch die bewusste Beschränkung bei der Erstellung von Software auf einige wenige Patterns zu einer Vereinfachung der Maintenance.

s. Abschn. 7.10, dass der Aufwand in Gl. 7.15 nur bedingt von der Erfahrung der Maintenancemannschaft abhängt.

Neben den psychologischen Effekten, die die Maintenance beeinflussen, gibt es auch soziologische Effekte. Ein Softwareentwickler, welcher Maintenance durchführt, muss sehr oft den Sourcecode anderer Softwareentwickler bearbeiten und verändern. Dabei muss er jedes Mal die innere Hürde des „not my code“ überwinden. Außerdem kommt vielen Softwareentwicklern Maintenancearbeit wie ein Abstellgleis vor; sie wird innerhalb der Gemeinschaft der Softwareentwickler als unproduktiv bzw. als nicht-kreativ empfunden. Diese unterschwellige Einstellung führt zu einer starken emotionalen Abwertung der Maintenancetätigkeit. So kann beobachtet werden, dass sehr ehrgeizige Softwareentwickler Maintenance als das Ende ihrer Karriere empfinden. Das Management tut hier einiges dazu, dieses Bild noch zu verstärken. Manchmal werden Positionen in Neuentwicklungen von Managern als eine Art „Bonus“ an Softwareentwickler verteilt; damit wird sehr direkt die geringe Wertschätzung von Maintenance seitens des Managements ausgedrückt. Hinzu kommt noch der Verdacht, dass der Softwareentwickler, welcher Maintenance macht, den Ursprungsentwickler kontrollieren möchte, indem er den Sourcecode[51] begutachtet. Für den Ursprungsentwickler sieht dies wie eine Kontrolle oder wie ein Anzweifeln seiner Fähigkeiten aus. Dieses Gefühl nachträglich kontrolliert zu werden führt oft zu Konflikten innerhalb der Softwareentwicklungsabteilungen, da der einzelne Softwareentwickler nicht in der Lage ist, dieses Gefühl zu äußern ohne sich dem Verdacht auszusetzen, dass er schlechte Qualität produziert; von daher wird nach einem anderen Ventil für die entstehenden Frustrationen gesucht.

7.12 Stochastische Modelle

Wie schon im letzten Abschnitt angedeutet, hat Maintenance einen sehr starken stochastischen Charakter, d.h. Maintenance wird von Wahrscheinlichkeiten geprägt. Dies ermöglicht es, ein einfaches Modell der Maintenance zu konstruieren.

Eines der einfachsten stochastischen Modelle für die Maintenance geht davon aus, dass die zu wartende Software wie ein einfacher Graph aufgebaut ist, s. Kap. 2. Ein Zufallsgraph $\mathcal{G}$ besteht aus N Knoten und K Kanten. Der einfachste Zufallsgraph hat eine zufällige Anzahl von Kanten K bei fester Anzahl von Knoten N. Die Wahrscheinlichkeit, dass zwei Knoten A_i und A_j miteinander verbunden sind, sei p. In einem solchen Zufallsgraphen gilt für die maximale Zahl von Kanten K_{max} und der mittleren Zahl von Kanten $\overline{K}$:

$$K_{max} = \frac{1}{2}N(N-1) \tag{7.16}$$

[51] Softwareentwickler identifizieren sich sehr stark mit dem von ihnen entwickelten Sourcecode. Sie sprechen immer von „meinem Code“.

$$\overline{K} = pK_{max}. \tag{7.17}$$

Wird Gl. 7.17 umgekehrt, so folgt:

$$p = \frac{2\overline{K}}{N(N-1)}. \tag{7.18}$$

Damit kann jeder Graph dargestellt werden als $\mathcal{G} = \mathcal{G}(N,p) = \mathcal{G}(\overline{K},p)$.
Für den Average Node Degree, Gl. 2.3, gilt:

$$\bar{N}_d = p(N-1) \approx pN$$

Für große N wird der Graph $\mathcal{G}$ regulär, d.h. alle Knoten haben dieselbe Anzahl von Kanten. In diesem vereinfachten Modell, unter der Annahme, dass der Graph stets regulär ist, stellt sich die Software als eine Reihe von Knoten und Kanten dar. Teilt man die Menge der Knoten in zwei Kategorien:

- stabile Knoten, welche eine lokale Veränderung nicht an Nachbarknoten weitergeben,
- labile Knoten, welche mit einer Wahrscheinlichkeit s ihre Veränderung an Nachbarknoten weitergeben der so genannte Ripple-Effekt, s. S. 181

so propagiert eine Veränderung durch den Graphen $\mathcal{G}$ mit einer gewissen Wahrscheinlichkeit. Die Wahrscheinlichkeit q, dass ein Knoten labil ist, ist einfach gegeben durch:

$$q = \frac{N_{labil}}{N} = 1 - \frac{N_{stabil}}{N}.$$

Für kleine $q, s \ll 1$ ergibt sich näherungsweise die Wahrscheinlichkeit, dass eine Änderung vom Knoten i zum Knoten j propagiert zu

$$\Lambda_{ij} = qs$$

und die Wahrscheinlichkeit, dass zwei Knoten über eine solche Veränderung verbunden werden zu:

$$\Lambda_{ij}^{neu} = pqs.$$

Ein so entstehender Subgraph hat die Eigenschaften eines Cayley-Baumes[52] mit r Zweigen und der Verbindungswahrscheinlichkeit qs. Dieser Cayley-Baum enthält im Mittel die Knotenzahl:

$$\overline{N_{Cayley}}(r) = \frac{1+qs}{1-r-qs}. \tag{7.19}$$

Die entstehenden Fragmente an Cayley-Bäumen folgen einem Expotentialgesetz mit

$$\sigma \sim N_{Cayley}^{-\frac{5}{2}} \mathrm{e}^{-cN_{Cayley}}.$$

[52] Auch als Bethe-Gitter bekannt.

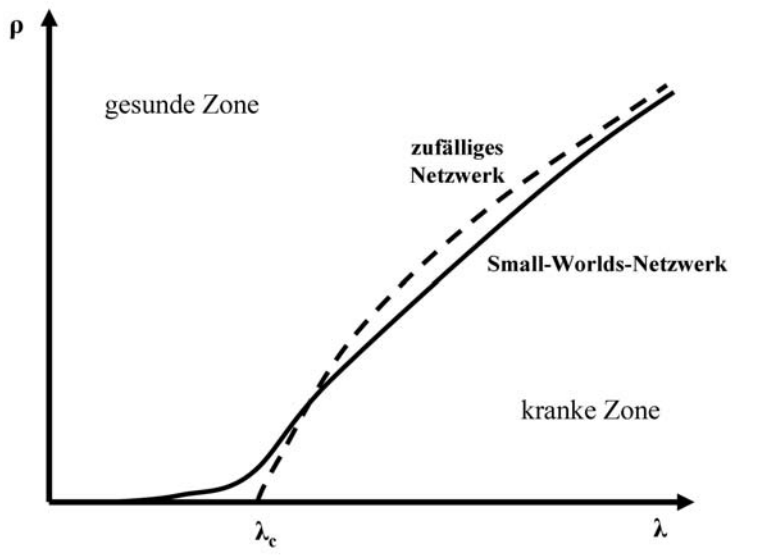

Abb. 7.13: Die Ausbreitung von Fehlern in einem Small-Worlds-Netzwerk

Das Modell der Beschreibung der Maintenance durch einen Graphen lässt sich jedoch drastisch verbessern, wenn berücksichtigt wird, dass große Softwaresysteme sich nicht wie reguläre Graphen verhalten, was bei einer zufälligen Defektverteilung zu Cayley-Bäumen führt, sondern besser durch Small-Worlds-Netzwerke, s. Abschn. 2.4, simuliert werden können. Da ein Small-Worlds-Netzwerk aus wenigen zentralen Hubs und einer Menge von anderen Knoten besteht, die weniger Verbindungen haben, breiten sich zufällige Effekte anders in einem solchen Netzwerk aus. Der typische Ripple-Effekt wirkt in einem solchen Netzwerk wie eine Infektion an einem Knoten und diese Infektion wird sich je nach Verteilung im Netzwerk wie eine Epidemie ausbreiten oder sie kann eingedämmt werden.

In einem einfachen Modell, das der Epidemiologie entlehnt ist, kann die Dichte der „infizierten“ Knoten ρ gegenüber der Ausbreitungsrate λ betrachtet werden, s. Abb. 7.13. Hierbei ist die Ausbreitungsrate gegeben durch:

$$\lambda = \frac{\nu}{\delta},$$

wobei ν die Wahrscheinlichkeit angibt, mit der ein Knoten „infiziert“ wird, wenn sein Nachbarknoten schon infiziert wurde, und δ gibt an, mit welcher Wahrscheinlichkeit „geheilte“ Knoten wieder „infiziert“ werden können. Für Netzwerke existiert ein kritisches λ_c, so dass für alle $\lambda > \lambda_c$ das gesamte Netzwerk „infiziert“ wird. Für Small-Worlds-Netzwerke ist dieser Schwellenwert kleiner als für rein zufällige Netzwerke; dies liegt daran, dass ein Hub,

wenn er in einem Small-Worlds-Netzwerk einmal „infiziert“ ist[53], diese „Infektion“ schnell weitergibt und sich sehr wahrscheinlich nach einer „Heilung“ erneut „infiziert“.

Die dazugehörige Feldgleichung ist, mit der Infektionswahrscheinlichkeit $\Theta(\rho_k)$:

$$\frac{\mathrm{d}\rho_k(t)}{\mathrm{d}t} = -\rho_k(t) + k\lambda\,(1-\rho_k(t))\,\Theta\,(\rho_k(t))$$

Wie kann nun eine Legacysoftware, welche einem solchen Small-Worlds-Netzwerk entspricht, „immunisiert“ werden? Dazu reicht es aus, die Hubs zu stärken, d.h. diese zu immunisieren.

Wie kann eine solche Immunisierung bei der Software aussehen? Darauf gibt es folgende Antworten:

- Kapselung, damit Änderungen lokal bleiben
- Entkoppelung, um die Gesamtkomplexität zu reduzieren
- Defensive Programmierung, um die lokale Stabilität zu gewährleisten

Die Vorgehensweisen wie Kapselung und Entkoppelung sind auch Maßnahmen, wie sie im Reengineering angewandt werden, s. Abschn. 5.9. Die größten Fortschritte bei der defensiven Programmierung, jenseits von Programmierrichtlinien und IF-THEN-ELSE-Zweigen, werden durch zwei einfache Maßnahmen erzielt:

- Boundschecking – Das Boundschecking besteht aus zwei Teilen. Zum einen wird überprüft, ob der vermeintlich verwendete Speicherplatz auch mit dem deklarierten übereinstimmt und zum anderen, ob der vordefinierte Wertebereich eingehalten wird. Ersteres ist in typsicheren Sprachen, wie Ada oder Java, nicht notwendig, erweist sich aber bei C und C++ als sehr wichtig. Das zweite ist etwas, das in allen Sprachen wichtig ist, da die Fachlichkeit oft den möglichen Wertebereich eines Datentyps drastisch einschränkt.
- Assertions – Innerhalb der fachlichen Domäne lassen sich oft Zusicherungen formulieren, welche Zwangsbedingungen auf den lokalen Variablen darstellen. Beispielsweise:

  $$\mid akt.Jahr - Geburtsjahr \mid +1 \geq Alter$$

  oder

  $$\sum_{Konten} Buchungen = 0.$$

  Solche Forderungen können stets aus der Domäne abgeleitet werden und sollten bei der Verletzung eine Fehlermeldung produzieren, damit Defekte frühzeitig erkannt werden.

[53] Dies erklärt die Beobachtung der Verteilung des berüchtigten „Love Bug“-Computervirus. Obwohl dieser nicht besonders aggressiv ist – er zählt zu den schwachen Viren – lag er noch 1 Jahr nach seiner Entdeckung und angeblichen Beseitigung auf der siebten Stelle der weltweiten Virushäufigkeit.

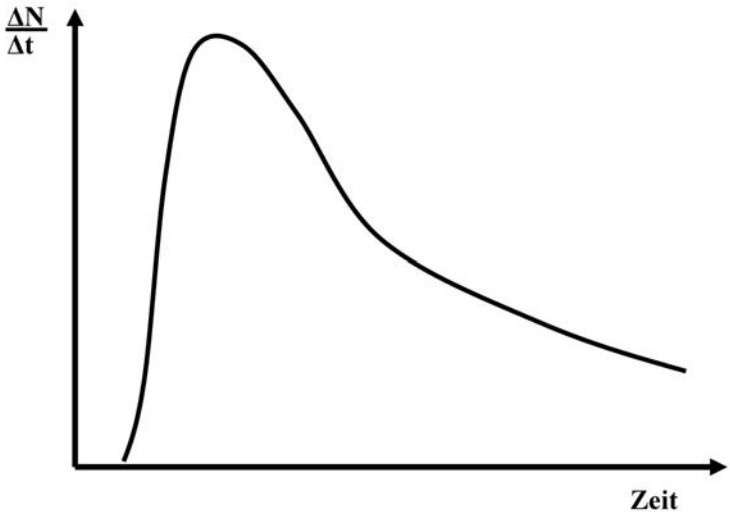

Abb. 7.14: Die Zahl der gefundenen Defekte pro Zeiteinheit $\frac{\Delta N}{\Delta t}$ als Funktion der Zeit in einem Release

7.13 Defektraten

Bisher wurde die Maintenance im Hinblick auf den einzelnen Defekt und die möglichen Resultate seiner Existenz und Behebung untersucht. Wie jedoch tauchen Defekte auf? Zwar ist es, nach der vorangegangenen Diskussion, einleuchtend, dass die Defekte stochastisch auftreten, jedoch ist nicht klar, wie die Wahrscheinlichkeit des Auftretens als Funktion der Zeit aussieht.

Aus Sicht eines einzelnen Releases oder eines festen Stands der Software erscheint die Zahl der Defekte zunächst als eine steil ansteigende Kurve, welche anschließend abflacht und sich asymptotisch einer Konstanten nähert, s. Abb. 7.14. Der Grund für dieses Verhalten ist, dass zunächst viele Fehler durch den jeweiligen produktiven Einsatz gefunden werden und dass sich durch die zunehmende Durchdringung der Software durch die Endanwender die Fehlerrate langfristig stabilisiert. Bei Systemen mit sehr hoher Entropie oder einem niedrigen Maintainability Index nähert sich die Asymptote nicht Null, sondern eventuell einem großen Wert. Solche Legacysysteme sind de facto nicht mehr beherrschbar.

Besonders interessant ist Abb. 7.15; hier wurde die Wahrscheinlichkeit p_+ dargestellt, weitere Defekte zu finden, wenn schon eine gewisse Anzahl N_D bekannt sind. Je mehr Defekte bekannt sind, desto größer ist die Wahrscheinlichkeit zusätzliche Defekte zu finden! Dies bedeutet nicht, dass die nachfolgenden Defekte kausal mit den vorhergehenden Defekten korreliert sind, sondern dass

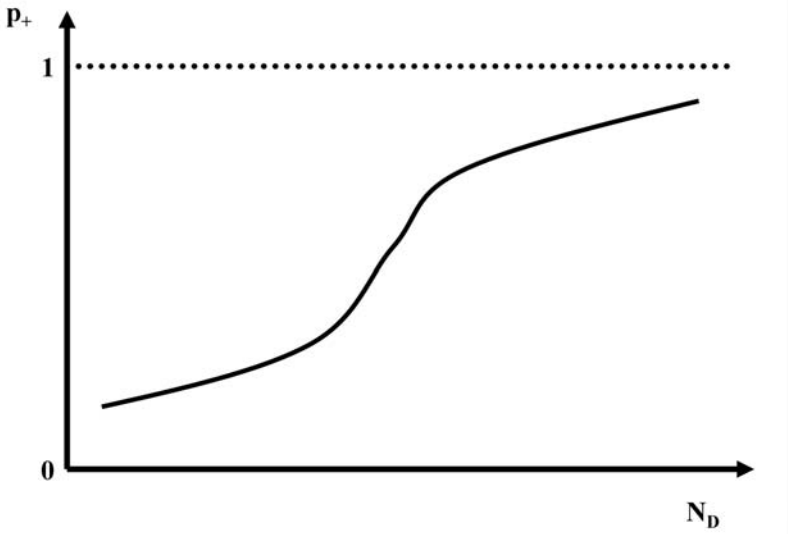

Abb. 7.15: Die Wahrscheinlichkeit p_+, dass noch mehr Defekte gefunden werden als Funktion der Zahl N_D der bereits gefundenen Defekte

eine Software, bei der viele Defekte gefunden werden, insgesamt eine große Menge an Defekten besitzt.

Betrachtet man die Defekte über alle Releases hinweg, so entsteht die bekannte „Badewannenkurve“, s. Abb. 7.16. Hier ist die Entwicklung einer Software ablesbar. Nach den anfänglichen „Kinderkrankheiten“, der linke Teil der Kurve, setzt ein Gebiet relativ konstanter Defektraten ein. In diesem Gebiet wirkt die korrektive und präventive Maintenance stabilisierend auf die Entropie, während die adaptive Maintenance die Entropie erhöht. Dies hat den Gesamteffekt, dass die Entropie relativ konstant bleibt. Ab einem gewissen Alter befindet sich die Software auf dem rechten Ast. In diesem Zustand tendieren Maintenancetätigkeiten dazu, die Entropie zu erhöhen und ein Ansteigen der Defektraten ist zu beobachten. Alle Legacysysteme befinden sich in der Mitte oder auf dem rechten Ast. Falls sie sich auf dem rechten Ast befinden, haben sie eine hohe Entropie und es muss versucht werden, sie durch Migrationsmaßnahmen in die Mitte von Abb. 7.16 zurückzuführen.

7.14 Services-Maintenance

Die Maintenance von Service-Orientierten-Architekturen, speziell von Webservices, ist anders als die Maintenance von traditioneller Software. Die treibenden Kräfte in traditioneller Software sind Defekte und Technologieverände-

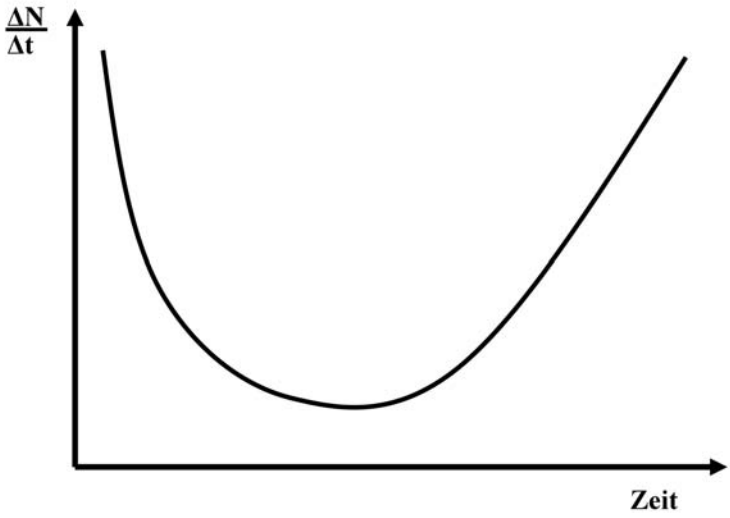

Abb. 7.16: Die Zahl der gefundenen Defekte pro Zeiteinheit $\frac{\Delta N}{\Delta t}$ als Funktion der Zeit über alle Releases

rungen. Im Gegensatz dazu leben Services in einer geänderten Welt, hier sind Systemgrenzen auf Dauer nicht mehr rigide, da die Services auf dem Markt eingekauft werden können.

Folglich werden Services erst zu dem Zeitpunkt ihrer Ausführung eingebunden mit der Folge, dass Kriterien wie:

- bester Service
- billigster Service
- schnellster Service
- aktuellster Service

plötzlich ausschlaggebende Größen werden. Aus diesem Blickwinkel heraus stellt sich Maintenance für das nutzende Unternehmen weniger als eine Frage der Pflege von Software, denn als eine Reaktion auf Marktgegebenheiten dar. Zwar müssen die Lieferanten auch ihre Services einer Maintenance unterziehen, nur merken die Benutzer dies nicht. Das so entstehende Modell ist nicht vergleichbar mit dem Einsatz und den Folgen von COTS-Software, s. Kap. 10, da bei den Services die Austauschbarkeit sehr viel besser, schneller und auch kurzfristiger bewerkstelligt werden kann.

8

Outsourcing

Entgegenkommen
Die ewig Unentwegten und Naiven
Ertragen freilich unsre Zweifel nicht.
Flach sei die Welt,
erklären sie uns schlicht,
und Faselei die Sage von den Tiefen.

Denn sollt es wirklich andre Dimensionen
Als die zwei guten, altvertrauten geben,
Wie könnte da ein Mensch noch sicher wohnen,
Wie könnte da ein Mensch noch sorglos leben?

Um also einen Frieden zu erreichen,
So laßt uns eine Dimension denn streichen!

Denn sind die Unentwegten ehrlich,
Und ist das Tiefensehen so gefährlich,
Dann ist die dritte Dimension entbehrlich.

Hermann Hesse, 1877-1962

Für jedes größere Unternehmen stellt sich auf Dauer die Frage des Outsourcings. Hinter dem Begriff Outsourcing versteckt sich die Delegation von Tätigkeiten, die zu einem Zeitpunkt im Unternehmen durchgeführt wurden, auf Dritte, welche außerhalb des Unternehmens sind. Die gegenteilige Bewegung, dass Zurückholen von vorher ausgelagerten Tätigkeiten, wird auch als Insourcing bezeichnet, insofern sind die Begriffe In- und Outsourcing nicht auf die Software und ihre Entwicklung beschränkt, sondern können auf jede Form der unternehmerischen Tätigkeit angewandt werden. Das vorliegende Buch kann das Outsourcing nicht in allen Facetten abhandeln, sondern beschränkt sich auf das Problem des Outsourcings im Kontext von Legacysoftware, welche per definitionem immer das Kerngeschäft des Unternehmens berührt.

Die möglichen Gründe für das Outsourcing sind divers, so gibt es Outsourcing, wenn die IT nicht das Kerngeschäft eines Unternehmens darstellt oder es innerhalb des Unternehmens nicht genügend qualifizierte Softwareentwickler[1]

[1] In diesem Fall wäre ein temporäres Insourcing, d.h. Consultingunternehmen oder Freelancer aufzunehmen, eine andere Möglichkeit.

gibt. Oft kann beobachtet werden, dass eine große Unzufriedenheit mit der IT innerhalb des Unternehmens besteht, oder dass die Softwareentwickler sich nur noch mit Maintenance beschäftigen und nicht mehr neue Software entwerfen können. In solchen Fällen liegt es nahe, an ein Outsourcing zu denken. Neben der Softwareentwicklung können auch die Infrastruktur, das Netzwerk oder ähnliche Teile ausgelagert werden. In Bezug auf Legacysoftware soll hier nur das Outsourcing von Softwareentwicklung und -maintenance betrachtet werden. Der Fall, dass aus organisatorischen und steuertechnischen Gründen der komplette Entwicklungsbereich eines Unternehmens ausgelagert wird und dieser dann die Rolle des Outsourcingpartners spielt, wird hier explizit nicht betrachtet, da diese Situation sich nicht grundsätzlich von einer Inhouselösung unterscheidet.

Obwohl das Outsourcing keine primäre Entscheidung der jeweiligen Entwicklungsmannschaft ist, sondern sehr viel stärker im Bereich der Managemententscheidungen liegt, wird jedoch jedes größere Unternehmen in seiner Organisationsstruktur, wie auch in der Handlungsfähigkeit von einer Outsourcinglösung betroffen sein. Speziell beim Vorhandensein von größeren Mengen von Legacysoftware stellt sich rasch die Frage des Outsourcings mit dem Hintergedanken, die mühsame und frustrierende Maintenancetätigkeit an andere, außerhalb des Unternehmens, delegieren zu können.

Häufig kann der Wunsch eines Unternehmens, Outsourcing vorzunehmen, auf eine tiefe Verunsicherung zurückgeführt werden. Wenn geschäftskritische Teile der Applikationslandschaft so komplex werden, s. Kap. 4, dass der Betrieb sowie jede sinnvolle Form der Maintenance, s. Kap. 7, stark gefährdet ist, liegt es nahe, nach einer drastischen Lösung zu suchen. In solchen Fällen wird oft ein Outsourcing des bestehenden Systems an Dritte vorgenommen. Allerdings wird mit dem Outsourcing die Vergangenheit vergessen, d.h. das aufgebaute Wissen über Verhalten und Fehler geht verloren: der neue Partner, welcher jetzt die outgesourcete Legacysoftware betreut, wiederholt alle schon einmal gemachten Fehler! Verstärkt wird dies noch, wenn das Outsourcing mit Hilfe eines Offshorings durchgeführt wird. In diesem Falle kommen dann noch kulturelle Differenzen und Missverständnisse hinzu. Unter Offshoring[2] versteht man das Outsourcing zu einem weit entfernten Anbieter, welcher beispielsweise in Indien oder China ansässig ist. Outsourcing nach naheliegenderen Zielen, so die osteuropäischen Staaten, Rumänien, Polen oder Ukraine, wird Nearshoring oder auch Eastsourcing genannt.

Historisch gesehen war das Jahr-2000-Problem, kurz das Y2K-Problem, die treibende Kraft hinter dem Offshoringboom. Das Y2K-Problem tauchte Mitte der neunziger Jahre auf, als bewusst wurde, dass ein großer Teil der Legacysoftware aus den sechziger und siebziger Jahren die Jahreszahlen bei einem Datum nur zweistellig verarbeiten konnte. In den meisten Fällen war die Software nicht originär für eine solche lange Lebensdauer konzipiert. Zur

[2] Für manche Unternehmen, so beispielsweise die öffentliche Hand, sind solche Szenarien aus politischen Gründen nicht öffentlich denkbar.

gleichen Zeit setzte der Internetboom ein, mit der Folge, dass Softwareentwickler gut bezahlte und subjektiv anspruchsvollere Jobs in der Internetindustrie, den ominösen Dot-Com-Unternehmen, der mühsamen Arbeit mit dem Y2K-Problem in Legacysystemen vorzogen. Auf dem Arbeitsmarkt in den USA und Europa war eine plötzliche Knappheit an Softwareentwicklern zu verspüren, welche sich mit dieser Legacysoftware auseinandersetzen wollten. In diese Lücke sprangen Offshore-Unternehmen, speziell, im Falle der USA, indische Unternehmen.

Tab. 8.1: Die verschiedenen Optimierungen des Outsourcings

Größe	**Outsourcingname**
Time to Market	Fastsourcing
Kosten	Costsourcing[3]
Skalenökonomie	Scalesourcing
Gewinn	Profitsourcing

Die Probleme des Offshorings werden erst nach einiger Zeit sichtbar. Typische Signale sind:

- Terminverschiebungen
- Kostenexplosion
- Qualitätsprobleme
- Abstürze

Jetzt wird plötzlich innerhalb der Fachbereiche realisiert, dass Outsourcing kein „Silver Bullet“, s. Abschn. 13.15, ist, und das Pendel schwingt zurück. Das Insourcing wird wieder zum Nonplusultra erklärt. Nach einer gewissen Zeit im Insourcing startet der Zyklus[4] wieder von neuem. Ob große Outsourcingprojekte wirklich ökonomisch sinnvoll sind, mag bezweifelt werden, da in den letzten 10 Jahren mindestens 50% der großen Outsourcingprojekte noch im ersten Jahr gestoppt wurden und 80% der als erfolgreich deklarierten Outsourcingprojekte keinerlei Einsparung ergaben.

Dieses Pendeln zwischen zwei Extremen kann ein Unternehmen stark gefährden, da es neben der Mitarbeitermotivationsproblematik in der Regel auch sehr kapitalintensiv und, vor allen Dingen, risikoreich ist. Da die Grundlage der meisten großen Unternehmen – sei es direkt oder indirekt – ihre IT ist, darf es nicht verwundern, dass Out- und Insourcingankündigungen direkte

[4] Oft wird der Zyklus durch einen personellen Wechsel an der Spitze des Unternehmens initiiert. Interessanterweise führen Wechsel, bei denen Führungskräfte von außerhalb des Unternehmens leitende Positionen innerhalb des Unternehmens erlangen stets zu Formen des Outsourcings, nie zu einem Insourcing, was auf ein grundsätzliches Misstrauen gegenüber der eigenen IT zurückzuführen ist.

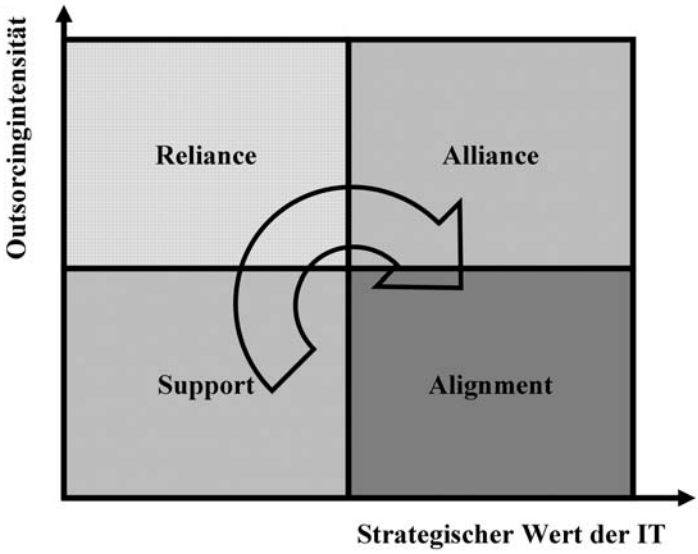

Abb. 8.1: Zunehmende organisatorische Ausrichtung durch Outsourcing

Auswirkungen auf den Marktwert der beteiligten Unternehmen haben. Professionelle Investoren reagieren sehr schnell auf solche Ankündigungen, was sich simultan im jeweiligen Aktienkurs niederschlägt.

In der Praxis ist das vollständige Outsourcing eher selten zu finden. Je nach Zielrichtung und Markt des Unternehmens wird Outsourcing so betrieben, dass dieses Ziel auch mit einer Mischung aus internen Kräften und den entsprechenden Outsorcingpartnern erreicht wird. Je nach der zu optimierenden Größe wird das Outsourcing unterschiedlich benannt, s. Tab. 8.1.

Einer der am meisten übersehenen Problemkreise des Outsourcings ist die starke Vernachlässigung des inhärenten Kommunikationsaspektes[5] bei der Softwareentwicklung. Die meisten Softwareprojekte scheitern nicht an der Technik, sondern an der Kommunikation[6] zwischen den Beteiligten. Zwar lässt sich ein Kommunikationskanal auch beim Offshoring aufbauen, jedoch sollte nicht die Wichtigkeit der simultanen Kommunikation übersehen werden.[7]

[5] Software wird von Menschen für Menschen gemacht.
[5] Das „klassische“ Outsourcing.
[6] Speziell die agilen Methoden berücksichtigen dies, s. S. 285 und Abb. 11.9.
[7] Ein Kammerorchester, bei dem der Cellist in Indien, der Geiger in Rumänien, die Oboe in London und der Dirigent in Berlin sitzt, dürfte ein Erlebnis der besonderen Art sein, insbesondere dann, wenn jeder eine andere Partitur benutzt.

Das Outsourcing sollte nie als singuläres Ereignis betrachtet werden, sondern als eine langfristige Strategie, s. Abb. 8.1. Die meisten Unternehmen starten mit kleineren Supportaufträgen, welche eine niedrige strategische Bedeutung haben, und wandern über die Reliance, das Sich-Verlassen auf den Outsourcer, jetzt wird er Provider genannt, zu einer strategisch ausgerichteten Allianz. Das Endziel der strategischen Ausrichtung ist das Alignment, die Ausrichtung des Outsourcingpartners an das eigene Unternehmen.

8.1 Vorgehensweisen

Damit überhaupt ein sinnvolles Vorgehen möglich ist, sollte ein gewisses Maß an quantifizierbarer Information gewonnen werden. Das einfachste Maß – es reicht für die meisten professionellen Outsourcingpartner aus – ist eine Schätzung der Menge der Funktionspunkte, s. Abschn. 2.3. Werden Größen wie Alter, eingesetzte Technologie, Datenvolumen und Zahl der verschiedenen Dateien oder Datenbanktabellen bzw. Datenstrukturen, mit erhoben, so kann anhand dieser Eckwerte meist eine recht komfortable Schätzung geliefert werden, zumindest von Seiten des Outsourcingpartners. Falls mehrere Anbieter[8] in Betracht gezogen werden, ergeben die Antworten auf die Kostenfrage die zu erwartende Bandbreite.

Tab. 8.2: Aufwände als Funktion der Anzahl der Funktionspunkte

V_{UFP}	$\overline{e}$	$\sum \overline{e}$
100	0.6204	62
1000	1.7277	1728
5000	3.5350	17675
10000	4.8116	48116
50000	9.8444	492223

In der Praxis schwanken die ermittelten Funktionspunkte recht stark, d.h. sie haben eine Varianz von ±30%.

Der mittlere Aufwand $\overline{e}$ für die Implementierung eines Funktionspunktes in Abhängigkeit der Gesamtmenge lässt sich durch die Gleichung 8.1 annähern:

$$\overline{e}_{FP} = \beta V_{UFP}^{\alpha}. \tag{8.1}$$

Die Werte von α und β sind von der Domäne, sowie der jeweiligen Organisation abhängig, s. Abb. 8.2. Die Formel Gl. 8.1 ergibt bei genauerer

[8] Oft werden solche Anfragen nur genutzt, um damit die bestehenden Outsourcingpartner oder die klassischen Insourcingpartner wie Consultingunternehmen preislich unter Druck zu setzen.

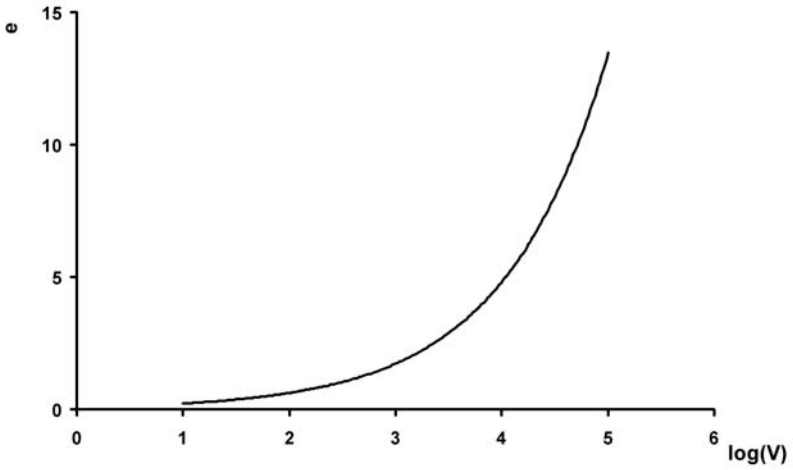

Abb. 8.2: Der mittlere Aufwand pro Funktionspunkt in Tagen mit den Werten $\alpha = 0.4448$ und $\beta \approx 0.08$ nach *Verhoeven*

Betrachtung, dass der Aufwand pro Funktionspunkt mit der Gesamtanzahl von Funktionspunkten ansteigt, s. Tab. 8.2.

Als nächster Schritt ist es notwendig, die eigenen Kosten den oben genannten gegenüberzustellen. Die meisten Unternehmen haben keine Vergangenheitsdaten, welche Auskunft über ihre Produktivität geben, aber die Produktivität lässt sich durch Gl. 8.2 annähern:

$$\overline{p}(V) \approx a + b\mathrm{e}^{-cV^d} \tag{8.2}$$

mit den Werten $a = 1.63$, $b = 38.373$, $c = 0.0622$ und $d = 0.4245$. Die mittlere Produktivität sinkt recht schnell ab, s. Tab. 8.3.

Tab. 8.3: Mittlere Produktivität in Funktionspunkten pro Mannmonat als Funktion der Anzahl der Funktionspunkte

V_{UFP}	$\overline{p}(V)$
100	26.35
1000	13.56
5000	5.43
10000	3.35
50000	1.71

Die Kosten Γ und die Dauer T ergeben sich recht einfach aus den Gleichungen 8.1 und 8.2 zu:

$$\Gamma = \frac{r * \overline{d}}{12} \frac{V}{\overline{p}(V)} \tag{8.3}$$

$$T = \frac{\gamma}{\overline{p}(V)}, \tag{8.4}$$

wobei in Gl. 8.3 r den gängigen Tagessatz und $\overline{d}$ die Anzahl der effektiven Arbeitstage pro Jahr darstellen, meist ca. 180 Tage. In Gl. 8.4 geht die Zahl der Mitarbeiter γ sowie die maximale Zahl an Funktionspunkten, welche ein einzelner Softwareentwickler möglicherweise bearbeiten kann, direkt ein.

Tab. 8.4: Kosten, Gl. 8.3 als Funktion der Anzahl der Funktionspunkte mit $r = 700$ und $\overline{d} = 180$

V_{UFP}	$Kosten$
100	34.200
1000	663.000
5000	8.300.000
10000	26.900.000
50000	264.000.000

Sowohl die Controller als auch die IT-Vorstände spielen gerne mit diesen Zahlen und den unterschiedlichen Faktoren. Besonders beeindruckt sind sie dann von dem Effekt, den gängigen Tagessatz r zu kappen, um damit „Einsparungen“ zu erzielen. Dies ist jedoch nicht besonders sinnvoll, da die Gleichungen 8.1–8.4 sich stets auf ein gut eingespieltes Team beziehen und die reinen Entwicklungskosten im Sinne der Implementation berücksichtigen. Ein Outsourcingunternehmen muss, bevor es überhaupt die Produktivität einer Inhousemannschaft erreichen kann, eine relativ flache Lernkurve durchlaufen, da eine Softwareentwicklung ohne Domänenwissen nicht sinnvoll machbar ist. Das Domänenwissen zu erwerben braucht Zeit, je nach Domäne 3-12 Monate.[9] Diese Zeit muss zunächst einmal investiert werden, was zur Folge hat, dass es alleine auf Grund der Lernkurve eine minimal sinnvolle Projektgröße für Outsourcing gibt. Die Effekte der Lernkurve sind vergleichbar mit den Auswirkungen beim Austausch einer einzelnen Person im Rahmen einer langlaufenden Maintenance, s. Abb. 8.3. Es dauert bis zu zwei Jahre, bis der neue Mitarbeiter die Produktivität des alten erreicht hat; anfänglich wird die Produktivitätskurve sogar negativ, da durch den neuen Mitarbeiter jetzt Kräfte

[9] In manchen Domänen noch länger, so beispielsweise Personalabrechnungssystemen im Bereich von Jahren. Aus dieser Perspektive betrachtet ist es einfacher, Personalabrechnungsfachleuten Programmierung beizubringen als Programmierern Personalabrechnungswissen.

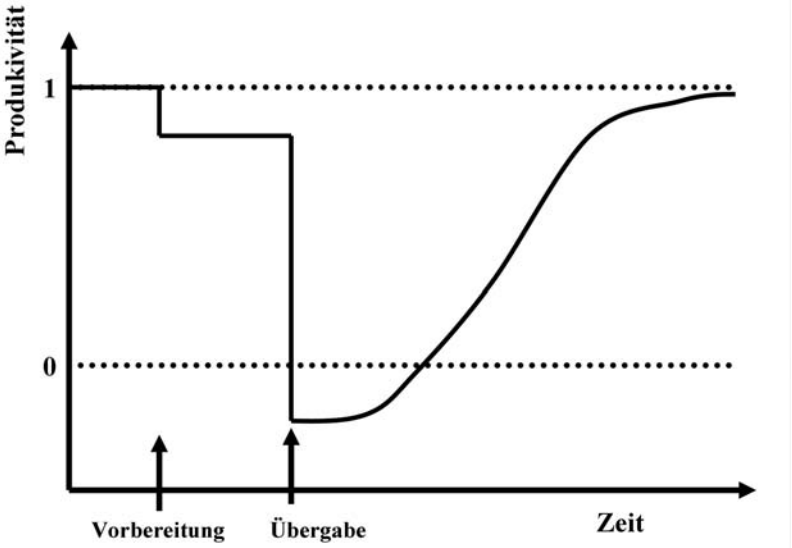

Abb. 8.3: Nachlassen der Produktivität beim Austausch eines Mitarbeiters

explizit gebunden werden. Dabei beläuft sich der Verlust, das Integral über die Kurve in Abb. 8.3, auf ungefähr ein Mannjahr!

Neben den Effekten der Lernkurve ist der stark erhöhte Bedarf an Qualitätssicherung zu beachten. Die erhöhten Qualitätssicherungskosten haben zwei Ursachen, zum einen hat Outsourcing fast immer einen Gewerkcharakter, mit der Folge, dass Abnahmekriterien und Ähnliches explizit überprüft werden müssen, zum anderen muss sichergestellt werden, dass der Outsourcingpartner wartbaren Sourcecode liefert, d.h. einen Sourcecode mit einem hohen Maintainability Index, s. Kap. 2, da die Software nach der Übergabe weitergepflegt werden muss. Zwar kann es sein, dass der Outsourcingpartner auch die Maintenance wahrnimmt, jedoch sollte man stets in der Lage sein, auch diesen „Partner“ zu ersetzen. Der Outsourcingpartner hat keinerlei Interesse daran, pflegbaren Sourcecode zu produzieren, zum einen ist es für ihn billiger, schnellen Wegwerfcode zu bauen, und zum anderen führt diese Nichtwartbarkeit des gelieferten Sourcecodes zu einer langfristigen Abhängigkeit des Kunden, beides erstrebenswerte Ziele für ein Outsourcingunternehmen.

Bei einem Offshoringansatz kommen noch drastische Kommunikationskosten hinzu. Offshoring setzt voraus, dass die Anforderungsspezifikation unzweideutig und sehr stabil ist. Bezüglich der Stabilität sollte berücksichtigt werden, dass auch die Anforderungen einer Evolution unterliegen, s. Abschn. 4.15, und davon ausgegangen werden kann, dass sich 40-70% der Anforderungen im Laufe des Projektes ändern können. Würde es gelingen, eine unzweideutige Spezifikation zu erstellen, so wäre diese automatisch verarbeit-

bar, ein Ziel, das hinter der Idee der MDA, Model Driven Architecture, steckt. Aber eine MDA benötigt keinen Outsourcingpartner, in Reinform benötigt sie nur eine Reihe von guten Generatoren. Außerdem ist die Schaffung von soliden Anforderungsspezifikationen die Hauptarbeit im Rahmen des Gesamtlebenszyklus, warum also den kleineren Teil outsorcen? Diese Tatsache, dass die Kosten für eine saubere Spezifikation immens hoch sind, wird durch die konkrete Arbeitsweise von Inhouseabteilungen oft nicht transparent gemacht, sondern unter dem Begriff Programmierung subsumiert, was zur fälschlichen Annahme führt, dass Offshoring sehr viel billiger sei. Folglich kann der Kostenfaktor bei der Betrachtung von Offshoring oder auch Outsourcing nicht der entscheidende Faktor sein.

8.2 Risiken

Zu den Risiken jedes Projektes gehört neben den oben angedeuteten Problemfällen auch das Risiko, dass der ursprüngliche Zeitplan nicht eingehalten werden kann. Verzögerungen in der Größenordnung von 50% sind durchaus üblich. Ein Teil dieser Verzögerungen wird durch klassische Projektmanagementfehler produziert, ein anderer, weitaus größerer Teil, ist die Folge des „Requirements Creeps“, s. Tab.4.6. Durch die Veränderung von nur 3,5% pro Monat erhöht sich die Anzahl der Funktionspunkte um den entsprechenden Anteil. Bei einem Projekt mit 10.000 Funktionspunkten und einer monatlichen Änderungsrate von 3,5%, führt das nun größere Projekt zu einem Absinken der Produktivität um 20% und einer Steigerung des Aufwandes um 80%. Allein diese einfachen Betrachtungen machen klar, wie risikobehaftet Softwareentwicklung im Allgemeinen und Outsourcing im Besonderen ist. Eine Folge dieses immens hohen Risikos ist die Tatsache, dass, laut Gartner Group, zwei Drittel aller Outsourcingprojekte in Europa als gescheitert gelten.

Die übliche Reaktion auf den „Requirements Creep“ ist die „Time Compression“, d.h. das Projekt muss nun in kürzerer Zeit fertig sein. Empirische Beobachtungen verschiedenster Projekte führen zu einer Abhängigkeit zwischen dem Aufwand W und der Dauer T in der Form:

$$W \times T^{3.721} \approx const. \tag{8.5}$$

Die Gl. 8.5 hat immense Auswirkungen; eine Verkürzung eines Vierjahresprojektes um sechs Monate führt zu einer Aufwandserhöhung um 65% und eine Halbierung der Zeit auf zwei Jahre führt zu einem über 13-fachen Aufwand. Da der Outsourcingpartner in der Regel ein Gewerk erstellt, hat er keinerlei ökonomisches Interesse an einer Komprimierung der Zeitachse! Folglich verlängert sich das Projekt um einige Zeit.

Erfolgreiche Outsourcingprojekte lassen sich von erfolglosen anhand der bisher betrachteten Größen unterscheiden. Für einen Erfolg von Outsourcing spricht Folgendes:

Tab. 8.5: Die Effizienz der Defektbeseitigung

Aktivität	**Effizienz**
Informelle Design Reviews	25-40 %
Formelle Design Reviews	45-65 %
Informelle Code Reviews	20-35 %
Formelle Code Reviews	45-70 %
Unit Tests	15-50 %
Regressionstests	15-30 %
Integrationstests	25-40 %
Performanztests	20-40 %
Systemtests	25-55 %
Akzeptanztests	25-35 %
Betatest < 10	25-40 %
Betatest > 1000	60-85 %

- weniger als 1% monatlicher „Requirements Creep“
- weniger als 5 Defekte pro Funktionspunkt
- mehr als 65 % der Defekte vor Abnahmetest beseitigt
- mehr als 94 % Defektbeseitigung vor der Installation

Gegen ein erfolgreiches Outsourcingprojekt sprechen:

- mehr als 2% monatlicher „Requirements Creep“
- mehr als 6 Defekte pro Funktionspunkt
- weniger als 35 % der Defekte vor Abnahmetest beseitigt
- weniger als 85% Defektbeseitigung vor der Installation

Speziell beim Outsourcing spielt das Auffinden und die Beseitigung der Defekte eine zentrale Rolle, da im Gegensatz zur Inhouseentwicklung andere Mechanismen vorherrschen. Bei den Inhouseentwicklungen können Softwaresysteme mit mehr Defekten in Betrieb genommen werden als bei ihren Outsourcingkollegen, da die Inhousemannschaft sehr viel kürzere Reaktionszeiten und auch eine höhere Flexibilität gegenüber den Fachbereichen hat, was sich in einem stärkeren Miteinander niederschlägt.

Leider ist die Effizienz von Defektbeseitigung je nach Position im Projektverlauf sehr unterschiedlich, s. Tab. 8.5. Die im Outsourcing übliche Methodik des Akzeptanztests ist, von ihrer Effizienz her, nicht besonders gut, da nur 25-35% der Defekte überhaupt so entdeckt werden.

Alle Defekte, die nach der Abnahme entdeckt wurden, müssen durch das Maintenanceteam beseitigt werden. Hier lässt sich die Faustregel von 8 Defekten pro Mitarbeiter und Monat in der Maintenance anwenden, d.h. eine große Maintenancemannschaft von ca. 15 Mitarbeitern kann etwa 1500 Defekte im Laufe eines Jahres beheben. Unter Berücksichtigung von ca. 5 Defekten pro Funktionspunkt, von denen 65% nach der Übergabe noch vorhanden sind, werden in diesem Szenario, mit seinen 10.000 Funktionspunkten und einem

monatlichen Wachstum von 3,5%, 34 Jahre benötigt, um alle Defekte zu beseitigen. Dies zeigt wiederum, wie immens wichtig Qualitätssicherungs- und Abnahmemaßnahmen beim Outsourcing sind.

Speziell beim Offshoring können kulturelle Differenzen deutlich zu Tage treten. Diese Differenzen sind oft in einer unterschwelligen Erwartungshaltung beider Seiten begründet und können nicht via Kommunikation beseitigt werden, da es sich um „unausgesprochene“ Regelwerke, auch als tacit knowledge[10], bekannt, handelt. So wird beispielsweise in Indien eine strenge Hierarchie gelebt, mit der Folge, dass indische Mitarbeiter Führung regelrecht erwarten, sie aber nicht verbal einfordern können. Westliche Führungskräfte sind es stärker gewohnt, Mitarbeitern Freiräume zu geben, in denen sie sich entfalten können; dies ist in der indischen Kultur ungewöhnlich. Diese Differenz an unausgesprochenen Erwartungen fördert auf Dauer Konflikte innerhalb großer Outsourcingprojekte.

8.3 Insourcing

Dem Problem des Outsourcings und seinen Facetten kann man sich jedoch auch über die gegenteilige Bewegung des Insourcings nähern. Es zeigt sich bei den Unternehmen der Trend, dass es zwar ein selektives Outsourcing oder die entsprechenden Versuche in fast allen Unternehmen gibt, dass jedoch Outsourcing besonders häufig vorkommt, wenn folgende Kriterien zutreffen:

- Das Unternehmen befindet sich in einer schlechten finanziellen Situation.
- Die bestehenden Softwaresysteme sind schlecht.
- Die IT-Abteilung und ihre Entwicklungen haben innerhalb des Unternehmens ein schlechtes Ansehen.

Alle diese Gründe führen, bedingt durch ihre subjektive Wahrnehmung innerhalb des Unternehmens, zu dem Versuch eines Outsourcings. Ein Insourcing lässt sich bei vier, praktisch archetypischen Situationen beobachten:

- IT *darf* Kosten senken – Die Unternehmen werden meistens unter äußerem Druck dazu gezwungen, ihre Kosten zu senken, mit der Folge, dass auch die IT ihre Kosten senken muss. In der Vergangenheit des Unternehmens hat es zwar von Seiten der IT immer wieder Versuche gegeben, die Kosten zu senken, diese wurden jedoch fast immer von den Fachbereichen ausgehebelt. In dieser Situation wird ein Outsourcing versucht. Die eigene IT-Abteilung bietet jedoch auch mit und nutzt die gleichen Techniken wie der zukünftige Outsourcingpartner, um ihre Kosten zu senken, da dies jetzt unter der Drohung eines Outsourcings gegenüber den Fachbereichen möglich ist. So verdreht diese Logik sich anhört, in vielen großen Unternehmen ist sie erfolgreich, sogar so erfolgreich, dass einige der Unternehmen nach gelungenem Insourcing selbst zu Outsourcingprovidern wurden.

[10] Siehe S. 286.

- Abbruch des Outsourcings – In den meisten Fällen ist ein Abbruch des Outsourcings auf massive und andauernde Verletzungen der Service Level Agreements sowie Terminverschiebungen oder einen drastischen Anstieg der Kosten zurückzuführen. Speziell bei Dienstleistungsverträgen mit Outsourcingpartnern ist zu beobachten, dass, wenn diese in eine Krise geraten, keine Time-Kompression durch die Outsourcer eingesetzt wird, bzw. die besten Mitarbeiter der Outsourcingpartner das „sinkende Schiff“ verlassen und in andere Projekte gehen.
- Verteidigung des Insourcings – Oft werden die eigenen IT-Abteilungen mit der Beurteilung von Outsourcing als strategische Denkweise oder in Form eines spezifischen Projektes beauftragt. Die IT-Manager nutzen oft diese Information, um den Wert ihrer eigenen Abteilung zu legitimieren. Ein solches Verhalten führt nicht zu Kosteneinsparungen und wird deshalb meistens zeitversetzt wieder in Frage gestellt.
- Der Wert der IT wird gewürdigt – Unternehmen, welche ihre eigene IT als einen Wert an sich, genauso wie Maschinen oder Know-how, erkannt haben, versuchen den Verlust dieses Wertes zu verhindern. In diesem Sinne ist Outsourcing ein langfristiger Wertverlust. Jedesmal, wenn ein Projekt an einen Outsourcingpartner gegeben wird, verliert das Unternehmen an intellektuellem Kapital, das Kapital, welches seine Mitarbeiter erlernen, wenn sie ein Entwicklungsprojekt erfolgreich beenden. Wenn sich dies wiederholt, wird irgendwann einmal der Outsourcingpartner mehr von dem Geschäft des Unternehmens verstehen als das Unternehmen selbst. Damit werden das Unternehmen und seine Mitarbeiter obsolet.

Es zeigt sich, dass oft die gleiche Argumentation für das Insourcing wie auch das Outsourcing benutzt wird. Aber um es nochmals klarzustellen: Unternehmen, die massiv Outsourcing betreiben, sehen ihre IT als eine „Commodity“ an, wie auch Strom, Wasser, Gas oder eine Kiste mit DIN-Schrauben.

9

Produktlinien

Come, let's return again, and suffice ourselves with
the report of it. Well, Diana, take heed of this
French earl: the honour of a maid is her name; and
no legacy is so rich as honesty.

All's Well that End's Well,
William Shakespeare

Die sehr hohen Kosten für die Entwicklung und die nachfolgende Maintenance von Software haben schon recht früh zur Idee der Wiederverwendung geführt, um damit den Aufwand für die Wiederholungen zu sparen. Über Jahrzehnte herrschte die Ansicht vor, dass Wiederverwendung der Schlüssel zum Erfolg sei. Ausgehend von der möglichen Wiederverwendung von Funktionen, im Sinne von Bibliotheken, in den siebziger Jahren über die Einführung von objektorientierten Sprachen, kam man Ende der achtziger Jahre zur Verwendung von objektorientierten Frameworks und in den Neunzigern zur Idee der komponentenorientierten Programmierung. Alle diese Versuche haben gemeinsam, dass der Ansatz eine Form der generischen unternehmensübergreifenden Wiederverwendung ist. Bis auf wenige Ausnahmen[1] sind die entsprechenden Versuche in der Praxis auch gescheitert. Die versprochenen Effekte der Wiederverwendung wurden nie erreicht. Der Grund für dieses Scheitern liegt darin, dass stets versucht wurde, zwei Dimensionen der Komplexität gleichzeitig zu überwinden: Die Wiederverwendung zwischen den einzelnen Produkten und, simultan, die Wiederverwendung über organisatorische Grenzen hinweg, was nach der Betrachtung von Conway's Law, s. Abschn. 13.14, sehr schwierig ist, da die Software ein Abbild der jeweiligen Organisations- und Kommunikationsstruktur des Unternehmens darstellt. Produktlinien hingegen stellen den Versuch dar, die Komplexität der Wiederverwendung auf eine Dimension, nämlich die des Produktes zu reduzieren.

Viele Unternehmen, die sich mit der Softwareentwicklung befassen bzw. als COTS-Software-Hersteller von dieser Software leben, sind mit der Herausforderung konfrontiert, sich von einer projektbasierten Softwareentwicklung zu einer wissensbasierten hin zu entwickeln. Historisch betrachtet sind die

[1] Ausnahmen für diese Beobachtung sind im Bereich der „Infrastruktur" zu finden, so z.B. GUI-Klassenbibliotheken oder Application Server, aber auch ganze Programmiersprachen wie Java oder .NET.

meisten mittelständischen Softwareunternehmen als Projektorganisation entstanden, da sie projektbezogene Individualsoftware geliefert haben. Ab einer gewissen Anzahl von Implementierungen wird jetzt der Druck, korrekterweise die Last, auf das Unternehmen durch die unterschiedlichen Implementierungen spürbar. Die hohen Aufwände an Maintenance, s. Kap. 7, resultieren in zu wenig freien Ressourcen, um überhaupt in der Lage zu sein, mit einer projektbasierten Vorgehensweise eine neue Implementierung bewerkstelligen zu können. Der Weg vom projektbasierten zum wissensbasierten Vorgehen ist der Versuch, das akkumulierte Wissen in Form der vorhandenen Artefakte wiederzuverwenden. Die hier angesprochenen Artefakte sind nicht nur die „klassischen“ Artefakte, wie beispielsweise Sourcecode, sondern alle Nebenprodukte, welche während der Entwicklung anfallen, von Ideen über die Testfälle, die Designentwürfe bis hin zu den Handbüchern.

Die Produktlinien verfolgen die Idee der strategischen Wiederverwendung von Produkten in einem Marktsegment, entweder innerhalb eines Unternehmens oder in sehr eng verwandten Organisationsformen. Die Idee der Produktlinien ist in der Softwareindustrie auf zunehmendes Interesse gestoßen. Einer der Gründe für dieses Interesse liegt in den beinahe inflationären Kosten für die Entwicklung und Maintenance von Softwaresystemen. Bedingt durch die hohe Startkomplexität und Volatilität ist die Erzeugung neuer Produkte sehr viel teurer geworden. Es liegt nahe, Ideen aus der Industrieproduktion zu benutzen, um ähnliche betriebswirtschaftliche Effekte in der Softwareindustrie zu erzeugen. In der klassischen Industrieproduktion gibt es zwei Basisideen zur Erhöhung der Ausbeute in der Produktion:

- Economy of Scale – Bei der Skalenökonomie verlässt man sich darauf, dass es billiger ist, eine große Anzahl von identischen Gütern zu erzeugen. Bei einer großen Menge sinkt der Stückpreis ab. Der betriebswirtschaftliche Hintergrund für diesen Ansatz liegt darin, dass die Kosten γ pro erzeugtem Produkt

  $$n\gamma = \Gamma_{gesamt}$$

  aus den variablen Kosten $\gamma_{variabel}$ und Fixkosten Γ bestehen:

  $$\Gamma_{gesamt} = \Gamma_{fix} + n\gamma_{variabel}.$$

  Da die Fixkosten sich auf alle produzierten Stücke n gleichmäßig verteilen, sinkt der Einzelproduktionspreis auf

  $$\gamma_{gesamt} = \frac{1}{n}\Gamma_{fix} + \gamma_{variabel}.$$

  Eine solche Betrachtung ist allerdings für die Softwareindustrie trivial, da der variable Anteil sehr klein gegenüber dem Fixteil ist. Hier zeigt sich, dass die Analogie zwischen Softwareentwicklung und Maschinenbau nur endlich weit trägt. Die klassische Forderung in der Produktion im Maschinenbau ist es, Kopien einer Vorlage möglichst billig herzustellen.

Das Kopieren von Software ist jedoch trivial. Dieser Vergleich ist auch der Grund, warum Unternehmen versuchen, ihre einmal erstellte Software an weitere Abnehmer zu verkaufen. Allerdings bricht diese Betrachtung zusammen, wenn die Software zum Verkauf partiell verändert werden muss.

- Economy of Scope – Hinter der Scope-Ökonomie steht die Idee, einen gemeinsamen Kern in verschiedenen Produkten wiederzuverwenden, d.h. bezüglich des Kerns wird eine Form der Skalenökonomie angenommen. Somit ergibt sich für die Gesamtkosten aller Produkte bei einer Scope-Ökonomie:

$$\begin{aligned} \Gamma_{alleProdukte} &= \sum_{i \in Produkt} \gamma_i \\ &= \gamma_{core} + \sum_{i \in Produkt} \Delta\gamma_i. \end{aligned}$$

Hierbei kann dann ein Teil der Gesamtentwicklungs- und Maintenancekosten auf den Bereich der „Core Assets“, der Kernfunktionalität, verlagert werden, welche dann zu jedem einzelnen Produkt einen kleinen Kostenbeitrag leisten:

$$\gamma_i = \Delta\gamma_i + \frac{1}{n}\gamma_{core}.$$

Auf Grund der Scope-Ökonomie suchen Softwarehersteller, welche keine COTS-Software, s. Kap. 10, produzieren, nach Wegen, einen solchen Produktlinienansatz möglich zu machen. Der Ansatz lohnt sich immer dann, wenn es verschiedene Softwareprodukte mit einem gemeinsamen Kern vom gleichen Hersteller gibt.

Die Definition einer Softwareproduktlinie spiegelt dies direkt wider:

> *... a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission ...*

Zielsetzung bei der Einführung eines Produktlinienkonzepts muss es also sein, eine begrenzte und klar definierte Menge an Kernfunktionalität zu identifizieren, welche eine produktübergreifende Gültigkeit besitzt. Dieser Menge der so genannten „Core Assets“ kann man sich auch anders nähern. Wird die Folge der Produkte als eine zeitliche Sequenz betrachtet, so sind die „Core Assets“ die wiederverwendbaren Teile des Ursprungsproduktes, welche sich in allen abgeleiteten Produkten wiederfinden. In der Praxis ist es jedoch nicht so einfach, da das „erste“ Produkt nicht unter dem Konzept der Softwareproduktlinie geschaffen wurde, sondern eher zufällig entstand. Auch weitere Produkte haben im Normalfall nur eine zufällige Wiederverwendung genutzt. Dies hat zur Folge, dass ein Unternehmen sich mit einer Palette von Produkten konfrontiert sieht, welche vermutlich eine gemeinsame Schnittmenge[2],

[2] Dies ist eine Vergröberung; in den meisten Fällen reicht es aus, wenn ein „Core Asset“ nur in einer Teilmenge der Produkte vorkommt, da die Scope-Ökonomie auch für Produktteilmengen funktioniert.

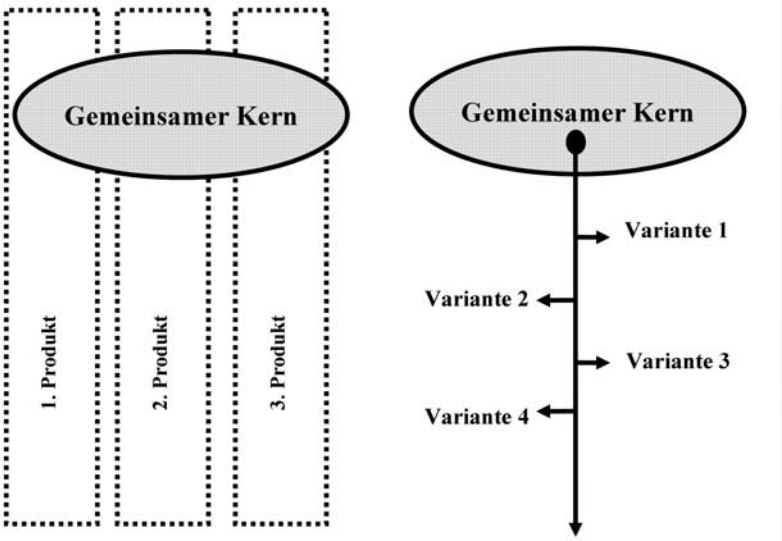

Abb. 9.1: Der gemeinsame Kern und die Varianten in den Produktlinien

die „Core Assets“ besitzen, die aber nicht explizit identifiziert wurde. Den Vorgang, alle „Core Assets“ zu identifizieren, nennt man Assetmining.

Eine andere Methode, die „Core Assets“ zu erhalten, ist es, sie zu kaufen. Ein typisches Beispiel für diese Strategie ist der Einsatz von Klassenbibliotheken in modernen Systemen oder Funktionsbibliotheken in Legacysystemen. Ein weiteres „Core Asset“ ist eine gemeinsame Softwarearchitektur, welche bei verwandten Produkten sehr oft vorhanden ist. Neben solchen Formen der Verwandtschaft bietet sich auch der Kauf von Modulen aus einem COTS-Software-System als „Core Asset“ an.

9.1 Einsatz

Nachdem sich die Wiederverwendung, die ja das primäre Ziel hinter dem Produktlinienansatz ist, auf Sourcecodeebene als nicht besonders effektiv herausgestellt hat – diese Form der Wiederverwendung führt zu massivem Bloating –, wird durch den Produktlinienansatz angestrebt, immer höherwertige sprich abstraktere Artefakte wiederzuverwenden. Im Rahmen einer Model Driven Architecture entspricht dies der Wiederverwendung des Modells.

Die Fragestellung, ob der Ansatz, begründet auf die Scope-Ökonomie, tatsächlich trägt, reduziert sich auf die Frage: Wie gut lässt sich ein „Core Asset“ wiederverwenden und wie hoch ist der Prozentsatz an „Core Assets“?

Um dies zu klären, ist es am günstigsten, die Fragestellung stärker zu präzisieren:

1 Lassen sich die „Core Assets“ direkt wiederverwenden? Welche Kosten entstehen? Wie soll das bewerkstelligt werden? Oder anders formuliert, ist die Wiederverwendung der „Core Assets“ effektiver als völlig neue Artefakte zu entwickeln?
2 Lassen sich die abstrakten „Core Assets“ wiederverwenden? Ein abstraktes „Core Asset“ ist beispielsweise ein Designpattern. Und wenn ja, lassen sich auch die abgeleiteten, d.h. implementierten Funktionen dieser abstrakten „Core Assets“ wiederverwenden?

Eine Wiederverwendung eines abstrakten „Core Assets“ ist nur dann sinnvoll, wenn das neue Produkt dieselben Anforderungen bezüglich der „Core Assets“ hatte wie das Altprodukt, denn Wiederverwendung ist immer komplette Wiederverwendung mit allen abhängigen, d.h. implementierten Teilen. Aus einem anderen Blickwinkel betrachtet, führt die Wiederverwendung des implementierten „Core Assets“ zu einer Wiederverwendung der Anforderung, dies wird oft als „Induced Requirements“ bezeichnet. Falls die induzierten Anforderungen sowieso Bestandteil des neuen Produktes sind, so resultiert die Wiederverwendung des „Core Assets“ in großen Einsparungen.[3]

Die zweite Möglichkeit ist, dass die vom „Core Asset“ produzierte Funktionalität nicht vollständig im neuen Produkt gebraucht wird. Die Folge einer Wiederverwendung wäre die Schaffung eines neuen Produktes mit überflüssiger Funktionalität. Das würde zwar den Produktionsprozess nicht unbedingt verteuern, allerdings fordert die erhöhte Komplexität in Form von mehr Test- und Maintenanceaufwand ihren Tribut. In Einzelfällen mag sich so ein Vorgehen lohnen, im Allgemeinen sollte man in dieser Situation aber von der Wiederverwendung des „Core Assets“ absehen.

Eine dritte Möglichkeit ist es, dass die Anforderungen an das neue Produkt Teilen der Anforderungen an das „Core Asset“ widersprechen. In diesem Fall müsste das „Core Asset“ strukturell verändert und an die neuen Anforderungen angepasst werden. Die Kosten für die Restrukturierung dürften diesen Ansatz sehr teuer machen.

Dieselbe Problematik[4] wie für die abstrakten „Core Assets“ entsteht, wenn ein weniger abstraktes „Core Asset“ wiederverwendet wird, nur dass sich dieses jetzt auf einem weniger abstrakten Niveau befindet. In der Praxis existiert immer eine gemeinsame Untermenge aller Produkte, welche als „Core Assets“ genutzt werden können.

Für den Einsatz einer Produktlinie ist es sehr hilfreich, die Begriffe:

[3] Die Einsparungen liegen jetzt nicht nur bei der Produkterstellung, sondern auch bei der Maintenance, da ein wiederverwendetes „Core Asset“ auch nur einmal gepflegt werden muss.

[4] Mathematisch betrachtet existiert natürlich auch die Nulllösung, d.h. jedes „Core Asset“ wird in jedem Produkt genau einmal implementiert. Diese Lösung führt aber den Produktlinienansatz ad absurdum.

- gemeinsamer Kern,
- Variabilität,

zu unterscheiden, s. Abb. 9.1. Der Sinn hinter dieser Differenzierung liegt in der Tatsache begründet, dass der gemeinsame Kern[5] die Kohäsion der einzelnen Produkte innerhalb der Produktlinie aufzeigt. Der gemeinsame Kern und besonders seine Wiederverwendung sind geradezu die Definition einer Produktlinie. Die explizite Auffindung und Beschreibung dieses gemeinsamen Kerns ist die zentrale Aufgabe bei der Einführung einer Produktlinie. Im Gegensatz zum gemeinsamen Kern werden durch die Variabilität die Unterschiede der einzelnen Produkte innerhalb einer Produktlinie transparent. Bezeichnenderweise sind es gerade die Differenzen, welche oft die entscheidenden Charakteristika der einzelnen Produkte darstellen. Doch in ihrer Verwendung sollten sie weniger wichtig sein als der gemeinsame Kern. Trotzdem muss jede Produktlinie Mechanismen zur Erreichung von Variabilität haben. Die Verfahren zur Erreichung der Variabilität sind eng verwandt mit den Erweiterungsmechanismen in den evolutionären Systemen, s. S. 74. Die Information über den gemeinsamen Kern sowie die Variabilität sind nicht statisch, d.h. auch sie entwickeln sich wie jeder andere Teil einer Software.

Hilfreich für die Implementierung der Produktlinie ist die Einführung von Variationspunkten[6], s. Abb. 9.1. Ein Variationspunkt ist eine gemeinsame Funktionalität der Produktlinie, die unterschiedlich implementiert werden kann oder wird. Der Variationspunkt selbst gehört zu allen Produkten, da er ja im gemeinsamen Kern liegt, aber jede Implementierung ist produktspezifisch. Der Vorteil in der Unterscheidung und der expliziten Nutzung dieses Konzeptes liegt darin, dass:

- der gemeinsame Kern komplett wiederverwendet wird. Diese Form der Wiederverwendung spart sowohl Entwicklungsaufwand als auch Zeit. Die Variationspunkte zeigen ganz genau die Stellen auf, an denen verändert werden muss und kann[7];
- die Formalisierung des Vorgehens in einer besseren Verwendung des Wissens über und innerhalb der Produktlinie resultiert.
- obwohl die Variationspunkte schon früh bekannt sein müssen, ihre Findung und Verwaltung recht kostengünstig ist.

[5] In der englischsprachigen Literatur wird die Summe der Funktionalitäten im gemeinsamen Kern Commonalities genannt.

[6] Innerhalb eines objektorientierten Frameworks werden die Variationspunkte als Hotspots bezeichnet.

[7] Die Java Interfaces sind ein Beispiel für ein technisches Konstrukt zur Unterstützung solcher Variationspunkte.

9.2 Kognitive Effekte

Die Wiederverwendung im Allgemeinen hat sehr spezifische Probleme, welche neben der Technik auch auf den speziellen kognitiven Prozess der Wiederverwendung zurückzuführen sind.

Ausgehend von der Idee eines schon vorhandenen Repository für die Produktlinien, d.h. Informationen sind vorhanden und zugänglich, können drei Formen von kognitiver Wiederverwendung unterschieden werden:

- Gedächtnis
- Assoziation
- Antizipation

Bei der Wiederverwendung durch Gedächtnis steht die Ähnlichkeit der aktuellen Problemstellung mit den Eigenschaften von den Softwareentwicklern bekannten, wiederverwendbaren Teilen im Vordergrund. Da die Beteiligten sich an die Teile erinnern können, werden diese sofort genutzt ohne einen Suchprozess zu durchlaufen. Damit dies aber möglich ist, muss sich der einzelne Softwareentwickler an die Eigenschaften der wiederzuverwendenden Teile erinnern können, d.h. sie müssen ihm bekannt[8] sein. Die Wiederverwendung durch Assoziation beruht, genau wie die Wiederverwendung durch das Gedächtnis, auf den vorhandenen Ähnlichkeiten, allerdings ist hier die Auffindung auf die Fähigkeit, Teile miteinander verknüpfen zu können, ohne sie zu kennen, reduziert. Die dritte Art der Wiederverwendung ist die Wiederverwendung durch die Antizipation. Hierbei weiß der einzelne Softwareentwickler nichts über die möglichen wiederverwendbaren Teile. Allein anhand der Tatsache, dass es möglicherweise solche Teile geben kann, wird die Suche eingeleitet.

Wenn jedoch Wiederverwendung möglich ist, warum ist sie so selten zu beobachten? Dafür gibt es zwei Gründe:

- Menschen, die mit einem Problem konfrontiert sind, versuchen, bekannte Strategien einzusetzen, um dieses Problem zu lösen. Dabei nutzen sie ihre Erfahrung, um eine Problemlösungsstrategie anzugehen, folglich werden einmal erfolgreiche Strategien[9] immer wieder eingesetzt. Für die meisten Softwareentwickler ist das Bauen mit Cut& Paste mit anschließender Veränderung eine erprobte Strategie zur Problemlösung und wird daher immer wieder eingesetzt.
- Bei Entscheidungsprozessen versuchen die meisten Menschen, eine Verlustminimierungs- und keine Gewinnmaximierungsstrategie einzuschlagen. Somit sind Menschen bei Entscheidungen sehr viel stärker für Verluste sen-

[8] Bekanntheitsgrad ist einer der Faktoren für den Softwaredarwinismus, s. Abschn. 13.1.

[9] Eine erfolgreiche Strategie ist beispielsweise, den „Experten“ nach der Lösung zu fragen und damit die Anstrengung, eine andere Strategie zu entwickeln, zu umgehen.

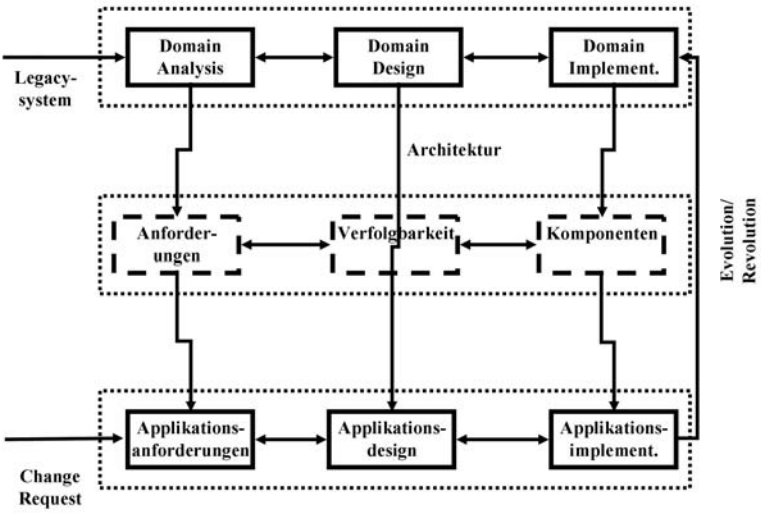

Abb. 9.2: Der Produktlinienprozess in abstrakter Sicht

sibilisiert[10] als für mögliche Gewinne. Aber Wiederverwendung bedeutet sofortiges intensives Nachdenken und Arbeiten bei unsicherem Ausgang, da unklar ist, ob das gefundene Teil tatsächlich wiederverwendet werden kann. Folglich werden mögliche Defizite eines wiederverwendbaren Teiles sehr viel stärker wahrgenommen als die Vorzüge, mit der Folge, dass Wiederverwendung nicht stattfindet.

9.3 Assetmining

Unter dem Begriff des Assetminings wird die systematische Suche nach möglichen Wiederverwendungskandidaten, den Assets, innerhalb der Legacysoftware verstanden, allerdings werden nur die „Core Assets“ tatsächlich wiederverwendet. Oft wird das Assetmining auch als Domänenanalyse bezeichnet. Das Assetmining kann definiert werden als ein Prozess, bei welchem Informationen aus der Entwicklung von Softwaresystemen wohldefiniert gefunden und gesammelt werden, um diese Information wiederverwendbar bei der Erzeugung eines neuen Systems oder der Veränderung eines bestehenden zur Verfügung zu stellen.

[10] Psychologisch gesehen werden Verluste etwa doppelt so stark wahrgenommen wie Gewinne. Verluste erzeugen Reue, Trauer und Schuldgefühle, wenn sie auf eigene Fehler zurückzuführen sind.

Tab. 9.1: Die verschiedenen Analyseansätze

	FODA	**JODA**	**RSEB**
Leitmotiv	Übergreifendes Design	Übergreifende Expertise	Übergreifendes Design
Prozess	steigende Variabilität	steigende Gemeinsamkeiten	steigende Variabilität
technologieabhängig	ja	ja	ja
sprachabhängig	nein	nein	nein
Architektur	produktbasiert	prozessbasiert	produktbasiert
Architekturform	Schichten	nein	Schichten

Insgesamt betrachtet gibt es viele Methoden zur Analyse von Assets, allerdings ist ihre Zahl im Bereich der Produktlinien geringer. Zu den verschiedenen Methoden für das Assetmining bei Produktlinien zählen:

- FODA, Feature-Oriented Domain Analysis
- JODA, Joint Object-Oriented Domain Analysis
- RSEB, Reuse-Driven Software Engineering Business

Alle diese Methoden lassen sich auf das Problemgebiet des Assetminings anwenden. Je nach vorliegender Legacysoftware bzw. Domäne sollte die Entscheidung für den jeweiligen Einsatz getroffen werden, s. Tab. 9.1.

Feature Oriented Domain Analysis, FODA

Diese Analyseform konzentriert sich auf die Identifizierung von Features einer Domäne. Als Methodik basiert FODA sehr stark auf der Structured Analysis und dem Structured Design.

Es werden zunächst die fachlichen Funktionen, unabhängig von der Software, identifiziert. Die Legacysysteme und Produkte, welche in einer solchen Domäne existieren, haben meist unterschiedliche und diverse Funktionalitäten, welche im Kontext von FODA „Capabilities“ genannt werden. Diese Capabilities werden als Assets, innerhalb der FODA-Methodik auch Features genannt, der Legacysoftware angesehen. Damit diese Assets modelliert werden können, sieht FODA einen Prozess mit drei Aktivitäten vor:

- Kontextanalyse – Das Ziel hinter der Kontextanalyse ist es, die Ausdehnung der zu untersuchenden Domäne zu modellieren. Dies geschieht durch die eingesetzten Struktur- und Kontextdiagramme.
- Domänenmodellierung – Die Domänenmodellierung identifiziert die Gemeinsamkeiten und Unterschiede, welche die einzelnen Applikationen innerhalb der Domäne klassifizieren. Unter diese Domänenanalyse fallen:
  - Feature-Analyse – Mit Hilfe der Feature-Analyse werden die Fähigkeiten und fachlichen Funktionen der jeweiligen Applikation beschrieben.

  - Informationsmodellierung – Zur Beschreibung der Entitäten und ihrer Relationen dient die Informationsmodellierung. Auf hoher Ebene entsteht hier ein Entity-Relationship-Modell.
  - Funktionsmodellierung – Die Funktionen, welche von einer Applikation geleistet werden, müssen in einer gut lesbaren Art und Weise dargestellt werden.
- Architekturmodellierung – Die Architekturmodellierung entspricht dem Architekturmining, s. S. 229.

Innerhalb von FODA wird die Domäne durch ein Struktur- und ein Kontextdiagramm definiert. Die gemeinsamen Charakteristika aller Applikationen innerhalb einer Domäne ergeben sich aus der Summe des Struktur- und des Kontextdiagramms. Das Strukturdiagramm zeigt die Pläne und die Ereignisse, während das Kontextdiagramm die Datenflüsse zwischen den verschiedenen Komponenten wiedergibt; erst bei einer gemeinsamen Betrachtung spiegelt sich die gesamte Applikation wider.

In der FODA-Methodik wird vorgeschlagen, die Domäne via Studium der Legacysoftware und dessen Entwicklungshistorie zu modellieren, folglich wird der FODA-Prozess getrieben durch das Streben nach zunehmender Variabilität.

Die Methodik führt zu einer produktbasierten Architektur, da die Domänenarchitektur auf Grund von Komponenten, Interfaces, Szenarios und den Connectoren der verschiedenen Subsysteme entsteht. In der Regel empfiehlt sich für FODA der Aufbau einer Schichtenarchitektur, bestehend aus vier Layers:

- Der Domänenarchitekturlayer – Dieser ist die Menge aller Prozesse und ihrer Wechselwirkung innerhalb des betrachteten Teiles der Gesamtdomäne.
- Ein Modulstrukturlayer – Das Modulstrukturdiagramm zeigt die unterschiedlichen Module auf. Ein Modul in FODA ist ein mehr oder minder willkürliches Paket von:
  - Funktionen
  - fachlichen Eigenschaften
  - Daten

  Diese so gefundenen Module haben meist eine gemeinsame Untermenge, welche in allen Applikationen innerhalb der Domäne enthalten ist.
- Der Common Utilities Layer – Dieser beinhaltet Systemfunktionen, die unabhängig von der jeweiligen Domäne sind. Ein typisches Beispiel für ein Element des Common Utilities Layer ist die Authentisierung oder die Datensicherung. Dass diese oft Bestandteile der Legacysoftware sind, zeigt den Bedarf an Neustrukturierung auf.
- Der Subsystemlayer – Dieser Layer beinhaltet das Betriebssystem, die Programmiersprachen und Ähnliches.

Joint Object-Oriented Domain Analysis, JODA

Hinter der Abkürzung JODA steckt der Ansatz, dass im Rahmen einer Domänenanalyse die Idee von Softwareobjekten sehr viel effektiver ist als eine Analyse, welche auf Funktionen und Unterprogramme abzielt. Gegenüber FODA entspricht dies einem Schwenk von der Structured Analysis zur objektorientierten Analyse. Mit Hilfe des Einsatzes von objektorientierter Methodik wird ein Domänenmodell gebildet und versucht, aus diesem dann Rückschlüsse auf mögliche Assets zu ziehen. Innerhalb dieser Vorgehensweise müssen sich die Assets in den Domänenobjekten widerspiegeln können.

Innerhalb von JODA werden sowohl die Anforderungen der Domäne als auch die Domänenstruktur modelliert. Die resultierenden Objekte stellen die wiederverwendbaren Teile und somit, per definitionem, die Assets dar. Damit eine Domänenmodellierung erreicht werden kann, sieht JODA drei Schritte vor:

- Domänenvorbereitung – In diesem Schritt werden die ersten Interviews durchgeführt und die bestehenden Legacysysteme analysiert. Eine weitere wichtige Tätigkeit ist, zu diesem Zeitpunkt, die Analyse zukünftiger Trends. Diese Fokussierung auf die Zukunft ermöglicht es, jenseits der konkreten Anforderungen für die Domäne zu blicken und eine Produktlinie mit Zukunftsaussichten zu schaffen. Außerdem wird mit diesem Schritt die Reife und Stabilität der bestehenden und zukünftig eingesetzten Technologie bewertet.
- Domänendefinition und Domänenabgrenzung – Dieser Prozess hat diverse Teilaktivitäten, zu denen unter anderem zählen:
  - Schaffung eines Domänenglossars – Erfahrungsgemäß ist eine divergierende Nomenklatur bei den Beteiligten ein sicheres Mittel, jedes Projekt zu gefährden.
  - Erstellung eines Subject-Diagramms – Ein Subject-Diagramm zeigt die Domäne mit ihren Kernentitäten und den Kernprozessen auf.
  - Erstellung einer „Lieferliste“ – Hierin wird das Subject-Diagramm in seine einzelnen Bestandteile zerlegt und deren Beziehungen untereinander werden aufgelistet. Meist handelt es sich um eine hierarchische Zerlegung, aber andere Formen der Zerlegung sind auch denkbar.
  - Vererbung – Vererbungsdiagramme zeigen den Zusammenhang zwischen der Modellierung der Domänenebene und den konkreten Implementierungen in Form der Legacysysteme. Im JODA-Sinne erben die implementierten Softwareobjekte von den fachlichen Domänenobjekten.
- Domänenmodellierung – Die Mittel der Domänenmodellierung innerhalb von JODA sind analog denen der „klassischen“ objektorientierten Analyse:
  - Package-Diagramme, damit die Bestandteile der Legacysysteme bezüglich ihres Deployments zuzuordnen sind.

- Objektmodelle; sie zeigen die Zusammenhänge innerhalb der jeweiligen Domäne auf. An dieser Stelle können auch Klassendiagramme unterstützend eingesetzt werden.
- Wiederverwendbare Businessszenarien
- Lebenszyklusmodelle für die Domänenobjekte
- Zustandsmodelle für die Domänenobjekte

Die JODA-Analyse ist Teil eines größeren Wiederverwendungslebenszyklus. Dieser Zyklus besteht aus:

- Geschäftsplanung
- Planung der Methodik
- Domänenengineering
- Applikationsengineering

Die Kriterien, ob die Domäne vollständig verstanden wurde, sind relativ einfach und münden in der Fragestellung, ob die vorhandene Expertise der Beteiligten zur Bewertung ausreicht. Die JODA-Methodik fokussiert auf die Interviews und das Reengineering. Da in der Methodik sehr große Mengen an Informationen eingesammelt werden, welche anschließend sukzessiv zu verdichten sind, ist das Ziel des Prozesses, zunehmende Gemeinsamkeiten, bzw. Abstraktionen, zu finden bzw. Abstraktionen zu bilden.

Die Verwendung von Szenarios in der Beschreibung der Architektur führt innerhalb von JODA zu einer stark prozessorientierten Domänenarchitektur.

Reuse-Driven Software Engineering Business, RSEB

In diesem von Ivar Jacobsen[11] stammenden Framework ist das Ziel, eine Reihe von Richtlinien und Modellen zu definieren, um hierdurch ein hohes Maß an Wiederverwendung zu erreichen. Das Framework beschäftigt sich mit den Geschäftsprozessen, der Architektur sowie der Organisation des Unternehmens. Die Prozesse werden in drei Kategorien eingeteilt:

- System Components Engineering – verantwortlich für alle anpassbaren veränderlichen Systemteile, welche spezifisch für eine Wiederverwendung gebaut wurden.
- Application Family Engineering – dahinter verbirgt sich die Gesamtarchitektur des Systems. An dieser Stelle wird auch entschieden, welche konkreten Systemkomponenten zur Erfüllung einer Aufgabe genutzt werden.
- Application System Engineering – Die Prozesse in dieser Kategorie erzeugen spezifische Applikationen aus den Systemkomponenten, welche exakt auf die jeweilige Domäne passen.

[11] Einer der drei „Amigos“, welche die UML „erfunden“ haben. Die beiden anderen sind James Rumbaugh und Grady Booch.

Innerhalb des Assetminings der Domäne konzentriert sich RSEB auf die Entwicklung der Domänenarchitektur im Rahmen des Application Family Engineerings und der Entwicklung der wiederverwendbaren Komponenten durch das System Components Engineering. Im Rahmen eines iterativen Prozesses werden verstärkt folgende Aktivitäten durchgeführt:

- Anforderungsaufnahme – Das RSEB-Framework modelliert die Anforderungen in Form von Use Cases. Jede unterschiedliche Art und Weise, mit der ein Benutzer oder ein anderes System die Legacysoftware bedienen kann, ist ein unterschiedlicher Use Case. Die Menge aller Interaktionen mit dem Legacysystem werden durch die Szenarios modelliert.
- Robustheitsanalyse – Bei der Robustheitsanalyse werden Kategorien von Analyseobjekten gesucht, welche dann in disjunkte Subsysteme eingeteilt werden können. Diese Subsysteme ergeben einen Ansatz für die späteren Komponenten im Rahmen von RSEB.
- Entwurf der zukünftigen Architektur

Die RSEB-Methodik definiert in seinem Sprachgebrauch eine Application Family als eine Menge von Applikationen mit verwandten Funktionen. Diese Menge von Applikationen arbeitet zusammen, um dem Endanwender seine Arbeit zu ermöglichen. Im Sinne von RSEB gehört eine angepasste Menge von Applikationen, welche durch ein Customizing oder ein Tailoring produziert werden kann, zur gleichen Applikation Family. Von daher nutzt RSEB das Charakteristikum des gemeinsamen Designs zur Untersuchung der jeweiligen Domäne.

Die RSEB-Methodik kann in ihrem Prozess genauso wie FODA darauf ausgerichtet werden, stets zunehmende Variabilität zu entdecken.

Auch RSEB hat als Ziel eine Schichtenarchitektur in der Domäne zu finden. Allerdings ist in RSEB ein Layer definiert als eine strukturierbare Untermenge, deren einzelne Elemente den gleichen Abstraktionsgrad besitzen. Die oberen Schichten stellen die Applikationen dar, während sich die unteren Schichten dann als sehr generisch erweisen. Dies hat zur Folge, dass in RSEB der Schwerpunkt auf der Beschreibung der Architektur und weniger auf dem Prozess selbst liegt.

9.4 Architekturmining

Einer der Erfolgsfaktoren für den erfolgreichen Einsatz einer Produktlinienstrategie ist die Schaffung der Softwareproduktlinienarchitektur. Diese Architektur reflektiert die allgemeinen und speziellen Teile des Gesamtsystems. An eine solche Produktlinienarchitektur gibt es einige Anforderungen. So sollte sie unbedingt Formen der expliziten Evolution unterstützen, sprich ein offenes System ermöglichen, s. Abschn. 4.14.3. Leider entstehen die Produktlinien nicht a priori im Top-Down-Verfahren mit der Definition einer Produktlinienarchitektur, sondern entstehen eher zufällig, wenn nämlich ein Unternehmen

feststellt, dass es ähnlich gelagerte Systeme produziert, welche eine ganze Weile parallel existieren und evolvieren. Häufig wird dies begleitet von dem Effekt der Softwaremitose, s. Abschn. 4.20. Zum Teil gehen sie auf einen gemeinsamen Kern zurück oder wurden durch eine Art „Cloning“ produziert, aber alle haben eine gewisse Zeit eine Form der Individualevolution[12] erfahren. Nach einiger Zeit wird die Erfahrung gemacht, dass die Maintenance vieler ähnlicher Produkte ökonomisch nicht sinnvoll ist und der Versuch gestartet, eine Produktlinie zu eröffnen.

Ein wichtiger Schritt zur Eröffnung der Produktlinie ist die Findung und Fixierung einer gemeinsamen Architektur. Den so entstehenden Prozess bezeichnet man als Architekturrekonstruktion, er ist eng verwandt mit dem gleichnamigen Prozess im Rahmen von Migrationen, s. Abschn. 5.10.2. Im Gegensatz zu der klassischen Migrationsarchitekturrekonstruktion, welche sehr intensiv die Architektur eines einzelnen Systems untersucht, spielt beim Architekturmining, dem Finden einer Produktlinienarchitektur, der Vergleich der einzelnen gefundenen Architekturen eine Rolle. Genau wie bei den fachlichen Funktionen ist auch die Architektur eines der Assets und unterliegt ähnlichen Betrachtungen.

9.5 Produktlinienwege

Der Einstieg in die Produktlinienstrategie lohnt sich nur für Unternehmen, die entweder langfristig aus ihrer aktuellen Legacysoftware eine COTS-Software, s. Kap. 10, produzieren wollen oder Dienstleister sind und deren Legacysoftware mit leichten Variationen bei unterschiedlichen Kunden eingesetzt wird.

Für beide Formen von Unternehmen stellt sich die Frage: Was ist die beste Methode, um mit ihr zu einer Produktlinienentwicklung zu kommen?

Bevor die konkret vorhandenen Legacysysteme untersucht werden, stellt sich die Frage nach dem Scope. Legacysysteme leben immer in dem Spannungsfeld zwischen Software auf der einen und unterschiedlichen Organisationsformen auf der anderen Seite. Daher sollte der Scope der Produktlinie von Anfang an klar abgegrenzt sein, denn wenn er zu groß oder zu klein gewählt wird, sind Produktlinien nur schwer zu realisieren:

- Wenn der Scope zu klein gewählt wird, entsteht vermutlich kein sinnvolles Produkt, bzw. der Aufwand zur Produktinitialisierung bei den einzelnen Installationen unterscheidet sich praktisch nicht von dem Aufwand bei vollständig parallelen Produkten.
- Wird der Scope zu weit gewählt so nähert man sich sehr schnell den Herausforderungen, denen ein COTS-Software-Hersteller gegenübersteht. Es mag zwar ein Ziel[13] sein, ein COTS-Software-Produkt zu entwickeln,

[12] Die euphemistische Bezeichnung für diese Form der Entwicklung ist: „projektgetriebene Weiterentwicklung“.

[13] Besonders der Vertrieb und das Management eines Softwareherstellers sehen sich gerne als COTS-Software-Unternehmen, welches „Industriestandards“ setzt.

aber dieses aus einem heterogenen Konglomerat von Legacysystemen abzuleiten, ist sehr risikoreich, da die beteiligten Legacysysteme in der Regel keinerlei Standardisierung folgen oder eine solche implizieren.

Neben organisatorischen Änderungen, wie beispielsweise der Einführung eines expliziten Produktmanagements, eines Vertriebs, eines Lizenzierungsschemas etc. sind auch Veränderungen am Entwicklungsprozess selbst notwendig, um in der Lage zu sein, produktlinienkonform zu entwickeln. Bedingt durch die Tatsache, dass es schon eine Legacysoftware gibt, welche bereits erfolgreich im Einsatz ist, kann ein radikaler Schnitt nicht sinnvoll sein. Besser ist es, evolutionär, in kleinen kontrollierbaren Schritten, vorzugehen. Eine evolutionäre Einführung hat folgende Vorteile:

- Die vorhandenen Kunden spüren die Umstellung nicht sofort, bzw. die Umstellung wird von ihnen nicht als ein hohes Risiko in Bezug auf die Servicefähigkeit des Unternehmens gesehen.
- Evolutionäres Vorgehen ermöglicht eine kontinuierliche Fortsetzung der vorhandenen Maintenance und des Supports gegenüber den Kunden.
- Das Entwicklungsteam kann sich langsam, quasi „on the job[14]“, auf die neue Technik einstellen.
- Die möglichen Veränderungen auf die eigenen Produkte werden sofort wirksam und können auch sehr zeitnah ihre Effektivität beweisen.
- Das in der Legacysoftware kodierte Domänenwissen bleibt erhalten und muss nicht komplett neu geschaffen werden.

Die Einführung von Produktlinien durchläuft eine Reihe von Prozessschritten:

1 Identifikation – Zunächst müssen alle möglichen und zugänglichen Features bestimmt werden, welche die natürlichen Kandidaten für die Assets sind. Am Ende der Prozesskette wird dann bewertet, ob es sich tatsächlich um „echte“ Assets handelt. Beim Reengineering, s. Abschn. 5.9, von Legacysoftware wird in den meisten Fällen ein Ansatz via Komponenten gewählt. Nicht so bei den Produktlinien, hier sind die Features das bestimmende Element, da die fachliche Wiederverwendung die Hauptrolle spielt.
2 Priorisierung – Die Zahl der Features und damit der Assetkandidaten ist meist sehr groß, da alle Legacysysteme eine große Mächtigkeit besitzen. Ein evolutionärer Ansatz verlangt, die Zahl der zu verändernden Features drastisch zu reduzieren, um so sinnvolle Pakete schnüren zu können. Neben einem aktiven Produktmanagement[15] ist an dieser Stelle die Beteiligung der Kunden sehr sinnvoll.

[14] *On the Job* bedeutet nicht, dass ein solches Training keine zusätzlichen Aufwände produziert.

[15] Softwareentwickler sind für diese Aufgabe in der Regel ungeeignet, da sie sehr selten eine Kunden- oder Marketingsicht einnehmen können.

3 Einsatzmöglichkeiten – Unabhängig von den Kunden muss das Produktmanagement zukünftige Einsatzgebiete oder Kundenkreise für die gewählten Features beurteilen.
4 Feature-Analyse – Diese Analyse besteht aus folgenden Teilschritten:
   - Featurelokation – Dieser wichtige Schritt versucht, eine Zuordnung des Sourcecodes zu den Features zu finden. Features werden von Seiten des Produktmanagements als Use Cases identifiziert. Eine Möglichkeit ist hier der Einsatz einer Profilingsoftware, um anhand von Testfällen die Wege der Benutzer durch die Legacysoftware zu bestimmen. Auf diese Weise lassen sich größere Codeteile identifizieren, die zu einem Testfall und damit zu einem Use Case und somit zu einem Feature gehören.
   - Featurestruktur und -abhängigkeiten – Im Grunde handelt es sich hierbei um eine Form des Design Recovery. Ziel ist es, durch die Codeteile, welche zu bestimmten Features gehören, abstraktere Blöcke zu finden, welche eine Implementierungsabhängigkeit bzw. -struktur aufzeigen.
5 Aufwandsschätzung – Anhand der Feature-Analyse kann nun der Aufwand für eine Änderung des bestehenden Systems, bzw. für die Einführung von parallelen Produkten, geschätzt werden. Wenn der Aufwand zu hoch ist, darf das entsprechende Feature-Reengineering nicht durchgeführt werden.
6 Reengineering – Hier werden die konkreten Veränderungen durchgeführt, s. Abschn. 5.9. Als Vorgehensmodell eignen sich besonders agile Methoden, s. Abschn. 11.4, unter diesen speziell FDD, s. Abschn. 11.4.4.

Der Einstieg in die Produktlinientechnologie führt zu einer grundlegenden Veränderung der Arbeitsweise des Softwareunternehmens. Unglücklicherweise lohnen sich Produktlinien nicht kurzfristig, sondern nur mittel- und langfristig, da der Return of Investment für Wiederverwendung erst nach einiger Zeit, etwa der dritten bis fünften Wiederverwendung, einsetzt.

Der Aufwandsverlauf zur Erstellung eines Produktes ändert sich drastisch. Traditionell liegt der Schwerpunkt des Aufwandes auf der Implementierung und dem Testen von Software, schematisch dargestellt in Abb. 9.3. Im Fall einer Produktinstanziierung mit Hilfe einer Produktlinie wird die Kurve in den Phasen Implementierung und Test, sogar in der Analyse und dem Design, sehr viel flacher, s. Abb. 9.4, da jetzt ein großer Teil der Vorarbeit wiederverwendet werden kann. Allerdings täuscht Abb. 9.4, denn es muss der zusätzliche Aufwand für das Assetmining berücksichtigt werden. Dieser entsteht zwar nur ein einziges Mal, kann aber sehr viel höher sein als der Aufwand für ein reguläres „tradionelles“ Produkt.

Der Fokus im Rahmen der Softwareentwicklung muss sich von der klassischen Implementierung zur Kenntnis der Domäne hin verschieben, es wird immer mehr Fachwissen innerhalb der Domäne benötigt. Dieser Wechsel im Schwerpunkt ist für die meisten Unternehmen ein drastischer Schritt, welcher psychologisch gut vorbereitet und begleitet werden muss.

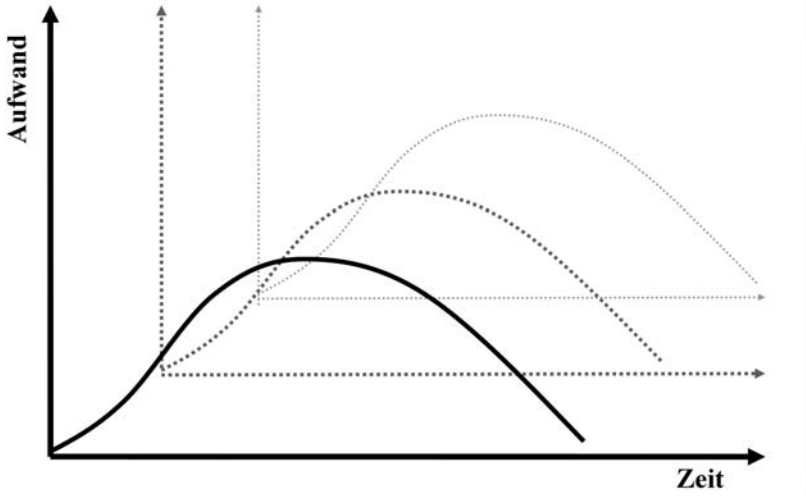

Abb. 9.3: Der „traditionelle“ projektbasierte Aufwandsverlauf

Damit ein Unternehmen die Produktlinienstrategie erfolgreich einsetzen kann, sollte es einer Reihe von „Best Practices“ folgen. Die Vorgehensweisen lassen sich in drei Kategorien einteilen:

- Softwaretechnik
- Methodik
- Organisation

Erst die Beachtung aller Kategorien führt zu einem Erfolg bei den Produktlinien. Dieser Anspruch steht im Gegensatz zu den vielen Versuchen, Produktlinien als ein Mittel der Applikationsentwicklung einzuführen. Solche Versuche beschränken sich dann meist auf die Softwarekategorie und vernachlässigen die beiden anderen, insbesondere die Organisationskategorie. Diese Form des Vorgehens ist ab initio zum Scheitern verurteilt, da die notwendige Wandlung die gesamte Organisation betrifft.

9.5.1 Softwaretechnik

Unter der Kategorie Softwaretechnik werden alle Vorgehensweisen subsumiert, welche notwendig sind, um die Assets, sowie die Produkte einer Produktlinie weiterentwickeln zu können. Die einzelnen Vorgehensweisen im Sinne von „Best Practices“ sind:

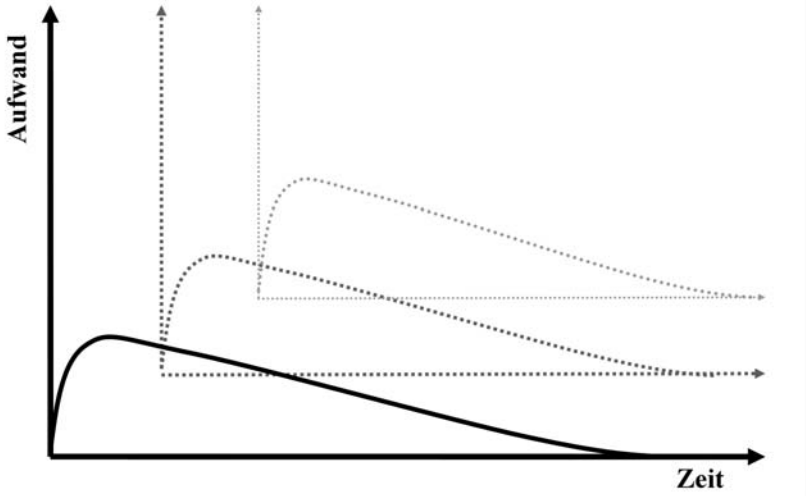

Abb. 9.4: Der produktlinienbasierte Aufwandsverlauf

- Vollständiges Verstehen der relevanten Domäne – Eigentlich sollte dies eine Selbstverständlichkeit sein, leider ist es das in der Praxis nicht! Unter dem Domänenwissen wird die Summe der Konzepte wie auch die Terminologie der Fachleute des jeweiligen Fachgebietes[16] verstanden. Üblicherweise wird das Domänenwissen aus verschiedenen Fachgebieten benötigt, um ein Produkt zu bauen. Da Wiederverwendung am effektivsten ist, wenn ein möglichst abstrakter Teil wiederverwendet werden kann, ist das tiefe Verständnis der Domäne unabdingbar. Der Schritt von der Auffindung von Kandidaten für die Assets hin zur Schaffung einer Menge von wiederverwendbaren Assets verlangt den Weitblick und das notwendige Abstraktionsvermögen, um genau die wiederverwendbaren Assets zu identifizieren. Die Leitlinie kann hier nur sein, die immer wiederkehrenden Probleme und Lösungen in einer Domäne zu identifizieren und diese entsprechend zu dokumentieren. Insofern herrscht hier eine große Verwandtschaft zu den Patterns, s. Kap. 13. Ohne ein Verständnis der Domäne ist keine Wiederverwendung möglich!
- Assetmining, s. Abschn. 9.3.
- Architekturmining, s. Abschn. 9.4.
- COTS-Software, s. Kap. 10 – Eine COTS-Software existiert unabhängig von der Produktlinie, sie ist de facto ein Zukaufprodukt. Wenn im Rahmen einer Produktlinie COTS-Software eingesetzt werden kann, so ist das

[16] d.h. der Domäne

letztendlich die Ausdehnung einer Skalenökonomie, s. S. 218, auf Bereiche jenseits des eigenen Unternehmens. Besonders verbreitet ist der Einsatz von Middleware und Datenbanken als COTS-Software in Produktlinien. In den letzte Jahren sind auch andere Teile, wie Reporting oder Buchhaltung, immer mehr als COTS-Software, in der Entwicklung von Produkten zugekauft worden. Der Trend geht zum Kauf immer größerer Komponenten mit immer mehr Funktionalität. Zum Teil, so beispielsweise bei Buchhaltungssystemen oder Lohnabrechnungssoftware in ERP-Produktsuiten, werden komplette Domänen als COTS-Software eingekauft und in die Produktlinie integriert. Damit dieses Vorgehen aber langfristig sinnvoll ist, muss die Produktlinie eine sehr generische Architektur haben, um diverse COTS-Software-Teile integrieren zu können. Außerdem benötigt die Produktlinie diverse Evolutions- und Integrationsstrategien, um den unterschiedlichsten COTS-Lieferanten gerecht zu werden.

- Softwareintegration – Bei der Softwareintegration werden die verschiedenen Komponenten zu einem Produkt zusammengefasst. Die verschiedenen Formen der Softwareintegration sind primär durch die Architektur der Produktlinie bestimmt.
- Maintenance, s. Kap. 7.

9.5.2 Methodik

Die Best Practices innerhalb der Methoden sind notwendig, um den Entwicklungs- und Maintenanceprozess von Produktlinien zu unterstützen. Die meisten Techniken in dieser Kategorie werden auch von den Systemintegratoren ausgeübt, gleichgültig ob eine Produktlinie im Einsatz ist oder nicht.

- Metriken und Traceability, s. Kap. 2 und Abschn. 7.8 – Ohne eine verständliche Auswahl an Metriken ist ein Fortschritt oder eine Entwicklung im Rahmen von Produktlinien nicht quantifizierbar. In Bezug auf die Metriken und die Traceability müssen diese nicht a priori aus den Artefakten der Softwareentwicklung gewonnen werden. Gerade wenn Legacysysteme involviert sind, bietet sich eine reichhaltige Informationsquelle in den Defektreports, der Änderungsdokumentation sowie der Kundenzufriedenheit.
- Produktlinienscoping – Hierunter wird die Definition der Grenzen der Produktlinie verstanden. Im Rahmen des Scopings wird auch zwischen dem gemeinsamen Kern und der Variabilität unterschieden. Wichtig ist, dass für die Menge der zu liefernden Produkte der gemeinsame Kern groß genug ist, um überhaupt eine Skalenökonomie zu rechtfertigen. Neben diesen mehr nach innen gewandten Aktivitäten gehört zum Produktlinienscoping auch die Tätigkeit des „klassischen“ Produktmanagements: Marktbeobachtung und Wettbewerbsanalysen. Das Produktlinienscoping ist nicht nur auf die Vergangenheit und den aktuellen Status quo ausgerichtet, sondern auch auf zukünftige Entwicklungen des Umfeldes, sowie der jeweiligen Produktlinie und der darin enthaltenen Produkte. Es ist sehr wichtig, eine Idealgröße

für den Scope der Produktlinie zu finden. Wird die Produktlinie zu groß gewählt, entstehen nur hoffnungslose generische Produkte; wird sie zu klein gewählt, versagt die Skalenökonomie.

- Configuration Management – Das Configuration Management ist eigentlich für jedes Entwicklungsvorhaben notwendig, gleichgültig, ob es sich dabei um Produktlinienentwicklung, einfache Produkte oder Projekte handelt. Im Rahmen einer Produktlinie werden alle Produkte durch ein einziges Configuration Management behandelt, im Gegensatz zu „regulären" Produkten, bei denen jedes ein eigenständiges Configuration Management besitzt. Das primäre Ziel des Configuration Managements im Falle der Produktlinien ist es, die Konstruktion einer beliebigen Version eines beliebigen Produktes, hervorgerufen durch die Wiederverwendung der Assets, zu ermöglichen. Eine der wichtigsten Fähigkeiten des Configuration Management ist, neben dem tatsächlichen Bau der einzelnen Produkte, die Fähigkeit zu einer aussagekräftigen Impact-Analyse, s. Abschn. 7.8, zu haben, da jetzt Veränderungen nicht nur ein einzelnes Produkt, sondern eine ganze Reihe von Produkten betreffen.
- Technisches Risikomanagement – Das technische Risikomanagement beschäftigt sich mit der Behandlung von Risiken innerhalb von Projekten. Die meisten Unternehmen haben praktische Erfahrungen im Risikomanagement, oft wird es nicht explizit als solches ausgewiesen, sondern implizit gehandhabt. Im Unterschied zu regulären Entwicklungsprojekten ist es bei der Produktlinienentwicklung notwendig, die Risiken über Projektgrenzen hinweg in den Griff zu bekommen, da die einzelnen Produkte über den gemeinsamen Kern sehr eng aneinander gekoppelt sind. Diese enge Koppelung sorgt für eine gute „Leitfähigkeit" von Risiken, so dass Risiken viel größere Schäden anrichten können.

9.5.3 Organisation

Die organisatorischen Veränderungen bei der Einführung einer Produktlinie sind, ähnlich wie bei der organisatorischen Migration, s. Abschn. 5.2, die am schwierigsten fassbaren.

- Organisationsstruktur – Jede Organisation hat eine Struktur, manche allerdings nur implizit. Aber diese Struktur[17] spiegelt sich in den Verantwortlichkeiten und Rollen aller Beteiligten wider. Eine projektorientierte Organisation verwaltet ihre Produkte auf Projektebene. Zwar haben faktisch alle projektgetriebenen Unternehmen eine Matrixorganisation, d.h. neben der Projektzuordnung des einzelnen Mitarbeiters existiert noch eine zweite Zuordnung im Sinne einer Personal- und Disziplinarverantwortung, aber aus Sicht der Produkte sind die Mitarbeiter jeweils einem oder, in

[17] Bei größeren Unternehmen sind diese Strukturen in den ominösen Organigrammen graphisch abgebildet.

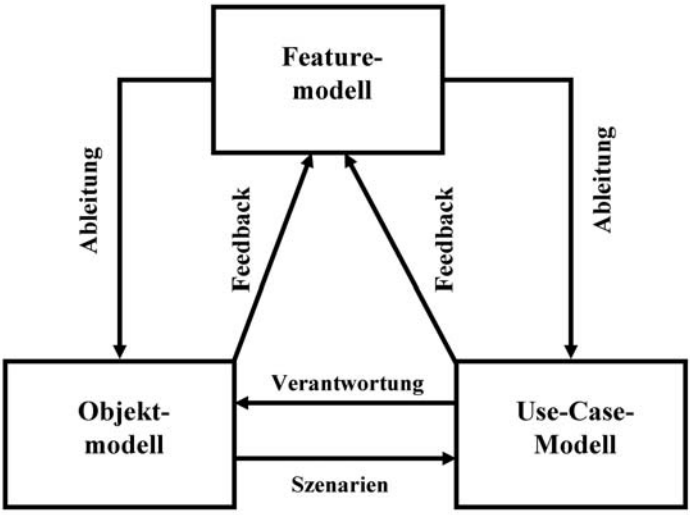

Abb. 9.5: Die drei Modelle in der Produktlinienentwicklung

Ausnahmefällen, mehreren Produkten zugeordnet. In einem Produktlinienansatz kann aber die Entwicklung und Maintenance des gemeinsamen Kerns nicht projektzentrisch stattfinden. Von daher muss das Unternehmen eine andere Organisationsstruktur, zumindest in Teilbereichen, finden.

- Betrieb – Die Wissensvermittlung, wie auch die Regularien für das Auffinden von Assets, sowie ihre jeweilige Nutzung müssen festgelegt werden. Diese Festlegungen beeinflussen drastisch den täglichen Betrieb einer Softwareentwicklungsorganisation.
- Institutionalisierung – Wenn sich die Erfahrungen bei der Einführung von Produktlinien innerhalb der Organisation bewährt haben, müssen alle Aktivitäten und Vorgehensweisen institutionalisiert und damit wiederholbar und formalisiert werden.

9.6 Featuremodell

Die Objektmodelle und Use-Case-Modelle gehören heute mit zum Standardrepertoire innerhalb der Softwareentwicklung. Innerhalb der Produktlinien kommt allerdings das Featuremodell hinzu, s. Abb. 9.5.

Ein Featuremodell beschreibt die Produktlinienfeatures in einer hierarchischen Struktur und ermöglicht es, sowohl die funktionalen Features, als auch

die Qualitätsattribute der Software innerhalb der „Core Assets“ sowie der abgeleiteten Produkte darzustellen. Der Vorteil des Featuremodells ist die einfache Kommunikation mit den Fachbereichen und dem Produktmanagement; da zusätzlich die Gemeinsamkeiten der „Core Assets“ und Unterschiede jedes Produktes aus Sichtweise des Kunden und Endbenutzers dargestellt werden, ist das Featuremodell auch ein ideales Vermittlungsmedium für Diskussionen mit den Kunden.

Auf Grund der Tatsache, dass der Erfolg einer Produktlinie auf dem Verständnis der Domäne beruht, empfiehlt es sich, bei neuen Produktlinien mit dem Featuremodell zu starten. Bei der Migration aus einer Legacysoftware hin zu einer Produktlinie ist dies viel schwieriger. Hier empfiehlt es sich meistens, mit dem Objektmodell und den Use-Cases zu starten, da diese, auch bei der regulären Maintenance unabhängig von einer Produktlinie, schon sehr hilfreich sind. Erst wenn diese beiden Modelltypen stabil sind, sollte der Schritt zum Featuremodell erwogen werden.

9.7 Typische Probleme

Bei der Ein- und Weiterführung von Produktlinien gibt es eine Reihe von typischen Problemen, welche in diesem Umfeld auftauchen. Neben den „klassischen“ Fragen nach Wiederverwendung und was ein „Core Assets“ ist und wie sich nach der Einführung von Produktlinien die Rollenverteilung zwischen Produkt-, Configuration Management und Softwareentwicklung verändert, gibt es auch organisatorische und produktlinienimmanente Phänomene.

Eines der Phänomene ist die Tendenz, nach der Einführung von Produktlinien ein stark erhöhtes Maß an Bürokratie und Regularien zu haben. Der Softwareprozess wird komplexer und verlangt deutlich mehr an Dokumentation. Außerdem müssen nun alle Änderungen begründet werden und können nicht mehr ad hoc vorgenommen werden. Zusätzlich muss eine neue Hierarchie von Führungskräften, Architekten, Eigentümern von Assets, Designern und so weiter geschaffen werden. Diese neue Hierarchie ist verantwortlich für die Erhaltung und Weiterentwicklung der Produktlinienarchitektur und sie muss auch eventuellen Veränderungen von Funktionen explizit zustimmen. Die Durchsetzung dieser Form der Architekturgovernance bedarf eines gewissen Niveaus an Bürokratie, gleichzeitig ist diese Bürokratie ein Risiko für die bisher gelebte Flexibilität[18]. Dieser Tendenz zur Bürokratisierung kann nur durch ein verstärktes Maß an Kommunikation effektiv begegnet werden.

Merklich wird die neue Struktur auch an der Dauer für die Einführung einer Änderung. Die Änderungen an den Funktionen müssen nun einen langen Weg über Gremien nehmen, um bewilligt zu werden. Am längsten brauchen

[18] Gemeint ist eine echte Flexibilität, die Fähigkeit, sich an Veränderungen bewusst und wohlüberlegt anzupassen. Leider wird oft das völlig chaotische Verhalten in Entwicklungsteams als Flexibilität bezeichnet.

Veränderungen, welche die Architektur betreffen, da hierfür die Abstimmungswege aus Sicherheitsgründen am längsten sind. Diese Verlangsamung und zum Teil Ablehnung von Änderungsvorschlägen ist eine notwendige Konsequenz aus der Einführung der entsprechenden Mechanismen für die Produktlinien.

Aus technischer Sicht taucht beim Einsatz von Produktlinien das Phänomen der überdesignten Plattform auf. Da die Basis der Produktplattform eine langfristige Evolution unterstützen muss, liegt die Tendenz nahe, die Architektur der Produktlinie so generisch zu gestalten, dass alle möglichen zukünftigen Veränderungen schon vorweggenommen wurden. Diese ist dann meistens zu generisch für die gewünschten Produkte. Es gibt unter Entwicklungsteams die selbstinduzierte Tendenz, stets die „beste“ Architektur zu suchen, welche alle denkbaren Situationen abhandeln kann. Die so entstehenden Produktlinienarchitekturen sind viel zu komplex, um sie vernünftig instanziieren zu können. Das Geheimnis einer guten Produktlinienarchitektur besteht darin zu wissen, wann sie adäquat genug ist.[19]

Das Ziel hinter dem Produktlinienansatz ist es, die Software innerhalb verschiedener Produkte gemeinsam nutzen zu können. Da aber die Besitzer der „Core Assets“ und die Nutzer dieser Core Assets unterschiedlich sind und meist auch unterschiedlichen Organisationsteilen angehören, entsteht ein Netzwerk von Abhängigkeiten. Diese Abhängigkeiten setzen sich oft kaskadenförmig innerhalb der Organisation fort. Nach Conway's Law, s. S. 385, lässt sich diese Kommunikationsstruktur in der Software wiederfinden. Das Resultat ist eine Form der Spaghetti-Abhängigkeit.

9.8 Evolution von Produktlinien

Nicht nur in der „regulären“ Software, sondern auch innerhalb der Produktlinien gibt es Evolution. Die unterliegenden Assets ändern sich im Laufe der Zeit. Diese Evolution der Produktlinienassets geschieht als Reaktion auf interne Kräfte bezüglich des Unternehmens, welches die Produktlinie entwickelt, und externe Kräfte:

- Ein neuer Standard, sei er technischer oder fachlicher Natur, zwingt die Produktlinienassets, diesen auch langfristig einzuhalten. Normalerweise ist diese Form der Evolution absehbar. Beispiele hierfür sind neue Datenaustauschstandards oder die Euroeinführung.
- Die Einführung einer neuen Technologie ist eine mögliche treibende Kraft hinter der Veränderung von Produktlinien.
- Eine Veränderung der Marktstrategie kann eine innere Kraft hinter der Evolution der Software sein. Meist handelt es sich dabei um die Hinzunahme neuer Funktionen oder die Veränderung des Look&Feel.

[19] Das Prinzip des „gerade gut genug“ wird auch bei den agilen Verfahren intensiv eingesetzt.

Die Evolution in einer Produktlinie kann allerdings an zwei unterschiedlichen Stellen stattfinden, zum einen bei den Core Assets, dem Kern der Produktlinie, und zum anderen bei den einzelnen Produkten. Außerdem müssen bestimmte Abhängigkeiten zwischen den Core Assets untereinander und den Core Assets und den Produkten beachtet werden, da ein Auseinandertreiben den Kern gefährden kann. Widersprechende Zielsetzungen können zu einer schnellen Erosion der Konsistenz innerhalb der Core Assets führen und damit die gesamte Produktlinie inklusive aller Produkte gefährden.

Nicht jede Veränderung einer Produktlinie ist evolutionär. Im Kontext von Produktlinien werden Veränderungen einzelner Produkte, welche sich nicht bei den Core Assets widerspiegeln, als phänotypische Änderungen[20] bezeichnet. Aber Evolution ist insgesamt betrachtet eine größere Bedrohung für die Core Assets als für eine Reihe von Objekten, die nur lose gekoppelt sind, da die Abhängigkeiten von den Core Assets sehr viel größer sind.

Die äußeren Kräfte für die Evolution sind analog den Kräften in jedem Geschäftsstrategiemodell:

- Neue Konkurrenten oder auch neue Produkte können Veränderungen in einer Produktlinie, bedingt durch den entstehenden Anpassungsdruck, erzwingen.
- Bestehende Konkurrenten können durch ihre Preis- oder auch Produktmanagementpolitik Unruhe in einen stabilen Markt bringen. Manchmal erzwingen neue De-facto-Standards auch Veränderungen der Produktlinie.
- Ersatz eines Lieferanten, bzw. Wegfallen eines Produktes, führt zu Veränderungen der Produktlinie, wenn dieses Produkt dort integriert oder zur Herstellung benutzt wurde.
- Die Käufer können durch ihr Verhalten eine Änderung erzwingen. Es gibt auch in der IT Modeströmungen, so ist zum Beispiel der Einsatz von Java Applets in Web-Anwendungen weitgehend „ausgestorben“.

In der Entwicklung verläuft die Veränderung in der Regel von den Produktentwicklern zu den Core-Asset-Softwareentwicklern, welche bestimmte Veränderungen in den Kern aufnehmen.

[20] Aus Sicht des einzelnen Produktes ist diese Veränderung natürlich eine Evolution.

10

COTS

...
More than my father's skill, which was the greatest
Of his profession, that his good receipt
Shall for my legacy be sanctified
By the luckiest stars in heaven:
and, would your honour
But give me leave to try success, I'ld venture
The well-lost life of mine on his grace's cure
By such a day and hour.

All's Well That Ends Well,
William Shakespeare

In der englischsprachigen Literatur wird die kommerzielle Software als COTS-Software, **C**ommercial **O**ff **T**he **S**helf Software, bezeichnet. Hierunter wird nicht nur die klassische Lizenzsoftware, wie beispielsweise Buchhaltungssysteme, sondern auch Software, welche stärker dem Infrastrukturbereich, wie z.B. ein Datawarehouse oder ein System zur Überwachung der Netzwerke, zuzuordnen ist, verstanden. Für die Unternehmen ist die Betrachtung dieser Software immanent wichtig. da mittlerweile eine hoher Prozentsatz des IT-Portfolios eines Unternehmens durch COTS-Software abgedeckt wird.

In Bezug auf Legacysysteme verdienen die COTS-Software-Systeme genauere Betrachtung, da es zum einen immer wieder die Tendenz gibt, ein Legacysystem oder Teile einer Legacysoftware durch COTS-Software zu ersetzen; zum anderen können COTS-Software-Systeme auch zu Legacysystemen werden. Außerdem sollte berücksichtigt werden, dass für ein Legacysystem, welches verschiedenste COTS-Software-Produkte beinhaltet, eine andere Vorgehensweise bei der Evolution und Maintenance notwendig ist als bei einem vollständigen Individualsystem.

Die COTS-Software wird üblicherweise definiert als eine Software, die den folgenden fünf Kriterien genügt:

1 Es handelt sich um ein kommerziell verfügbares Softwareprodukt, d.h. es kann gekauft, geleast oder lizenziert werden.
2 Der Sourcecode steht nicht zur Verfügung, aber die Dokumentation ist ein Bestandteil des COTS-Software-Produktes.

3 Es gibt mehr oder minder periodische Releases mit neuen Funktionalitäten oder technischen Veränderungen.
4 Die COTS-Software ist nutzbar, ohne dass Interna der Software verändert werden müssen.
5 Die konkret vorliegende COTS-Software existiert in mehreren identischen Kopien auf dem Markt.

Die Software aus dem Open-Source-Bereich ist nach diesem Kriterienkatalog keine reine COTS-Software, für die nachfolgende Diskussion ist dies jedoch ohne größeren Belang, da die Open-Source-Community als Gemeinschaft wie ein Lieferant reagiert. Auch Produkte, welche nur von bestimmten Organisationen erworben werden können und auf dem freien Markt nicht verfügbar sind, wie beispielsweise JBF, das Java Banking Framework, fallen unter den Begriff der COTS-Software im erweiterten Sinne.

Es gibt allerdings einen merklichen Unterschied zwischen Open-Source-Software und der Software von kommerziellen Anbietern: Erfolgreiche kommerzielle Software leidet unter einem *„feature bloating“*. Es werden zuviele Funktionalitäten in die Software eingebracht, obwohl der Endanwender diese überhaupt nicht braucht.[1] Dieser Überfluss an Funktionalität soll beim Endanwender Abhängigkeit gegenüber dem Hersteller erzeugen. Die Hersteller von Open-Source-Software verzichten meist bewusst auf unnötige Funktionalität.

Obwohl die Unveränderbarkeit der Interna ein Definitionsmerkmal für COTS-Software ist, sollte beachtet werden, dass der Unterschied zwischen Anpassung, auch Tailoring genannt, und der Veränderung der Interna oft fließend ist. Unter Anpassung versteht man die Veränderung der Software in der Art und Weise, wie der Hersteller der COTS-Software diese Veränderungen vorgesehen hat; dies funktioniert analog zu den evolutionären Systemen, s. Kap. 4. Es gibt aber eine Reihe von COTS-Software-Produkten, bei denen der Hersteller nur die Kernkomponenten gegen Veränderungen schützt und zusätzliche Teile für die Veränderung, auch im Sourcecode, zulässt. Im Allgemeinen ist der Grad der Abweichung von der ursprünglichen Form ein gutes Maß für den Aufwand, der bei jedem neuen Release der COTS-Software zu leisten ist.

Die Einsatzgebiete von COTS-Software sind divers, sie rangieren von einem einzelnen COTS-Software-System bis hin zum Aufbau eines Gesamtsystems aus reinen COTS-Software-Teilen. Traditionell werden die COTS-Software-Produkte unterteilt in eine System- und eine Applikationssoftware, s. Abb. 10.1. Der applikative Teil wird zusätzlich noch in horizontale und vertikale Applikationen unterschieden. Horizontale Applikationen sind Software, welche branchenübergreifend eingesetzt werden kann, wie z.B. eine Finanzbuchhaltung oder eine Textverarbeitung. Unter der vertikalen Software wird die branchenspezifische Software verstanden.

[1] Bekanntes Beispiel hierfür ist das starke Größen- und Featurewachstum von Microsoft Word. Die von einem durchschnittlichen Endanwender genutzte Funktionalität macht nur einen Bruchteil der in Word enthaltenen aus.

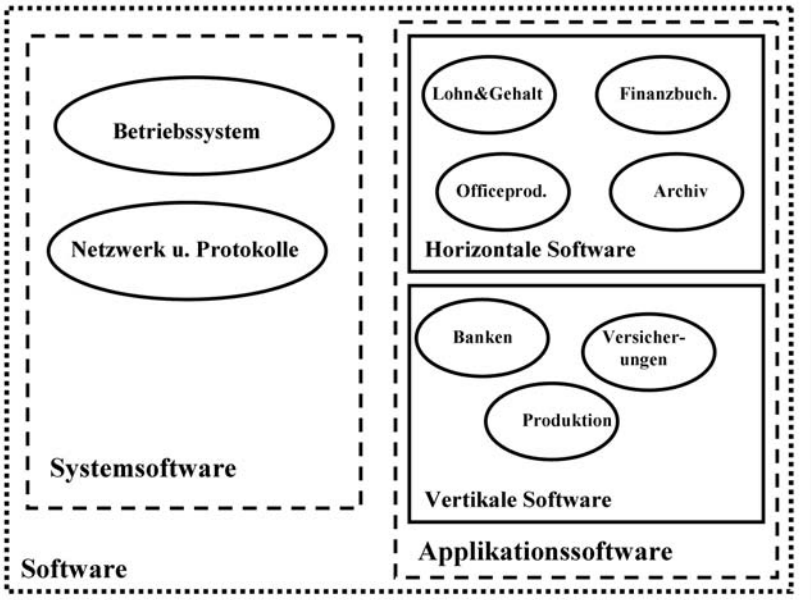

Abb. 10.1: Die typische COTS-Software-Landschaft

In Bezug auf ein Legacysystem stellt sich die Frage nach dem vollständigen oder teilweisen Ersatz von Teilen der Legacysoftware durch eine käuflich erworbene COTS-Software. Unter einer Produktsuite oder auch einem Lösungssystem wird eine Kollektion von COTS-Software, die von einem Lieferanten gebündelt verkauft wird, verstanden. Das heißt nicht, dass dieser COTS-Software-Lieferant seine verkaufte Produktsuite auch vollständig selbst produziert hat. Auch in der Welt der COTS-Software-Hersteller wird stark arbeitsteilig vorgegangen und ein Teil der COTS-Software wird zugekauft. Besonders Klassenbibliotheken, Reportingsysteme und Datenbankverbindungssoftware sind beliebte Zukaufprodukte.

Für den Einsatz dieser Produktsuiten ist ein signifikantes Maß an Anpassung sowohl der COTS-Software, als auch der Organisation des erwerbenden Unternehmens für einen Erfolg unabdingbar. Am meisten verbreitet ist COTS-Software in den Bereichen der Buchhaltung, Lohnabrechnung, Produktionsplanung und -steuerung, sowie Personalverwaltung. Bekannteste Lieferanten sind hier SAP, PeopleSoft, Oracle oder Microsoft.

Zu den üblichen Vorurteilen in Bezug auf die COTS-Software zählen:

- COTS-Software spart Geld.
- Da es COTS-Software ist, muss die Software nicht getestet werden.
- Bei COTS-Software muss man sich nicht um die Architektur kümmern.
- COTS-Software braucht keine Maintenance.

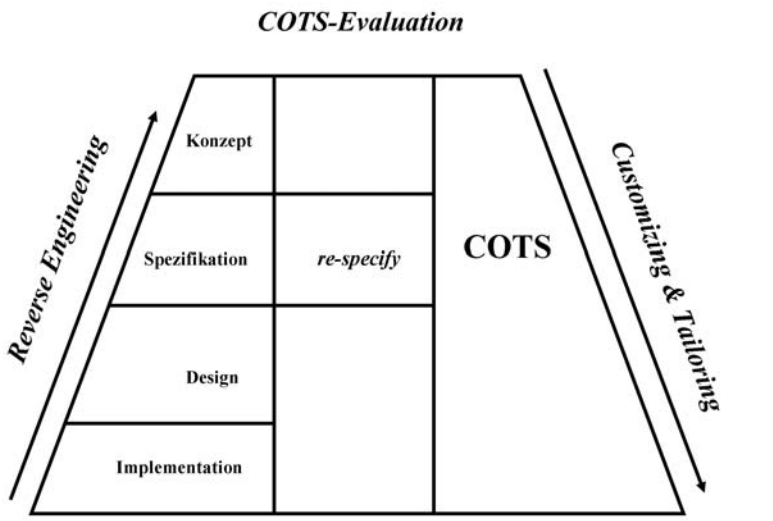

Abb. 10.2: Reengineeringpyramide mit einem COTS-Ersatz, s. Abb. 5.9

Nicht immer ist eine COTS-Software die beste Wahl, vor allen Dingen, wenn eine solche Software eingesetzt wird, um die Kernprozesse des Unternehmens zu unterstützen. In diesem Fall kann die normative Kraft der COTS-Software dazu führen, dass das Differenzierungsmerkmal des Unternehmens gegenüber Konkurrenten am Markt verloren geht. Der Verlust des Differenzierungsmerkmals kann für ein Unternehmen fatale Folgen haben, da es mit diesem Verlust auch einen Vorsprung gegenüber Konkurrenten am Markt verlieren kann.

10.1 Teilersatz

Für Unternehmen, die Legacysysteme einsetzen, stellt sich immer mehr die Frage: Können einzelne Teile oder das gesamte Legacysystem durch eine COTS-Software ersetzt werden? Häufig ist hierbei gar nicht so sehr die Frage nach einem kompletten Ersatz des gesamten Legacysystems gestellt, da ein solches Unterfangen seine eigenen Risiken und Problematiken hat, s. Abschn. 10.2, sondern welche Kosten in Betracht gezogen werden müssen, wenn Teile der Legacysoftware durch eine COTS-Software ausgetauscht werden.

Aus Sicht der Migration, s. Abb. 5.9, ist der Teilersatz, wie auch der Komplettersatz durch COTS-Software eine spezielle Form des Reengineering, s. Abb. 10.2. Obwohl sich in der Theorie eine COTS-Software genauso einfach oder genauso schwierig in ein Legacysystem integrieren lassen muss wie ein

selbsterstelltes Produkt, zeigt die Praxis, dass es nicht so einfach ist. Es sind eine Reihe von Aktivitäten durchzuführen, welche ihre eigenen Gesetzmäßigkeiten haben:

1 Anforderungsanalyse
2 Evaluation und Selektion der COTS-Software
3 Kauf der COTS-Software
4 Adaption der COTS-Software
5 Adaption des restlichen Legacysystems
6 Design, Implementation und Test des Glue-Codes
7 Systemintegration
8 Lizenzverwaltung, COTS-Software-Produkt-Feedback
9 Evaluation und Einbau der COTS-Software-Upgrades
10 Fehlerbeseitigung

Manche dieser Aktivitäten sind relativ klar und einfach umzusetzen, andere jedoch besitzen ein hohes Maß an Risiko und Aufwand.

Anforderungsanalyse

Die Anforderungsanalyse ist eine Aktivität, die auf alle Fälle, unabhängig vom möglichen Einsatz einer COTS-Software oder dem Bau von Individualsoftware, gemacht werden sollte. Die „Make-or-Buy“ Entscheidung[2] sollte immer ein Bestandteil der Anforderungsanalyse sein. Sich die passende COTS-Software auszusuchen, ohne eine explizite und intensive Anforderungsanalyse durchzuführen, ist ein Ding der Unmöglichkeit.

Erstaunlicherweise geschieht dies trotzdem! Die große Menge an Shelfware[3] beweist ganz augenscheinlich, wie häufig COTS-Software erworben und dann nie eingesetzt wird. Es lassen sich für dieses Phänomen drei Gründe ausmachen:

- Mangelnde Endbenutzerbeteiligung – Häufig sind in den Unternehmen die IT-Abteilungen oder ein Zentraleinkauf Entscheidungsträger darüber, welche Software erworben wird. Beide Organisationsbereiche tendieren dazu, recht selbstherrlich zu glauben, dass sie besser als der jeweilige Fachbereich wüssten, was dieser an Software braucht. Der Grund für dieses Gedankengut und die nachfolgende, oft falsche, Kaufentscheidung ist unterschiedlich:

[2] Die „Make-or-Buy“-Entscheidung offenbart eine gewisse Polarität in den Unternehmen, da die Softwareentwicklungsabteilungen generell zu einem „make“ und die Fachabteilungen zu einem „buy“ tendieren.

[3] Shelfware ist Software, welche gekauft, aber nie eingesetzt wurde. Sie bleibt auf dem Regal stehen, daher der Name Shelfware. In gewisser Weise ist Shelfware der Traum jedes COTS-Softwareherstellers, da der Kunde Lizenzen bezahlt, aber das Produkt nie einsetzt. Der Nichteinsatz führt zu einer hohen Stabilität des Produktes, da dies jetzt keinen Änderungen unterliegt.

  - Zentraleinkauf – Für einen Zentraleinkauf sind alle Güter, auch die Software, die er einkauft, austauschbar. Zentrale Einkaufsabteilungen reagieren primär auf die Kostenfrage. Auf Grund der Idee der Austauschbarkeit wird dann der im Einkaufspreis günstigste Anbieter genommen. Austauschbarkeit ist zwar im Bereich von DIN-genormten Schrauben, Nägeln und Stahlträgern gegeben, nicht jedoch bei höchst komplexen Produkten wie Software oder speziellen Dienstleistungen. Mangelnde Fachkenntnisse auf Seiten des Zentraleinkaufs verschärfen noch die Differenz zwischen der Kaufentscheidung und dem eigentlich benötigten Produkt, mit der Folge, dass häufig Shelfware oder die falsche Dienstleistung gekauft wird. Ironischerweise ist dies langfristig gesehen sogar sehr viel teurer!
  - IT-Abteilung – Die IT-Abteilungen in Unternehmen sind primär technisch orientiert und entscheiden daher auch nach den entsprechenden Gesichtspunkten. Folglich werden COTS-Software-Produkte primär aus technischen und erst sekundär aus vermeintlich fachlichen Gesichtspunkten betrachtet. Neben der mangelnden Fachlichkeit neigen die sehr technisch orientierten IT-Abteilungen dazu, relativ komplexe Produkte zu erwerben. Bei einem komplexen Produkt erhält der Käufer, subjektiv betrachtet, sehr viel mehr für sein Geld als bei einem weniger komplexen Produkt. Leider ist diese subjektive Einschätzung betriebswirtschaftlich falsch, da in den meisten Fällen ein Zuviel an Funktionalität zur Verwirrung der Endbenutzer beiträgt und diese dann die verwirrenden Produkte nicht weiter einsetzen.[4] Außerdem wird Software als die Domäne der IT-Abteilung empfunden, daher betrachten sich die meisten IT-Abteilungen nicht als Dienstleister der Fachbereiche, sondern als Selbstzweck für Kauf, Programmierung und Betrieb von Software. Insofern sind Kaufentscheidungen von IT-Abteilungen auch, bis zu einem gewissen Grad, eine Machtdemonstration.
- „Vendor-Lock-In“ – Viele heutige Softwareanbieter haben eine sehr breit aufgestellte Produktpalette. Oft besitzen die Verkäufer der Softwareanbieter gute Kontakte zum Management des kaufenden Unternehmens mit der Folge, dass Kaufentscheidungen stark über das tatsächliche oder auch vermeintliche Beziehungsgeflecht des Managements motiviert sind. Solche Beziehungsgeflechte[5] können so stark sein, dass Anforderungsanalysen komplett ignoriert werden. Bei der öffentlichen Hand, welche einen Teil ihrer COTS-Software-Einkäufe ausschreiben muss, funktionieren diese Netzwer-

[4] Eine weit verbreitete Differenz ist, dass Fachbereiche auf Grund der höheren Eingabegeschwindigkeiten einer Tastatur, bei erfahrenen Benutzern, Systeme bevorzugen, welche sich vollständig über die Tastatur steuern lassen, während IT-Abteilungen als unerfahrene Benutzer stets die Maus verwenden.

[5] Auch „old boys' network“ genannt.

ke etwas subtiler. Hier werden manche Ausschreibungen so formuliert, dass nur der gewünschte Anbieter[6] sie auch erfüllen kann.

- Softwarebudgetierung – In großen Unternehmen wird für den Erwerb von Software ein Budget zur Verfügung gestellt. Wird dieses nicht innerhalb eines vordefinierten Zeitraumes aufgebraucht, so verfällt es. Meistens ist dieses Budget zweckgebunden und kann nicht umgewidmet werden, mit der Folge, dass zum Ende des Budgetzeitraumes dieses, subjektiv gesehen, aufgebraucht werden muss, um COTS-Software zu erwerben. Die Nichtausschöpfung eines Budgets[7] bedeutet in den meisten Fällen, dass man dieses Geld anscheinend nicht braucht und daher das Folgebudget gekürzt werden kann. Als Konsequenz dieser Denkweise wird jede Menge von nutzloser Software erworben.

Unabhängig von diesen Problemfällen muss die Anforderungsanalyse in derselben Art und Weise durchgeführt werden wie bei einem großen Individualsoftwareprojekt. Unternehmen mit einem hohen Legacysoftwareanteil tendieren dazu, Forderungen zu formulieren, die im Endeffekt bedeuten, dass sie eine identische Kopie des Legacysystems haben möchten. Eine solche Forderung kann jedoch nicht erfüllt werden, da die individuellen Merkmale der Legacysoftware nur schwer reproduzierbar sind. Die Anforderungsanalyse muss daher auch weiter gehen und einen Teil der Implementierung der Geschäftsprozesse, welche für die spezifische Ausprägung der Legacysoftware verantwortlich waren, in Frage stellen. Man sollte sich der Anforderungsanalyse nähern, ohne von Anfang an eine COTS-Software-Zensur im Kopf zu haben!

Evaluation

Es ist üblich, sich während der Konzeption oder der frühen Anforderungsanalyse die Frage nach der Möglichkeit eines COTS-Software-Einsatzes zu stellen. Hierbei ist es wichtig herauszufinden, ob es eine solche Software wirklich gibt und ob diese die gewünschte Funktionalität haben könnte. Wenn diese Verfügbarkeit geklärt ist, muss die Evaluation beginnen. Es gibt diverse Evaluationskriterien und -gründe, als Basismenge sollte man aber immer folgende Untersuchungen durchführen:

- Stimmt die vom Hersteller versprochene Funktionalität mit der tatsächlichen Funktionalität überein? Die meisten Vertriebsunterlagen der Hersteller zeigen die jeweiligen Produkte immer in einem sehr günstigen Licht. Unabhängige Aussagen zu bekommen ist extrem schwer, da klassische Quellen, wie Consultingunternehmen, s. S. 252, oder andere Unternehmen, oft

[6] Es sind Fälle bekannt, bei denen der gewünschte Anbieter die Formulierung der Ausschreibung sogar „hilfreich“ unterstützt hat.

[7] Der Vertrieb der Softwareanbieter ist in dieser Situation unterstützend tätig. Es werden problemlos Rechnungen und Lieferscheine ausgestellt, obwohl in dem entsprechenden Jahr die Software noch nicht zur Verfügung steht.

sehr subjektiv geprägte Aussagen treffen. Eventuelle Nachfragen bei anderen Unternehmen, welche das entsprechende Produkt einsetzen, können zu falschen Interpretationen führen:
  - Die Prozesse und die Organisation des anderen Unternehmens sind unterschiedlich. Da Software nur im Kontext der Prozesse und des jeweiligen Unternehmens beurteilt werden kann, ist hier die Vergleichbarkeit fragwürdig.
  - Je kapitalintensiver die COTS-Software für das Unternehmen war, desto stärker ist der Rechtfertigungszwang es überhaupt erworben zu haben. Dies hat zur Folge, dass, speziell vom IT-Management, die sehr teuer erworbenen COTS-Software-Produkte gelobt werden, obwohl diese überhaupt nicht adäquat sind. Teure Produkte gelten oft auf Grund ihres hohen Preises als exklusiv und gut, bzw. das Management gesteht nicht gerne Fehlentscheidungen ein, welche viel Geld kosteten.
- Werden alle nichtfunktionalen Anforderungen getroffen? So beispielsweise:
  - Performanz, speziell auch bei großen Datenmengen[8]
  - Betriebssystemplattformen
  - Hardwareplattformen
  - Datenbanken
  - Betriebsverfahren, speziell Datensicherung
  - Sicherheitskonzept, speziell Authentisierung und Autorisierung
  - Stabilität
- Stimmt die Funktionalität mit der geforderten Funktionalität überein?
- Ist die COTS-Software kompatibel mit der restlichen Landschaft? Hier haben sich unterschiedliche Treiberversionen als sehr tückisch herausgestellt. Besonders komplex wird dieses Treiberproblem, wenn ein einzelnes Datenbanksystem verwendet wird, aber die unterschiedlichsten COTS-Software-Produkte innerhalb des Unternehmens unterschiedliche Versionen des Datenbanktreibers benötigen; oft sind diese, obwohl vom gleichen Hersteller, nicht kompatibel. Eine ähnliche Situation zeichnet sich mittlerweile im Umfeld von COTS-Software ab, welche eine Java Virtuelle Maschine benötigt, nicht selten sind 2-3 Versionen des JREs auf einem Rechner in größeren Unternehmen vorzufinden.
- Ist es überhaupt ökonomisch sinnvoll? Für die Kosten sind allerdings nicht nur die Lizenzkosten, sondern alle anfallenden Kosten während des Einsatzzyklusses der Software entscheidend, speziell die Adaptionskosten sind im Vergleich zu den Lizenzkosten teilweise exorbitant hoch. Für den Einsatz der COTS-Software sind aber nicht nur die Lizenz- und Adaptionskosten bei der Einführung ein Faktor, sondern auch die Frage nach den dauernden Unterhaltungs- bzw. den Folgekosten. Jeder Releasewechsel auf Seiten des Herstellers produziert Folgekosten, diese können je nach Grad

[8] Beispieldaten sind notorisch kleine Bestände, bei denen die Skalierungseffekte von Datenmengen keine Rolle spielen.

des Tailorings oder nach Infrastrukturlastigkeit[9] der COTS-Software immens sein.

Es ist nicht empfehlenswert, die zu evaluierende Software vollständig isoliert zu begutachten, quasi im „luftleeren Raum“. Aussagen, welche so gewonnen werden, sind meistens nicht auf eine echte produktive Umgebung anwendbar. Im Gegensatz dazu ist eine Evaluation in einer vollständigen Produktionsumgebung nicht sinnvoll, da hier das Risiko eines Ausfalls der Produktionsumgebung viel zu hoch ist. Es empfiehlt sich daher, einen Mittelweg zu wählen und die zu evaluierende COTS-Software mehrfach parallel in einzelnen Testumgebungen zu prüfen. Diese Testumgebungen simulieren jeweils einen oder mehrere Aspekte der tatsächlichen Produktionsumgebung. Hierdurch kann das Risiko beträchtlich reduziert werden. Typische Fehler bei der Evaluation der COTS-Software sind:

- Einmalige Evaluation – Oft wird in den Glauben verfallen, es würde ausreichen, die COTS-Software einmal zu evaluieren, und anschließend sei diese COTS-Software unveränderlich stabil. Dies ist leider nicht so, s. Abschn. 10.3. Jeder neue Release muss erneut vollständig evaluiert werden, da nicht von einem langfristig stabilen Verhalten ausgegangen werden kann. Auch die COTS-Software unterliegt einer häufigen, ja beinahe kontinuierlichen Veränderung[10].
- Best of Breed – Dieser Fehler taucht meist bei Softwaresystemen auf, welche durch Integration von COTS-Software zu einem Gesamtsystem entstehen. Nicht immer bildet die Integration von den „Besten“ ein wirklich gutes Gesamtsystem[11], viel entscheidender sind hierbei kompatible Architekturen und verwandte Vorgehensweisen.
- IT-Lastigkeit – Einzig die IT-Abteilung mit der Auswahl der COTS-Software zu beauftragen, führt zu einer Entscheidung, welche nicht die spezifischen fachlichen Anforderungen der Endanwender berücksichtigt. Auf diese Weise wird oft eine fachlich nicht adäquate COTS-Software erworben, die anschließend zum Fundus der Shelfware beiträgt.
- Marketing – Speziell bei großen Softwarepaketen wird die Auswahl des COTS-Lieferanten sehr viel stärker durch seinen Bekanntheitsgrad bzw. durch das Beziehungsgeflecht des Lieferanten bestimmt, als durch die tatsächliche Anwendbarkeit der COTS-Software.

Die größte Schwierigkeit bei der Evaluation ist die Tatsache, dass sich die individuellen COTS-Software-Produkte nicht unabhängig voneinander evaluieren

[9] Die Kosten für die Migration von Windows NT auf Windows XP sind für die Unternehmen sehr groß. Von daher tendieren größere Unternehmen dazu, solche Migrationen möglichst lange hinauszuzögern.

[10] Oft wird diese Veränderung nicht durch eine Änderung der entsprechenden Domäne oder Technologie ausgelöst, sondern durch den Markt. Mittlerweile führen Aufkäufe von Softwarefirmen zu einer immer häufigeren Ablösung bestehender COTS-Software-Produkte.

[11] Oder sportlich formuliert: Nicht jeder Star ist ein guter Teamplayer.

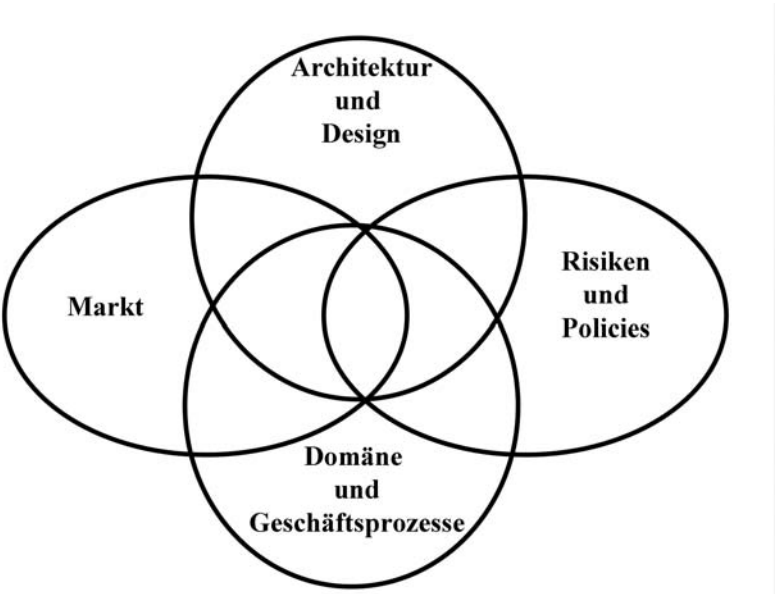

Abb. 10.3: Die vier Einflusssphären bei der Evaluation von COTS-Software

lassen. Eine zusätzliche Schwierigkeit ergibt sich oft dadurch, dass die gesamte zu verwendende COTS-Software ausgewählt werden muss, bevor eine neue Architektur definiert werden kann. Aber es gibt auch versteckte psychologische Faktoren bei der Evaluation; so bevorzugen Endbenutzer meist Produkte mit einer möglichst hohen fachlichen Funktionalität, während Softwareentwickler generell zu komplexeren Lösungen tendieren, auch wenn sie selten genutzt werden, solange sie als technologisch fortschrittlich gelten.

Die Unternehmen, welche Erfahrung in der Softwareentwicklung besitzen, sehen sich bei der Evaluation der COTS-Software mit einem neuen Phänomen konfrontiert. Im Gegensatz zur Softwareentwicklung, wo die Phasen Systemumgebungsdefinition, Architektur, Design und Implementation in einer zeitlichen Sequenz folgen und aufeinander aufbauen, sind diese Betrachtungsweisen bei der COTS-Software simultan vorhanden und müssen auch gleichzeitig betrachtet werden, s. Abb. 10.3. Zu den simultan zu berücksichtigenden Einflüssen gehören:

- Markt – Auf dem Markt gibt es, in der Regel, verschiedene miteinander konkurrierende COTS-Software unterschiedlichster Hersteller. Diese haben durchaus stark differierende Eigenschaften, unterschiedliche Standards, Marktpotentiale, Bekanntheitsgrade, usw.
- Architektur – Die Architektur und das Design beschreiben die Elemente des Systems und die Beziehung zwischen den einzelnen Elementen. Hierzu

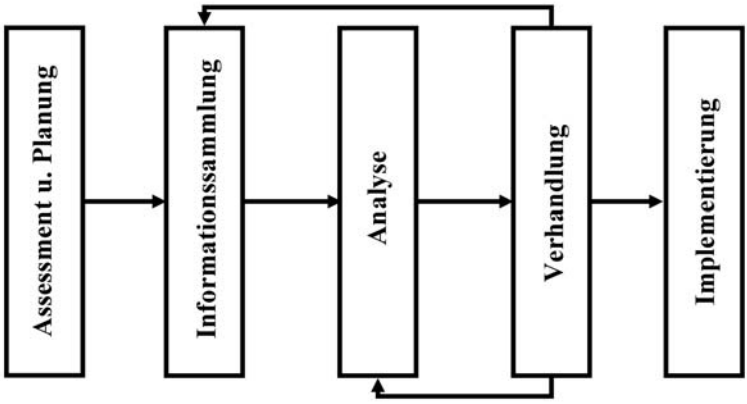

Abb. 10.4: Der Evaluationsprozess für COTS-Software

zählen Eigenschaften wie Datenhaltung, Performanz, Usability, Ergonomie, usw.
- Domäne und Geschäftsprozesse – Die Anforderungen des aussuchenden Unternehmens; diese sollen ja gerade unterstützt und implementiert werden.
- Risiken und Policies – Neben Fragen der IT-Governance und Enterprisearchitektur hat jedes Projekt spezifische Risiken, die bewusst adressiert werden müssen.

Ein möglicher Prozess zur Evaluation von COTS-Software ist in Abb. 10.4 dargestellt. Dieser Prozess ist nur dann vernünftig durchführbar, wenn er hochiterativ eingesetzt werden kann, da erst nach einer Implementierung tatsächlich ein sinnvolles Assessment stattfinden kann. Die einzelnen Schritte dieses Prozesses sind:

- Assessment und Planung – Dieser Schritt enthält die Zieldefinition für die anstehende Iteration und die typischen Planungsaktivitäten.
- Informationssammlung – Hier werden die Anforderungen, sowie die Informationen über die COTS-Software gesammelt. Aber auch eine Risikobetrachtung sollte hier vorgenommen werden.
- Analyse – In diesem Schritt müssen alle bisher gesammelten Informationen analysiert werden.
- Verhandlung – Im Rahmen der Verhandlungen müssen die verschiedenen Beteiligten ein juristisches Übereinkommen abschließen.

- Implementierung – Letztendlich muss die gefundene Lösung auch implementiert werden. Oft spricht man von einem Assembly, speziell dann, wenn das gesamte Softwaresystem aus COTS-Software aufgebaut wird.

Auf Grund der sehr hohen Dokumentenlastigkeit – schließlich werden jede Menge juristischer Verträge geschlossen – sind agile Vorgehensmodelle, s. Abschn. 11.4, für eine COTS-Evaluierung denkbar ungünstig. Die hohe Iterativität legt es nahe, einen RUP oder EUP-Prozess zu wählen, s. Abschn. 11.2, allerdings sind einige wichtige Veränderungen an diesen Prozessen vorzunehmen, da sich bei COTS-Software das resultierende System nicht inkrementell aufbaut, sondern oft völlig chaotisch verhält. Insofern ist ein Vorgehensmodell zu wählen, das sowohl ein hohes Maß an Parallelität als auch Fehlentscheidungen unterstützen kann. Bei mehreren konkurrierenden COTS-Software-Produkten kann es sein, dass erst nach einer langen Testphase der momentane Favorit verworfen wird und wieder von vorne begonnen werden muss. Aus dieser Sicht ist das Vorgehen chaotisch und nicht inkrementell.

Da die Unternehmen oft mit dem gesamten Prozess überfordert sind, bzw. die im Unternehmen beteiligten Strömungen nicht mehr objektiv urteilen können, wird nach einer neutralen Instanz für den Auswahlprozess gesucht. Diese neutrale Instanz soll dann, möglichst objektiv und quantifizierbar, die entsprechenden COTS-Software-Produkte evaluieren. Es liegt daher nahe, ein Consultingunternehmen mit genau dieser Aufgabe zu betreuen. Leider sind die Berater, je nach dem Consultingunternehmen, nicht immer wirklich hilfreich bei der Evaluation, da große Consultingunternehmen meistens Kompetenzen und Mitarbeiter für die Implementierung von Produktsuiten aufgebaut haben. Da diese aber langfristig ausgelastet werden müssen, tendiert das Consultingunternehmen dazu, eine solche COTS-Software zu empfehlen, die eines sehr intensiven Tailorings bedarf, welches wiederum besonders gut vom empfehlenden Consultingunternehmen geleistet werden kann. Es sind Fälle bekannt, in denen die Kosten für das Tailoring die Lizenzkosten um Größenordnungen überstiegen haben.

Kauf

Bei den Kaufbetrachtungen ist nicht nur das Augenmerk auf die Lizenzkosten zu richten, denn es gibt noch eine Anzahl von weiteren Faktoren, welche sich als Kostentreiber herausstellen. Dazu zählen:

- Schulungskosten für die Benutzer bzw. den IT-Betrieb
- Upgradekosten für neue Hard- und Software, welche durch den Kauf der COTS-Software nötig geworden sind. Häufig handelt es sich hierbei um Datenbanklizenzen für die Datenhaltung der COTS-Software. Je näher die COTS-Software an der Infrastruktur ist, s. Abb. 10.5, speziell bei Datenbanksystem- und Betriebssystemsoftware, desto höher ist die Wahrscheinlichkeit, dass durch den Einsatz dieser infrastrukturähnlichen Software auch alle dieselbe Infrastruktur nutzenden Applikationen beeinflusst

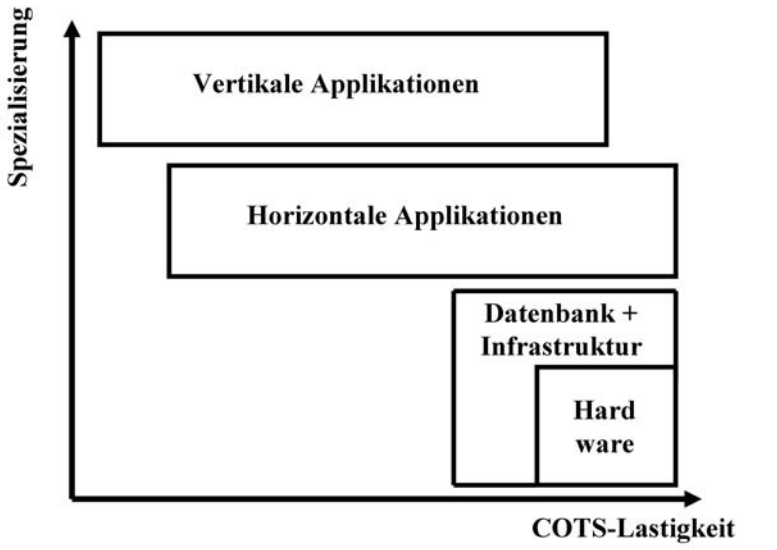

Abb. 10.5: Die unterschiedlichen Spezialisierungsgrade der COTS-Software

werden. Je genereller und verbreiteter der Einsatz der jeweiligen COTS-Software, desto größer sind die möglichen Auswirkungen auf Änderungen quer durch die gesamte IT-Landschaft eines Unternehmens. Aber auch Änderungen in spezielleren COTS-Software-Produkten, beispielsweise der vertikalen Software, können, neben der Anpassung des Glue-Codes, ganze Veränderungsketten in den generelleren COTS-Software-Produkten nach sich ziehen. Diese Änderungen wiederum können sich negativ auf die restliche vertikale und horizontale Software auswirken. Auf diese Weise entsteht eine Art Dominoeffekt der Änderung.

- Kosten für die Softwarepflege des COTS-Software-Produktes – Die Kosten für die Pflege, sie beinhaltet in der Regel das Recht auf Fehlerbeseitigung und Anpassung an neue technische Gegebenheiten, beläuft sich, je nach Branche, auf 10 bis 25 % der Lizenzkosten pro Jahr.
- Kosten für die Implementierung von Individualsoftware durch den COTS--Software-Hersteller – Im Fall der notwendigen Erweiterung der COTS-Software, um der Anforderungsanalyse zu genügen, kann diese Erweiterung nur durch den COTS-Software-Hersteller durchgeführt werden. Solche Formen der Anpassung erreichen durchaus Kosten in derselben Größenordnung wie die Lizenzkosten. Während der Evaluierungsphase wird dies meist nicht offensichtlich, erst beim konkreten Einsatz im Fachbereich entdecken die Endbenutzer plötzlich Zusatzanforderungen an die Software. Da der Weg zurück nun verbaut ist, lässt sich der COTS-Software-Hersteller

diese nachträglichen Änderungen[12] teuer bezahlen. Sein Quasimonopol ermöglicht ihm diese Machtstellung.

- Kosten für die Adaption – Die Kosten für die Anpassung der COTS-Software an das Unternehmen übersteigen in manchen Fällen die Lizenzkosten um das Vielfache. Besonders COTS-Software, welche die Kernprozesse des Unternehmens unterstützen soll, erweist sich hier als sehr teuer. COTS-Software für die Nichtkernprozesse verlangt in der Regel eine sehr viel geringere Adaption. Das liegt daran, dass die Kernprozesse eines Unternehmens hochgradig spezialisiert sind, denn die Kernprozesse sind die Prozesse, anhand derer sich das Unternehmen von Konkurrenzunternehmen differenzieren kann. Bei unterstützenden Prozessen, z.B. der Buchhaltung, ist es einfacher; hier können horizontale Applikationen eingesetzt werden. Diese lassen sich branchenweit hochgradig standardisieren, so dass der Adaptionsaufwand sich meist in Grenzen hält. Der hohe Grad an Spezialisierung[13] der Kernprozesse erzwingt häufig hohe Adaptionskosten.
- Adaptionsfolgekosten – Ein hoher Grad an Adaption führt zu hohen Folgekosten[14] bei der Anpassung zukünftiger Releases des COTS-Software-Herstellers, da a priori nicht sichergestellt ist, ob nach dem Releasewechsel das System noch lauffähig bleibt. Aus diesem Grund enthalten die meisten COTS-Software-Systeme Mechanismen zur Softwareevolution, s. Abschn. 4.14.

COTS-Software-Adaption

Die Adaptionsphase beinhaltet die Summe aller Anstrengungen, die unternommen werden müssen, um die COTS-Software an die lokalen Gegebenheiten anzupassen. Diese Adaptionen fallen in 3 Basiskategorien:

- Erzeugung von Meta- und Betriebsdaten
- Adaption der funktionalen und nichtfunktionalen Eigenschaften
- Datenmigration

Zu den Meta- und Betriebsdaten gehören alle fachlich und technisch steuernden Elemente, so z.B. die Namen der Datenbank, die Autorisierung und Namen der Benutzer, aber auch die Anpassung von Oberflächen oder Vorlagen auf den Unternehmensstandard. So müssen die Rechnungen, welche eine

[12] Der COTS-Software-Hersteller wird vermutlich intern eine Produktlinienstrategie, s. Kap. 9, einsetzen, um seine tatsächlichen Kosten gering zu halten.

[13] Die COTS-Software-Hersteller sind sehr schnell bereit, ihre Produkte als Standardsoftware zu bezeichnen, allerdings ist es fragwürdig, ob nach den immensen Adaptionskosten der Standard noch erhalten ist. Außerdem definiert jeder Softwarehersteller selbst, was ein Standard ist, daher gibt es heute einen Microsoft- und einen SAP-Standard.

[14] Die Kosten können so hoch sein, dass neue Upgrades bewusst nicht eingeführt werden, um Risiken und Kosten zu vermeiden.

Fakturiersoftware erzeugt, Unternehmensadresse, Gerichtsstand, Handelsregistereintrag, Bankverbindungen etc. enthalten.

Der größte Aufwandsblock ist aber in der Adaption der funktionalen und nichtfunktionalen Eigenschaften verborgen. Dies geschieht entweder durch den Hersteller selbst, oder durch ein Consultingunternehmen. Die meisten Consultingunternehmen leben in einer fast „symbiotischen“ Beziehung zum COTS-Software-Hersteller, dieser bildet die Mitarbeiter der Consultingunternehmen aus und stellt sicher, dass diese eine gewisse Basisqualität besitzen. Alle großen COTS-Software-Hersteller haben entsprechende Ausbildungs- und Zertifizierungsprogramme für die Consultingunternehmen. Diese enge Bindung hat zur Folge, dass die Consultingunternehmen meist die Produkte des entsprechenden COTS-Software-Herstellers aktiv empfehlen. Große COTS-Software-Hersteller geben den betriebswirtschaftlichen Fakultäten bekannter Universitäten äußerst preisgünstige Lizenzen ihrer Software, so z.B. SAP und Microsoft. Die Motivation dahinter ist, dass sich die meisten Entscheidungsträger von Unternehmen aus Absolventen der bekannten Betriebswirtschaftsfakultäten rekrutieren. Diese haben sich als Studenten an die „Standard“-Software gewöhnt und sind dann als Entscheider dem ihnen bekannten COTS-Software-Herstellern stärker zugeneigt.

Die Datenmigration ist ein essentieller Bestandteil für den Einsatz der COTS-Software. Erstaunlicherweise wird diese Aktivität oft in ihrer Komplexität und ihrem Aufwand stark unterschätzt. Die COTS-Software-Hersteller leben, gemessen an dem Durchdringungsgrad der Unternehmen bezogen auf den Softwareeinsatz, in einem Verdrängungswettbewerb. Folglich ist die Wahrscheinlichkeit sehr hoch, dass die COTS-Software eine bestehende Software, entweder vom COTS-Software-Typus oder Individualsoftware, ablösen muss. Dies hat jedoch zur Folge, dass die vorhandenen Daten übernommen werden müssen; sie komplett neu zu erfassen ist in der Regel keine sinnvolle Alternative. Auf Grund von Inkompatibilitäten und der häufig schlechten Datenqualität ist dieses Unterfangen recht teuer, s. Abschn. 4.18.

Legacysystem-Adaption

Bei den meisten Projekten mit der Zielsetzung, bestehende Legacysoftware durch COTS-Software zu ersetzen, ist zu beobachten, dass implizit vorausgesetzt wird, dass das restliche, verbleibende Legacysystem durch den Einsatz der COTS-Software nicht beeinflusst wird. Diese Annahme ist in den meisten Fällen falsch! De facto dient Software zur Unterstützung und Implementierung von Geschäftsprozessen, welche genauer betrachtet diverse Abfolgen von Prozessschritten, manchmal sogar regelrechte Prozessketten, darstellen. Die einzelnen Schritte werden häufig von unterschiedlichen Softwareteilen unterstützt. Es liegt folglich in der Natur der Sache, dass das restliche Legacysystem in der Lage sein muss, Daten und Prozesse für die COTS-Software vor- oder nachzubereiten. Dies hat zur Konsequenz, dass sich die Legacysoft-

ware sehr wohl verändern muss. Solche Veränderungen sind ein Spezialfall der adaptiven Maintenance, s. Kap. 7.

Glue-Code

Der Glue-Code, wörtlich übersetzt „Kleister-Code“, ist ein implementationsspezifischer Teil, welcher zwischen dem restlichen Legacysystem und dem neuen COTS-Software-System notwendig ist, um das Gesamtsystem zusammenzuhalten. Dieser Glue-Code nimmt praktisch eine Zwischenstellung ein, auf der einen Seite ist er kein Bestandteil des COTS-Software-Systems, auf der anderen Seite auch kein Teil der Legacysoftware. Dieser Glue-Code benutzt die Schnittstellen des COTS-Software-Systems, interpretiert dessen Inhalt und überträgt diesen dann an die Legacysoftware. Typisches Beispiel für einen Glue-Code ist die Gesamtmenge der Schnittstellensoftware, welche bei einer Enterprise Application Integration geschaffen werden muss.

Ein zweites Einsatzgebiet von Glue-Code ist der Glue-Code in seiner Funktion als eine Art Isolationsschicht. Auf Grund der Aufwände, bedingt durch die zukünftigen Änderungen der COTS-Software, möchte man den Impact, s. Abschn. 7.8, möglichst gering halten. Von daher ist es sinnvoll, das restliche System gegenüber der COTS-Software zu isolieren.

Ein solches Vorgehen ist auch beim Einsatz von COTS-Klassenbibliotheken sinnvoll, da eine Veränderung dieser die damit erstellte Software gefährden kann, der Glue-Code schafft die Möglichkeit, diese Veränderungen in den Griff zu bekommen.

Die Entwicklung von Glue-Code hat die große Problematik, dass die COTS-Software in der Regel unbekannt ist und daher die Aufwände für die Einarbeitung und den sinnvollen Entwurf von Glue-Code recht hoch sein können. Erschwert wird dies durch die Geschlossenheit der COTS-Software, klassische Werkzeuge der Softwareentwicklung, wie beispielsweise Debugger, sind oft nicht einsetzbar.

Systemintegration

Wie bei jedem System ist auch bei den Veränderungen einer Legacysoftware durch den Einsatz von COTS-Software vor der Inbetriebnahme ein intensiver Integrationstest durchzuführen. Hier sollten die Aufwände sehr viel höher sein als bei selbsterstellter Software[15], da der nicht vorhandene Sourcecode der COTS-Software immense Gefahren beinhalten kann. Die COTS-Software muss vom Systemintegrator wie eine „Blackbox“ behandelt werden, d.h. die COTS-Software ist wie eine geschlossene Einheit bezüglich der Tests anzusehen. Ironischerweise geschieht in der Praxis häufig das Gegenteil; die im eigenen Haus erstellte Software besitzt einen geringeren Stellenwert, sie gilt

[15] Dieser Forderung wird leider in den seltensten Fällen entsprochen!

als schlecht und fehleranfällig im Vergleich zur COTS-Software. Diese Einstellung ist das Ergebnis der restriktiven Informationspolitik der COTS-Software-Hersteller, diese verheimlichen ihre Defekte in der Software[16], während bei einer Software aus dem eigenen Haus die Defekte quasi öffentlich sind. Damit diese komplexe Testproblematik in den Griff zu bekommen ist, erscheint es sinnvoll, inkrementellen Integrations- und Teststrategien den Vorzug zu geben.

Lizenzverwaltung

Die Lizenzen sind Kostenfaktoren, vor allem der Kauf von unnötigen Lizenzen ist ein Problem. Die Lizenzen für Spezialsoftware sind meist noch gut verfolgbar, am komplexesten ist es bei Lizenzen für Datenbanken und Officesoftware. Deren großer Verbreitungs- und Nutzungsgrad innerhalb eines Unternehmens macht die Einführung eines expliziten Softwarelizenzmanagements sinnvoll. Aber auch sich plötzlich verändernde Lizenzpolitiken der Hersteller können in einem Unternehmen Unsicherheit erzeugen.

Softwareupgrades

Genau wie bei der Erstevaluation zur Entscheidung, ob eine COTS-Software eingesetzt werden soll, muss auch bei der Frage verfahren werden, ob ein Upgrade auf die bestehende Software eingesetzt werden kann.

Es gibt diverse Gründe, warum ein spezifisches Upgrade eingesetzt werden sollte. Rangierend von einer einfachen Fehlerbeseitigung über die Schaffung völlig neuer Funktionalitäten bis hin zu Adaptionen an neue Technologien: Alle diese Quellen können Ursachen für die Produktion eines Upgrades sein. Allerdings hat ein neuer Upgrade auch Schattenseiten; prinzipiell können alle schon angesprochenen Aktivitäten bei einem Upgrade wieder notwendig sein, von der Evaluation, da ja zuviel Funktionalität auch unerwünscht sein kann, über die Adaption der COTS-Software, der Legacysoftware, bis hin zur Änderung des Glue-Codes.

Manchmal ist es unerlässlich, einen Upgrade einzusetzen. Dies kann geschehen, wenn der COTS-Software-Hersteller die für ihn alten, sich im Einsatz befindenden Releases vom Markt nimmt und auch jedwede Haftung oder Support für solche Releases verweigert. Besonders tückisch ist dieses Vorgehen, wenn es sich bei der COTS-Software um Betriebssystemsoftware handelt, da jetzt von den Upgrades eine Unmenge von anderen Applikationen inklusive des gesamten Legacysystems betroffen sind. Die Kosten einer solchen Umstellung können immens sein, wie viele Unternehmen im Rahmen ihrer Umstellung auf Windows XP leidvoll erfahren mussten.

[16] Siehe auch Fußnote S. 170.

Fehlerbeseitigung

Bezüglich der Beseitigung von Defekten ist zu beachten, dass jetzt die Zeiten gegenüber der Fehlerbeseitigung bei einer selbsterstellten Individualsoftware zunehmen. Diese Zunahme hat diverse Ursachen. Dazu zählen:

- Das schwierige Auffinden und Identifizieren des Fehlers, da der Sourcecode nicht zur Verfügung steht.
- Die Prioritäten beim COTS-Software-Hersteller sind in der Regel anders als beim kaufenden Unternehmen. Dies kann soweit führen, dass der Lieferant sich weigert, den vermeintlichen Defekt zu beseitigen.
- Oft muss der Glue-Code nach einer Fehlerbeseitigung angepasst werden.

Die Fachbereiche müssen, bezüglich der Reaktion und vor allen Dingen bezüglich der Lösungszeiten auf Fehlermeldungen, ihrerseits umdenken. Bei selbsterstellter Software sind die Fachbereiche sehr kurze Lösungszeiten, manchmal im Stundenbereich, gewohnt, nun aber werden die Zeiten von der ersten Meldung bis zur Behebung, die Lösungszeiten, stark gestreckt. Die Lösungszeiten gehen vom Stunden- und Tagesbereich für die Individualsoftware in den Bereich von Monaten bis Jahren für die COTS-Software über. Zwar nimmt der Hersteller die Meldungen meist recht schnell entgegen, jedoch dauert es bis zur tatsächlichen Defektbehebung relativ lange.

10.2 Ersatz

Der vollständige Ersatz der Legacysoftware durch COTS-Software verläuft ähnlich der Bewertung, ob einzelne Teile der Legacysoftware ersetzt werden sollten, s. Abschn. 10.1. Allerdings gibt es hier einige Besonderheiten. Zum einen werden nicht nur Teile des Legacysystems betrachtet, sondern die Legacysoftware in ihrer Funktionalität als Ganzes, und zum anderen werden auch andere Teile des Enterprise-Stacks, s. Abb. 5.2, in Frage gestellt. Wenn der Ersatz der Legacysoftware ins Auge gefasst wird, sollte das bestehende System in seine funktionalen Bausteine zerlegt werden und für jeden der einzelnen funktionalen Bausteine sollte versucht werden, einen praktikablen COTS-Software-Ersatz zu finden.

Häufig sind komplette funktionale Pakete, auch Produktsuites genannt, besser für den Ersatz als der Versuch, ein System vollständig aus einzelnen Komponenten unterschiedlichster Lieferanten aufzubauen. Einzelne Komponenten tendieren dazu, ein höheres Maß an Inkompatibilität zu haben als ein Produktpaket eines Herstellers. Diese Aussage ist zwar in der Theorie korrekt, die Praxis lehrt aber etwas Vorsicht darin, diese Auffassung walten zu lassen. Auf Grund der zunehmenden Konzentration der Softwarehersteller sind einige von ihnen dazu übergegangen, vollständige Fremdsoftware, die wiederum COTS-Software ist, nun allerdings von einem anderen Hersteller, in ihre Produktsuites aufzunehmen, ohne jedoch für eine reibungslose Integration der

unterschiedlichen Komponenten zu sorgen. Die so entstehenden Produktpakete weisen oft eine Vielzahl von unterschiedlichen Architekturen auf, welche die Risiken von Fehlern erhöhen.

In den meisten Fällen ist der Komplettersatz jedoch erwägenswert, da die COTS-Software-Hersteller das Phänomen der Sedimentation ausnützen, um effektiver zu produzieren. Bei der Sedimentation bewegt sich ein Teil der Supportfunktionalität von der Applikation in die Middleware, von der Middleware in das Betriebssystem und von dort in die Hardware. Beispiele hierfür sind Netzwerkverbindungsprotokolle, die heute Bestandteile aller modernen Betriebssysteme sind. Aber auch Supportfunktionalität, wie beispielsweise ein Printspooler oder ein Jobscheduler, welche früher Bestandteil einer Legacysoftware waren, sind heute oft in der Infrastruktur wiederzufinden. Ein besonders schönes Beispiel für Sedimentation ist Lastverteilung in Webservern; diese wanderte zunächst in die Betriebssysteme und ist heute sogar in der Netzwerkhardware implementiert.

Durch die Sedimentation kann der COTS-Software-Hersteller einen Teil seiner Funktionalität outsourcen, dies hat andererseits zur Folge, dass die COTS-Software-Hersteller sich in einem permanenten Migrationsprozess, s. Kap. 5, befinden.

Die effektivste Form des Ersatzes ist jedoch eine andere: Der Einkauf eines kompletten COTS-Software-Paketes als Ersatz für die Legacysoftware und die vollständige Umstellung der Geschäftsprozesse und der Organisation auf das neue COTS-Software-Paket. Dies macht nur dann Sinn, wenn die Legacysoftware nicht gerade den wettbewerbskritischen Vorsprung des Unternehmens gebildet hat. Die Widerstände gegen ein solches Unterfangen sind in der Regel immens, s. Abschn. 5.2. Obwohl hier der Fokus auf einer Reorganisation sowie dem Business Process Reengineering liegt, bleibt doch auch hier ein softwaretechnischer Teil zurück: Die Datenmigration! Eine solche Datenmigration kann zu einem sehr risikoreichen Prozess werden, speziell dann, wenn die Datenqualität nicht besonders hoch ist, s. Abschn. 4.18.

Erfahrungsgemäß handelt es sich bei einem COTS-Ersatz um ein sehr großes Unterfangen für jedes Unternehmen. Die üblichen Laufzeiten eines solchen COTS-Software-Ersatzprojektes bis zum Ende der Migration betragen mehr als 2 Jahre. Es ist damit zu rechnen, dass in dieser Zeit ca. 1-4 Releases der originalen COTS-Software auf den Markt gekommen sind. Diese neuen Releases der Software müssen ihrerseits auch wieder in Betrieb genommen werden und sollten genauso intensiv geprüft werden wie der allererste Release. Die Prüfung stellt natürlich einen immensen Aufwand dar; wenn zusätzlich noch ein hoher Anteil an Anpassungen in der COTS-Software bei der Implementierung vorgenommen wurde, so können die langfristigen Kosten sehr hoch werden. Zum Teil übersteigen diese Kosten innerhalb weniger Jahre sogar die gesamten Lizenzkosten.

10.3 Softwareevolution und COTS

Die Gesetze der Evolution von Software, s. Abschn. 4.2, wurden bisher sehr stark aus dem Blickwinkel der Softwareentwicklung selbst betrachtet. An dieser Stelle stellt sich die Frage, wie sich die COTS-Software in diese Entwicklungsgesetze einfügt.

Aus Sicht eines Systemintegrators oder eines Endbenutzers sind einige der Evolutionsgesetze identisch, da die treibenden Kräfte, welche manchmal zur Identifikation des jeweiligen Gesetzes führen, nicht direkt beobachtbar sind.

Gesetze I und VI

Bei der Auswahl der COTS-Software muss die Adäquatheit der Software sowohl gegenüber der Systemumgebung als auch den Domänenanforderungen gegenüber überprüft werden. Die Anforderungen werden spezifiziert und die entsprechenden Schnittstellen zu anderen Systemen werden festgelegt, genau wie bei jeder anderen Form der Softwareentwicklung. Umgekehrt werden auch von dem Hersteller der COTS-Software implizite und explizite Annahmen über die Domänen gemacht.

Die Veränderungen der Umgebung, nach dem *I.*[17] und *VI.*[18] Gesetz der Softwareevolution, resultieren in der Notwendigkeit, die COTS-Software zu adaptieren. Die COTS-Software würde ansonsten, genau wie jede andere Software, schnell degradiert. Im Gegensatz zur Individualsoftware wird die Adaption vom COTS-Software-Hersteller vorgenommen. Damit diese Adaptionen erreicht werden können, gibt es folgende grundsätzliche Mechanismen:

- lokale Veränderung der COTS-Software[19],
- Veränderung der Systemumgebung, d.h. die Soft- und Hardware, in der das COTS-Software-System eingebettet ist,
- den COTS-Software-Hersteller auffordern, die COTS-Software entsprechend zu verändern.

Die zweite Variante ist von Anfang an zu verwerfen, da sie nur in ganz speziellen Ausnahmefällen tatsächlich funktioniert. Die erste Variante, lokale Änderungen der COTS-Software, ist zwar möglich, hat jedoch langfristig zur Konsequenz, dass das Unternehmen selbst für die Maintenance verantwortlich wird. Diese Entkoppelung mag nicht sofort merkbar zu sein, aber einige Releases später können die Effekte immens werden. Eine Maintenancelawine droht! Diese hohen Maintenancekosten waren aber oft der Grund für den Wechsel

[17] Gesetz I: Software, welche genutzt wird, muss sich kontinuierlich anpassen, ansonsten wird sie sehr schnell nicht mehr nutzbar sein.

[18] Gesetz VI: Die funktionalen Inhalte einer Software müssen stetig anwachsen, um der Endbenutzererwartung auf Dauer gerecht zu werden.

[19] Die lokale Veränderung der Software führt zu dem Phänomen der Softwaremitose, s. Abschn. 4.20.

zu einem COTS-Software-System. Eine solche Vorgehensweise führt den Einsatz von COTS-Software ad absurdum. Die einzige konsequente Variante ist es, den Hersteller der COTS-Software aufzufordern, diese Änderung zu implementieren und im nächsten Release zur Verfügung zu stellen. Nur reagiert dieser nicht auf ein Unternehmen alleine, sondern auf die Anforderungen diverser Unternehmen, welche die COTS-Software einsetzen. Aber nach dem ersten und sechsten Gesetz ist dies für den COTS-Software-Hersteller zunehmend schwieriger, mit der Folge, dass die Einführung von COTS-Software den Maintenanceaufwand langfristig erhöhen und die Qualität des Gesamtsystems rapide degradieren lässt. Diese Erscheinung tritt auch dann ein, wenn am Anfang alle Anforderungen durch die COTS-Software exakt abgedeckt wurden. Oder anders formuliert: Auf Grund der sehr viel größeren Domäne und der erhöhten Zahl an Endbenutzern wächst die Unschärfe zwischen Domäne, implementierter Software und Applikationsmodell schneller an als bei anderen Formen der Software, s. Abb. 4.8.

Alle diese Probleme gehen auf die sehr viel größere Unschärfe von COTS-Software zurück. Die Nutzung von COTS-Software, obwohl ursprünglich vorteilhaft, hat zur Folge, dass die Komplexitätsprobleme sich langfristig betrachtet noch verschärfen! Anders formuliert wird in das Gesamtsystem des Unternehmens eine Applikation eingekoppelt, die COTS-Software, welche die Komplexität und Entropie des Gesamtsystems erhöht.

Gesetz II

Die Folge der stetig ansteigenden Komplexität geht auch an der Maintenance der COTS-Software nicht spurlos vorbei, hier gelten dieselben Gesetzmäßigkeiten[20], wie bei jeder anderen Form von Software. Allerdings produzieren sowohl das größere Volumen der Software, bzw. ihre größere inhärente Komplexität, wie auch die erhöhte Anzahl von widersprüchlichen Anforderungen, bedingt durch die diversen Kunden, recht schnell eine hohe Steigerungsrate der Komplexität. Mathematisch gesehen ist die Komplexitätssteigerung $\mathrm{d}\mathcal{K}$ proportional zur vorhandenen Komplexität:

$$\frac{\mathrm{d}\mathcal{K}}{\mathrm{d}t} \sim \mathcal{K}.$$

Die entstehende Komplexitätskurve

$$\mathcal{K} \approx \mathcal{K}_0 \mathrm{e}^{\alpha t}$$

wächst speziell bei der COTS-Software meistens sehr schnell.

[20] Gesetz II: Die Komplexität einer Software wächst stetig an, außer es wird Arbeit investiert, um diese Komplexität zu reduzieren.

Gesetze III, IV und VII

Diese Gesetze[21,22,23] betreffen die Organisation des COTS-Software-Herstellers und haben insofern nur bedingte Auswirkungen auf die Unternehmen, welche COTS-Software integrieren. Allerdings haben diese organisatorischen Zwangsbedingungen zur Folge, dass die Unschärfe im einsetzenden Unternehmen auch schnell zunimmt, da die Differenz der verschiedenen Organisationen, von COTS-Software-Hersteller und -nutzer, zusätzlich zur Unschärfe beiträgt. Da nach dem Conway'schen Gesetz, s. Abschn. 13.14, eine Softwareentwicklung immer auch ein gewisses Abbild der Organisation darstellt, finden sich in der COTS-Software auch Spuren der Herstellerorganisation wieder. Dies mag zunächst etwas absurd erscheinen, aber es lohnt sich vor Augen zu führen, dass Softwareentwickler ein mentales Modell der Kommunikationsstruktur haben müssen, um die Arbeitsweise des Endanwenders reflektieren zu können. Was liegt also näher, als das eigene Unternehmen, das COTS-Software-Unternehmen, als Vorbild für das innere Kommunikationsmodell des Softwareentwicklers zu machen? Folglich finden sich in der COTS-Software Spuren der Organisation des COTS-Software-Herstellers wieder.

Zusammenfassend kann gesagt werden, dass COTS-Software zwar einen kurzfristigen Gewinn darstellen kann, langfristig aber zu einem Import an Komplexität mit all seinen Konsequenzen führt. Aus dem Blickwinkel der Entropie werden zwei Systeme gekoppelt, bei denen das COTS-Software-System die höhere Entropie besitzt und damit, implizit durch die Koppelung, das Gesamtsystem des Kunden in der Entropie erhöht.

Es ist wahrscheinlich, dass die COTS-Software sehr viel stärker durch die allgemeine Technologieentwicklung sowie die Marktnachfrage getrieben wird als durch Anforderungen von tatsächlichen Kunden. Dies hat zur Folge, dass die COTS-Software den mittlerweile sehr kurzen Technologiezyklen folgt, was sehr schnell zu einer degradierenden Software führt.

10.4 Defekte in COTS-Software

Der typische Test von COTS-Software auf einer Komponentenbasis reicht nicht aus, um damit alle Defekte innerhalb der einzelnen Komponente zu finden, gleichgültig wie intensiv getestet wird. Gleichzeitig sinkt mit der zunehmenden Komplexität und Heterogenität des Gesamtsystems die Wahrscheinlichkeit, dass überhaupt Defekte identifiziert werden können. Insofern

[21] Gesetz III: Die Evolution von Software ist selbstregulierend mit Produkt- und Prozessmetriken, welche nahe an einer Normalverteilung liegen.

[22] Gesetz IV: Die mittlere effektive Aktivitätsrate, welche für eine evolvierende Software angewandt wird, bleibt während der Lebensdauer der Software konstant.

[23] Gesetz VII: Die Inhalte von aufeinanderfolgenden Releases innerhalb einer Software sind statistisch konstant.

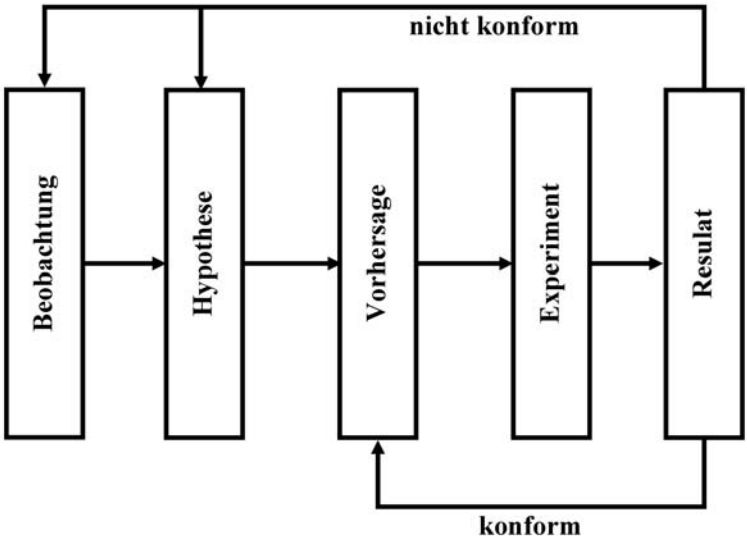

Abb. 10.6: Vorgehensweise bei der Defektsuche in COTS-Software

verstärkt der Einsatz von COTS-Software das „klassische“ Problem der Testbarkeit von komplexen Softwaresystemen.

Die Defekte[24], welche im Zusammenhang mit COTS-Software auftauchen, sind besonders ärgerlich, da das Unternehmen mit dem Erwerb der Software dem Lieferanten ein großes implizites Vertrauen entgegengebracht hat. Aber auch die Reaktionen von Seiten des Herstellers sind für COTS-Software besonders, vor allen Dingen:

- andere Prioritäten beim Hersteller als beim einsetzenden Unternehmen,
- „alte“ Releases werden nicht mehr unterstützt

Diese Reaktionen sind bei der Individualsoftware nicht zu beobachten! Aber auch die Bedeutung sowie die Vorgehensweise bei einer Methode, wie der des „Debuggings“ hat sich verschoben. Durch das Nichtvorhandensein des Sourcecodes bzw. der Entwicklungsumgebung konzentriert sich Debugging nicht mehr auf das Auffinden von fehlerhaften Codezeilen, das klassische Debugging, sondern beschäftigt sich mit der Beobachtung des Verhaltens eines großen undurchsichtigen Produktes. Auf Ursachen kann nur durch das Verhalten des gesamten Produktes indirekt geschlossen werden. Mit der zusätzlichen Unsicherheit, ob das COTS-Software-Produkt den Defekt beinhaltet, oder ob die Integration in das Legacysystem fehlerhaft war – die COTS-Software-

[24] Zur Definition des Begriffs Defekt, siehe Fußnote S. 160.

Hersteller sind hier in der Regel nicht besonders hilfreich.[25] Unabhängig von der Ursache ist die Person, die versucht, den Defekt einzugrenzen, nie identisch mit der Person, welche die COTS-Software entwickelt hat, was dazu führt, dass sie keinerlei Wissen über Designentscheidungen, interne Abläufe oder frühere Defekte hat. Da Softwareentwicklung immer ein Kompromiss aus verschiedenen Forderungen wie:

- Performanz
- Sicherheit
- Zuverlässigkeit
- Wartbarkeit
- Ergonomie

ist, wird das Fehlen genau dieses Wissens zu einem großen Manko. Das Ergebnis des Debuggings ist folglicherweise auch keine Fehlerkorrektur, sondern eine detaillierte und, vor allen Dingen, wiederholbare Beschreibung des Defektes. Interessanterweise hört das Testen von COTS-Software nach der Evolutionsphase auf, was dazu führt, dass ein großer Teil der Defekte erst in der Betriebsphase entdeckt wird und dort zu massiven Problemen führen kann. Die dann gefundenen Defekte führen öfters zu einem echten Ausfall des Gesamtsystems.

Aber selbst wenn der Defekt identifiziert und reproduzierbar ist, heißt dies noch lange nicht, dass er behoben wird, da die Prioritäten des Herstellers oft ganz andere sind. Wenn die COTS-Software einen Defekt hat, ist der erste Impuls des einsetzenden Unternehmens, den Lieferanten aufzufordern, eine Problemlösung zu liefern. In den meisten Fällen dreht der Lieferant die Nachweispflicht um, d.h. er verlangt vom Käufer nachzuweisen, dass dessen Integration korrekt ist. Besonders spannend ist dies, wenn mehrere COTS-Software-Hersteller beteiligt sind, diese schieben sich dann im Kreis die Verantwortung gegenseitig zu, wenn es dem einsetzenden Unternehmen erst einmal gelungen ist, nachzuweisen, dass die Integration korrekt ist. Diese Reaktion ist nicht unverständlich, da der COTS-Software-Hersteller mit einer Vielzahl von Fehlermeldungen konfrontiert wird, die zum größten Teil auf unzureichendes Training, Fehlbedienung oder auch falsche Integration zurückzuführen sind.

Am einfachsten ist es, die Aufmerksamkeit des Herstellers zu gewinnen und damit die Wahrscheinlichkeit zu erhöhen, dass der Defekt in der COTS-Software behoben wird. Dies kann entweder durch eine klare Eingrenzung und Reproduzierbarkeit des Defektes oder durch massiven Druck auf die Leitung des COTS-Software-Unternehmens geschehen.

Die Eingrenzung und Reproduzierbarkeit des Defektes wird erreicht durch eine Vorgehensweise, s. Abb. 10.6, welche der Wissenschaftstheorie der Naturwissenschaften entlehnt ist. Zunächst wird eine Beobachtung gemacht. Auf

[25] Besonders enervierend ist es, wenn die Antworten der Servicehotline oder des Supports des COTS-Software-Herstellers lauten:
„Da sind Sie aber der Allererste ...“
„Das kann gar nicht sein, haben Sie denn auch das Handbuch gelesen?“
„... dafür ist unsere Software auch nicht gedacht ...“

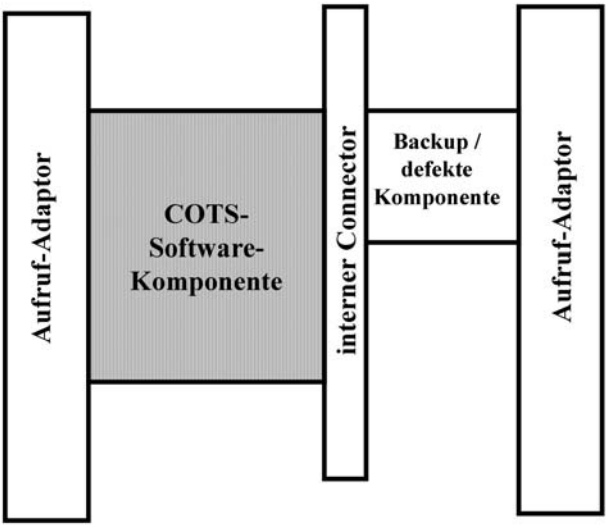

Abb. 10.7: Isolation von COTS-Software-Komponenten

Grund dieser wird eine Hypothese über die COTS-Software als System aufgestellt. Diese Hypothese ermöglicht es, eine Vorhersage über das Verhalten der COTS-Software abzuleiten, das nachfolgende Experiment liefert ein Resultat, welches entweder der Vorhersage entspricht oder von dieser abweicht. Im Fall der Abweichung ist diese Abweichung eine zusätzliche Beobachtung und es muss eine neue Hypothese formuliert werden. Im Fall der Konformität wurde die Hypothese bestätigt. Im Gegensatz zu den Naturwissenschaften, wo es das Ziel ist, die Hypothesen – hier werden sie Modelle genannt – zu falsifizieren, steht bei der Defektsuche die Konformität des Resultats im Vordergrund.

10.5 COTS-Softwareisolation

Wie schon aus der obigen Diskussion über Defekte zu erkennen ist, stellt COTS-Software immer ein Risiko dar; speziell ein Risiko im Sinne von Defekten und Ausfall. Neben diesem rein funktionalen Risiko kann a priori nicht sichergestellt werden, dass sich die COTS-Software auch architektonisch sauber in eine Gesamtarchitektur einfügt. Aus diesen Gründen ist es sehr sinnvoll, jede COTS-Software innerhalb des Gesamtsystems zu isolieren. Das gilt sowohl für die Schnittstellen zum Legacysystem, wie auch für Schnittstellen zu anderen COTS-Software-Komponenten. Eine solche Isolation verhindert zwar nicht den Ausfall einer einzelnen COTS-Software-Komponente, sie hält je-

doch das Gesamtsystem am „Leben“ und ermöglicht in gewisser Weise eine Fehlertoleranz.

Schematisch gesehen ist eine solche Isolation in Abb. 10.7 dargestellt. Die eigentlichen COTS-Software-Komponenten sind nur über einen Adaptor zugänglich. Andererseits werden auch alle Aufrufe von Seiten der COTS-Software-Komponente durch einen eigenständigen Adaptor gekapselt. Für den Fall, dass die COTS-Software einen Defekt aufweist, wird eine zweite „Ersatzkomponente“ aktiviert. Damit eine solche Isolation sinnvoll eingesetzt werden kann, sind eine Reihe von Fragen zu klären:

- Die Spezifikation des Verhaltens der COTS-Software-Komponente von Seiten des Herstellers – Diese muss überprüfbar und gegebenenfalls auch mit Hilfe von Assertions verifizierbar sein.
- Das Verhalten der COTS-Software-Komponente, wie es vom Systemintegrator erwartet wird – Diese Spezifikation muss nicht identisch mit der Spezifikation des COTS-Software-Herstellers sein. Allerdings sollte sie, in der Regel, restriktiver sein.
- Angesammelte Erfahrungen über typische Defektsituationen oder bekannte Work-Arounds
- Inakzeptables Verhalten von Seiten der COTS-Software-Komponente
- Die Spezifikation des restlichen Systems im Sinne einer Umgebung für die jeweilige COTS-Software-Komponente.

Wenn diese Spezifikationen vorliegen, kann mit Hilfe der Adaptoren das so festgelegte Verhalten bis zu einem gewissen Grad erzwungen werden, bzw. das Gesamtsystem verhält sich im Falle eines Defektes „gutmütiger“.

11

Entwicklungsprozesse

But I remember, when the fight was done,
When I was dry with rage and extreme toil,
Breathless and faint, leaning upon my sword,
Came there a certain lord, neat, and trimly dress'd,
Fresh as a bridegroom; and his chin new reap'd
Show'd like a stubble-land at harvest-home;
He was perfumed like a milliner;
And 'twixt his finger and his thumb he held
A pouncet-box, which ever and anon
He gave his nose and took't away again;
Who therewith angry, when it next came there,
Took it in snuff; and still he smiled and talk'd,
And as the soldiers bore dead bodies by,
He call'd them untaught knaves, unmannerly,
To bring a slovenly unhandsome corse
Betwixt the wind and his nobility.
With many holiday and lady terms
He question'd me; amongst the rest, demanded
My prisoners in your majesty's behalf.
I then, all smarting with my wounds being cold,
To be so pester'd with a popinjay,
Out of my grief and my impatience,
Answer'd neglectingly I know not what,
He should or he should not; for he made me mad
To see him shine so brisk and smell so sweet
And talk so like a waiting-gentlewoman
Of guns and drums and wounds,–God save the mark!–
And telling me the sovereign'st thing on earth
Was parmaceti for an inward bruise;
And that it was great pity, so it was,
This villanous salt-petre should be digg'd
Out of the bowels of the harmless earth,
Which many a good tall fellow had destroy'd
So cowardly; and but for these vile guns,
He would himself have been a soldier.

King Henry IV,
William Shakespeare

11.1 Komplexe Systeme

Die Softwaresysteme, die wir heute als Legacysysteme bezeichnen, haben alle gemeinsam, dass sie folgende Charakteristika besitzen:

- Sie haben eine sehr hohe Entropie.
- Sie lassen sich nur sehr schwer verändern.
- Die Software hat ein so hohes Maß an Volatilität erreicht, dass sie faktisch nicht mehr veränderbar ist. Der Maintainability Index ist sehr niedrig geworden.
- Das System besitzt unbekannte und undokumentierte Teile.
- Kein einzelner Softwareentwickler kann das System komplett verstehen.

Aus systemtheoretischer Sicht, handelt es sich folglich um ein komplexes System, welches nicht auf einer einfachen Parametrisierung oder einer singulären Kausalität beruht. Neben der schieren Größe solcher Legacysysteme enthalten sie auch eine Unmenge an Informationen und Gestaltungsmöglichkeiten, so viele, dass kein einzelner Softwareentwickler sie mehr komplett überschauen kann.

Der traditionelle Ansatz, diese Probleme anzugehen, beruht darauf, ein Modell des Systems zu entwickeln, welches nur wenige, aber dafür wichtige Aspekte beschreibt, um damit das System als Ganzes zu kontrollieren. Die Einführung von Schichten oder Komponenten fällt in diese Kategorie von Maßnahmen. Die gesamte Idee des Information Hiding und der Kapselung will die Komplexität auf wenige Größen reduzieren. Historisch gesehen stammt dieser Ansatz aus der Newtonschen Mechanik, wo sich durch ihn eine einfache mechanische Maschine beschreiben lässt. Aber alle diese mechanistischen Systeme haben gemeinsam, dass sie

- eine geringe Detailkomplexität – sie haben nur wenige einfache Teile,
- und wenige Wechselwirkungen zwischen den Teilen – eine geringe dynamische Komplexität,

besitzen.

Solche Systeme verhalten sich stets streng kausal und lassen sich recht einfach vorhersagen, da aus dem Wissen über die einzelnen Teile auf das Verhalten des Gesamtsystems geschlossen werden kann. Diese Systeme haben die Eigenschaft, dass sie sich bezüglich ihrer Grundgröße deterministisch verhalten und damit auch exakt vorhersagbar sind.

Die Legacysysteme verhalten sich jedoch völlig anders; sie sind irreversibel und entziehen sich einer einfachen Kausalitätsbeziehung, da hier die dynamische Komplexität, d.h. die Wechselwirkung zwischen den einzelnen Teilen, überwiegt. In manchen Fallen führt die gleiche Tätigkeit zu einem etwas anderen Zeitpunkt zu drastisch anderen Ergebnissen. Wiederholungen sind meistens nicht durchführbar.

Für den Entwicklungsprozess ist neben der Komplexität des Systems auch die Beteiligung des Menschen am Entwicklungsprozess selbst einer der wichtigsten Punkte. Der Mensch kann in diesem komplexen System jedoch nur

dann sinnvoll agieren, wenn er das System „beherrscht“. Diese Beherrschung bedeutet nicht komplette und vollständige Kontrolle über alle Aspekte des Systems, wie es ein mechanistisches Weltbild impliziert, sondern die Möglichkeit zur Steuerung der Komplexität, ohne sie detailliert kennen zu müssen. Damit der einzelne Softwareentwickler dies erreichen kann, muss er die komplexen Systeme anders begreifen, als er es aus dem Newtonschen Denken gewohnt ist. Stochastische Modelle in der Maintenance sind ein Ansatz, jedes hochgradig komplexe System unter Kontrolle zu bringen.

Charakteristisch für komplexe Systeme sind folgende Eigenschaften:

- Offenheit – Alle Systeme sind offen. Jedes System steht in Wechselwirkung mit seiner Umgebung. Von daher lässt sich ein System nur in seinem jeweiligen Kontext verstehen, der soziotechnische Kontext ist daher auch Bestandteil der Definition eines Legacysystems. Innerhalb der Vorgehensmodelle bedeutet dies zunächst, dass alle möglichen Endanwender auch mit berücksichtigt werden müssen, da sie für das System die Umwelt darstellen. Da die Systeme in permanenter Wechselwirkung mit ihrer Umgebung leben, sind sie zwei Kräften ausgesetzt: Der Selbsterhaltung und dem Anpassungsdruck durch die Systemumgebung. Die Selbsterhaltung erfolgt durch die ständige Selbstreproduktion – der Maintenance – des Systems. Wenn das Softwareentwicklerteam als System verstanden wird, gehorcht es ähnlichen Kräften; hier ist die Selbsterhaltung durch die Projekt- und Firmenkultur bestimmt. Diese Kultur wird durch ständige Wiederholung ritualisiert und schafft somit eine Abgrenzung gegen andere Gruppen.
- Flexibilität – Jedes System besitzt eine Reihe von Freiheitsgraden. Unter der Freiheit des Systems versteht man die Möglichkeit, dass das System sich zwischen verschiedenen Alternativen entscheiden kann. Die Flexibilität eines Systems ist umso höher, je mehr Entscheidungsmöglichkeiten es hat. Die Gegenbewegung, die Einschränkung von Freiheit, wird als Macht bezeichnet, wobei hier zwischen intrinsischer – die Macht des Systems, sich selbst einzuschränken – und externer Macht – die Umwelt, in der Regel andere Systeme, üben Macht auf das betrachtete System aus – unterschieden wird. Die meisten Entwicklungsmethodiken setzen ihre externe Macht ein, um damit die Flexibilität einzuschränken. Durch die eingeschränkte Flexibilität werden ganze Lösungsräume unzugänglich gemacht. Auffällig wird dies, wenn sich die Umgebung verändert: Plötzlich müsste ein anderer Lösungsraum wahrgenommen werden, was aber durch die Zwangsbedingung unmöglich geworden ist. Speziell die agilen Methoden, s. Abschn. 11.4, haben dies als einen ihrer Kernpunkte erkannt. Der heute vorherrschende Streit zwischen den Verfechtern der traditionellen Entwicklung und denen der agilen Methoden, und in gewissem Grad auch zwischen den Anhängern des Rational Unified Process bzw. Enterprise Unified Process, beruht auf der Frage: Wo liegt die Trennlinie zwischen Stabilität und Flexibilität? Wird die Stabilität überbewertet, so führt das so gewählte Vorgehen zu einer Starre, wird der Flexibilität zu viel Ge-

wicht gegeben, so entsteht ein Chaos. Die Frage nach dem Maß an Flexibilität, welches ein System braucht, lässt sich nicht generell beantworten. Als Faustregel lässt sich jedoch beobachten: Je dynamischer die Systemumwelt, desto flexibler muss das System sein. Diese Faustregel erklärt auch die architektonische Entscheidung für Webservices, s. Abschn. 12.9, welche ein hohes Maß an Flexibilität aufweisen, erzwungen durch die hohe Dynamik des B2B- und B2C-Wirtschaftssektors. Außerdem lässt sich hieraus ableiten, dass bei niedriger Dynamik des Systemumfeldes starre, monolithische Systeme durchaus adäquat sein können. Was hier für die IT-Systeme formuliert wurde, gilt völlig identisch für den Entwicklungsprozess: Je dynamischer die Anforderungen, desto flexibler der Prozess, und je statischer die Anforderungen, desto stabiler kann der Entwicklungsprozess sein.

- Dimensionalität – Alle komplexen System sind mehrdimensional. Die Vorstellung, dass nur ein einziger Parameter ausreicht, um ein System zu steuern, ist ein Relikt der Modellbildung der Naturwissenschaften. Der Vorteil einer Eindimensionalität ist eine einfache Kausalität, d.h. es gibt genau eine Ursache, welche genau eine Wirkung zeigt. Dieses Denkschema kann ins Absurde führen: Viele Projekte erhöhen die Qualität der Tests, um die Qualität des Produktes zu erhöhen. Dass hier das letzte Glied der Kette benutzt wird, um damit die Fehler der prozessuralen Vergangenheit zu eliminieren, entgeht den meisten Beteiligten. Die Ursachen sind jedoch in komplexen Systemen sehr viel vielfältiger. Speziell im Entwicklungsprozess nehmen nichtmessbare Größen, wie Motivation und Zufriedenheit, eine steuernde Position ein, ohne dass die Kausalität direkt verfolgbar ist.
- Emergenz – Unter Emergenz wird das Auftreten von Eigenschaften eines Systems verstanden, welche sich nicht aus den Teilen des Systems ableiten lassen. Alle komplexen Systeme zeigen diese Eigenschaft der Emergenz. Das Ziel muss es sein, eine hohe Emergenz zu erreichen, da dann ein echter Mehrwert für das Unternehmen geschaffen wird. Da die klassische analytische Denkweise die Emergenz nicht erklären kann, ist sie auch ungeeignet, das Auftreten von Emergenz vorherzusagen. Außerdem sind emergente Eigenschaften nicht messbar, was sowohl ihre Definition als auch die Feststellung ihrer Existenz stark erschwert. Es gibt aber drei Prinzipien, die Voraussetzungen für das Auftreten von Emergenz sind:
  1 Die Emergenz entsteht immer durch die Wechselwirkung der Teile.
  2 Ohne ein gewisses Mindestmaß an dynamischer Komplexität entsteht keine Emergenz. Umgekehrt formuliert: Starre Systeme zeigen keine Emergenz!
  3 Durch den ständigen Reproduktionsprozess der Systemteile bildet und reproduziert sich Emergenz. Dies ist auch unter dem Begriff Feedback bekannt.

  Das Verhältnis zwischen den Teilen und der Gesamtheit ist iterativ und co-evolutionär, s. Kap. 4, denn Emergenz ist der Prozess, durch den neue Ordnungen aus der Selbstorganisation der Teile entstehen.

Praktische Beobachtungen zeigen ein ähnliches Bild: Je größer die Freiheiten der Beteiligten, desto besser ist das Ergebnis, und je besser das Ergebnis, desto mehr Motivation, was wiederum zu mehr Freiheiten führt. Allerdings führt die ständige Reproduktion des Systems zum Anstieg der Entropie.

- Nichtintuitivität – Alle komplexen Systeme sind per se nichtintuitiv. Man kann dies auch als eine Art Definition für komplexe Systeme nutzen, d.h. intuitive Systeme sind nicht komplex! Durch den hohen Grad an Wechselwirkungen lässt sich die Auswirkung einer Veränderung nicht eindeutig vorhersagen. Ursachen und Wirkungen sind oft überhaupt nicht mehr unterscheidbar, was zu Kausalitätszyklen führt. Allein die Beobachtung eines Systems durch Messung führt schon zu einer Veränderung des Systems. Dies ist eine Eigenschaft, welche die komplexen Systeme mit der Quantenmechanik gemeinsam haben; auch hier zerstört die Messung den messbaren Zustand. Jede Entscheidung verändert das System, mit der Folge, dass dieselbe Entscheidung zu einem späteren Zeitpunkt „falsch“ sein kann, da jeder Eingriff ein neues System produziert. Neben der Kausalität zeigen komplexe Systeme einen Hang zur Zeitverzögerung. Oft lassen sich Kausalitäten allein auf Grund der zeitlichen Distanz nicht mehr zuordnen, was die Steuerung immens erschwert. Diese Nichtintuitivität ist eine der Ursachen für die Nichtvorhersagbarkeit der Maintenance.

Wenn wir die komplexen Systeme als komplex wahrnehmen und das Augenmerk primär auf die Eigenschaften des Systems als Ganzes richten, so lassen sich diese Systeme auch erfassen und steuern. Dies hat allerdings seinen Preis, denn die Idee der Kausalität auf der Ebene der Einzelteile muss aufgegeben werden. Dafür halten statistische und thermodynamische Betrachtungsweisen Einzug in die Welt der komplexen Systeme; daher auch die Nutzung stochastischer Modelle in der Maintenance.

In komplexen Systemen gibt es noch ein weiteres Phänomen, welches vor allen Dingen bei Entscheidungsprozessen auftaucht und ähnlich der Counterfinality, s. S. 192, ist: „Shifting the Burden“. Einer kurzfristigen Lösung wird gegenüber einer langfristigen stets der Vorrang gegeben, obwohl sie das vorliegende Problem eher langfristig verschärft. Das „Shifting the Burden“ ist eines der großen Risiken hinter den agilen Methoden, s. Abschn. 11.4, speziell hinter dem eXtreme Programming. Ein bekanntes Beispiel für dieses Verhaltensmuster ist die Tendenz, eine mögliche Verletzung eines Meilensteins durch massive Überstunden[1] zu korrigieren, mit dem Erfolg, dass Frustration und Defekte langfristig gesehen stark zunehmen.

[1] Ironischerweise empfinden die meisten Führungskräfte dieses Vorgehen als Beweis für das Engagement der Beteiligten und fordern es explizit ein. Es ist durchaus üblich, dass ein Projektleiter über einen längeren Zeitraum hinweg eine 80-Stunden-Woche hat. Nur wird dieser auf Dauer keine guten Ergebnisse produzieren!

Alle Legacysysteme sind komplexe Systeme, allerdings mit der Besonderheit, dass sie schon ein gewisses Alter erreicht haben. Dieses Alter impliziert auch eine spezielle Form des Equilibriums, in dem sich die Legacysysteme befinden; es ist tiefer und „stabiler“ als bei jüngeren Systemen, wobei nicht nur die Software, sondern auch die dazugehörige Organisation ein gewisses „Alter“ erreicht hat. Dies betrifft speziell:

- Maintenanceorganisation
- softwarenutzende Fachbereiche

In den angesprochenen Organisationsteilen sind, parallel zum Alterungsprozess der Legacysoftware, Verkrustungen entstanden, die nur schwer wieder aufzubrechen sind; schließlich bildet die Software auch Teile der Organisation ab. Bezüglich der Fachbereiche wird ein solches Unterfangen im Rahmen der Migration angegangen, bezüglich der Maintenanceorganisation durch die Einführung oder Umgestaltung des Entwicklungsprozesses.

In der Literatur wird bei den Entwicklungsprozessen der Schwerpunkt stets auf die Ersterstellung der Systemsynthese gelegt, dafür wurden die unterschiedlichen Entwicklungsprozesse auch originär konzipiert als Maßnahmen, den Informationsfluss aus der Domäne wohlgeordnet und gesteuert in Software zu verwandeln. Im Zusammenhang mit einem Legacysystem jedoch stehen immer die Maintenance und die mögliche Transformation bzw. Evolution der Legacysoftware im Vordergrund.

11.2 Rational Unified Process

Der Rational Unified Process, RUP, s. Abb. 11.1, ist ein zyklisches Vorgehensmodell, welches als Diagrammsprache die Unified Modeling Language, UML, benutzt. Ursprünglich von der Firma Rational entwickelt, wurde er rasch bekannt und ist heute sehr weit verbreitet. Seit mehr als einem Jahrzehnt ist er das Vorzeigevorgehensmodell in Teilen der IT-Industrie. Die Prozessdefinitionen basieren auf der UML, welche wiederum ein anerkannter Standard ist und de facto die heutige Modellierungssprache für objektorientierte bzw. komponentenbasierte Systeme darstellt. Der Rational Unified Process ist adaptierbar auf eine große Zahl von Umgebungen und Projekt- bzw. Problemgrößen, daher ist sein heute weit verbreiteter Einsatz nicht verwunderlich. Einen zusätzlichen Impuls für die Marktdurchdringung hat in den letzten Jahren der Aufkauf von Rational durch IBM gegeben.

Der Rational Unified Process basiert auf drei Prinzipien:

- Die Geschäftsvorfälle, die Use Cases, sind die steuernden Elemente. Die Grundlage für das Systemdesign, sowie für die Implementierung ist eine eindeutige und unzweifelhafte Festlegung der Requirements an das zu verändernde System.

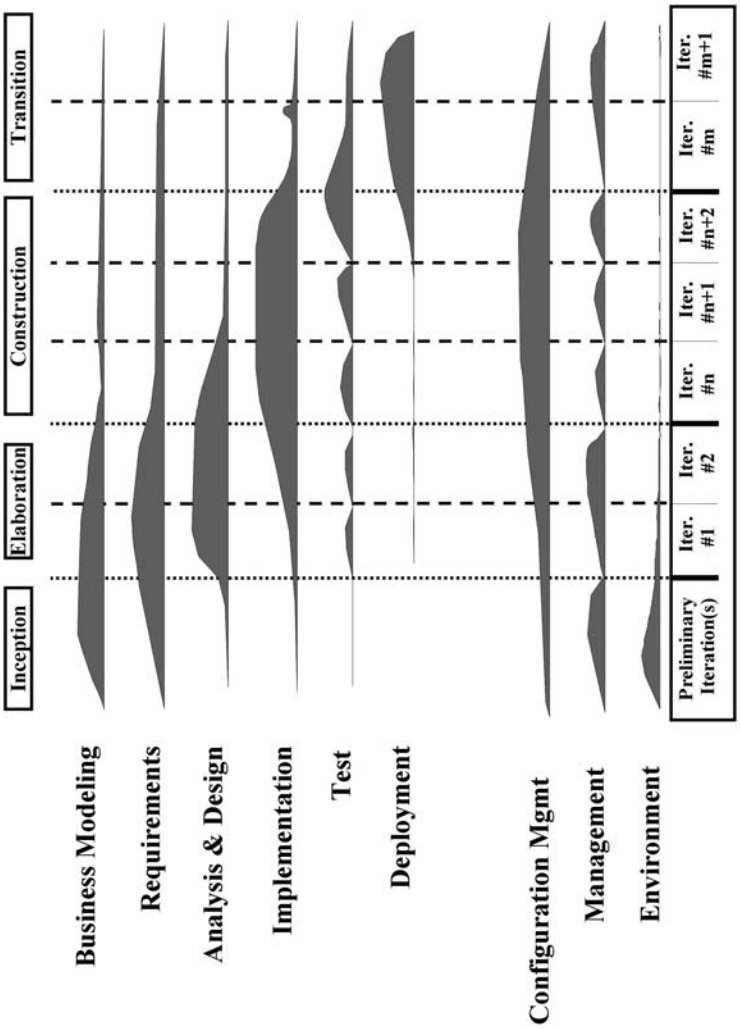

Abb. 11.1: Der Rational Unified Process ©Rational

- Der Rational Unified Process ist ein iterativer und inkrementeller Prozess, s. Abb. 11.2. Die klassischen wasserfallbasierten Modelle hatten stets das Problem der „Big Bang"-Integration am Ende des Entwicklungsprozesses. Der Rational Unified Process übernimmt die iterative, inkrementelle, risikogesteuerte Vorgehensweise vom Spiralmodell. Während der einzelnen Iterationen werden Eichungen am Verfahren zwecks besserer Schätzung, sowie Verbesserungen am Produkt und dessen Eigenschaften vorgenom-

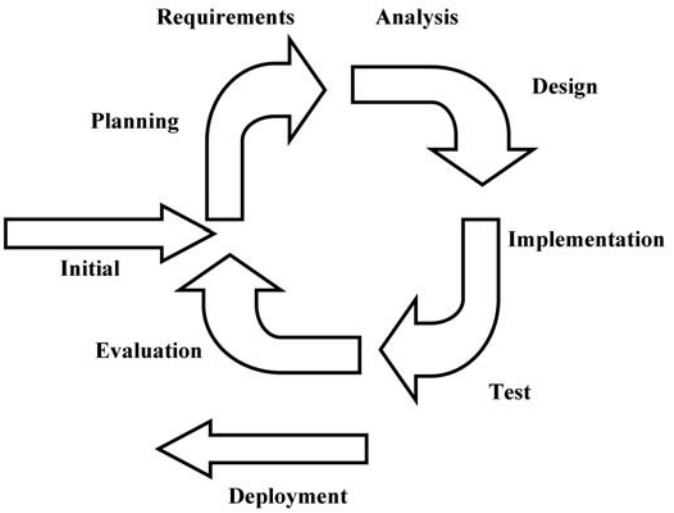

Abb. 11.2: Die Zyklen innerhalb des Rational Unified Process

men. Innerhalb der einzelnen Iteration erfolgt das Arbeiten dann sequentiell.
Der Sinn und Zweck hinter diesem Vorgehen ist es, Fehler und Defizite in regelmäßigen Abständen zu erkennen und korrigierend eingreifen zu können, insofern gibt es eine Risikoorientierung. Je früher ein Defekt gefunden wird, desto einfacher ist er zu korrigieren.

Zusätzlich zu den Prinzipien beinhaltet der Rational Unified Process eine Reihe von Tätigkeiten, auch Disziplinen[2] genannt. Diese einzelnen Disziplinen, s. Abb. 11.1, sind:

- Business Modeling – In der Geschäftsprozessmodellierung werden die fachlichen Anforderungen erfasst.
- Requirements – Hier werden die Requirements an das zu bauende System festgelegt.

[2] Bezüglich der Machtfrage siehe S. 269.

> *Die Disziplin „verfertigt" Individuen: sie ist die spezifische Technik einer Macht, welche die Individuen sowohl als Objekte wie als Instrumente behandelt und einsetzt.*
>
> *Überwachen und Strafen*
> Michel Foucault

- Analysis & Design – Im Rahmen der Analyse wird ein UML-Klassendiagramm erstellt und später durch das Design für die Implementation entworfen.
- Implementation – Diese Disziplin beinhaltet die Umsetzung der durch die Analyse und das Design entworfenen Teile in Code. Die Softwareentwickler können hier wieder die Use Cases zu Hilfe nehmen, da in diesen das gewünschte Verhalten und die gewünschte Funktionalität der Software beschrieben wird.[3]
- Test – Nach jeder Iteration muss das gesamte System auf die Funktionalität und auf Fehler getestet werden. Dabei dienen die Use Cases und ihre Szenarien als Testcasevorlagen.
- Deployment – Das Deployment beschäftigt sich mit der Installation des Gesamtsystems für den späteren Betrieb. Dabei wird sichergestellt, dass alle nötigen Softwareteile vorhanden sind, um die Software fehlerfrei installieren zu können.
- Configuration & Change Management – Die Aufgabe des Change Managements ist es, eine Infrastruktur und Prozesse zur Verfügung zu stellen, welche die systematische Erfassung der sich ändernden Requirements erlauben. Das Configuration Management hat die Versionierung des Systems und den korrekten Einsatz der Softwarereleases als Aufgabe.
- Project Management – Der Projektmanager kümmert sich um die Planung und das Controlling der einzelnen Phasen und Iterationen.
- Environment – Die Environment-Disziplin soll sicherstellen, dass alle Mitarbeiter in der Entwicklungsumgebung mit den identischen Werkzeugen arbeiten. Faktisch muss hier die Entwicklungsinfrastruktur gemanaged werden.

Der Rational Unified Process hat, trotz seiner Verbreitung, einige schwerwiegende Defizite.

- Das erste Defizit ist seine intensive Nutzung der UML. Diese wiederum ist für die Modellierung von Komponenten und objektorientierten Systemen konzipiert, mit der Folge, dass der Rational Unified Process eine starke Tendenz zu solchen Systemen hat. Unternehmen, welche einen hohen Anteil an Legacysoftware haben oder mit einer Koexistenz verschiedenster Systeme leben, tun sich schwer, diese nicht objektorientierten Systeme mit dem Rational Unified Process zu verändern.
- Das zweite Defizit ist, dass der Rational Unified Process nicht alle Phasen des Lebenszyklus von Software abdeckt. Die späteren Phasen, Maintenance und Migration, werden fast gar nicht angesprochen. Erweiterungen des Rational Unified Process hin zum Enterprise Unified Process, s. Abschn. 11.3, beseitigen dieses Problem. Von daher ist der reine Rational Unified Process für Legacysysteme in den meisten Fällen nur bedingt geeignet.

[3] Im Rahmen eines MDA-Ansatzes wäre die Implementierung ein rein generativer Vorgang.

- Das dritte Defizit ist ein Erbe des Wasserfallmodells. Die Phasenorientierung des Rational Unified Process mit den entsprechenden Meilensteinen ist zwar relativ leicht planbar, erweist sich jedoch als eine Hemmschwelle. Der Rational Unified Process impliziert, dass innerhalb des Projektes phasensynchron gearbeitet werden muss. Der Prozess kann Parallelität und verschiedene Entwicklungszustände in den einzelnen parallelen Phasen nicht gut verkraften. Bei den hochkomplexen Legacysystemen ist eine solche Koordination notwendig.
- Das vierte Defizit ist die Bindung des Rational Unified Process an Phasen, nicht an die Ergebnisse der Phasen, d.h. die Iterationen definieren sich durch die Wiederholung der Tätigkeiten und nicht durch die Verbesserung der Produkte der Tätigkeiten. Hier tritt die Prozessorientierung in den Vordergrund, obwohl öfters eine Produktorientierung sehr viel wünschenswerter ist. Diese Tatsache ist bei großen Projekten sehr bedenklich, da diese generell über Teilprojekte kontrollierbar gemacht werden. Die einzelnen Teilprojekte müssen aber ihre Ergebnisse, d.h. ihre Produkte, auf das jeweils andere Teilprojekt abstimmen bzw. ihre Planung auf die Fertigstellung von Zulieferprodukten anderer Teilprozesse ausrichten, was im Rational Unified Process, als Ergebnis der Prozessorientierung, sehr schwierig ist.
- Das fünfte und vielleicht schwerwiegendste Defizit ist die mangelnde Fähigkeit zur Rekursion und Hierarchie. Die Legacysysteme sind sehr komplex und diese Komplexität lässt sich nur durch Schaffung einer Hierarchie beherrschen. Der Rational Unified Process bietet jedoch keinerlei Mittel an, komplexe Probleme in kontrollierbarer Art und Weise zu lösen, obwohl es in der Systemtheorie wie auch in der klassischen Informatik Strategien gibt, solche hochkomplexen Systeme zu beherrschen[4].

11.3 Enterprise Unified Process

Da die meisten Unternehmen Legacysysteme im Einsatz haben, muss der Rational Unified Process genau für diesen Einsatz erweitert werden. Auch der mögliche Einsatz von COTS-Software wird jedoch im Rational Unified Process so gut wie gar nicht unterstützt. Durch die starke Entwicklungslastigkeit des Rational Unified Process fehlen die Teile jenseits des Deployments, der Bereich der Softwareevolution, vollständig.

Der Rational Unified Process muss folglich auf die Teile ausgedehnt werden, mit denen sich alle Unternehmen die meiste Zeit beschäftigen, den Bereich der Softwareevolution, s. Kap. 4 und Abb. 11.3. Das so erweiterte Modell wird als Enterprise Unified Process, EUP, bezeichnet. Die Erweiterung betrifft in den Phasen die beiden folgenden Zeitabschnitte:

[4] *Divide an Conquer* ist eine solche Strategie um große Systeme zu beherrschen.

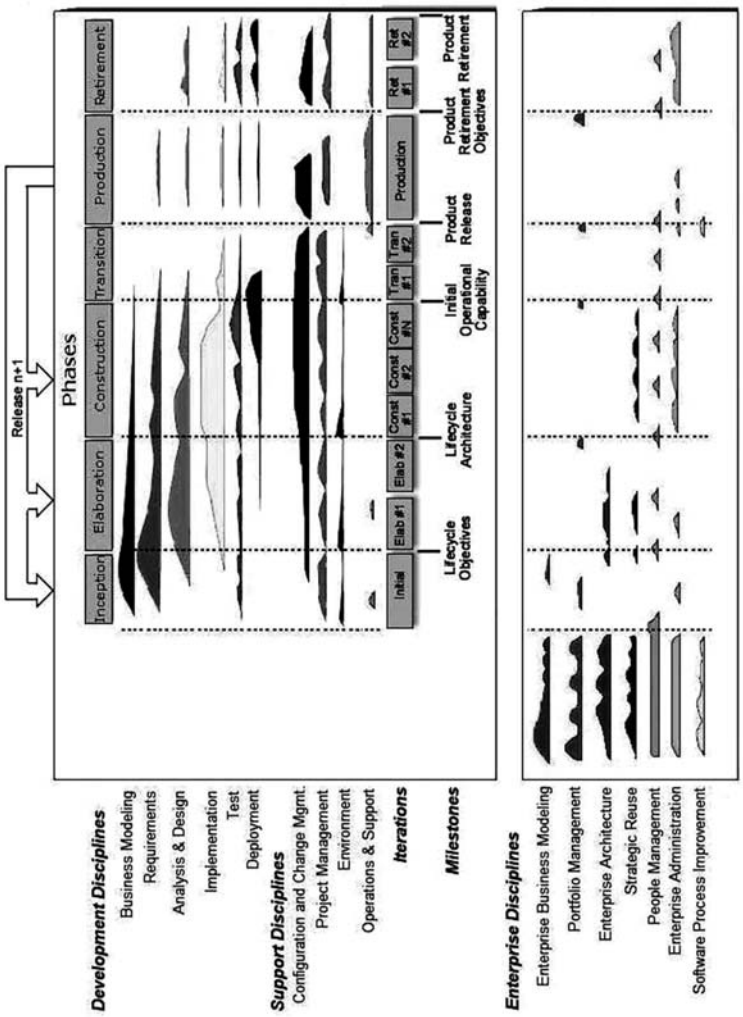

Abb. 11.3: Der Enterprise Unified Process ©Scott W. Ambler

- Production – Die Produktion ist die eigentliche Zielsetzung hinter der Entwicklung. Interessanterweise scheinen viele Softwareentwickler diese Tatsache geradezu zu verdrängen. Aus Softwareentwicklersicht ist die Entwicklung das Wichtigste, nicht die Produktion. Aus unternehmerischer Sicht ist jedoch stets die Produktion der wichtigste Teil bei einem System. Diese Diskrepanz zwischen Softwareentwicklungssicht und unternehmerischer Sicht mag erklären, warum Make-or-buy Entscheidungen innerhalb der

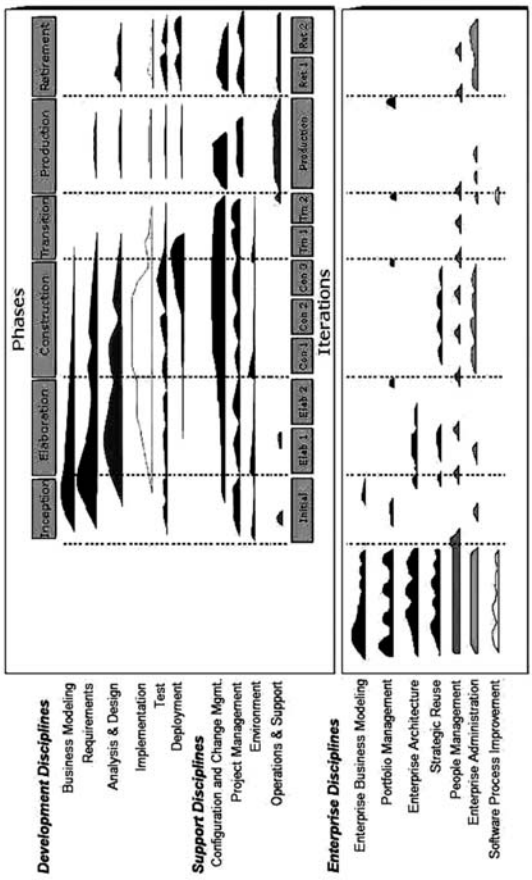

Abb. 11.4: Die Enterprise Disziplinen und das Tailoring ©Scott W. Ambler

Softwareentwicklungsabteilung falsch angesiedelt sind, da diesen Abteilungen der entsprechende Fokus für solche Entscheidungen fehlt. Außerdem sollte berücksichtigt werden, dass Software etwa 80% seines Lebenszyklus in der Wartung verbringt, so dass die Wichtigkeit dieser Phase nicht zu unterschätzen ist. Das Ziel hinter der Produktionsphase ist es, die Software in Produktion zu halten, bis sie durch einen Update oder neuen Release ersetzt werden kann. Nähert sich der Lebenszyklus des Produktes dem Ende, so steht die Ausphasung bzw. die Retirementphase an. Es gibt innerhalb dieser Phase keine Iterationen, da jeder neue Upgrade immer in vorhergehenden Phasen startet und die Phase nur für jeweils einen einzigen Release Gültigkeit besitzt, s. Abb. 11.3. Die großen Pfeile über dem Diagramm zeigen, dass, auch nachdem ein Release erfolgreich in Produk-

tion geht, die Entwicklung der Folgereleases zeitlich parallel abläuft und in den vorhergehenden Phasen angesiedelt ist. Je nach Entwicklungsproblematik, z.B. Bugfixing oder neuen fachlichen Funktionalitäten, beginnt der Entwicklungszyklus des Nachfolgerelease in der Inception- oder Constructionphase.

- Retirement – Irgendwann ist die Software so sehr gealtert, dass ihr Einsatz nicht mehr zu vertreten ist. Dann ist es an der Zeit, diese Software revolutionär durch eine neue Software abzulösen, mit Hilfe der Transformationen wird versucht, diesen Zeitpunkt in die Zukunft zu verschieben. Obwohl diese Phase für ein Unternehmen elementar wichtig ist, ist sie doch relativ unbekannt in den Entwicklungsmodellen, mit Ausnahme des Enterprise Unified Process. Das Ziel der Retirementphase ist die erfolgreiche Entfernung eines schon produktiven Systems aus der Produktion. Innerhalb der Retirementphase laufen folgende Aktivitäten ab:
  - Vollständige Analyse und Dokumentation des Systems bezüglich der Interfaces zu anderen Systemen – Was passiert mit den anderen Systemen im Unternehmen, wenn das untersuchte verschwindet? Oft ist die Koppelung an Legacysysteme erstaunlich stark, aber en detail nur sehr wenig bekannt. Diese Aktivität reduziert das Risiko, mit einem nicht funktionsfähigen Gesamtsystem abzuenden.
  - Anpassung der nun entkoppelten Nachbarsysteme – Häufig hat das Retirement eines Legacysystems eine Art Dominoeffekt, d.h. andere Systeme werden auch obsolet bzw. die Investments in die Entkoppelung sind exorbitant hoch.
  - Transformation und Archivierung der Daten – Obwohl das operative System außer Dienst gestellt wird, stellen die in ihm enthaltenen Daten einen hohen Wert für das Unternehmen dar und sollten entsprechend migriert werden.
  - Configuration Management der zu entfernenden Software, damit diese für ein Rückfallszenario wieder eingesetzt werden kann.
  - Integrationstests des verbleibenden Gesamtsystems auf Funktionalität und Durchgängigkeit der relevanten Geschäftsprozesse.

  Zu den Zielen der Retirementphase zählen:
  - das vollständige Entfernen des Systems,
  - minimale Auswirkungen auf den einzelnen Benutzer,
  - minimale Unterbrechungen des Geschäftsprozesses.

Neben diesen zusätzlichen Phasen sind jedoch, um in der Sprache des Rational Unified Process zu bleiben, zusätzliche Disziplinen notwendig, s. Abb. 11.4:

- Development Disciplines:
  - andere Disziplinen – Die Ausdehnung des Rational Unified Process, s. Abb. 11.1, auf den Enterprise Unified Process, s. Abb. 11.3, benötigt eine konkrete Ausprägung und Anwendung der „traditionellen“ Entwicklungsmethodik durch den Rational Unified Process, obwohl die

Art der Ausprägung irrelevant ist, d.h. jede an den Rational Unified Process angelehnte Methodik ist für diese „linke Ecke“ verwendbar, ohne den Rest des Enterprise Unified Process zu beeinflussen.
  - Operations & Support Discipline – Das Arbeitsgebiet dieser Disziplin ist die Operation und Unterstützung der bestehenden Software. Wie die Erfahrungen aus dem Rechenzentrumsbetrieb bzw. Erfahrungen bei der Einführung von Softwarepaketen mit vielen Benutzern zeigen, kann diese Disziplin hochkomplex sein. Schon während der Konstruktionsphase des Vorgehensmodells muss die spätere Produktion vorgedacht und eventuell softwaretechnisch unterstützt werden. Üblicherweise bilden Betriebsdokumentation sowie Schulungen der zukünftigen Einsatzkräfte den Schwerpunkt in dieser Disziplin. Diese Tätigkeiten setzen sich bis in die Retirementphase fort.
- Enterprise Disciplines: Diese völlig neue Kategorie an Disziplinen macht Software für ein großes Unternehmen erst wirklich produktiv nutzbar; sie haben einen stärkeren Schwerpunkt auf den systemübergreifenden bzw. organisatorischen Teilen des Lebenszyklus. Zu den einzelnen Disziplinen in dieser Kategorie gehören:
  - Program & Portfolio Management – Die Übersicht und Steuerung der Projekte innerhalb des Gesamtunternehmens ist hier angesiedelt. Hier werden die Entscheidungen getroffen, welche entweder auf Grund der Unternehmensstrategie oder auf Grund betriebswirtschaftlicher Gesichtspunkte Aussagen über die Zielsetzung und Priorisierung von Software treffen.
  - Enterprise Modeling – Die Modellierung der Kernprozesse des Unternehmens auf abstraktem Niveau ist eine Form der vorweggenommenen Wiederverwendung.
  - Enterprise Architecture – Die Erschaffung einer applikationsübergreifenden Architektur ermöglicht eine Steuerung auf Architekturebene.
  - Enterprise Asset Management – Die Steuerung der Hard- und Softwaregüter bzw. Lizenzen, die das Unternehmen in der Vergangenheit erworben hat, gehört zum Asset Management. Wichtig ist hier der Aufbau einer Datenbasis, welche die Wechselwirkungen der einzelnen Assets untereinander beschreibt und wirkungsvoll kontrollieren kann. Typischerweise zählen auch Service Level Agreements zu den Enterprise Assets.
  - People Management – Dies ist wohl die schwierigste Disziplin: Die Führung der Mitarbeiter des Unternehmens, welche an dem System beteiligt sind. Da die meisten IT-Projekte an den Menschen und nicht an der Technik scheitern, ist diese Disziplin zwar eine der wichtigsten, aber gleichzeitig eine der am wenigsten[5] beachteten.

[5] Vermutlich wird sie gerade auf Grund ihrer Schwierigkeit verdrängt.

- Strategic Reuse Management – Eine echte Wiederverwendung kann nur geplant stattfinden, da die zufällige Wiederverwendung nicht effektiv steuerbar ist.
- Standards & Guidelines Management and Support – Die Weiterentwicklung der eingesetzten Standards ist im Zuge einer Reifung des Unternehmens immanent wichtig. Typischerweise sind ISO-9000 oder CMMI-Projekte hier stark vertreten.
- Software Process Management – Die Entwicklung des Softwareentwicklungsprozesses und der Qualitätssicherung sowie die Fortschreibung der Best Practices fallen in den Bereich dieser Disziplin. Letztendlich stellt sie eine Metaebene dar, da die Anpassungen des Enterprise Unified Process hier vorgedacht werden müssen.

11.4 Agiles Manifest

Im Frühjahr des Jahres 2001 trafen sich in Utah eine Reihe von Experten mit Erfahrung im Bereich eXtreme Programming. Im Zentrum der damaligen Diskussion stand die Frage, wie sich eXtreme Programming mit anderen so genannten Lightweight[6]-Prozessen vergleicht und was alle gemeinsam haben. Die Bezeichnung lightweight verführt zur Annahme eines unstrukturierten oder disziplinlosen Prozesses. Diese Annahme ist von Grund auf falsch, alle Lightweight-Prozesse, und besonders eXtreme Programming, verlangen ein sehr hohes Maß an Disziplin.

Das Ergebnis der Utah-Konferenz war das agile Manifest. Es lautet im Original:

> **The Manifesto for Agile Software Development**
>
> Seventeen anarchists agree:
>
> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
> - Individuals and interactions over processes and tools.
> - Working software over comprehensive documentation.
> - Customer collaboration over contract negotiation.
> - Responding to change over following a plan.
>
> That is, while we value the items on the right, we value the items on the left more. We follow the following principles:

[6] Salopp gesagt bezieht sich der Ausdruck *lightweight* für die *agilen* Methoden und *heavyweight* für V-Modell, RUP und EUP auf das „Gewicht“ der entstehenden Dokumentation, s. S. 288.

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity – the art of maximizing the amount of work not done – is essential.
- The best architectures, requirements and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Das agile Manifest ist in seinen Grundzügen die konsequente Abwendung von der klassischen strukturierten Analyse mit ihrem sehr dokumentenlastigen Vorgehensmodell, aber auch die Abwendung von anderen Prozessen, wie Rational Unified Process, Enterprise Unified Process und V-Modell.

Der zweite interessante Punkt ist der Wechsel von der Werkzeuggläubigkeit hin zu der Fähigkeit des Individuums, komplexe Probleme zu lösen. Auch das Outsourcing wird von der Gruppe sehr kritisch gesehen. Das agile Manifest ist stark von den Ideen selbstregulierender und selbstorganisierender Systeme beeinflusst, deshalb hat das Management nur eine Moderatorfunktion.

Obwohl die agilen Vorgehensweisen gegenüber dem V-Modell und dem Rational Unified Process sehr kurzfristig wirken, resultieren sie aus den Erfahrungen, welche im Umgang mit komplexen Systemen gewonnen werden konnten. Oft werden agile Vorgehensweisen, speziell von CMMI-Anhängern, als „Hacking“ bezeichnet. Dies ist nicht so! Der Ausdruck „Hacking“ bedeutet Tun ohne Nachdenken, also Aktionismus.

Da komplexe Systeme stets nichtintuitiv sind, versagen sehr oft langfristig geplante Strategien. Solche Strategien verlangen in der Regel ein hohes Maß an Kausalität, welches bei komplexen Systemen nicht gegeben ist. Die agilen

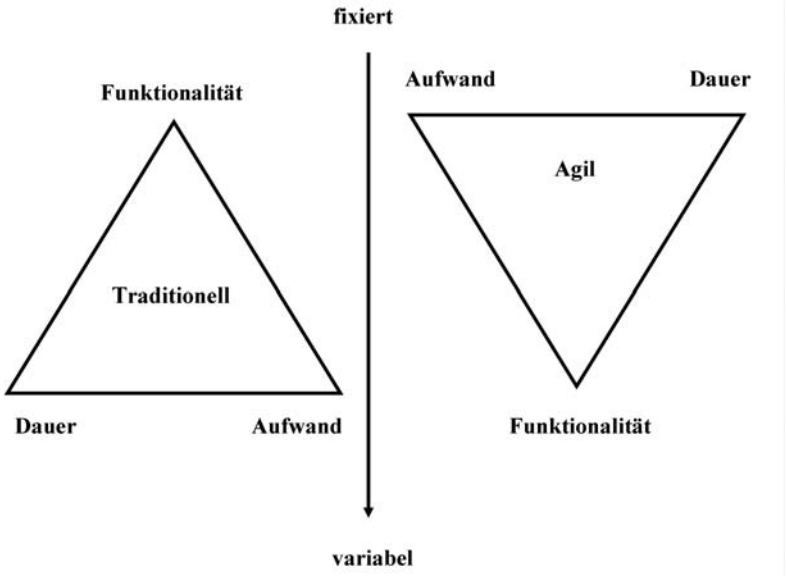

Abb. 11.5: Traditionelle gegenüber agilen Methoden

Vorgehensweisen verzichten darauf, da nach der agilen Philosophie das System sowieso viel zu komplex ist, um es kausal steuern zu können.

Die Zukunft ist stets von starker Unsicherheit geprägt, wodurch eine Planung prinzipiell nur geringen Wert hat. Softwareentwickler sind selbstverantwortlich und wollen möglichst gute Ergebnisse abliefern, deshalb ist eine aufwändige externe Qualitätssicherung nicht notwendig. Die Hintergründe für die entsprechenden Wertformulierungen sind relativ leicht zu sehen:

- Die meisten Projekte scheitern an Individuen, nicht an der eingesetzten Technik. Wenn man berücksichtigt, dass nur 20% aller IT-Projekte mehr oder minder erfolgreich abgeschlossen wurden, liegt hier ein breiter Erfahrungsschatz vor, welcher diese Beobachtung empirisch stützt. Dies erklärt den Fokus auf Menschen und Kommunikation und nicht auf Prozesse und Technik. Umgekehrt betrachtet: Die erfolgreichste Softwareentwicklung des letzten Jahrzehnts, Linux, beruhte weder auf einem gemeinsamen „postulierten“ Prozess, noch auf dem Einsatz von Werkzeugen außerhalb der Linuxplattform, sondern auf der intensiven Kooperation vieler anonymer individueller Softwareentwickler.
- Sehr ausführliche, und „formal“ korrekte, Dokumentation nimmt bei größeren Softwareprojekten leicht mehrere tausend Seiten ein. Eine Größe, die kein einzelner Softwareentwickler mehr liest oder komplett versteht. Die Sinnhaftigkeit solcher Dokumentationen wird zu Recht bezweifelt, daher liegt der Fokus auf dem Produkt, weniger auf der Dokumentation. Bei

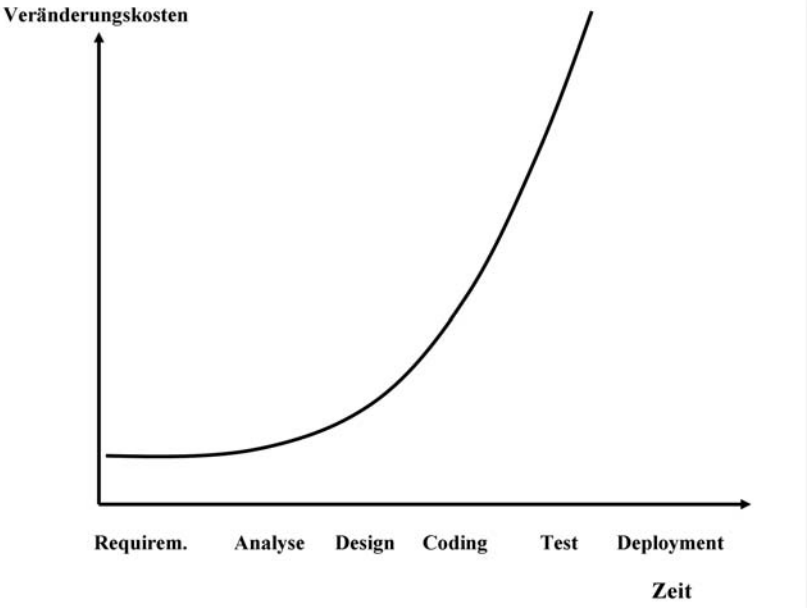

Abb. 11.6: Kosten in der traditionellen Entwicklung mit einem fast exponentiellen Verlauf

Legacysystemen ist dies zu einem gewissen Grad kontraproduktiv, da ein Verstehen ohne Dokumentation schwer ist. Allerdings ist eine überflüssige Dokumentation, welche nicht zielgerichtet ist, genauso hinderlich.

- Die meisten Spezifikationen sind nicht so eindeutig, wie es für eine Umsetzung günstig wäre. Diese Tatsache stellt auch ein Problem für das Offshoring[7] dar, da hier meist noch kulturelle Unterschiede hinzukommen, welche das „tacit knowledge gap“ vergrößern. Eine eindeutige, vollständige Spezifikation wäre via Parser dann automatisch umsetzbar, was eine der Zielvorstellungen hinter der Model Driven Architecture ist. Um diesen Tatsachen Rechnung zu tragen, wird die Zusammenarbeit mit dem Kunden und die Veränderung von Plänen als wichtiger Wert an sich angenommen.
- Sehr große Projekte haben oft Laufzeiten über zwei Jahre; dieser Zeithorizont ist jedoch oft größer als die Halbwertszeit einer eingesetzten Technologie, folglich führt ein starres Festhalten an einem einmal gefassten Plan zur Schaffung eines obsoleten Produktes. Die Maintenance eines Legacysystems dauert mehrere Jahrzehnte, entsprechend gut müssen die Mechanismen zur Kontrolle der Evolution des Systems sein.

Die einzelnen Prinzipien hinter dem agilen Manifest weisen auf eine langjährige praktische Erfahrung im Umgang mit Softwareentwicklung hin:

[7] Siehe auch Kap. 8.

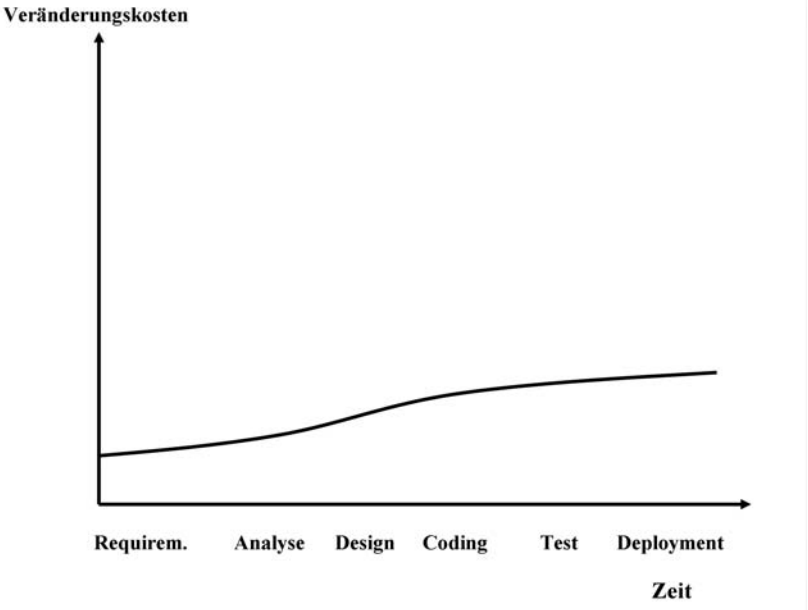

Abb. 11.7: Kosten in der agilen Entwicklung mit relativ flachem Verlauf

- Die traditionelle Projektmanagersicht ist, dass sich der Mehrwert des Kunden von selbst aus dem Projekterfolg ergibt. Dieser wiederum ist, in traditioneller Sicht, das Ergebnis der Planeinhaltung. Leider ändern sich die Umwelt und die Einflüsse auf ein Projekt heute so rapide, dass der entstehende Mehrwert für den Kunden öfters neu berechnet werden muss.
- Wandel ist in der traditionellen Methodik ein sehr problematisches Gebiet, da selbst die sorgfältigste Planung durch die anfallenden Änderungen stark gestört wird. Aus diesem Grund würden die meisten Projekte am liebsten ein Art Freeze Zone für sich selbst einrichten, um nicht mit dem Phänomen des Wandels konfrontiert zu werden. Die Praxis zeigt jedoch, dass sich zwischen 20-60% der Anforderungen an eine Software im Laufe des Projektes verändern, d.h. Softwareprojekte befinden sich im permanenten Wandel und Stasis ist in diesem Gebiet die große Ausnahme. Daher ist es sinnvoll, sich auf den Wandel einzustellen, s. Abb. 11.6 und 11.7.
- Obwohl iterativ-inkrementelle Vorgehensmodelle schon sehr lange im Einsatz sind und alle diese Vorgehensmodelle kurze Zyklen präferieren, leiden die meisten heutigen Projekte an dem Phänomen, nur wenige Zyklen auszuführen. Dafür sind diese Zyklen aber sehr lang. Diese langen Zykluszeiten ihrerseits produzieren die hohen Kosten, s. Abb. 11.6 und führen zu einem Stau an Änderungen. Im Gegensatz hierzu senken kurze Zyklen die Risiken, sowie die Kosten, s. Abb. 11.7.
- Ein großer Teil der Kommunikation findet auf nonverbaler Ebene statt, für diesen Teil der Kommunikation sind Dokumente wenig geeignet. Die-

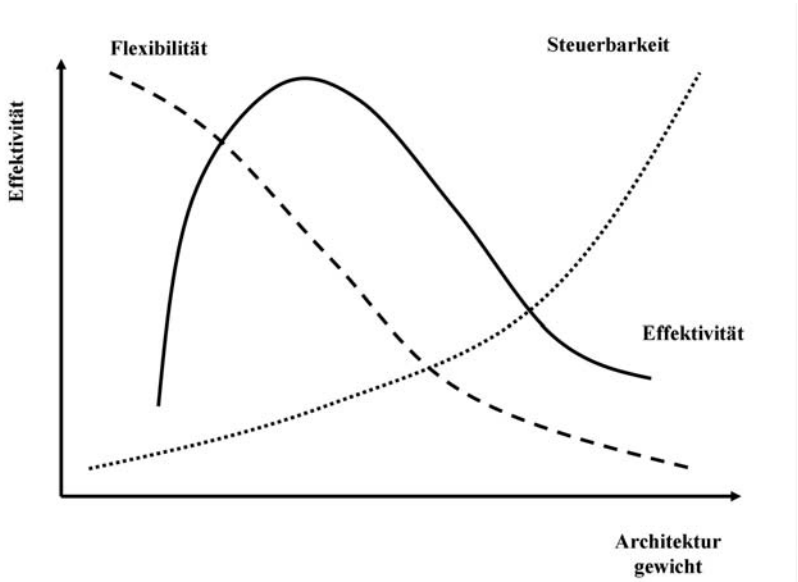

Abb. 11.8: Effektivität als Ergebnis von Flexibilität und Steuerbarkeit

se Beobachtung geht einher mit der Beobachtung über das Scheitern von Projekten. Ein anderer Punkt ist das Vorhandensein von tacit knowledge, dabei handelt es sich um stillschweigendes bzw. implizites Wissen. Ein solches Wissen lässt sich leider nicht dokumentieren, es ist aber für jedes Entwicklungsvorhaben extrem wichtig. Ein Teil der Unternehmenskultur basiert auf dieser Form des Wissens. Offensichtlich gibt es in jedem Unternehmen eine Menge so genannter ungeschriebener Gesetze, welche zum tacit knowledge gehören, s. Abb. 11.9.

- Viele Entwicklungen produzieren eine große Menge an Dokumentation, ohne tatsächlich etwas Lauffähiges vorweisen zu können. Je früher Defekte oder Fehlannahmen entdeckt werden, desto risikoärmer ist das ganze Vorgehen, was selbstverständlich auch für die Maintenance sinnvoll ist.
- Aus dem eXtreme Programming wurde das zeitlimitierte Modell übernommen. Entwicklung und Maintenance sind primär intellektuelle Leistungen und können nicht durch bloße Anwesenheit erzwungen werden. Bei allen Menschen ist die Zeit, in der sie kreativ sein können, sehr limitiert, von daher macht es wenig Sinn, Überstunden anzuhäufen.
- Im Gegensatz zum traditionellen Rapid Prototyping, welches den Schwerpunkt stärker auf das Rapid legt und damit qualitativ minderwertige Prototypen erstellt, liegt bei der agilen Methodik der Schwerpunkt auf qualitativ hochwertigem Design zusammen mit einem Up-to-date-Wissen über aktuelle Technik. Die Basisidee ist hierbei, das Design kontinuierlich zu

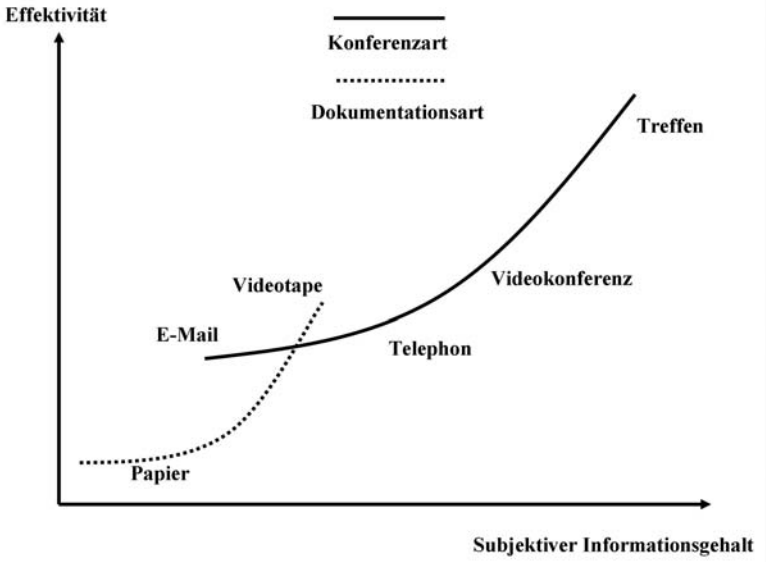

Abb. 11.9: Die verschiedenen Kommunikationsformen

verbessern. Die Idee des permanenten Refaktorings kann die Entropie senken.

- Die Erfahrung zeigt, dass Teams ohne allzu große Restriktionen und mit einer hohen Motivation sehr gute Ergebnisse produzieren können.
- Optimierung kann nur vom Team selbst ausgehen, damit sie effektiv ist.

Alle Ansätze des agilen Manifests folgen einem reichhaltigen Erfahrungsschatz und einer, zumindest empirisch, nachweisbaren Systematik. Mit dem Manifest wurde ein strategisches Programm für die zukünftige Entwicklung von IT-Systemen formuliert.

Die starke Orientierung hin zum Menschen und weg von der Technik ist eine zentrale Eigenschaft aller agilen Methodiken. Die Orientierung am Individuum Mensch zeigt sich auf verschiedene Weise in agilen Prozessen.

Eines der Schlüsselelemente ist, dass der Prozess angenommen und nicht aufgezwungen wird. In einer klassischen Umgebung werden der Softwareentwicklungsmannschaft vom Management oder einer Stabsstelle, z.B. der Abteilung Methoden&Verfahren, die Softwareprozesse oft aufgedrängt.[8] Diese Machtasymmetrie weckt sehr oft den Widerstand[9], besonders dann, wenn die Beteiligten des Managements sehr weit vom Entwicklungsteam weg sind, oder

[8] Siehe hierzu auch die Diskussion über komplexe Systeme und Macht, Abschn. 11.1.

[9] In manchen Unternehmen wird die Abteilung Methoden&Verfahren unter der Hand als „Kasperltruppe“ bezeichnet.

das Verfahren sehr abstrakt ist. Solche Versuche scheitern in der Regel, da der einzelne Mitarbeiter einen großen Teil seiner Energie darauf verwenden wird nachzuweisen, dass der Prozess so nicht funktionieren kann. Damit ein neuer Prozess angenommen werden kann, ist die aktive Teilnahme aller Beteiligten notwendig, sonst existiert bei allen Beteiligten kein Commitment.

In letzter Konsequenz bedeutet dies, dass nur die Softwareentwickler selbst die nötige Entscheidung treffen können, einer adaptiven Methodik zu folgen. Das trifft in besonderem Maße auf eXtreme Programming zu, welches ein sehr hohes Maß an Disziplin bei der Durchführung erfordert.

Wichtig für alle agilen Methodiken ist, dass die Softwareentwickler in der Lage sein müssen, alle technischen Entscheidungen zu treffen. Das eXtreme Programming folgt dieser Argumentation, indem es in seinem Planungsprozess festlegt, dass nur die Softwareentwickler schätzen dürfen, wie lange sie für eine bestimmte Arbeit brauchen.

In der Literatur findet sich oft der Unterschied zwischen „Lightweight"- und „Heavyweight"- Prozessen. Unter einem „Heavyweight"-Prozess wird eine Softwareentwicklung verstanden, die jede Menge an Artefakten produziert, welche von dem entsprechenden Vorgehensmodell, sei es V-Modell, RUP oder EUP, gefordert werden, gleichgültig ob diese Artefakte zum Ergebnis einer wartbaren Software beitragen oder nicht. Ein „Lightweight"-Prozess benutzt nur eine minimale Menge an Artefakten, nämlich gerade die Artefakte, welche notwendig zur Zielerreichung sind. Zwar versuchen alle agilen Methodiken, „Lightweight"-Prozesse zu bilden, umgekehrt ist jedoch nicht jeder „Lightweight"-Prozess ein agiler Prozess.

Obwohl sich keines der agilen Verfahren wirklich explizit mit der Maintenance beschäftigt, stehen sie ihr doch viel näher, als die klassischen oder Unified-Prozesse. Das übliche Vorgehen in der Maintenance mit seinem zeitlich sehr kurzen, spontan angetriebenen Bugfixing passt zu den agilen Verfahren, da diese primär kurze Zeiten und direkt verifizierbare Qualität fordern. Auch das permanente Refaktoring ist ein Aspekt, welcher der Maintenance zugute kommt.

Die klassischen und Unified-Prozesse lassen sich nur sehr schwer auf die permanente Anforderungsänderung und Fehlerbeseitigung einstellen, welche sehr stark während der Evolutions- und Servicingphase eines Legacysystems auftauchen.

11.4.1 EXtreme Programming

Von allen Methoden ist das eXtreme Programming, oft auch als XP bezeichnet, die älteste und bekannteste der agilen Vorgehensweisen. Die Wurzeln des eXtreme Programming liegen im Smalltalk, insbesondere in der engen Zusammenarbeit von Kent Beck und Ward Cunningham. Beide verfeinerten ihre Praktiken in vielen Projekten in den frühen 90ern und bauten ihre Ideen eines Softwareentwicklungsansatzes aus, der sowohl adaptiv als auch menschenorientiert ist. Der Sprung von informellen Vorgehensweisen zu einer eigenen

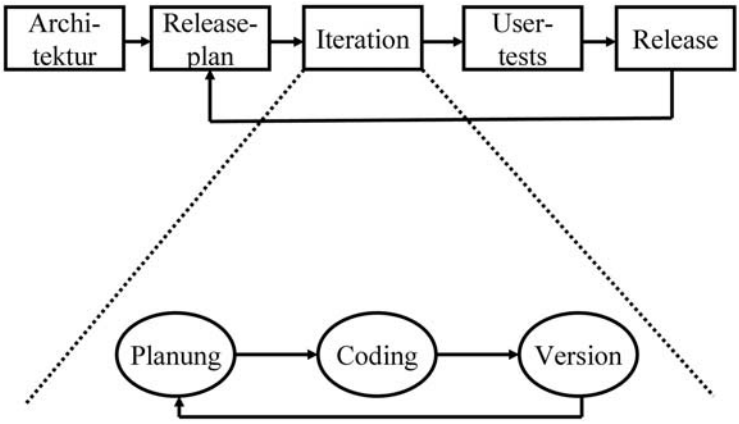

Abb. 11.10: Der EXtreme-Programming-Verlauf

Methodik geschah im Frühling 1996. Ein Review eines Abrechnungsprojektes bei Chrysler gilt heute als das erste offizielle EXtreme-Programming-Projekt. Das Verfahren des EXtreme-Programmings besitzt vier unverzichtbare Grundwerte:

- Kommunikation,
- Feedback,
- Einfachheit,
- Mut.

Der Rest ist eine Ansammlung von meist bekannten und mehrfach erfolgreich verwendeten Praktiken, die sich nach Ansicht von Kent Beck besonders zur Entwicklung eignen:

> *. . . none of the ideas in XP are new. Most are as old as programming. There is a sense in which XP is conservative – all its techniques have been proven . . .*

Das eXtreme Programming baut auf einem Dutzend Praktiken auf, welche alle EXtreme-Programming-Projekte befolgen sollten. Viele dieser Praktiken sind altbewährte und getestete Techniken, die außerhalb von eXtreme Programming jedoch nicht bewusst, sondern meistens nur instinktiv befolgt wurden. Das eXtreme Programming lässt einerseits diese erprobten Techniken wieder aufleben und verknüpft sie andererseits zu einem neuen Ganzen,

wobei sich diese Techniken im Sinne einer Emergenz, s. Abschn.11.1, teilweise gegenseitig verstärken.

Eine der herausragendsten Praktiken ist die starke Betonung des Testens. Obwohl alle Vorgehensmodelle das Testen erwähnen – es scheint zum „guten Ton“ zu gehören – leben es die meisten mit ziemlich geringer Priorität. Das eXtreme Programming jedoch erhebt das Testen zum Fundament der Entwicklung, wobei jeder Softwareentwickler während des Schreibens des Programms Tests schreibt. Die Tests werden in einen kontinuierlichen Integrations- und Erstellungsprozess einbezogen, was eine sehr stabile Basis für zukünftige Evolution hervorbringt. Umgekehrt betrachtet ist dieses Verfahren auch sehr gut geeignet, um damit eine permanente Evolution zu unterstützen.

Auf dieser Basis baut eXtreme Programming einen iterativen Evolutionsprozess auf, welcher darauf beruht, das jeweilige Basissystem mit jeder Iteration erneut zu refaktorisieren, was auch bei einem Legacysystem sehr gut funktioniert, da das Refaktoring die Entropie absenken kann. Die ganze Tätigkeit konzentriert sich auf die aktuelle Iteration, wobei keine Arbeit für antizipierte zukünftige Bedürfnisse gemacht wird. Das Ergebnis ist ein Prozess, der diszipliniert, aber aufregend ist, indem er Disziplin und Adaptivität auf eine solche Weise kombiniert, die ihn wahrscheinlich zur „am besten“ entwickelten aller agilen Methodiken macht.

Die Anwendung von eXtreme Programming im Umfeld eines Legacysystems kann nur eins bedeuten: Ziel muss es sein, ein hohes Maß an Einfachheit und Flexibilität zu erreichen. Dieses Ziel für den Einsatz in einem Legacysystem kann in mehreren Schritten erreicht werden:

1 Zunächst muss ein Überblick über die aktuell vorhandenen Quellen geschaffen werden. Technisch bedeutet dies, eine Online-Dokumentation des Sourcecodes zu generieren und diese allen zur Verfügung zu stellen. Diese Dokumentation muss selbstverständlich nach jeder Änderung neu erzeugt werden.
2 Involvierung der Endbenutzer; diese müssen direkte Teammitglieder werden. Die Endbenutzer haben in der Regel einen hohen Frustrationsstau und eine Menge an unerfüllten Wünschen, welche sie jetzt explizit äußern können. Für die Fachbereiche ist dieser Punkt oft verwirrend, da sie sich in der Vergangenheit als die Auftraggeber verstanden haben, welche sich nach der Auftragsvergabe zurücklehnen können und nicht in den aktuellen Prozess aktiv involviert sind.
3 Die Änderungswünsche müssen in kleine Iterationen aufgeteilt werden. Für jeden einzelnen Änderungswunsch muss eine Impact-Analyse, s. Abschn. 7.8, vorgenommen sowie die jeweils betroffenen Sourcecodedateien identifiziert werden. Wie in jedem eXtreme-Programming-Projekt wird auch hier die Schätzung des Aufwandes durch die Softwareentwickler vorgenommen und die Priorisierung durch die Endanwender.
4 Jetzt beginnt die Arbeit für jeden der einzelnen Releases. Jeder dieser Releases durchläuft folgende Schritte:

- Testcases für den Sourcecode schreiben, welche von dem Change Request betroffen sind, und die Tests durchführen.
- Der Sourcecode muss in den meisten Fällen einem Refaktoring unterzogen werden, damit falscher Code, schlechtes Design und Ähnliches entfernt werden kann. Auf das Refaktoringergebnis werden die Testcases aus dem letzten Schritt zur Überprüfung der Konsistenz angewandt.
- Jetzt ist der Sourcecode stabil genug zur Modifikation und er wird entsprechend dem Change Request abgeändert, s. Abb. 4.15.
- Ein erneutes Refaktoring, Retesting und Generierung der Dokumentation schließt sich an.

Diese Schritte entsprechen denen im regulären eXtreme Programming. Es gibt jedoch auch merkliche Differenzen:

- Die resultierende Produktivität pro Iteration ist deutlich niedriger als im Falle von Nicht-Legacysoftware. Hintergrund ist das Fehlen der Testcases, welche ja erst bei der versuchten Modifikation erzeugt werden, und die hohe Anfangskomplexität des Sourcecodes.
- Die hohe Komplexität des Sourcecodes impliziert auch eine erhöhte Zeit, welche für das Verstehen des Codes notwendig ist, wobei noch unklar ist, inwiefern das Pairprograming[10] die Fähigkeit zum Verstehen des Codes erhöht.

11.4.2 Adaptive Software Development

Das Adaptive Software Development, kurz ASD, behandelt hauptsächlich Probleme, die bei der Entwicklung großer und komplexer Systeme entstehen. Die Methodik ist eine inkrementelle und iterative Entwicklung mit der Nutzung von permanentem Rapid Prototyping. Das Adaptive Software Development „balanciert auf der Kante des Chaos“. Die Methodik hat als Zielsetzung, ein Framework anzubieten, welches Projekte gerade noch vor dem Chaos schützt, aber das Entstehen von Software und die Kreativität des einzelnen Softwareentwicklers sowie die des Teams nicht unterdrückt. Alle Adaptive-Software-Development-Projekte haben drei Grundsätze:

- Die Spekulation wird an Stelle einer Planung verwendet, weil bei einem Plan Unsicherheit als Schwäche angesehen wird. Eventuelle Abweichungen von der Planung stellen einen Misserfolg im klassischen Projektmanagement dar. Das Planen wird als ein Paradoxon, quasi wie ein Fremdkörper, in einer adaptiven Umgebung gesehen, denn die Ergebnisse sind natürlich

[10] Unter Pairprograming wird die Eigentümlichkeit verstanden, dass sich zwei Entwickler stets den Sourcecode gemeinsam betrachten und bearbeiten. Der eine nimmt die Änderungen vor und der andere sichert simultan die Qualität der Änderung.

unvorhersagbar. In der traditionellen Planung sind Planabweichungen Fehler, die korrigiert werden müssen. In einer adaptiven Umgebung jedoch zeigen Abweichungen den Weg zur richtigen Lösung.
- Das Zusammenarbeiten der Softwareentwickler hebt die Bedeutung des Teamworks hervor, damit sich ständig verändernde Systeme überhaupt entwickeln können. In dieser unvorhersagbaren Umgebung müssen die Menschen auf reichhaltige Art zusammenarbeiten, um mit der Unsicherheit umgehen zu können. Der Fokus des Managements muss daher weniger darin liegen, den Mitarbeitern zu sagen, was diese tun sollen, als vielmehr darin, zur Kommunikation anzuregen, damit die Mitarbeiter selbst kreative Lösungen hervorbringen können.
- Das Lernen hebt die Notwendigkeit hervor, Fehler zu erkennen und auf die Tatsache, dass sich Anforderungen während der Entwicklung ändern können, zu reagieren. Lernen als solches ist ein andauerndes und wichtiges Merkmal, das annimmt, dass Pläne und Entwürfe sich im Laufe der fortschreitenden Entwicklung ändern müssen.

Besonders bei den Legacysystemen ist das Lernen wichtig und zwar in dreifacher Hinsicht:

- Die Domäne muss erlernt werden, da sonst eine der notwendigen Voraussetzungen für eine erfolgreiche Evolution fehlt.
- Die Zieltechnologie muss erlernt werden. Ohne eine intensive Kenntnis der Zieltechnologie ist weder der Bau eines Produktes noch eine sinnvolle Form der Migration möglich.
- Die Implementierung des bestehenden Legacysystems muss erlernt werden. Die dritte notwendige Voraussetzung ist die ausreichende Kenntnis der Architektur, des Designs und des Sourcecodes des betreffenden Legacysystems.

11.4.3 Scrum

Bei Scrum handelt es sich um einen Lightweight-Managementprozess, also nicht um einen eigentlichen Softwareentwicklungsprozess, der sich auf Projekte von verschiedenen Größenordnungen anwenden lässt. Die Grundeigenschaften von Scrum sind:

- Kleine Teams, welche die Kommunikation und den informellen Informationsaustausch fördern.
- Anpassungsfähigkeit bezüglich Veränderungen von Technologie und Benutzeranforderungen.
- Häufige Erstellung von neuen Produktversionen, welche inspiziert, angepasst, getestet und dokumentiert werden können. Diese Grundeigenschaft ist eine ideale Voraussetzung zum Arbeiten mit Legacysystemen.

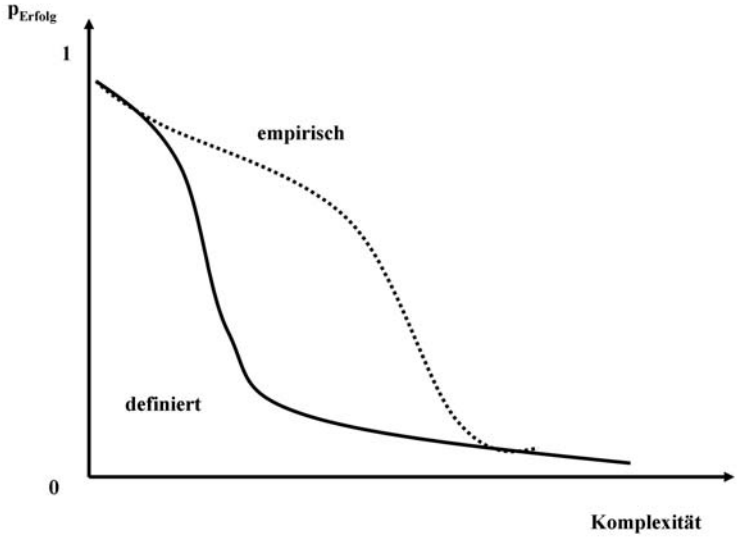

Abb. 11.11: Lieferwahrscheinlichkeit von definierten gegenüber empirischen Prozessen

- Aufteilung der zu erledigenden Arbeit in kleine, voneinander möglichst unabhängige Teilaufgaben – eine vom Prozess gesteuerte Form der Aufteilung in Pakete beeinflusst die Evolution und Maintenance positiv.
- Die Möglichkeit, ein Projekt jederzeit als beendet zu erklären, sei dies aus zeitlichen, finanziellen, wettbewerbstechnischen oder anderen Gründen.

Andere Modelle konzentrieren sich auf die Tatsache, dass ein definierter und wiederholbarer Prozess nur funktionieren kann, wenn man definierte und wiederholbare Probleme mit definierten Mitarbeitern in definierten und wiederholbaren Umgebungen lösen will. Im Gegensatz dazu sieht Scrum den Entwicklungsprozess als inhärent unvorhersagbar an, s. Abb. 11.11.

Scrum unterteilt ein Projekt in Iterationen von etwa einem Monat, welche Sprints genannt werden. Bevor man einen Sprint beginnt, definiert man die erforderliche Funktionalität, die der Sprint erzeugen soll und lässt dann das Team in Ruhe, um die geforderten Ergebnisse zu liefern. Der Zweck ist, die Anforderungen während des Sprints stabil zu halten. Obwohl dies eine Freeze Zone ist, ist sie doch relativ kurz.

Auch das Management behält eine wichtige Rolle während der Sprints. In der Methodik ist verankert, dass jeden Tag das gesamte Team ein kurzes Treffen abhält, welches Scrum[11] genannt wird und namensgebend ist. Während dieses Scrums werden dem Management die Blockaden präsentiert. Dies sind

[11] *Scrum* stammt aus der Umgangssprache und bedeutet so viel wie Gedränge.

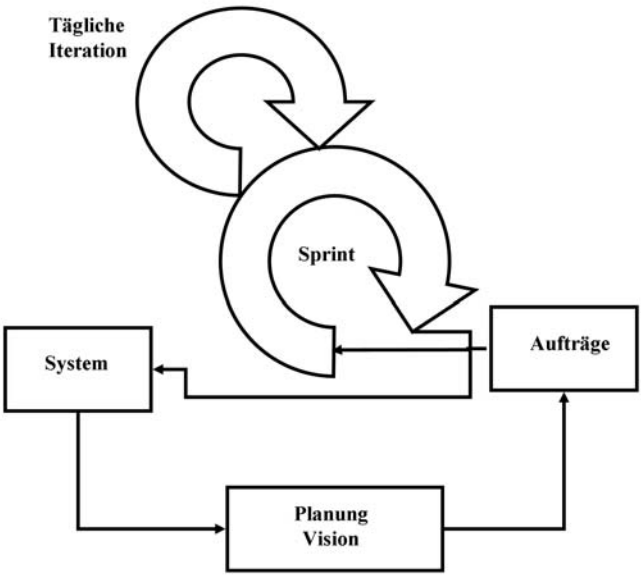

Abb. 11.12: Der allgemeine Scrum-Prozess

Hindernisse, welche das Management auflösen muss. Interessant ist hier die Umkehrung der klassischen Rolle: Das Management wird damit ein Dienstleister für das Entwicklungsteam.

Das Scrum wird sehr oft für die Maintenance von bestehender Software genutzt, da das Hinzufügen eines neuen Features auch immer den Start einer Iteration und das Beenden der Iteration die Existenz eines Produktes bedeutet, gleichgültig, ob es sich um ein neues Feature, ein Change Request oder die Behebung eines Defektes handelt. Das Vorgehen von Scrum lässt sich gut auf alle Typen der Maintenance übertragen.

11.4.4 Feature-Driven Development

Das Feature-Driven Development ist eine „customer oriented“ agile Methodik mit kleinen Teilresultaten. In einem größeren Softwareprojekt mit Java in Singapur wurde das Feature-Driven Development Mitte der neunziger Jahre entwickelt und erstmals eingesetzt. Obwohl die Methodik für den Einsatz mit einer objektorientierten Programmiersprache und mit UML als Modellierungssprache konzipiert worden ist, eignet sie sich auch gut für den Einsatz bei Produktlinien, s. Kap. 9. Die wesentlichen Merkmale dieser agilen Methodik sind einerseits die Zerlegung in fünf Prozesse und andererseits die kurzen Entwicklungsschritte sowie die laufende Fertigstellung von Teilprogrammen, die so genannten Features.

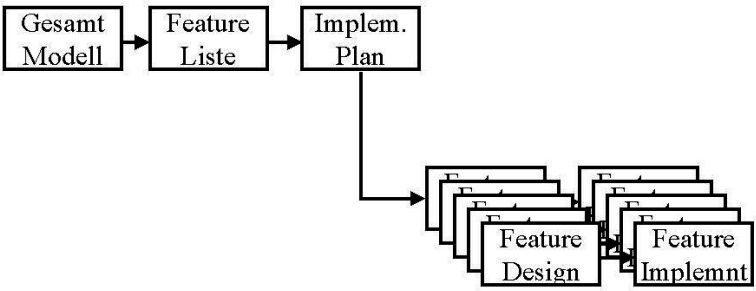

Abb. 11.13: Feature-Driven Development

Das zentrale und namensgebende Element der Methodik ist das Feature, ein kleiner Entwicklungsschritt, der stets nur einen Bruchteil des Gesamtsystems darstellt. Dieser kleine Entwicklungsschritt wird wie folgt definiert:

The features are small „useful in the eyes of the client" results.

Die einzelnen angesprochenen Features entsprechen den Assets bei den Produktlinien. Das gesamte Legacysystem wird in selbstständige kleine Features zerlegt, die in weiterer Folge einzeln weiter bearbeitet werden. Die Größe eines solchen Features wird durch die Bedingung eingeschränkt, dass ein Feature in maximal zwei Wochen entwickelt werden sollte, andernfalls muss es weiter aufgeteilt werden, bis diese Vorgabe erfüllt werden kann. Der Vorteil einer solchen Vorgabe wird in der Motivation der Softwareentwickler und in der Messbarkeit der Resultate gesehen. Durch die häufige Erreichung eines brauchbaren und sichtbaren Resultats, d.h. alle zwei Wochen, kann die Motivation der Softwareentwickler und der Kunden aufrecht erhalten werden. Der Abschluss eines Features kann zudem zur Feststellung des Projektfortschritts verwendet werden. Da ein Feature als kundenorientiertes Resultat definiert ist, ist die Fertigstellung eines Features ein dem Fachbereich oder Produktmanagement vermittelbarer Meilenstein.

Die fünf Prozesse des Feature-Driven Developments können in Projektplanung und Projektausführung eingeteilt werden, s. Abb. 11.13.

Die Planungsphase wird für jeden Release oder jedes Projekt einmal durchlaufen und beinhaltet:

1 die Entwicklung eines Gesamtmodells,
2 das Erstellen einer Feature-Liste,
3 die Planung pro Feature.

Jeder dieser drei Prozesse kann als Meilenstein für die Entstehung einer fertigen Projektdefinition angesehen werden. Nach deren Abschluss folgt die zweite Phase der Softwareentwicklung, die Projektausführung. Diese Phase umfasst die beiden letzten Prozesse, welche für jedes Feature eigens durchlaufen werden sollten und pro Feature nie länger als zwei Wochen dauern.

Die Ausführungsphase jedes Features enthält:

5 den Entwurf pro Feature,
6 die Implementierung pro Feature.

Auf diese Art und Weise kann ein Legacysystem langfristig evolviert oder transformiert werden.

11.4.5 Agile Modelle und Dokumente

Ein Modell ist eine Abstraktion des zu beschreibenden Systems mit Hilfe von Dokumentation und Diagrammen. Für die Diagramme hat sich UML als Industriestandard durchgesetzt. Aber Dokumentation kann alles sein, von einer beschriebenen Serviette oder einem Bierdeckel bis hin zu einem PDF-Dokument in einem DMS. Modelle sind neben dem Sourcecode die typischen Artefakte einer Softwareentwicklung. Interessanterweise ist das Fehlen von adäquaten Dokumenten geradezu ein Charakteristikum für Legacysysteme.

Bei den agilen Methodiken stellt sich jedoch stets die Frage: Wann ist ein Modell gut genug? Die Antwort lässt sich anhand von einigen Kriterien geben:

- Jedes Modell muss einen Sinn und Zweck haben, wenn es keine Verwendung für ein Diagramm oder Dokument gibt, wird es auch nicht erzeugt. Agile Methodiken streben hier einen Minimalismus an.
- Jedes agile Modell muss verständlich für die jeweilige Zielgruppe sein, dies bedeutet, dass fachliche Spezifikationen in der Sprache der Fachbereiche und technische Dokumentation wie z.B. der Datenbankentwurf in der Sprache der Datenbankadministratoren produziert werden.
- Agile Modelle sind hinreichend exakt. Hinreichend exakt bedeutet, dass es im Modell Toleranzen bzw. auch Fehler geben kann, solange davon grundsätzliche Themen nicht berührt sind. Dies ist kein Aufruf, qualitativ minderwertige Modelle zu bauen, sondern ein Hinweis auf die permanente Änderung der Umgebung und des Modells. Vollständige Exaktheit lässt sich in der Software nur an trivialen Beispielen zeigen, bei allen realen Modellen lässt sich deren Exaktheit nicht nachweisen. In der Systemtheorie, s. Abschn. 11.1, entspricht diese Forderung der Konsistenz der Teile.
- Agile Modelle sind hinreichend konsistent. Konsistenz in der Systemtheorie, s. Abschn. 11.1, entspricht der vollständigen Beschreibung der Wechselwirkungen der einzelnen Teile. In einer idealen Welt wäre alles konsistent,

aber die reale Welt ist nicht ideal, von daher reicht in den meisten Fällen eine gewisse Konsistenz aus.
- Agile Modelle sind hinreichend detailliert. Je detaillierter ein Modell, desto aufwändiger ist seine Erstellung, außerdem ist es viel schwerer zu verstehen als ein weniger detailliertes Modell. Für die meisten Entscheidungen reicht ein sehr niedriger Detaillierungsgrad aus.
- Agile Modelle müssen einen Mehrwert für das Unternehmen produzieren. Bei jedem Modell gilt es abzuwägen, welche Kosten das Modell produziert und welche Ersparnis es erzeugt. Die Mehrwertdiskussion ist eng mit der Sinnhaftigkeitsfrage verknüpft. Sinnlose Modelle haben einen negativen Mehrwert.
- Die Modelle müssen so einfach wie möglich sein. Je einfacher ein Modell, desto leichter ist es zu vermitteln, zu pflegen und zu produzieren.

11.5 Agile Maintenance

Dadurch, dass alle agilen Vorgehensmodelle das Vorhandensein der Veränderung als einen der Kernpunkte der Softwareentwicklung ansehen, sind sie auch gute Vorgehensmodelle für die Maintenance. Außerdem machen agile Methoden keine A-priori-Annahmen über die Entstehung der Software, was sie zusätzlich für Legacysoftwaresysteme prädestiniert. Die Fähigkeit, schnell und effizient auf Veränderungen zu reagieren, beeinflusst die Fähigkeit, adaptive und perfektive Maintenance, s. Kap. 7, vorzunehmen, in positiver Art und Weise, s. Abb. 11.6 und 11.7.

Die iterative Natur der agilen Verfahren erlaubt es ihnen, auch Veränderungen in den sehr späten Zeiten des Lebenszyklus gut verkraften zu können. Aber agile Methoden haben auch einen Nachteil: Innerhalb der agilen Entwicklungsmethodik ist Dokumentation ein lästiges Übel, welches oft vernachlässigt wird mit der Folge, dass die Maintenance erschwert wird. Theoretisch ist dies nicht der Fall, da das Wissen über das zu wartende System ja im Team bleibt, in der Praxis aber liegen die Turn-around-Zeiten für Mitarbeiter in Entwicklungs- und Maintenanceteams in der Größenordnung von wenigen Jahren; dies hat zur Folge, dass das Wissen über die Software ohne die Dokumentation tatsächlich verloren geht.

12

Architekturen und Sprachen

... Thanks, fortune, yet, that, after all my crosses,
Thou givest me somewhat to repair myself;
And though it was mine own, part of my heritage,
Which my dead father did bequeath to me.
With this strict charge, even as he left his life,
'Keep it, my Pericles; it hath been a shield
Twixt me and death;'
–and pointed to this brace;–
'For that it saved me, keep it; in like necessity–
The which the gods protect thee from!
–may defend thee.'
It kept where I kept, I so dearly loved it;
Till the rough seas, that spare not any man,
Took it in rage, though calm'd have given't again:
I thank thee for't: my shipwreck now's no ill,
Since I have here my father's gift in's will.

Pericles, Prince of Tyres,
William Shakespeare

12.1 Legacyarchitekturen

Die meisten Legacysysteme sind nach bestimmten architektonischen Richtlinien entstanden, welche aber oft durch eine Phase langanhaltender Evolution, s. Kap. 4, verwässert wurden. Nach mehreren „Renovierungszyklen“ ist oft die ursprüngliche Struktur des Systems nur noch schwer erkennbar. Trotzdem lassen sich fast alle Legacysysteme auf 3 grundsätzliche Systemarchitekturen zurückführen:

- Dateibasierte Systemarchitektur
- Datenbankzentrische Systemarchitektur
- Teleprocessingmonitor-Systemarchitektur

wobei hinter dem Begriff Systemarchitektur nicht der individuelle Aufbau der einzelnen Applikation – das wäre die Softwarearchitektur – betrachtet wird, sondern das Zusammenspiel der einzelnen Applikationen innerhalb eines Legacysystems.

12.1.1 Dateibasierte Systemarchitektur

Ein dateibasiertes System ist die älteste noch im Einsatz befindliche Legacysystemarchitektur, s. Abb. 12.1. Vorstufe hierzu waren die Punchcardsysteme oder Applikationen mit Lochstreifenlesern und -stanzern, oder auch Bandgeräte. Diese unterscheiden sich von den dateibasierten Systemen einzig darin, dass die I/O-Devices jetzt nur im exklusiven Lese- oder Schreibzugriff genutzt werden können. Strukturell sind sie folglich fast identisch, was sich auch in der Art des Lösungseinsatzes zeigt. In dieser Architektur geschieht der gesamte Datenaustausch über den Einsatz von Dateien. Die meisten solcher dateibasierten Legacysysteme entstanden ursprünglich sehr klein, d.h. es wurde zu Beginn ein einzelnes Programm erstellt, welches meist auf nur eine Datei zugriff. Dieses Programm war fast immer ein Batchprogramm. Oft war dessen einzige Aufgabe, eine größere Datenmenge zu sortieren oder bestimmte Werte in den Daten aufzufinden.

Die angesprochene Datei besteht logisch gesehen aus Sätzen, auch Records genannt. Jeder dieser Sätze hat einen Schlüssel und einen Inhalt, welcher eine spaltenorientierte Unterstruktur besitzt, s. Abb. 12.2. Es existieren auch „schlüssellose“ Dateien, in diesem Fall dient die Nummer des Records als „Schlüssel“. Die Dateien können rangieren von

- einfachen sequentiellen Dateien
- ISAM-Dateien – Index Sequential Access Method
- KSDS – Key Sequenced Datasets
- VSAM – Virtual Sequential Access Method

Alle diese Dateizugriffsmethoden unterscheiden sich zwar in der Implementierungstechnik, sind aber architektonisch identisch. Das Programm benutzt einen Schlüssel, um einen bestimmten Satz in der Datei zu finden, verarbeitet diesen Satz und schreibt ihn wieder in die gleiche Datei zurück, oder in eine andere Datei. Die Idee, über dateibasierte Zugriffe mit Schlüsseln zu agieren, setzt sich, genauer betrachtet, in den heutigen XML-Datenströmen[1] fort.

Aus dem Blickwinkel eines einzelnen Programms funktioniert das dateibasierte Verfahren recht gut, besonders, wenn man berücksichtigt, dass Sprachen wie COBOL eigene Befehle beinhalten, um sich genau dem Problem des schlüsselbasierten Dateizugriffs zu widmen.

Problematisch wird diese Form der Architektur, wenn sich die so geschaffenen Systeme ausdehnen. Dies kann in zwei Richtungen geschehen. Zum einen kann versucht werden, mehrere Instanzen des gleichen Programms auf einer Datei laufen zu lassen, und zum anderen kann versucht werden, die unterschiedlichsten neuen Programme mit den bestehenden Daten ausführen zu lassen. Beide Entwicklungsrichtungen führen langfristig zu Problemen. Bei diesen Problemen sind die entsprechenden Legacysysteme heute angekommen.

Wenn es mehrere Instanzen des gleichen Programms gibt, welche auf eine gemeinsame Datei zugreifen, so tauchen sehr schnell Inkonsistenzen auf, da

[1] Die ominösen win.ini-Dateien sind ein anderes Beispiel für einen solchen Einsatz.

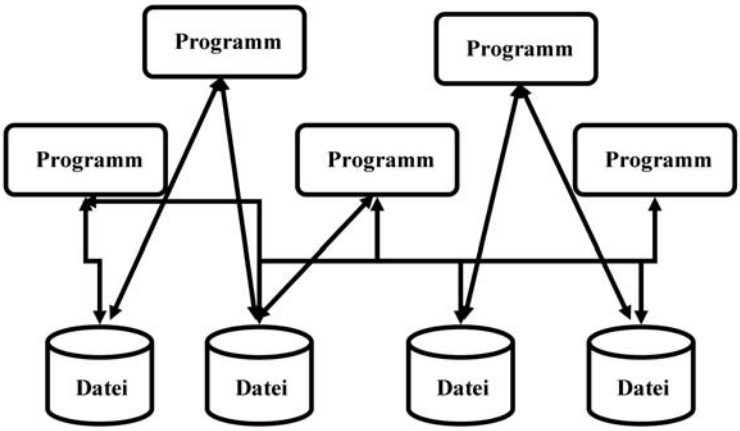

Abb. 12.1: Dateibasierte Systemarchitektur

jedes Programm annimmt, die Datei exklusiv für sich zu haben. Es werden Annahmen über Änderungen oder aktuelle Zustände getroffen, die während des parallelen Ablaufs keine Gültigkeit mehr besitzen. Der Ausweg aus diesem Problem ist die Serialisierung. Hierbei werden alle parallelen Änderungen sequentiell verarbeitet und so die Konsistenz sichergestellt. Bei Batchsystemen ist dies recht einfach, da viele Batchsprachen, wie beispielsweise JCL von IBM, die exklusive Zuteilung von Ressourcen, in diesem Fall Dateien, erzwingen. Der Nachteil ist hierbei jedoch die lange Laufzeit, da jetzt die einzelnen Batchjobs streng sequentiell gestartet werden müssen, bzw. das zweite Programm wartet mit seiner Verarbeitung, bis das erste Programm die entsprechende Ressource, sprich Datei, wieder freigegeben hat. Der andere mögliche Ausweg ist die Einführung von Transaktionen mit Hilfe eines Transaktionsmonitors, welcher sowohl die Konsistenz, als auch die Serialisierung sicherstellt.

Das zweite Problem entsteht, wenn unterschiedliche Programme, d.h. unterschiedliche Logiken auf dieselbe Datei zugreifen. Selbstverständlich existiert auch jetzt das Problem der Parallelität, allerdings wird es nun durch die unterschiedliche Evolution der verschiedenen Programme verstärkt. Selbst wenn beide Programme zu Beginn mit der identischen Datenstruktur starteten, benötigen sie nach einiger Zeit trotzdem unterschiedliche Datenstrukturen auf Grund der unterschiedlichen Evolution, d.h. ein einfacher Satzaufbau wie in Abb. 12.2 reicht nun nicht mehr für alle Programme aus. Die Reaktion auf dieses Dilemma war, redundante Dateien einzuführen, eventuell noch mit

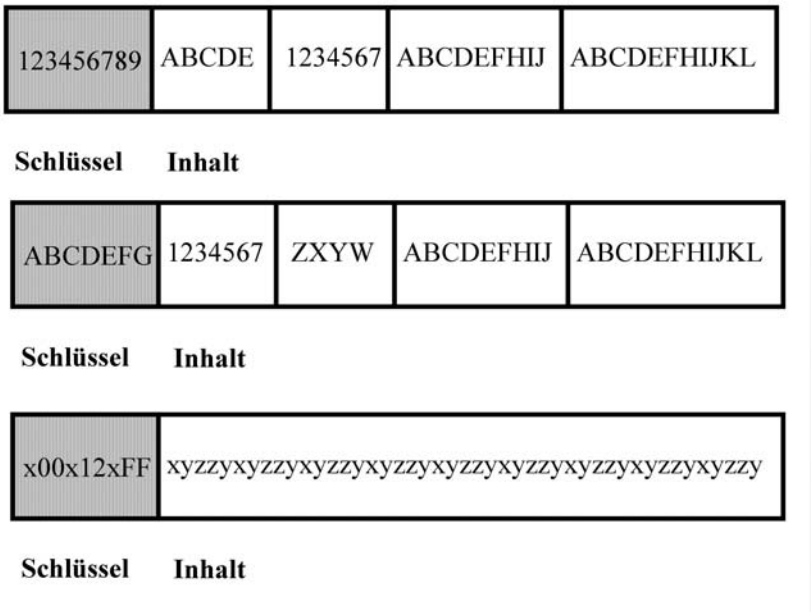

Abb. 12.2: Satzstruktur von Dateien

Übersetzungsprogrammen, den Datamappern[2], zwischen diesen beiden Satzstrukturen. Diese Entwicklung hatte zur Folge, dass immer mehr Dateien entstanden.

Neben diesen bewusst geförderten Redundanzen hat der Parallelzugriff mehrerer Programme auf dieselbe Datei eine Verminderung der Datenqualität zur Folge. Zwar kann, in manchen Systemen, der Datentyp des Schlüsselfeldes, in engen Grenzen, erzwungen werden, innerhalb der Daten des Satzes sind jedoch keine systemunterstützenden Datentypisierungen möglich, mit der Folge, dass ein beliebiger Datenwust in solchen Dateien steht, s. auch Abschn. 4.18.

Die Programme, welche für diese Architektur entwickelt wurden, haben typische Zugriffsmuster auf ihre jeweilige Datenhaltung, und zwar gleichgültig, ob es sich um Batch- oder Dialogprogramme handelt, da Dialoge in solchen Systemen nur für wenige, in der Regel unter 100, gleichzeitige Benutzer ausgelegt sind. Bei einer so kleinen Anzahl von Nutzern kann noch davon ausgegangen werden, dass einfache Lockmechanismen auf Satzebene innerhalb einer Datei ausreichen, um damit ein gewisses Maß an Sicherheit zu gewährleisten. Diese Lockmechanismen waren historisch gesehen der Grund, von einer reinen sequentiellen Zugriffslogik auf eine indexsequentielle Logik auszuweichen. Eine besonders leicht wiederzuerkennende Struktur in dateibasierten Legacysystemen ist die Schlüsselsuche mit Gruppenwechsel:

[2] Analoge Verfahren werden noch heute in der Datenmigration, s. Abschn. 5.11, eingesetzt.

```
 Fuellen des Suchkeys
 Finden des ersten Satzes;
MNEXT:  Verarbeitung des Satzes:
        Nachlesen einer anderen Datei:
        GOTO ADATEI;
MRETURN:    Lesen naechster Satz;
            Gruppenwechsel im Key?
            ja:     GOTO  MENDE;
            nein:   GOTO  MNEXT;
MENDE:
 Weitere Operationen;
...
...
...
ADATEI: Nachlesen mit Key andere Datei;
        Finden des ersten Satzes;
ANEXT:  Verarbeitung des Satzes:
        Nachlesen einer anderen Datei:
        GOTO ADATEI;
ARETURN:    Lesen naechster Satz;
            Gruppenwechsel im Key?
            ja:     GOTO  AENDE;
            nein:   GOTO  ANEXT;
```

Hierbei können die Kriterien für den Gruppenwechsel divers sein, d.h. alle Vergleiche sind zulässig. Je nach Implementierung können manche Systeme nur nach vorne, d.h. zu größeren Schlüsseln hin lesen, in diesen Fällen wird entweder vorab eine umgekehrte Verarbeitungsrichtung festgelegt, oder es wird ein zweiter Schlüssel auf dieselbe Datei mit umgekehrter Reihenfolge definiert. Ein weiteres Merkmal für diese Form der Datenhaltung ist das logische Löschen. In den dateibasierten Systemen wird im Grunde kein Satz gelöscht. Löschen in diesem Umfeld bedeutet den entsprechenden Satz als gelöscht zu markieren und später durch eine applikationsneutrale Spezialsoftware löschen zu lassen; dies ist selbstverständlich nur im Falle von Dateien mit gleichzeitigem Lese- und Schreibzugriff notwendig. Diese Löschsoftware kopiert meistens die unmarkierten Sätze in eine temporäre Datei und benennt diese am Ende in die Originaldatei um.

12.1.2 Datenbankzentrische Systemarchitektur

Der nächste Entwicklungsschritt nach den dateibasierten Systemen, sind die datenbankzentrischen Systeme. Charakteristisch für diese Form von Systemen ist das Vorhandensein einer Datenbank, s. Abb. 12.3. Logisch gesehen, ist eine Datenbank stets die Kombination aus einem Datenmodell, welches die Semantik der jeweiligen Domäne als Modell enthält, und einem Datenbankmanagementsystem, DBMS. Das DBMS liefert die Infrastruktur sowie

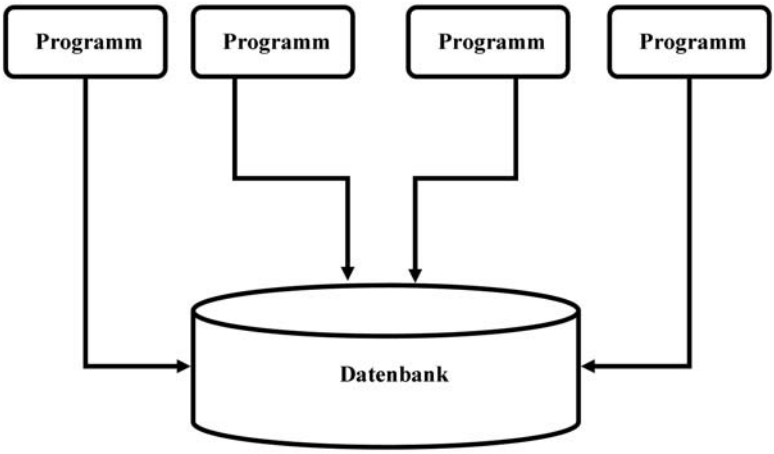

Abb. 12.3: Datenbankzentrische Systemarchitektur

Administration und Zugriffsrechte auf die Datenbank, wobei das eingesetzte DBMS stets applikationsneutral ist.

Die Datenbanken in Legacysystemen zerfallen in drei Kategorien:

- Hierarchische Datenbanken
- Netzwerk-Datenbanken
- Relationale Datenbanken

Zwar bieten die Datenbankmanagementsysteme in allen drei Fällen Funktionalität im Sinne der Ressourcenkontrolle, sowie Serialisierung an, jedoch unterscheiden sich die Implementierungsparadigmen der verschiedenen Systeme grundlegend voneinander.

Das DBMS ermöglicht, neben der Serialisierung und der Ressourcenkontrolle, auch ein gewisses Maß an Ausfallsicherheit, indem ein Transaktionsprotokoll zur Verfügung gestellt wird. Zusätzlich existieren in den unterschiedlichsten Datenbankmanagementsystemen sehr ausgeprägte Werkzeuge zur Unterstützung der Softwareentwicklung bzw. des Betriebs der Applikation. Eine Form der Unterstützung der Softwareentwicklung ist der Export des Datenmodells der Datenbank in ein Programmiersprachenformat, im Falle von COBOL als COBOL-Copybooks. Der Einsatz dieser Generierungswerkzeuge ermöglicht es bis zu einem gewissen Grad, Datentypen einzusetzen und damit das Redundanzdilemma des dateibasierten Zugriffes zu umgehen.

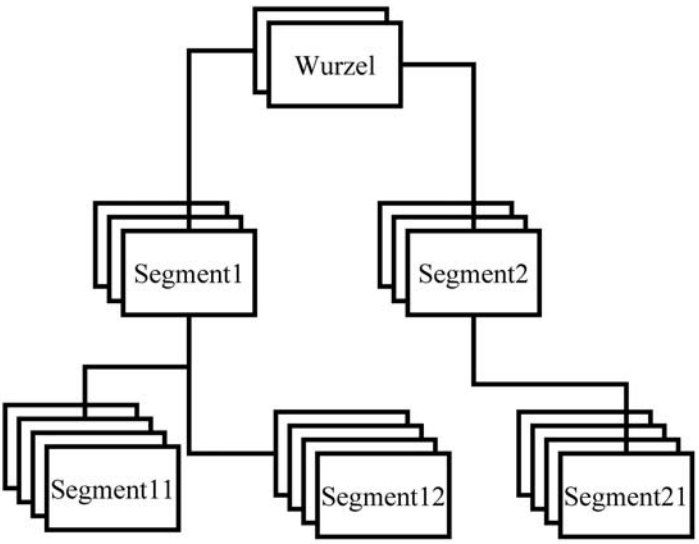

Abb. 12.4: Segmentstruktur einer hierarchischen Datenbank

Hierarchische Datenbanken

Hierarchische Datenbanken sind, wie es der Name schon suggeriert, dadurch gekennzeichnet, dass sie Hierarchien abbilden. Hierarchien sind spezielle Formen der Beziehungen von Objekten der Domäne, s. Abb. 12.4. Da fast alle Organisationen hierarchisch strukturiert sind, wird hier in gewisser Weise auch die Organisation als solche abgebildet. Dies ist konsistent mit Conway's Law, s. Abschn. 13.14.

Am Beispiel des Datenbankmanagementsystems IMS, heute von IBM, lassen sich hierarchische Datenbanken gut verdeutlichen. IMS, manchmal auch DL/I genannt, wurde von 1966-68 unter dem Namen ICS, Information Control System, durch IBM, North American Rockwell und Caterpillar Tractors entwickelt. Ende der sechziger Jahre wurde das ICS in IMS, Information Management System, umbenannt und wird von IBM bis heute ständig weiterentwickelt.

Innerhalb von IMS werden die einzelnen Teile der Hierarchie als Segmente bezeichnet. Diese Segmente sind untereinander verknüpft; besonders markant ist das Wurzelsegment, es dient als Startpunkt der Hierarchie. Die Datenbank ermöglicht, als einfachste Operation, das Auffinden eines Segmentes mit Hilfe eines Schlüssels, dies kann sogar über mehrere Hierarchiestufen hinweg geschehen. Von diesem so gefundenen Segment an wird dann die Datenbank sequentiell gelesen. Da alle Segmente eine eindeutige Reihenfolge innerhalb der Datenbank haben, was darauf zurückzuführen ist, dass sie intern als VSAM-

Dateien implementiert sind, wird von diesem Punkt an die Datenbank bis zum Ende gelesen. Jeder Wechsel in der Hierarchie gibt eine spezielle Datenbankmeldung an das entsprechende Applikationsprogramm. Diese Kombination des Schlüssellesens, gefolgt von einer sequentiellen Verarbeitung mit einer Abfrage auf den Wechsel von Hierarchieebenen, ist typisch für IMS-Programme:

```
 Fuellen des Suchkeys
 Finden des ersten Satzes:
                  Code=GU
NEXT:   Verarbeitung des Satzes:
        Lesen des naechstes Segment:
                Code=GN oder Code=GNP
        Statuscode des Zugriffs?
            ' ':            GOTO  NEXT;
            GE,GA,GK,GB:    GOTO  WECHSEL;
WECHSEL:
 Weitere Operationen;
...
```

Innerhalb von IMS-DB lassen sich einige Zugriffs- und Returncodes recht einfach merken:

- **GU** get unique – das Aufsetzen mit einem Suchkriterium
- **GN** get next – das Lesen des nächsten Satzes
- **GNP** get next within parent – das Lesen des nächsten Satzes bei festem Elternsegment
- **' '** – Segment erfolgreich gefunden
- **GE** – Segment nicht gefunden
- **GA** – Wechsel der Hierarchieebenen
- **GK** – gleiche Hierarchieebenen aber anderer Segmenttyp
- **GB** – Ende der Datenbank

De facto werden IMS-Datenbanken in den meisten Fällen aus COBOL-Programmen heraus angesteuert. Obwohl auch der Einsatz von Assembler oder PL/I möglich ist, da die entsprechenden Bibliotheken vorhanden sind, wurden die meisten IMS-Legacysysteme auf COBOL-Basis gebaut.

Obwohl IMS-Datenbanken als sehr schnell gelten, gibt es im Bereich großer Buchungssysteme, d.h. solcher mit sehr vielen Endanwendern, Performanzprobleme. Für eine spezielle Sorte von Datenbanken, und zwar solche mit sehr vielen Zugriffen, aber wenigen Daten pro Zugriff, was typisch für Buchungssysteme ist, gibt es spezielle IMS-Datenbanksysteme, die so genannten Fast-Path-Datenbanken. Diese erlauben nur einen Segmenttyp, das Wurzelsegment. Die hohe Performanz der Fast-Path-Datenbanken wird dadurch erreicht, dass die Datenbanken komplett in den Hauptspeicher einer Mainframe geladen werden und dort resident bleiben, außerdem ermöglicht der Verzicht auf unterschiedliche Segmenttypen eine schnellere Verarbeitung.

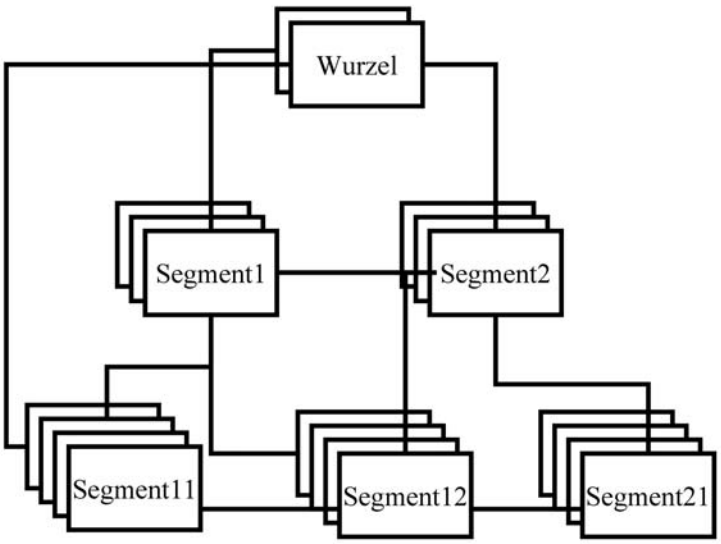

Abb. 12.5: Segmentstruktur einer CODASYL Datenbank

Netzwerk-Datenbanken

Die hierarchischen Datenbanksysteme gehen stets von einer strengen Zerlegung innerhalb der Domäne aus, d.h. es wird ein Baum von Segmenten aufgebaut. Leider entspricht dies nicht immer den Anforderungen an eine Domänenmodellierung. Oft ist es notwendig, Blätter und Knoten eines Baumes miteinander zu verknüpfen, wobei diese Verknüpfungen orthogonal zur Hierarchie sind. Da dies in den hierarchischen Datenbanken nur über Umwege, beispielsweise über das explizite Mitführen des Schlüssels des jeweiligen anderen Segmentes funktioniert, wurde nach einem eleganten Ausweg gesucht. Dieser wurde im Netzwerkmodell[3] gefunden. Ein Netzwerkmodell erlaubt es, neben den Hierarchien auch die Segmente direkt miteinander zu verknüpfen, s. Abb. 12.5.

Der Ansatz eines Netzwerkmodells wurde von der CODASYL Data Base Task Group, DBTG, entwickelt, daher resultiert auch der Name CODASYL-Datenbanksystem. Der Begriff CODASYL ist von **Co**nference on **Da**ta **Sy**stems **L**anguages abgeleitet. CODASYL entstand in den USA aus einer Konferenz am 28.-29. Mai 1959. Diese Konferenz hatte das Ziel, über die Entwicklung einer gemeinsamen Programmiersprache zu beraten, welche in kompatibler Weise auf Computern der verschiedenen Hersteller funktionieren sollte. Das erste Produkt von CODASYL war die Programmiersprache COBOL, ein weiteres die Entwicklung von standardisierten Verfahren zur Definition (Data

[3] Nicht zu verwechseln mit einem LAN, Local Area Network.

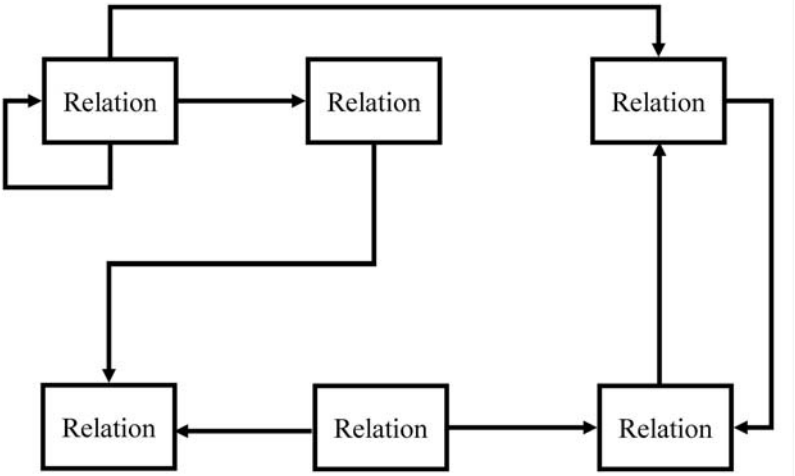

Abb. 12.6: Struktur einer relationalen Datenbank

Definition Language) und Bearbeitung (Data Manipulation Language) von Netzwerk-Datenbanksystemen. Netzwerk-Datenbanken werden, ähnlich der hierarchischen Datenbank IMS, vorwiegend für administrative Applikationen eingesetzt. Die bekanntesten Systeme sind:

- IDMS – Integrated Database Management System
- UDS – Universelles Datenbanksystem von Siemens
- IDS-2 von Honeywell-Bull

Das hierarchische Modell geht in das Netzwerkmodell über, wenn man zulässt, dass ein und dieselbe Satzart in verschiedenen Unterordnungsbeziehungen verwendet werden kann. Eine Trennung zwischen den Hierarchien, die für IMS kennzeichnend ist, gibt es beim Netzwerkmodell nicht. Die Hierarchien sind sämtlich in einem Netz enthalten, welches aus den einzelnen Segmenten Beziehungen zwischen diesen Segmenten erstellt. Auch bei Netzwerk-Datenbanken ist die übliche Verarbeitungsart die über Applikationsprogramme. Das Zugriffsmuster ist dem von IMS sehr verwandt, jedoch viel flexibler. Im Unterschied zu IMS kann der Einstieg grundsätzlich bei jedem Segment stattfinden. Auch Rückrelationen, als Rückwärtslesen bezeichnet, können so konstruiert werden.

Relationale Datenbanken

Die relationalen Datenbanken gehen auf Codd, 1970, zurück. Diese Datenbanken unterscheiden sich in folgenden Punkten grundlegend von den hierarchischen Datenbanken:

- Es gibt keine Hierarchien mehr. Die einzelnen Segmente, hier Relationen genannt, sind alle auf einer Ebene und werden via Fremdschlüssel miteinander verknüpft.
- Relationale Datenbanken besitzen ein eigenes Datentypschema, welches zur Laufzeit erzwungen wird. Dieses Datentypschema ist unabhängig von den Applikationsprogrammen und struktureller Bestandteil der Datenmodelle.
- Während hierarchische und Netzwerk-Datenbanksysteme für Einzelzugriffe konzipiert sind – es wird immer genau ein Segment gelesen – sind die relationalen Datenbanken für Mengenoperationen ausgelegt.

Eine relationale Datenbank, s. Abb. 12.6, besteht aus einer Reihe von Relationen, auch Tabellen genannt, und Beziehungen dieser Relationen, auch relationale Integrität genannt. Diese Relationen stellen die Daten dar, d.h. sie entsprechen den Segmenten in den anderen Datenbanksystemen. Die Verknüpfungen zwischen den Segmenten werden Foreign Keys oder Fremdschlüssel genannt, sie sichern die Integrität der relationalen Datenbank.

Die bekanntesten Vertreter der relationalen Datenbanken sind:

- DB2 von IBM
- Oracle von der gleichnamigen Firma
- Informix von IBM
- SQL-Server von Microsoft
- MySQL, ein Open-Source-Produkt
- ADABAS-D von der Software AG[4]

In den meisten großen Legacysystemen ist DB2 oder ADABAS anzutreffen, im Fall von ADABAS dann sehr häufig in Verbindung mit Natural, einer eigenständigen Programmiersprache.

Im Gegensatz zu den hierarchischen Datenbanken werden bei den relationalen Datenbanken unterschiedliche Zugriffsmuster für Batch und Online-Applikationen eingesetzt; dies hat primär Performanzgründe. Technisch gesehen benutzen relationale Datenbanken Indizes, dies sind Strukturen, welche Zuweisungen zwischen einem Schlüsselwert und dem physischen Speicherplatz der diesem Wert entsprechenden Tupeln darstellen. Ein Tupel ist eine mit konkreten Werten gefüllte Relation, sie entspricht einer Zeile in einer Tabelle. Diese Indizes sind inhaltlich betrachtet transparent für die Applikationen, da

[4] Durch das gekapselte Datadictionary wird ADABAS-D von Puristen nicht zu den reinen SQL-Systemen gezählt.

sie nur zur internen Datenbankverwaltung dienen. Allerdings gilt als Daumenregel: Je mehr Indizes existieren, desto schneller kann gelesen und je weniger Indizes, desto schneller können Sätze verändert werden.

Bei Online-Programmen, welche relationale Datenbanken einsetzen, entsteht meist ein Muster auf Grund der Eigenschaft, dass solche Programme einige Relationen lesen und nur ganz wenige Relationen inhaltlich verändern. Auf Grund der mengenorientierten Abfragesprache SQL, Structured Query Language, werden die Verknüpfungen schon in der Datenbank aufgelöst und als Ergebnismenge an das Applikationsprogramm gegeben.

Im ersten Schritt wird eine Menge deklariert, welche durch Boolesche Ausdrücke in der WHERE-Klausel festgelegt ist. Auf dieser Menge wird dann ein CURSOR, eine Art Zeiger definiert, welcher im OPEN geöffnet wird. Durch den FETCH-Befehl werden die einzelnen Sätze der Ergebnismenge dem Programm zur Verfügung gestellt, Operationen ausgeführt und das Endergebnis dann via UPDATE in die Datenbank zurückgeschrieben. Da die wenigsten Programmiersprachen für Legacysysteme dynamisch Speicherplatz allokieren können, muss die mengenorientierte Verarbeitung von SQL auf eine Einzelsatzverarbeitung analog den dateibasierten Systemen abgebildet werden. Im Falle des CURSORs wird auch vom Holen eines Records gesprochen.

```
      Define CURSOR as SELECT ... WHERE...
      OPEN CURSOR

NEXT:   FETCH Ergebnis;
        Falls keine Menge {
        COMMIT;
        Veraenderung:   GOTO UPDT;
        Neue Daten:     GOTO ISRT;
        Loeschen:       GOTO DELT;
        }
        Operationen;
        GOTO NEXT;

UPDT:   UPDATE Menge SET... WHERE...;
        COMMIT;
        GOTO END;
ISRT:   INSERT INTO Menge VALUES ...;
        COMMIT;
        GOTO END;
DELT:   DELETE Menge WHERE...;
        COMMIT;
        GOTO END;
END:    Ende der Verarbeitung
```

Zwar lässt sich theoretisch ein so genannter Cursor-for-Update deklarieren, so dass die selektierten Mengen direkt nach dem Lesen verändert wer-

den, ein solcher CURSOR hat jedoch den gravierenden Nachteil, große Teile der Tabelle für andere Benutzer zu blockieren, von daher werden in Online-Programmen die obigen Strukturen bevorzugt. Da das COMMIT die Transaktion beendet und die allokierten Mengen wieder freigibt, kann sich die Menge zwischen dem NEXT-Schritt und den ändernden Operationen UPDT, ISRT und DELT durch einen anderen Nutzer verändert haben. Diesem Problem wird durch die Verwendung von Zeitstempeln, welche die exakte Uhrzeit der letzten Veränderung des jeweiligen Satzes festhalten, begegnet. Diese Zeitstempel sind bei den Änderungen, UPDT und DELT, wiederum Bestandteil des Mengendefinitionskriteriums.

Im Fall eines Batches wird auf sehr komplexes Suchen in der Regel verzichtet, hier steht die Veränderung einzelner Relationen im Vordergrund. Bei sehr großen Tabellen, in denen sich viel ändern soll, wird oft ein Umweg über Datenbankwerkzeuge gegangen:

```
Entladen der Daten;

Sortieren der Daten, z.B. DFSORT;

Batchprogramm veraendert Datei;

Sortieren der veraenderten Daten, z.B. DFSORT,

Loeschen der Indizes;
Loeschen der Fremdschluessel;

Laden der Daten ohne Transaktionssicherheit;
Aufbau der Indizes;
Aufbau der Fremdschluessel;
```

Interessanterweise wird diese Erfahrung der Legacysysteme, zwischen Online-Programmen und Batches zu unterscheiden, in neueren Softwarepaketen häufig ignoriert. Speziell objektorientierte Entwicklungen tendieren dazu, eine Zugriffsschicht zwischen der Applikation und der Datenbank zu legen, welche eine Abbildung zwischen der objektorientierten Welt und der relationalen Datenhaltung vollzieht. Diese Zugriffsschichten sind in der Regel nicht für Batchprogramme und große Mengenverarbeitungen ausgelegt! Forciert wird dieses Problem durch Werkzeuge, welche automatische Datenbankmodelle aus dem objektorientierten Modell generieren: Zwar sind diese Modelle semantisch korrekt, zumindest aus Sicht des objektorientierten Modells, jedoch enthalten sie zuviele Relationen, um für große Datenmengen geeignet zu sein.

Es gibt noch eine zweite Beobachtung bezüglich inperformanter Zugriffsschichten. In manchen Unternehmen wurde die IMS oder CODASYL-Datenbank durch eine relationale Datenbank abgelöst. Dieser starke Wandel der Architektur konnte jedoch, meist aus Kostengründen, nur partiell vollzogen werden. Die Konsequenz war hier oft die Einführung einer Zugriffsschicht,

welche die traditionellen IMS- und CODASYL-Zugriffe intern auf relationale Datenbankzugriffe umstellt. Solche „Datenbankwrapper“, s. Abschn. 5.4.1, sind hochgradig inperformant mit der Konsequenz, dass das migrierte System oftmals stark abgelehnt wurde, da es subjektiv als viel zu langsam empfunden wurde. Dieses Dilemma lässt sich auch nicht beheben, da die Zugriffsmuster, siehe oben, einfach zu verschieden sind, um eine sinnvolle und performante algorithmische Abbildung zu gewährleisten.

12.1.3 Transaktionen

Transaktionen finden sich überall in der realen Welt, speziell im Bereich des Austausches von Geld oder Gütern. Innerhalb der Software ist eine Transaktion eine Abfolge von Veränderungen in einem System, meist einer Datenbank, welche entweder alle gemeinsam akzeptiert oder alle summarisch abgelehnt werden. Die Transaktionen ermöglichen es der Software, eine Datenbank so zu nutzen, als gehöre jeder Instanz eines beliebigen Programms die Datenbank alleine. Transaktionen sind also ein Mittel, um die Veränderung der Datenbank durch unterschiedliche Programme zu serialisieren, d.h. in eine Reihenfolge zu bringen. Entweder gelingt dies oder nicht, es gibt keine anderen Zustände.[5] Alle Transaktionen haben als garantierte Eigenschaften[6]:

A Atomizität – Die Transaktion wird entweder vollständig verarbeitet oder gar nicht.
C Konsistenz – Durch die Transaktion geht die Datenbank von einem konsistenten Zustand in einen anderen konsistenten Zustand über und alle strukturell definierten Zwangsbedingungen werden erfüllt.
I Isolation – Jeder Zwischenzustand, gleichgültig ob konsistent oder inkonsistent, ist für andere Transaktionen nicht sichtbar.
D Ausfallsicherheit – Das Ergebnis einer Transaktion bleibt unabhängig von Hard- oder Softwarefehlern stets erhalten.

Solche Transaktionen werden typischerweise explizit gesteuert, dabei gibt der Softwareentwickler an, von wo bis wo die Transaktion reicht:

```
Operation;
Operation;
BEGIN TRANSACTION;
    Operation;
        DB-OPERATION;
        DB-OPERATION;
        DB-OPERATION;
    IF OKAY THEN
```

[5] Zumindest theoretisch. In der Praxis kann ein Transaktionskonzept auch umgangen werden.

[6] Oft als ACID-Eigenschaften bezeichnet. Nach: **A**tomicity, **C**onsistency, **I**solation, **D**urability.

```
        COMMIT TRANSACTION;
    ELSE
        ROLLBACK TRANSACTION;
    END-IF
Operation;
```

Solche Transaktionsmuster gibt es in verschiedenen Ausprägungen. Meist handelt es sich bei Legacysystemen um obige flache Transaktionen. Es sind jedoch auch Systeme bekannt, welche solche Transaktionen schachteln können. Ein gewisser Sonderfall ist das 2-Phase-Commit, bei diesem werden zwei Datenbanken angesprochen. Beide liegen jedoch in einer flachen Transaktion. Die Koordinierung der jeweiligen Datenbanken übernimmt aber der Transaktionsmanager der Datenbanken und nicht das Applikationsprogramm.

Eine abgewandelte Form ist die der Transactionqueues, s. auch Abschn. 12.7; hierbei werden zwischen zwei Programmen In- und Outputqueues eingesetzt, welche innerhalb einer Queue eine gewisse Transaktionsfähigkeit sicherstellen, jedoch nicht queueübergreifend. Queuing heißt, dass Programme über Queues miteinander kommunizieren. Dadurch ist es nicht notwendig, dass diese Programme zeitlich parallel ausgeführt werden. Eine Queue bezeichnet eine Datenstruktur, welche in der Lage ist, Nachrichten zu speichern. Auf einer Queue können entweder die Applikationen, oder andere Queue-Manager Nachrichten ablegen. Die Queues existieren unabhängig von den Applikationen, welche diese Queues benutzen. Die Queues selbst können im Hauptspeicher oder aber, persistent, auf jedem beliebigen Medium liegen. Jede Queue gehört zu genau einem Queue-Manager. Letzterer ist verantwortlich für die Verwaltung der Queues, er legt die empfangenen Messages auf der entsprechenden Zielqueue ab. Queues sind entweder lokal, d.h. sie existieren in ihrem lokalen System, oder nichtlokal, dann nennt man sie remote. Eine Remote-Queue ist einem anderen Queue-Manager, der nicht zu dem lokalen System gehört, zugeordnet. Die Applikationen stellen Nachrichten in die Queues bzw. empfangen welche aus den Queues. Dabei benutzen die Applikationen den Queue-Manager; damit kann eine Applikation eine Nachricht an eine andere Queue senden und eine andere Applikation diese Nachricht von derselben Queue empfangen. Beim asynchronen Messaging führt die sendende Applikation ihre eigene Verarbeitung weiter aus, ohne dass sie auf eine Antwort bezüglich der gesendeten Nachrichten wartet. Im Gegensatz dazu warten beim synchronen Messaging die sendenden Applikationen auf eine Antwort, bevor sie die Verarbeitung fortsetzen.

12.1.4 Teleprocessingmonitor-Systemarchitektur

Ein Teleprocessingmonitor erlaubt es, eine große Anzahl von Terminals scheinbar gleichzeitig zu bedienen. Die bekanntesten Vertreter dieses Programmtypus sind IMS/DC und CICS, Customer Information Control System, beide von IBM. Ihre Aufgabe ist es, den Input von verschiedenen Quellen zu puffern und diesen dann zu verarbeiten, s. Abb. 12.7. Strukturell betrachtet ist

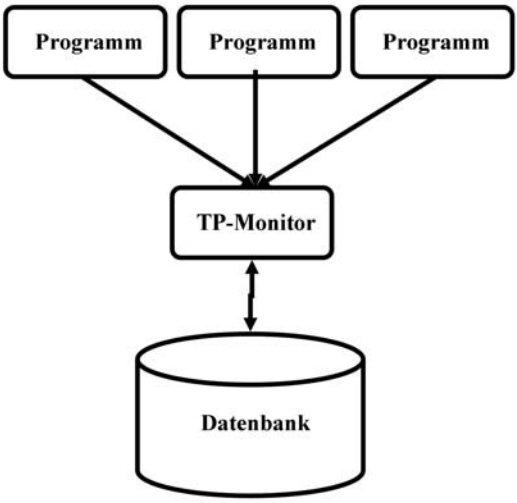

Abb. 12.7: Teleprocessingmonitor-Systemarchitektur

der Hauptunterschied zwischen IMS/DC und CICS, dass im IMS/DC-Fall das System wie eine Art Bibliothek gerufen wird, während im CICS-Fall das CICS das Hauptprogramm darstellt und nur die implementierten Unterprogramme aufruft. Obwohl die Hauptimplementierungssprache im Teleprocessingumfeld COBOL ist, besteht ein durchschnittliches CICS-Online-Programm nur aus 50-70% COBOL-Statements, der Rest sind CICS-Befehle.

Diesen TP-Systemen, Teleprocessingsystemen, ist gemein, dass sie generell reentrantfähig programmiert werden. Dies bedeutet, dass die einzelne Instanz eines Programms kein „Gedächtnis“ besitzt und sich einer Datenbank bedienen muss, damit sie sich an den letzten Zustand „erinnern“ kann. Ablauftechnisch sind die reentrantfähigen Applikationen Unterprogramme, welche vom TP-Monitor aufgerufen werden. Da die Applikationen Unterprogramme sind und damit im gleichen Adressraum des Betriebssystems liegen wie der TP-Monitor, sind sie auch in der Lage, eine komplette Instanz eines TP-Monitors, in der Fachsprache eine IMS- oder CICS-Region, zum Absturz zu bewegen. Ein Fehler im Programm reicht dafür aus!

Das angeschlossene Terminal benutzt einen Kanal, um seine Daten in einen Inputbuffer, im IMS/DC-Kontext Message-Queue genannt, zu stellen, s. Abb. 12.8. Hierbei muss es sich nicht unbedingt um ein traditionelles 3270-Terminal handeln, der Kanal kann von einem beliebigen Programm aus gefüllt werden. Das eigentliche Applikationsprogramm liest nun die Message-Queue aus – für die Applikation ist eine Message-Queue eine spezielle IMS-Datenbank mit einem besonderen Namen –, führt einige Operationen durch und schreibt

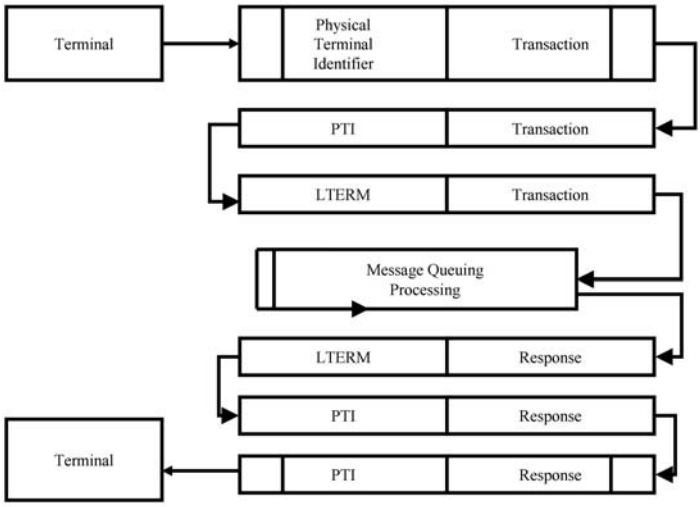

Abb. 12.8: Teleprocessingmonitorablauf

in die Outputqueue, welche wiederum eine IMS-Datenbank ist, zurück. Die Kommunikation mit den Terminals oder Clients übernimmt vollständig der TP-Monitor IMS/DC.

Durch die Reentrantfähigkeit besitzt das Programm kein Gedächtnis über vorherige Zustände, es ist zustandslos. Damit man trotzdem noch in der Lage ist, zustandsbehaftet zu arbeiten, wird eine spezielle Datenbank, die SPA, Scratch Pad Area, angesprochen. Diese speichert einen unstrukturierten Datenbereich, welcher kleiner als 32 kB sein muss, zwischen, und überlässt es dem Programm, diesen selbst zu strukturieren und zu verwalten, s. Abb. 12.9. Eine allgemeine Programmstruktur eines IMS/DC-Programms sieht wie folgt aus:

```
ENTRY TP-NAME;
GET-UNIQUE INPUTQUEUE;
GET-UNIQUE Srcatch-Pad-DB;
Wiederherstellung des Zustands;
Operation;
...
Operation;
INSERT  Srcatch-Pad-DB;
INSERT  OUTPUTQUEUE;
```

Innerhalb der oben angesprochenen Operationen sind natürlich auch Datenbankzugriffe oder andere Aufrufe möglich, in diesem Fall stehen alle Ressourcen, inklusive des Datenbanktransaktionsmonitors, unter der Kontrolle

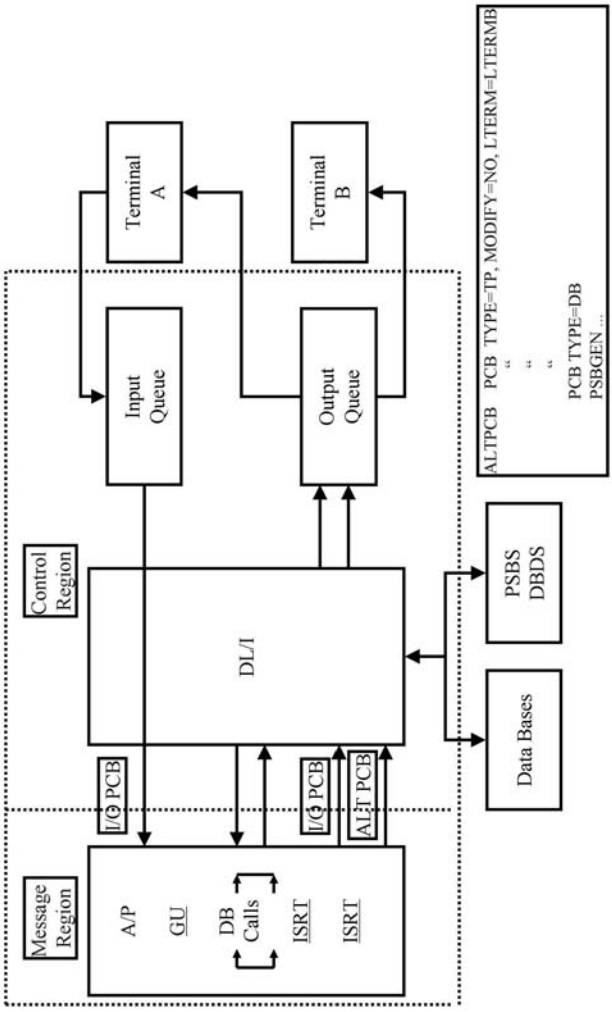

Abb. 12.9: Teleprocessingmonitorablauf, detailliertes Bild für IMS/DC

des TP-Monitors. Die Technik der Scratch-Pad-Area kann auch genutzt werden, um von einem Programm aus innerhalb des TP-Monitors ein anderes Programm aufzurufen, quasi Transaktionen zu verketten. Auf diese Weise ist sogar eine Rekursion[7] im COBOL-Umfeld implementierbar.

[7] Rekursionen sind in anderen Sprachen wie C oder C++ sehr weit verbreitet. Interessanterweise sind Rekursionen auch in Java möglich, sie werden jedoch äußerst selten genutzt.

12.1.5 Sonstige Eigenschaften

Andere typische Konstrukte innerhalb von Legacysoftware lassen sich oft auf frühere Hardware- bzw. Softwaregegebenheiten zurückführen. Dazu zählen:

- I/O-Batching: Alte Mainframemaschinen lagerten Programme aus dem Hauptspeicher aus, um die Input- und Outputoperationen durchzuführen. Da diese Operationen relativ teuer waren, gewöhnte man es sich an, ganze Blöcke zu lesen und auch zu schreiben, das I/O-Batching. Solche Batchzyklen führen dazu, dass beispielsweise erst nach jedem 100ten Satz geschrieben wird. Eine Parallele hierzu ist bei sehr batchlastigen Datenbankprogrammen zu beobachten, auch hier wird erst nach einigen hundert Sätzen ein Commit durchgeführt.
- Scratchfiles: Auf Grund der geringen Hauptspeicherausstattung alter Rechner war es notwendig, Daten zwischenzuspeichern. Dies geschah in temporären Dateien, den Scratchfiles.
- Overlays: In manchen Rechnerarchitekturen war die Menge an adressierbarem Hauptspeicher drastisch limitiert. Um dies zu umgehen wurde die Overlaytechnik eingesetzt, dabei wurde ein bestimmter Bereich im Hauptspeicher des Programms mehrfach in verschiedenen Kontexten genutzt.
- Gepackte Werte: Um Hauptspeicher zu sparen, wurden oft sehr ausgedehnte Schemata zur Kodierung von Informationen genutzt und diese dann als Zahl gespeichert. Beispielsweise sagt die erste Zahl etwas über die Art von Objekten aus.

Das Verstehen von Legacysystemen ist nicht ganz einfach, da jede Generation von Softwareentwicklern eigene „Dialekte“ entwickelt hat. Beispielsweise war es in den sechziger Jahren üblich, die Anweisungen für den Listendrucker direkt im Sourcecode zu verankern, was zu kryptischen Zahlen wie 1 für Seitenvorschub, oder 0 für Leerzeile in der ersten Spalte eines Programms führte. Dieser Dialekt ist für heutige Softwareentwickler nicht mehr verständlich, zum Teil sogar unbekannt, da die Notwendigkeit für den Einsatz solcher Mittel fehlt. Eine Legacysoftware mit einem Alter von ca. 30 Jahren hat mehrere Generationen[8] von Werkzeugen und Softwareentwicklern gesehen, welche alle ihre Spuren im Sourcecode hinterlassen haben, mit der Folge, dass heute nur die „gebündelte Historie“ sichtbar ist. Manche Autoren nennen diese „gebündelte Historie“ ein „Lava-Flow-Pattern“; in ähnlicher Weise, wie flüssige Lava erkaltet und dabei unbewegliche Strukturen bildet, die quasi „einfrieren“, und anschließend von einem neuen Ausbruch überdeckt werden, bleiben die Spuren ehemaliger Entwickler im Sourcecode vorhanden.

[8] Bei einer Lebensdauer von 5-10 Jahren pro Werkzeug sind dies immerhin 3-6 Werkzeuggenerationen.

12.2 Legacysprachen

Die Programmiersprachen werden üblicherweise in vier Generationen eingeteilt:

- First-Generation Languages – Das Programmieren in Binärform, was extrem hardwarenah ist.
- Second-Generation Languages – Darunter fallen alle Sprachen, welche symbolische Adressen benutzen, speziell Assembler. Bei der Sprache C existieren Diskussionen darüber, ob es sich um eine Second-Generation oder eine Third-Generation Language handelt. Die starke Hardwarenähe legt es nahe, C hier einzuordnen.
- Third-Generation Languages – Diese werden auch als Hochsprachen bezeichnet, herunter fallen die klassischen Sprachen wie:
  - Pascal
  - COBOL
  - Fortran
  - Ada
  - PL/I
  - RPG
- Fourth-Generation Languages – Diese werden auch als 4GL bezeichnet. Alle 4GL sind proprietär und oft nichtprozedural.

Obwohl sich der Ausdruck Legacysoftware sehr viel stärker auf das Phänomen des Alters und der Maintainability einer Software an sich bezieht, als auf die konkrete Implementierungssprache, gibt es doch einige typische Sprachen, in denen Legacysoftware geschrieben wurde. Diese typischen Sprachen sind:

- Assembler – Assembler ist typisch für Systeme aus den sechziger Jahren. Der jeweils gewählte Dialekt ist dabei sehr maschinenabhängig. Die bekanntesten, noch heute vorzufindenden Dialekte sind: /370-Assembler bei einer Mainframe und 8080-Assembler beim PC.
- COBOL – Die Sprache COBOL ist die Standardsprache für betriebswirtschaftliche Applikationen. Von daher ist die COBOL-basierte Legacysoftware im gesamten Bereich der Buchhaltung, der Planung, Lagerwirtschaft und ähnlichen Domänen vorzufinden.
- Fortran – Die Sprache Fortran ist vor allen Dingen im wissenschaftlichen Bereich sowie im Umfeld der Numerik vorzufinden.
- C – Größere C-Programme existieren in den Bereichen CAD-Systeme, Oberflächenentwicklung und, bedingt durch die Hardwarenähe, im gesamten Telekommunikationssektor.

Selbstverständlich gibt es auch noch andere Programmiersprachen[9], welche in einer Legacysoftware vorkommen. Außer den diversen 4GL-Sprachen findet

[9] Historisch gesehen ist die Gesamtanzahl von entstandenen Programmiersprachen deutlich größer als 100.

man folgende Programmiersprachen in verschiedenen Legacysystemen öfters vor:

- PL/I
- RPG
- CSP
- Pascal
- C++
- C
- APL

Statistisch gesehen spielen diese Sprachen aber, wie auch alle 4GL-Sprachen, nur eine untergeordnete Rolle.[10]

Tab. 12.1: Sprachvergleich

Sprache	**Heterogene Strukturen**	**Festkomma-zahlen**	**Report-generator**	**Datenbank**
COBOL	ja	ja	teilw.	ja
Fortran	teilw.	nein	nein	nein
C / C++	teilw.	nein	nein	ja
Visual Basic	nein	nein	nein	teilw.[11]
Java	teilw.	nein	nein	teilw.[12]

Betriebswirtschaftliche Applikationen bilden den größten Anteil an der vorhandenen Legacysoftware. Eine solche Applikation stellt verschiedene Anforderungen an die jeweilige Programmiersprache, die zum Einsatz kommt. In Tab. 12.1 wurden verschiedene Sprachen nach den in ihrem originären Sprachumfang vorhandenen Elementen dargestellt.

Das am schwierigsten zu verstehende Konstrukt in der Implementierung ist die Nutzung globaler Variablen. Da in COBOL alle Variablen global zur Verfügung stehen, sind es Softwareentwickler gewohnt, mit diesem Konzept gut zu arbeiten und sehen die entsprechenden Maßnahmen im Code, speziell durch die Namensgebung, vor. In anderen Sprachen wie C oder Fortran ist dies anders. Wenn hier globale Variablen eingesetzt werden, so resultiert dies immer in dem Problem, dass Änderungen starke nichtlokale Auswirkungen haben. In C, wie auch in Fortran, sind die Entwickler gewohnt, über Funktionsaufrufe Algorithmen abzubilden, globale Variablen sind ein Weg, genau diesen Mechanismus zu umgehen.

[10] Ein schwacher Trost für ein Unternehmen, welches ein Legacysystem auf dieser Basis im Einsatz hat.
[11] via ODBC.
[12] via JDBC.

12.2.1 COBOL

Die Sprache COBOL, **Co**mmon **B**usiness **O**riented **L**anguage, wurde eingeführt, um betriebswirtschaftliche Applikationen entwickeln zu können, insofern ist COBOL eine domänenspezifische Sprache, s. Tab. 12.1. Sie ist primär für die gute Lesbarkeit, im Vergleich zu Assembler, gebaut worden. Eine der Folgen hinter diesem Anspruch ist die Tatsache, dass COBOL die meisten Funktionen enthält, welche ein Softwareentwickler für betriebswirtschaftliche Applikationen gebrauchen kann:

- Fixed-format-Datenstrukturen
- Festkommaarithmetik
- Reportgenerator für die Datenstrukturen
- Dateimanipulationen
- Einbettung in Transaktions- und Teleprocessingmonitore
- Datenbankzugriffe

Obwohl COBOL für betriebswirtschaftliche Applikationen sehr gut geeignet ist, gerieten die domänenspezifischen Sprachen, so auch Fortran für den naturwissenschaftlichen Bereich, in den sechziger Jahren aus der Mode.[13] Beginnend in den sechziger Jahren mit ALGOL, PL/I und später auch C, C++ sowie neuerdings Java und C#, wurden domänenneutrale Sprachen die große Neuerung. Parallel zu diesen Sprachen wurden Fortran und COBOL weiterhin sehr stark eingesetzt und entwickelten sich auch fort. Obwohl COBOL als Sprache, völlig ungerechtfertigt, verpönt ist, bleibt es, ironischerweise die Sprache mit der größten Verbreitung und dem größten Wachstum an Sourcecode weltweit. Es wird geschätzt, dass heute mehr als 200 Milliarden COBOL-Codezeilen in aktiver Software vorhanden sind, mit einem Wachstum von ca. 5 Milliarden Zeilen pro Jahr!

Leider ist es in der Sprache COBOL nicht möglich, die Typdeklaration von der Variablendeklaration zu trennen, mit der Folge, dass sich, wenn zweimal dieselbe Variablendeklaration gebraucht wird, die Typdeklaration wiederholen muss. Die COBOL-Softwareentwickler umgehen dies durch den Einsatz von Copybooks, in denen der Typ deklariert wird; durch den Befehl

```
COPY copybook  REPLACING  xxxxx BY  yyyyyy
```

wird der Variablen der Typ zugeordnet. Diese Spracheigenschaft führt zu sehr aufgeblähtem Sourcecode mit einer hohen Redundanz. Durch die fehlenden Typdefinitionen ist eine Gruppierung anhand des Typs unmöglich. Praktisch bedeutet dies, dass anhand des Typs nicht entschieden werden kann, wozu

[13] Ein treffender Ausdruck, da es auch in der IT-Welt regelrechte Modewellen gibt, welche, genau wie bei der „klassischen“ Bekleidungsmode, durch das komplexe Zusammenspiel aus Produzenten, hier Compiler- und Entwicklungsumgebungslieferanten, den Medien und den Verbrauchern, hier den Entwicklern, erzeugt werden.

die Variable gehört. Das Fehlen von Aufzählungstypen macht es nicht einfacher. Zwar existieren so genannte 88-Stufen, diese entsprechen aber nicht den Aufzählungstypen in CORBA oder C++, sondern stellen logische Variablen dar, welche entweder wahr oder falsch sind. Hinzu kommt, dass jede Variable überall in einer Sourcedatei verwendbar ist; COBOL kennt keine lokalen, sondern nur globale Variablen.

Die Aufteilung eines COBOL-Programms in SECTIONs und PARAGRAPHs führt zwar zu einer Struktur, durch die Globalität der Variablen muss der Softwareentwickler diese aber stets „im Kopf“ haben, um zu wissen, wie er einen Algorithmus implementieren kann. Durch das Fehlen einer Parameterstruktur lässt sich auch keine Prüfung durchführen.

Obwohl COBOL zu den Sprachen gehört, welche ideal für betriebswirtschaftliche Applikationen sind, wird es oft geschmäht und mit wiederkehrender Häme wird regelmäßig das Ende von COBOL verkündet. Interessanterweise ist COBOL genau wie Java sehr portierbar; heute findet sich COBOL auf allen gängigen Betriebssystemplattformen wieder, so auch unter .NET.

12.2.2 C

Unter den Sprachen spielt C eine gewisse Sonderrolle, ausgehend von Kernighan und Ritchie wurde C dominant in technischer Software. Im Grunde ist C eine Systemprogrammiersprache, dies gibt der Sprache die Möglichkeit, mit Speicheradressen und Verweisen zu arbeiten. Die Eigenschaft von C, Speicherarithmetik und Integerarithmetik zu mischen, ist für viele hardwarenahe Entwickler attraktiv, allerdings stellt diese Eigenschaft eine immense Fehlerquelle dar, ein Grund, warum es eine direkte Speicheradressierung in Java oder C#[14] nicht mehr gibt. Man hat in beiden Fällen aus den „Fehlerquellen“ von C gelernt.

Eines der großen Probleme im Rahmen von C ist der massive Einsatz des C-Präprozessors. Zwar ermöglicht dieser, Definitionen und Konstanten separat von einem Programm in einer Headerdatei zu halten, er wird jedoch oft ausgenutzt, um eine völlig eigene Syntax zu erfinden:

```
HEADER:
#inlucde<stdio.h>
#define SCHLEIFE(N) for( int i=0;i<n;i++){
#define NEXT }

SOURCE:
void function(int k) {
 SCHLEIFE(k)
 Operation;
NEXT }
```

[14] C# ist der .NET-Nachfolger von C++.

Dieses Beispiel zeigt, wie „leicht“ es ist, in C eine Pascal-ähnliche Syntax aufzubauen. Ein anderes Beispiel ist die mehrfache Ausführung solcher Headerdateien:

```
HEADER: #ifdef PASS
    #undef      WERT
    #define     WERT    2
    #undef      PASS
    #define     PASS2
#elif defined   PASS2
    #undef      WERT
    #define     WERT    3
    #undef      PASS2
#else
    #undef      WERT
    #define     WERT    1
    #define     PASS
#endif
```

Dieses Konstrukt ist schwer verständlich und resultiert darin, dass sich bei jedem Durchlaufen der Headerdatei der Inhalt von WERT um 1 erhöht.

Diese Beispiele machen deutlich, warum es unter den C-Programmierern den „obfuscated C contest“[15] gibt. Das Ziel hinter diesem Wettbewerb ist es, den unverständlichsten C-Code zu schreiben.

12.2.3 Fortran

Die Sprache Fortran wurde von John Backus als Reaktion auf den schwierigen Umgang mit Assembler entwickelt, insofern entstand sie parallel zu COBOL. Die Programmiersprache Fortran, früher hieß sie FORTRAN, **For**mular **Tran**slation, gibt es seit Ende 1953. Das Fortran war die erste höhere Programmiersprache, für die es einen Compiler gab.

Die Überlegenheit von Fortran bestand immer im Bereich der technischen und naturwissenschaftlichen Applikationen; auf diesen Gebieten ist Fortran auch heute noch die Sprache der Wahl. Unternehmen, welche Fortran nutzen, können auf einen riesigen weltweit vorhandenen Schatz an existierenden Fortran-Programmen zurückgreifen, welcher benutzt und gepflegt wird. Es gibt viele Domänen, in denen es zu Fortran auch heute noch keine ernsthafte Alternative gibt.

Um die Investitionen in die bestehenden FORTRAN-77-Programme zu schützen, ist das vollständige FORTRAN-77 in Fortran 90 enthalten. Während einerseits bewährte FORTRAN-77-Spracherweiterungen übernommen wurden, ist die Sprache hauptsächlich um solche neuen Sprachelemente weiterentwickelt worden, die sich in anderen Sprachen bereits bewährt haben. Es

[15] *www.ioccc.org*

hat sich inzwischen gezeigt, dass der Aufwand und die Probleme bei der Umstellung großer FORTRAN-77-Pakete vergleichsweise gering sind.

Fortran 90 unterstützt moderne Hardware-Architekturen wesentlich besser als frühere Fortran-Sprachen oder als andere große Sprachen. Das gilt ganz besonders für Vektorrechner, zum Teil auch für Parallelrechner. Das geschieht allerdings nicht etwa so, dass neue Hardware-Abhängigkeiten in der Sprache entstanden sind, sondern solche Aspekte wurden deutlich reduziert und neue Sprachelemente so definiert, dass sie keine Hindernisse für moderne Architekturen darstellen. Das Fortran 90 ist unabhängig von einem bestimmten Architekturmodell.

Das unangenehmste Konstrukt in Fortran sind vermutlich die COMMON-Blöcke. Sie erlauben es nicht nur, globale Variablen einzuführen, sondern auch noch Variablen zu überdefinieren, ähnlich den COBOL-Redefines. Allerdings geschieht diese Redefinition in Fortran nicht explizit durch ein Statement, sondern implizit, indem der COMMON-Block in einem anderen Programm anders aufgebaut ist. Neben den COMMON-Blöcken produzieren IMPLICIT-, genau wie INCLUDE- und BLOCKDATA-Strukturen Probleme bei der Weiterentwicklung.

12.3 Neuere Architekturen

Die bisher dargestellten Strukturen und Sprachen stellen einen Querschnitt durch die Welt der Legacysysteme dar, doch wie sieht die neue Welt aus, die Umgebung, in die hinein transformiert werden soll?[16]

Im Gegensatz zu den „alten“ sind die „modernen“ Sprachen fast immer mit einer Architekturform verknüpft. Im Fall von J2EE oder .NET ist es sogar noch weitergehend, hier existiert zuerst die Architektur und dann die Sprache, Java oder C#.

12.3.1 XML

Die Nutzung von XML, der e**X**tensible **M**arkup **L**anguage, hat sich in den letzten Jahren auf einen großen Teil der gesamten Softwareindustrie ausgedehnt. XML ist heute der De-facto-Standard für den Datenaustausch im Internet. XML ist ein GML-, **G**eneralized-**M**arkup-**L**anguage-Abkömmling und bildet eine textbasierte Metasprache, genauer gesagt ist XML ein SGML-, **S**tructured-**G**eneralized-**M**arkup-**L**anguage-Abkömmling. Die XML wird üblicherweise genutzt, um neue „Protokolle“ zu definieren und die Daten in strukturierter Form zu übermitteln. Im Gegensatz zu HTML, welches auch von SGML abstammt, wird XML zur Beschreibung von Daten und HTML meistens zur Beschreibung von Darstellungen genutzt.

[16] S. Kap. 6.

Die strukturierte und für Menschen gut[17] lesbare Form macht dieses Austauschformat einsetzbar an Stellen, wo man früher binäre Formate[18], wie beispielsweise CORBA oder EDIFACT, verwendet hätte.

Die Sprache XML erlaubt es, Daten in Elementen oder Attributen – beide können benannt werden um eine Semantik zu vermitteln – zu speichern. Die Elemente der XML werden durch so genannte *begin*- und *end*-Tags und den möglichen Inhalten dieser Tags charakterisiert. Dieser Inhalt kann aus Daten oder aus anderen Elementen bestehen, durch diese Form der Schachtelung bildet ein XML-Dokument immer einen Baum ab.

Ein einfaches XML-Dokument sieht wie folgt aus[19]:

```
<?xml version=''1.0''?>
    <nachricht>
        <to> Harry </to>
        <from> Stefan </from>
        <header> Erinnerung </header>
        <body> Harry, hol schon mal den Wagen! </body>
    </nachricht>
```

Dieses XML-Dokument kann nun verarbeitet, transformiert, überprüft, gespeichert oder weitergeleitet werden, je nach der beteiligten Applikation. Die Struktur des XML-Dokuments kann durch eine XSLT, e**X**tended **S**tyle Sheet **L**anguage **T**ransformation, umgewandelt werden, dabei können Elemente verschwinden oder neue Elemente hinzugefügt werden. Bei der Überprüfung wird zunächst verifiziert, dass das XML-Dokument bezüglich der XML-Syntax korrekt und vollständig ist, so beispielsweise, dass es für jedes $< tag >$ ein $< /tag >$ gibt. Zusätzlich dazu kann ein DTD-, **D**ata-**T**ype-**D**efinition-, oder ein XML-Schema genutzt werden, um damit explizit Namensräume und Datentypen überprüfen zu können. Beide Verfahren stellen ein Typschema für XML-Elemente dar.

Das zu obigem Beispiel gehörende DTD könnte lauten:

```
<!ELEMENT nachricht(to, from, header, body)>
<!ELEMENT to(#PCDATA)>
<!ELEMENT from(#PCDATA)>
<!ELEMENT header(#PCDATA)>
<!ELEMENT body(#PCDATA)>
```

[17] Softwareentwickler sehen XML als gut und einfach lesbar an, „normale“ Endanwender sind da anderer Ansicht.

[18] Schon die Bibel beschreibt binäre Kommunikation:

Eure Rede aber sei: Ja, ja; nein, nein; was darüber ist, das ist vom Übel.

Matthäus **5**, 37

[19] Der Satz: „*Harry, hol schon mal den Wagen!*“, wird in keiner Derrick-Folge ausgesprochen!

Ein analoges XML-Schema wiederum hätte etwa folgendes Aussehen:

```
<?xml version=''1.0''?>
<xs:schema
        xmlns:xs=''http://www.w3.org/2001/XMLSchema''
        targetNamespace=''http://www.w3schools.com''
        xmlns=''http://www.w3schools.com''
        elementFormDefault=''qualified''>
    <xs:element name=''nachricht''>
    <xs:complexType>
        <xs:sequence>
        <xs:element name=''to'' type=''xs:string''/>
        <xs:element name=''from'' type=''xs:string''/>
        <xs:element name=''header'' type=''xs:string''/>
        <xs:element name=''body'' type=''xs:string''/>
        </xs:sequence>
    </xs:complexType>
    </xs:element>
</xs:schema>
```

Anhand des DTD oder des XML-Schemas kann nun überprüft werden, ob das XML-Dokument der Struktur und den Datentypen der jeweiligen Definition genügt. Die XML-Schemata sind aber in ihrem Aufbau mächtiger als die DTDs und haben sich heute de facto durchgesetzt.

12.3.2 Multichannelarchitektur

Die heutige IT-Landschaft ist von einer Proliferation von verschiedenen Formen von Endgeräten geprägt, dazu zählen:

- proprietäre Applikationen
- browserbasierte Clients
- PDAs
- Webservices
- Handys
- Drucker
- Druckstraßen
- Archive
- PDF
- E-Mail
- ...

Die sehr hohe Änderungsrate der Präsentationsschicht, s. Abb. 12.10 sowie Tab. 12.2, legt es nahe, eine Architektur zu wählen, welche den schnellen Austausch der Präsentation unterstützt. Durch die Existenz paralleler Präsentationen entsteht eine Multichannelarchitektur.

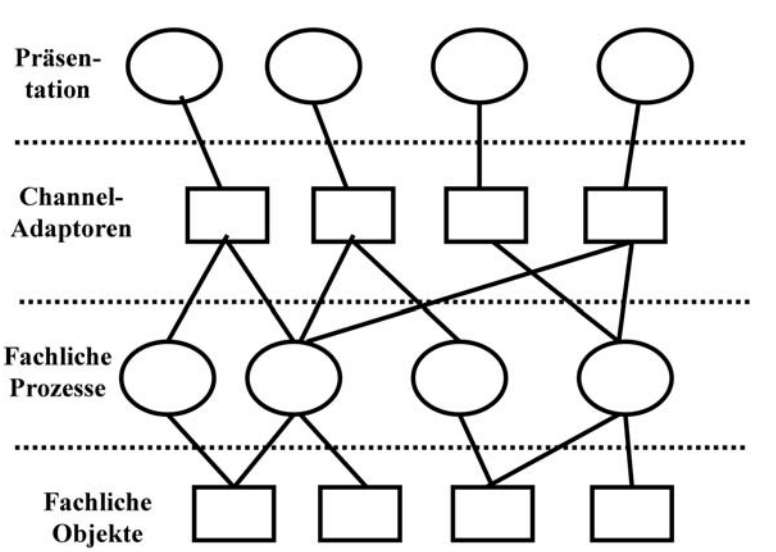

Abb. 12.10: Multichannel-Topologie

In den meisten Fällen ist die treibende Kraft hinter einer Multichannelarchitektur der Versuch, diverse Eingangskanäle zu unterstützen. Die Basisidee der Multichannelarchitektur wiederholt sich, in abgewandelter Form, bei der Enterprise Application Integration, s. Abschn. 12.6, und der Java Connector Architecture, s. Abschn.12.4.8. Die Multichannelarchitektur lebt davon, dass diverse Eingangskanäle ihre Eingangsdaten auf ein „internes“ Format, das kanonische Format, abbilden und diese kanonische Datendarstellung anschließend von den geschäftsprozessunterstützenden Legacysystemen weiterverarbeitet wird, ohne dass der konkrete Eingangskanal bekannt sein muss, s. Abb. 12.11. Aber nicht nur in Bezug auf den eingehenden Datenstrom ist eine Multichannelarchitektur sinnvoll, auch der ausgehende Datenstrom kann völlig identisch unterstützt werden.

Tab. 12.2: Änderungszeiten der einzelnen Schichten in Abb. 12.10

Schicht	**Änderungszeit**	**Stabilität**
Fachliche Objekte	5-10 Jahre	systematisch
Fachliche Prozesse	1,5-3 Jahre	quasi-systematisch
Channel-Adaptoren	3-24 Monate	opportunistisch
Präsentation	1-100 Tage	hyperopportunistisch

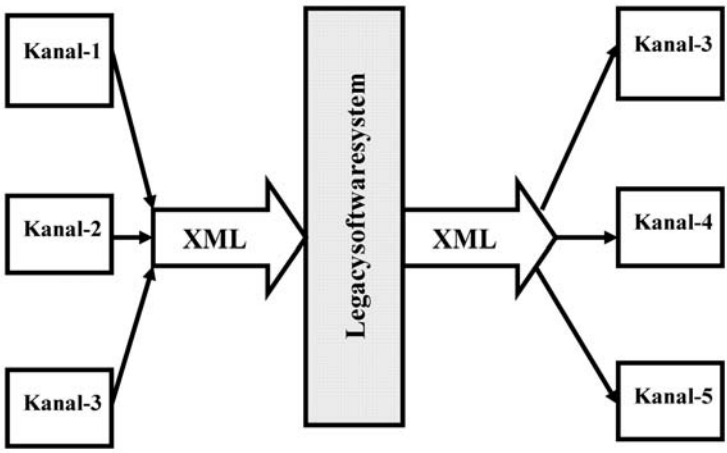

Abb. 12.11: Die Multichannelarchitektur

Besonders elegant ist der Einsatz von XML für das kanonische Format. Den großen Vorteil bezieht die Multichannelarchitektur aus der Reduktion der Zahl der Komponenten und Schnittstellen, s. Abb. 12.11.

Für die Zahl der Schnittstellen N_S und die Zahl der Komponenten N_K ergibt sich, jeweils mit und ohne eine Multichannelarchitektur:

$$N_S = \begin{cases} 2n_{Kanal}n_{Prozess} & ohne\ Multichannel \\ 1 + 2n_{Kanal} & mit\ Multichannel \end{cases}$$

$$N_K = \begin{cases} n_{Kanal}n_{Prozess} & ohne\ Multichannel \\ 1 + n_{Kanal} & mit\ Multichannel \end{cases}$$

Ab einer gewissen, aber kleinen Anzahl von Kanälen, $n_{Kanal} = \mathcal{O}(1)$ ist eine Multichannelarchitektur in Bezug auf die Zahl der Komponenten und Schnittstellen immer besser. Die kleinere Zahl von Schnittstellen und Komponenten macht eine Maintenance wiederum einfacher, da die Komplexität niedriger ist als bei einem anderen System.

Aber nicht nur die Anzahl von Softwareobjekten ist geringer, auch die Geschwindigkeit, mit der ein neuer Kanal aufgebaut werden kann, ist deutlich höher, wenn eine Multichannelarchitektur eingesetzt wird, da das eigentliche Legacysystem nicht oder nur minimal verändert werden muss. Neben dem rein

technischen Aspekt einer Multichannelarchitektur resultiert aus ihr auch eine geänderte Vertriebssicht auf die Unternehmensdaten. Durch die gewählte Konstruktion ist das bevorzugte Kommunikationsmedium des Kunden leicht zu identifizieren. Für das Marketing sind dies sehr interessante Daten, da unterschiedliche Medien nach unterschiedlichen Marketingmaßnahmen verlangen.

Aus Sicht der Entropie – völlig analog verhält sich hier auch ein beliebiges Komplexitätsmaß – ergibt sich:

$$S_{trad} \approx n_{Kanal} S^0_{Legacy} \tag{12.1}$$

$$S_{MChannel} \approx 2n_{Kanal} S_\delta + S^1_{Legacy}, \tag{12.2}$$

wobei S^0 die Entropie der „normalen“ Legacysoftware und S^1 die Entropie nach der Umstellung auf das kanonische Format darstellt.

Obige Näherung setzt voraus, dass die einzelnen Applikationen in Gl. 12.1 nur sehr schwach koppeln bzw. die Kanäle unabhängig voneinander sind. Unter der Annahme, dass

$$S^1 > S^0$$

und

$$S^0 \gg S_\delta$$

gilt, folgt, dass bei einer hinreichend großen Zahl von Kanälen:

$$S_{trad} > S_{MChannel}.$$

In Bezug auf Größen wie Maintainability Index und Volatilitätsindex ist die Situation schwerer zu beurteilen. Vermutlich hat für die Legacysoftware das kanonische Format einen niedrigeren Maintainability Index und einen höheren Volatilitätsindex als für die „normale“ Legacysoftware. Allerdings dürfte die geringere Anzahl an Systemen den Unterschied, insgesamt betrachtet, wettmachen.

12.3.3 Web-Enabling

Eine Spezialform des Wrapping ist das Web-Enabling. Bei diesem wird die Legacysoftware auf ein System, bestehend aus Webservern und HTML-Seiten, abgebildet. Das Ziel hinter dieser Form des Wrapping ist es, Informationen aus der Legacysoftware in das Internet zu stellen. Zwar ist es prinzipiell möglich hiermit ganze Applikationen, welche zuvor direkt in der Legacysoftware eine Oberfläche hatten, für das Internet zugänglich zu machen, dies ist aber eher ungewöhnlich. In der Regel wird diese Form des Web-Enabling nur für ausgewählte Funktionen der Legacysoftware vorgenommen, damit diese dann, via Internet, den Endkunden direkt zur Verfügung stehen können. So beispielsweise eine Kontenanzeige für das Online-Banking.

Architektonisch am einfachsten für diese Aufgabe ist es, eine simple 3-Tier-Architektur, s. Abb. 12.12, zu verwenden. Hierbei existiert eine Präsentationsschicht, welche auf einen Webserver zugreift, der die Middle Tier darstellt. Das Backend wird durch die Legacysoftware gebildet.

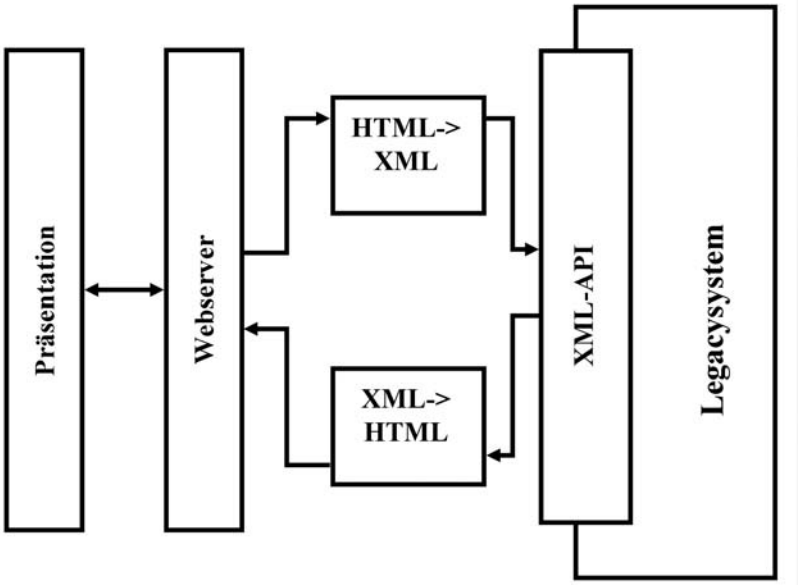

Abb. 12.12: Eine einfache Web-Enabling-Architektur mit einer XML-API

In Wirklichkeit sieht eine solche Architektur komplexer aus. Schlüsselidee ist jedoch stets die Verwendung eines Komponentenwrappers. In einem solchen Fall wird eine konkrete Funktion der Legacysoftware mit einem funktionalen Wrapper versehen und die Wiederverwendung hat den Charakter einer Blackbox-Wiederverwendung.[20] Der Komponentenwrapper ist aus Sicht des Komponentenclients eine Whitebox und kann daher durch Dekoratoren und Adaptor ergänzt werden. Dieses hohe Maß an Funktionalität und Anpassbarkeit macht diese Form der Wiederverwendung interessant.

Obwohl es in der Vergangenheit auch Versuche gegeben hat, direkt HTML in der Legacysoftware zu bilden, erweist sich diese Form der Koppelung als nicht sehr tragfähig, da sie zu starr ist. Der Einsatz einer XML-Schnittstelle für die Blackboxwiederverwendung ist sehr zu empfehlen, da hierdurch eine Entkoppelung der Schnittstelle stattfinden kann.

Die Einführung eines solchen Migrationspfades, s. Kap. 5 und Kap. 6, erweist sich als sehr flexibel, wenn simultan eine Multichannelarchitektur, s. Abschn. 12.3.2, angestrebt wird. Allerdings sollte stets berücksichtigt werden, dass es eine solche Verwendung des Legacysystems nur erlaubt, einzelne Funktionen zur Verfügung zu stellen. Dies limitiert naturgemäß die Einsatzvielfalt eines solchen Szenarios.

[20] Der Begriff Blackbox-Wiederverwendung impliziert, dass über die konkrete Implementierung der Legacysoftware nichts bekannt ist oder sein muss.

12.4 Java 2 Enterprise Edition

12.4.1 Geschichte

Die Ursprünge von J2EE, der Java 2 Enterprise Edition, liegen in der Entwicklung der darunter liegenden Sprache Java. Die Programmiersprache Java wurde ursprünglich von Sun entwickelt, um damit Haushaltsgeräte steuern zu können, daher legt Java auch großen Wert auf die Plattformunabhängigkeit. Obwohl die Sprache ursprünglich für diverse einfache Geräte entwickelt wurde, folgte Sun sehr schnell dem Trend der aufkommenden Internetrevolution und erweiterte Java zu einer Sprache für die Clientseite im Internet. Nach der Schaffung von Java-Applets und Java-Beans kam relativ rasch das JDBC, die Java Database Connectivity, als Verbindungspaket für Datenbanken hinzu. Damit war die erste Brücke hin zur Serverseite für Java geschlagen.

Ziemlich bald stellte es sich heraus, dass Java für eine browserbasierte Clientseite nur bedingt geeignet ist. Das größte Problem ist die hohe Latenzzeit, bis die notwendigen Java-Bibliotheken über das Internet geladen worden sind. Ohne diesen Ladevorgang jedoch kann der Client nicht arbeiten, da er das Programm[21] durch den Ladevorgang erst bezieht. Trotz dieser Schwierigkeiten lieferte Java eine Reihe von Eigenschaften und Bibliotheken, welche die Entwicklung von Web-Applikationen stark vereinfachten.

Zurzeit werden von Java drei verschiedene Plattformen unterstützt, s. Abb. 12.13:

- J2ME, die Micro Edition – Ein Paket zur Entwicklung von Software auf Embedded Devices, wie z.B. Telefone, Palmtops usw.
- J2SE, die Standard Edition – Dies ist die wohl bekannteste Java-Plattform mit allen Funktionalitäten, welche im regulären Sprachumfang vorhanden sind, inkl. JDK, Java Development Kit.
- J2EE, die Enterprise Edition – Konzipiert für die Schaffung von Enterprise-Applikationen und Koppelungen an Legacysysteme, daher werden hier die J2EE und ihre Eigenschaften in den Mittelpunkt gerückt.

12.4.2 Überblick

Mit der Java Enterprise Edition, J2EE, liegt eine integrierte Plattform für die Entwicklung portabler Softwarekomponenten auf dem Server vor. Die J2EE spezifiziert also eine komplette Architektur zur Entwicklung verteilter mehrschichtiger Applikationen, s. Abb. 12.15.

Die J2EE ist ein komponentenorientiertes Framework, welches technische Basisservices insbesondere auf Serverseite anbietet, die bei der Entwicklung von komplexen, verteilten Systemen helfen. Ein zentrales Element hierfür sind

[21] Korrekterweise lädt der Client zunächst die notwendigen Klassen, welche anschließend durch einen Just-In-Time-Compiler übersetzt und von der Java Virtual Machine interpretiert werden.

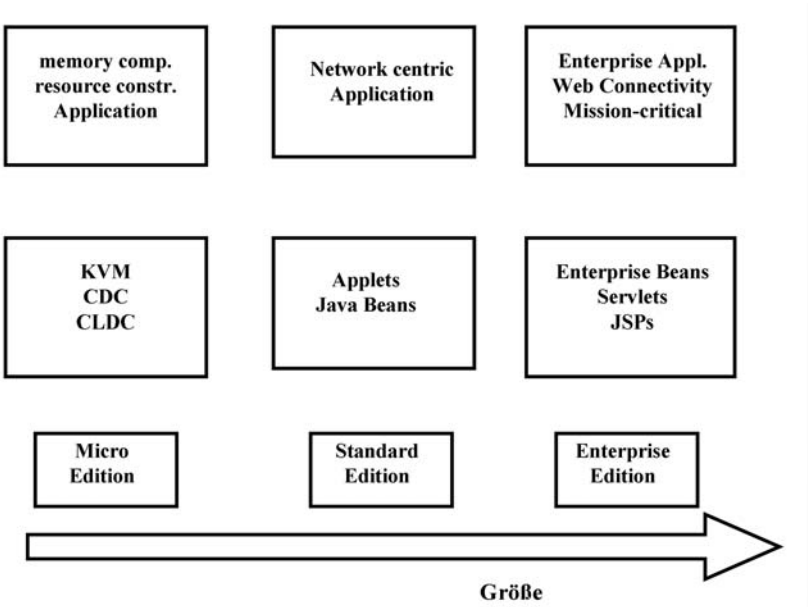

Abb. 12.13: Die verschiedenen Java-Plattformen

die Enterprise Java Beans, welche die eigentlichen Komponenten für den Einsatz auf Application Servern darstellen. Dennoch handelt es sich bei J2EE um eine reine Spezifikation, nicht um ein Produkt. Die J2EE enthält zwar eine Referenzimplementierung der J2EE-Architektur, Java ist als Implementierungssprache aber nicht zwingend vorgeschrieben. In der Praxis wird dieser feine Unterschied allerdings nicht gemacht und beides wird praktisch synonym gebraucht.

Im Grunde stellt J2EE eine verteilte Application-Serverumgebung dar, s. Abb. 12.18, liefert also die Grundlagen und Mittel zur Realisierung der Ebene der Geschäftslogik in der Applikationsschicht, in J2EE vereinfacht auch „Middle Tier“ genannt. Hauptbestandteile sind eine Laufzeitinfrastruktur für den Einsatz und eine Anzahl von Schnittstellen für die Entwicklung von Applikationen. Es wird nicht vorgeschrieben, wie eine Laufzeitumgebung für Geschäftslogik implementierende Komponenten im Einzelnen auszusehen hat, sondern es werden Rollen und Schnittstellen der verschiedenen beteiligten Parteien definiert. Dadurch lässt sich eine Trennung von Applikation und Laufzeitumgebung erreichen, welche es der Laufzeitumgebung ermöglicht, den Applikationen Services des zugrunde liegenden Rechnersystems einheitlich zur Verfügung zu stellen. Komponenten können so auf verschiedenen Betriebsplattformen mit gleichen Umgebungsbedingungen rechnen.

Obwohl es innerhalb der offiziellen J2EE-Spezifikation keinen Zwang gibt, eine bestimmte Architektur zu nutzen, empfiehlt sie doch zum Aufbau eines Clients das Model View Controller Pattern, MVC-Pattern, s. Abschn. 13.11,

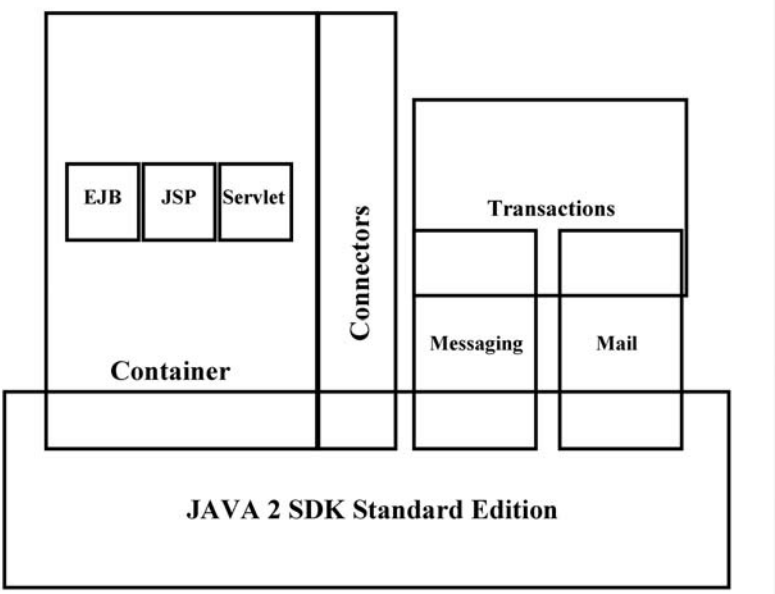

Abb. 12.14: Die Teile der J2EE-Umgebung

wobei häufig das MVC-Pattern in der Form verwendet wird, dass das darin befindliche Modell das eigentliche Legacysystem darstellt.

12.4.3 Container

Der Begriff des Containers ist der zentrale Begriff im Umfeld des serverseitigen J2EEs, s. Abb. 12.16. Ein Container ist ein Softwareteil, welches in einem Server abläuft und für eine Reihe von Komponenten zuständig ist. Umgekehrt stellt der Container die Ausführungsumgebung für die J2EE-Komponenten dar.

Durch den Einsatz des Containerkonzepts erlaubt es die J2EE-Architektur, die Trennung von Entwicklung und Deployment aufrecht zu erhalten und gleichzeitig einen hohen Grad an Portabilität für den „Middle Tier“ zur Verfügung zu stellen. Neben der reinen Laufzeitumgebung „managed“ der Container den vollständigen Lebenszyklus der in ihm enthaltenen Komponenten und stellt einen Teil der Infrastruktur im Bereich Ressourcen Pooling sowie Security zur Verfügung. Innerhalb der J2EE-Spezifikation existieren vier verschiedene Typen von Containern:

- Application Container, es handelt sich hierbei um Stand-alone-Java-Applikationen.
- Applet Container – Der Applet Container stellt die Laufzeitumgebung für Applets zur Verfügung.

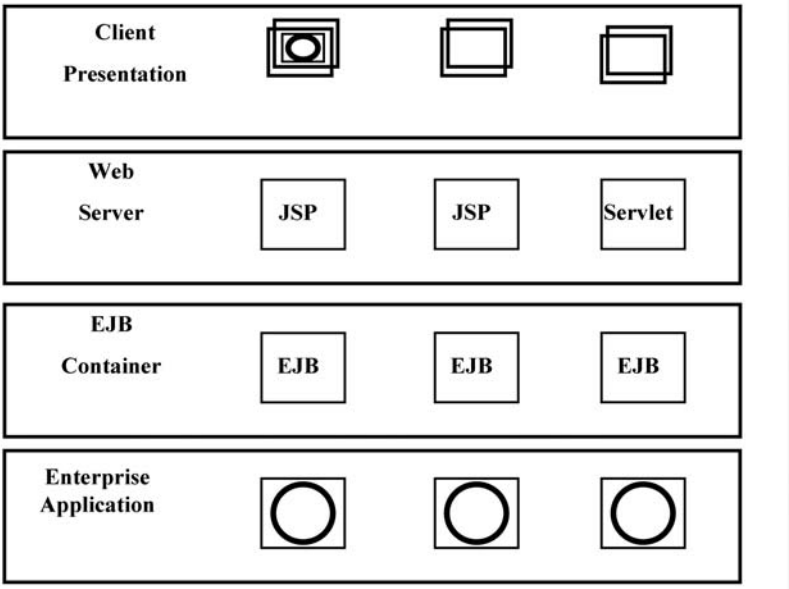

Abb. 12.15: Die allgemeine J2EE-Architektur

- Web Container – Dieser ist für zwei Typen zuständig:
  - Servlets,
  - Java Server Pages, JSPs.

  Grob vereinfacht gesehen, spielt der Web Container die Rolle eines Webservers, welcher mit HTTP umgehen kann und als Host für die Servlets und Java Server Pages zur Verfügung steht.
- Enterprise Container – Dieser enthält die Enterprise Java Beans, EJBs. Von diesen Enterprise Java Beans existieren drei Typen:
  - Session Beans
  - Entity Beans
  - Message Beans

  Der Enterprise Container stellt den in ihm enthaltenen Beans eine Reihe von Funktionen zur Verfügung. Dazu zählen:
  - Lifecycle-Services – Der Lebenszyklus der EJBs wird im Container verwaltet. Jedes einzelne Enterprise Java Bean durchläuft die Zustände:
    - neu erzeugt
    - verändert
    - gelöscht

    Die zentrale Idee hinter den Enterprise Java Beans ist es, einen abstrakten Persistenzmechanismus zu kreieren. Die Enterprise Java Beans liefern einen generischen, objektorientierten Zugang zur Persistenz von Daten, ohne dass zunächst spezifisches Wissen über Datenhaltung vorhanden sein muss. Der EJB-Container implementiert dann

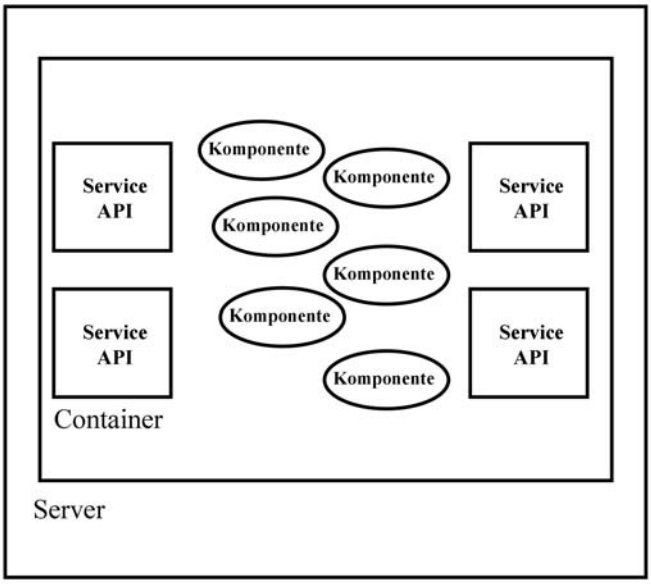

Abb. 12.16: Der Container bei J2EE

selbstständig die Vorgaben des Softwareentwicklers bezüglich Transaktionsverhalten und Speichermechanismen oder auch Zugriffsschutz. Auf der anderen Seite produzieren die Enterprise Java Beans auch ein gewisses Maß an Komplexität, was dazu führt, dass die EJB-Container oft der Performanzflaschenhals per se sind. Der Container sorgt auch für den Abgleich mit der Datenbank, um immer den aktuellen Zustand seiner EJB-Objekte mit der Datenbank zu synchronisieren. Eine weitere Aufgabe des Containers ist die Verwaltung eines Objekt-Caches für den schnelleren Zugriff auf die Objekte.

- Object Persistency – Für die persistente Speicherung der Objektdaten ist der Container zuständig. Er kann die Objekte in einer Datenbank speichern und sie auch mittels einer Objekt-Id eindeutig identifizieren. Die Object Persistency ist auch in der Lage, Relationen zwischen den Objekten abzubilden.
- Transaction Management – Der Container verwaltet einen Pool von Datenbankverbindungen und ist in der Lage, die Datenbanktransaktionen zu steuern.
- Security – Innerhalb des Containers existiert das J2EE Security Model, welches, ähnlich einer Datenbank, die Zugriffe für die Benutzer regelt.
- Remote Client Connectivity – Mit Hilfe des JNDI, des Java Naming and Directory Interface, lässt sich auf EJB-Objekte auch remotely zugreifen. Dieser Zugriff von Seiten eines Clients erfolgt mit dem RMI/IIOP-Protokoll, Remote Method Invocation.

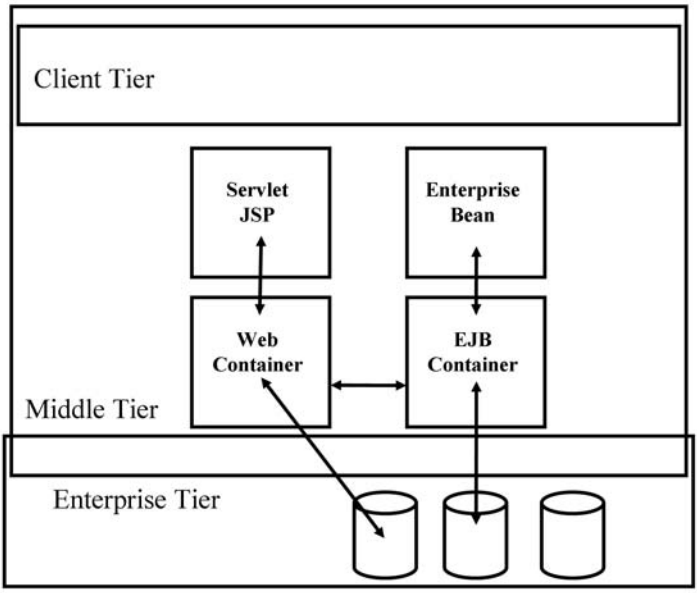

Abb. 12.17: Die Zusammenarbeit der verschiedenen Container

12.4.4 Enterprise Java Beans

Der EJB-Container, Enterprise Java Bean Container, befindet sich immer innerhalb des Application Servers, der auch als J2EE-Server bezeichnet wird, s. Abb. 12.18. Der Application Server ist in der Lage, mehrere EJB-Container simultan zu unterstützen. Dadurch ist es möglich, Applikationen zu installieren, welche getrennt voneinander administriert werden können. Die Hauptaufgabe des EJB-Containers ist die Delegation von Anfragen an die EJBs, weiterhin übernimmt er die Rückmeldungen der Ergebnisse eines Enterprise Java Beans an den Client. Der EJB-Container stellt einem Enterprise Java Bean eine Laufzeitumgebung zur Verfügung und verwaltet zudem deren Lebenszyklus und sichert ihre Ansprechbarkeit. Diese Absicherung ist nach der Spezifikation von J2EE ein Service des Containers. Fordert ein Client ein EJB an, so erzeugt der EJB-Container eine EJB-Instanz. Damit der EJB-Container die EJBs in verschiedene Zustände bringen kann, existieren Callback-Methoden innerhalb des Enterprise Java Beans, welche der Container nutzen kann. Da der Application Server die Aufgabe hat, eine große Zahl von Clients zu unterstützen, entsteht das Problem, dass mit der Anzahl von Clients meist die Anzahl der erzeugten bzw. notwendigen Enterprise-Java-Bean-Instanzen steigt. Um die Systemlast zu reduzieren, besteht für den EJB-Container die Möglichkeit, nicht benötigte EJBs in so genannten Pools zu verwalten. Als vorteilhaft erweist sich die Tatsache, dass die EJB-Instanzen bereits existieren und somit die Zeit für deren Instanziierung wegfällt. Befinden sich Bean-Instanzen im

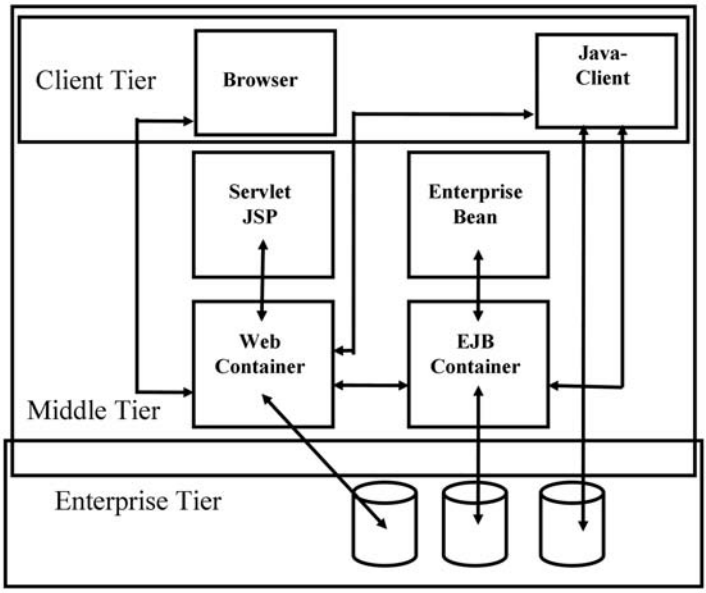

Abb. 12.18: Der J2EE Application Server.

Zustand „pooled“, so können sie in den Zustand „ready“ überführt werden und umgekehrt. Bei einem Aufruf durch einen Client wird zuerst überprüft, ob eine Instanz des benötigten Enterprise Java Bean im Pool verfügbar ist. Außerdem kann der EJB-Container eine EJB-Instanz aus dem Arbeitsspeicher in eine Datenbank auslagern, wenn sie über einen definierten Zeitraum hinweg nicht genutzt wurde. Die Aktivierung dieser Instanz wird durchgeführt, sobald ein erneuter Zugriff auf die persistente Instanz erfolgt. Auch Thread- und Prozessmanagement werden durch den EJB-Container zur Verfügung gestellt.

Da Daten persistent zu speichern sind, ermöglicht der EJB-Container Zugriffe auf die entsprechenden Ressourcen, in der Regel eine relationale Datenbank. Sollen Änderungen von Daten nur dann vorgenommen werden, wenn mehrere, voneinander abhängige Operationen erfolgreich absolviert wurden, dann lassen sich die Transaktionen nutzen, welche durch den EJB-Container bereitgestellt werden. Die so bereitgestellten Services können sowohl durch den aufrufenden Client, als auch durch ein anderes Enterprise Java Bean genutzt werden. Einem Enterprise Java Bean ist es nicht möglich, mit dem Server direkt zu kommunizieren, sondern nur mit dem EJB-Container. Eine weitere Funktion des EJB-Containers ist die Bereitstellung eines Namens- und Verzeichnisservices. Über den Namensservice kann ein EJB ortstransparent gefunden werden, wobei das „Java Naming and Directory Interface“, JNDI, genutzt wird. Nachdem das EJB angefordert wurde, erzeugt der EJB-

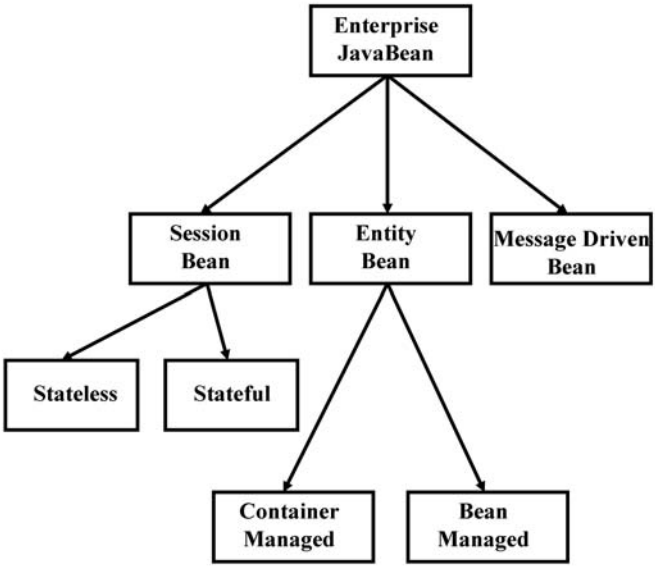

Abb. 12.19: Die Enterprise-Java-Beans-Typen

Container ein neues Objekt des EJBs oder entnimmt ein bereits existierendes Objekt aus einem Pool.

Sollen mehrere Aktionen zusammenhängend auf der relationalen Datenbank ausgeführt werden, wobei andere Aktionen nicht in der Lage sein sollen, diese zu beeinflussen, empfiehlt sich die Verwendung von Transaktionen. Dies kann vor allem dann von Vorteil sein, wenn es von Bedeutung ist, ob wirklich alle Datenbankoperationen korrekt ausgeführt wurden.

Ein Hauptmerkmal der EJB-Architektur ist die Transaktionsunterstützung. Die Bereitstellung der impliziten und expliziten Transaktionsunterstützung durch die EJBs setzt neue Maßstäbe in der Entwicklung von Web-Technologien bzw. in der Realisierung robuster web-basierter Applikationen. In der Entwicklung von web-basierten Applikationen übernimmt die Transaktionsunterstützung somit in der Praxis einige Implementierungsaufgaben, die bisher zum Teil, jede einzeln, realisiert werden mussten. Transaktionen ermöglichen dem Softwareentwickler die Implementierung der Applikationslogik, die atomare Aktionen über mehrere Persistenzmedien in verteilten Umgebungen durchführt. Dies umschließt alle ACID-Prinzipien einer Transaktion, s. S. 312. Die EJB-Spezifikation benutzt hierzu die Java Transaction, JTA, welche von dem EJB-Container bereitgestellt wird. Das Transaktionsmodell der EJB-Spezifikation unterstützt jedoch nur flache und keine geschachtelten Transaktionen.

Die üblichen Objektnetze eignen sich normalerweise nur sehr wenig für Transaktionen, da es schwer ist, die große Zahl von Beziehungen, welche ein

Objektnetz besitzt, abzubilden. Dies liegt darin begründet, dass die Veränderung eines Objektes häufig die Erzeugung und Löschung anderer Objekte bedingt. Die Datenbanken sind mit diesem Problem nicht konfrontiert, da hier, durch die referentielle Integrität, die Konsistenz strukturell verankert ist. Letztendlich bedeutet dies, dass die Applikation selbst für die Konsistenz verantwortlich ist, da im Umfeld von Enterprise Java Beans nur die Transaktionen auf der Objektebene sichergestellt werden.

Die Enterprise-Java-Beans-Spezifikation erlaubt Transaktionen auf Objektebene durch den Einsatz von primären Schlüsseln für jedes Entity Bean. Die Referenzen zwischen den Objekten werden nicht durch direkte Java-Referenzen abgebildet, sondern sind Aufgabe des Application Servers, der diese Schlüssel nutzt. Der Application Server verfolgt die Veränderungen der Zustände von Objekten bzw. deren Relationen untereinander.

12.4.5 Session Beans

Die Session Beans sind in den meisten Fällen Geschäftsprozesse, wobei ein Session Bean exklusiv für einen Client ausgeführt wird. Session Beans können Daten ändern, beispielsweise über JDBC[22], und dies, falls erforderlich, mittels Transaktionen absichern. Die Lebensdauer eines Session Bean entspricht der Sitzungsdauer des Clients. Es ist jedoch auch möglich, dass das Session Bean vorzeitig beendet wird. Die Session Beans können „stateful“ oder „stateless“ implementiert werden, s. Abb. 12.19.

In einem Stateless Session Bean werden ausschließlich Daten verarbeitet, welche bei einem Methodenaufruf explizit übergeben wurden. Innerhalb des Session Beans ist es nicht möglich, Daten zu speichern und bei einem späteren Methodenaufruf zu nutzen. Auf Grund dessen besitzen alle stateless Session Beans des gleichen Typs die gleiche Identität. Dies ermöglicht die Einrichtung eines Pools von Stateless-Session-Bean-Instanzen, damit die Zeit für die Initialisierung wegfällt. Somit ist es möglich, dass mehrere Clients parallel und ohne Verzögerung auf die Session-Bean-Instanzen zugreifen können.

Die Stateful Session Beans bieten die Möglichkeit, Daten innerhalb dieser zu speichern. Demnach kann ein Client bei einem erneuten Methodenaufruf auf die Daten eines früheren Aufrufs zurückgreifen. Auch bei den Stateful Session Beans besteht die Möglichkeit, die Daten explizit zu übergeben. Da die Möglichkeit besteht, ein Session Bean über mehrere Methodenaufrufe einem Client zuzuordnen, besitzen die Stateful Session Beans des gleichen Typs unterschiedliche Identitäten.

[22] JDBC ist das standardisierte Java Interface für den Zugriff auf relationale Datenbanken. Der Name leitet sich von dem Microsoftstandard ODBC ab. Die ersten JDBC-Implementierungen nutzten explizit das ODBC als Basis und bildeten so genannte JDBC-ODBC-Bridges.

12.4.6 Entity Beans

Die Entity Beans sind Objekte, welche die Daten einer Datenbasis als Objekt darstellen bzw. mit bestimmten Daten assoziiert werden. Es ist möglich, Entity Beans durch mehrere Clients gleichzeitig zu nutzen. Um alle Integritätsbedingungen einzuhalten, ist es empfehlenswert, alle Methoden der Entity Beans innerhalb einer Transaktion auszuführen. Die Entity Beans können ihre Daten in einer Datenbank ablegen und so persistent werden. Die Lebensdauer eines Entity Bean ist an die Bestandsdauer der Daten in der jeweiligen Datenbank geknüpft. Im Gegensatz zu einem Session Bean überlebt ein Entity Bean die Zerstörung des jeweiligen EJB-Containers.

In Bezug auf die Transaktionen existieren zwei Untertypen:

- Container Managed Transaction Demarcation – Bei dem impliziten Transaktionsmanagement übernimmt der EJB-Container die vollständige Verwaltung der Transaktionsunterstützung. Über die Transaktionsattribute im Deploymentdescriptor wird das Verhalten der Transaktionssteuerung von EJBs oder deren einzelner Methoden bestimmt. Die Transaktionsattribute werden entweder von dem Bean-Provider oder Application Assembler im Deploymentdescriptor genau festgelegt.
- Bean Managed Transaction Demarcation – Im Fall des expliziten Transaktionsmanagements muss der Softwareentwickler die gesamte Transaktionsunterstützung selbst steuern. Somit lassen sich Transaktionskontexte in der Applikationslogik von dem Softwareentwickler selbst festlegen. Hierdurch kommt es zu einer Vermischung zwischen Applikations- und Transaktionslogik. Jegliche Änderung in der Transaktionslogik führt zu einem erneuten Compilieren der Applikationslogik. Aus diesem Grunde ist das implizite Transaktionsmanagement dem expliziten in den meisten Fällen vorzuziehen.

Auch beim Eintreten eines Fehlers während der Transaktionssteuerung werden unterschiedliche Fehlerbehandlungsmechanismen für das implizite und explizite Transaktionsmanagement durchgeführt. Während bei dem impliziten Transaktionsmanagement grundsätzlich ein Rollback von dem EJB-Container veranlasst wird, erhält der EJB-Client beim expliziten Transaktionsmanagement eine Exception. Der EJB-Client entscheidet nach der Exception selbst, ob er ein Rollback oder eine andere Aktion ausführt. Session Beans und Message-Driven Beans unterstützen sowohl implizites als auch explizites Transaktionsmanagement. Dagegen sind Entity Beans nur durch implizites Transaktionsmanagement zu realisieren.

12.4.7 Message Beans

Die Message-Driven Beans, kurz auch Message Beans oder MBeans genannt, werden durch Nachrichten angesprochen und sind an sich zustandslos. Sobald der EJB-Container eine Nachricht aus der Java Message Service Queue,

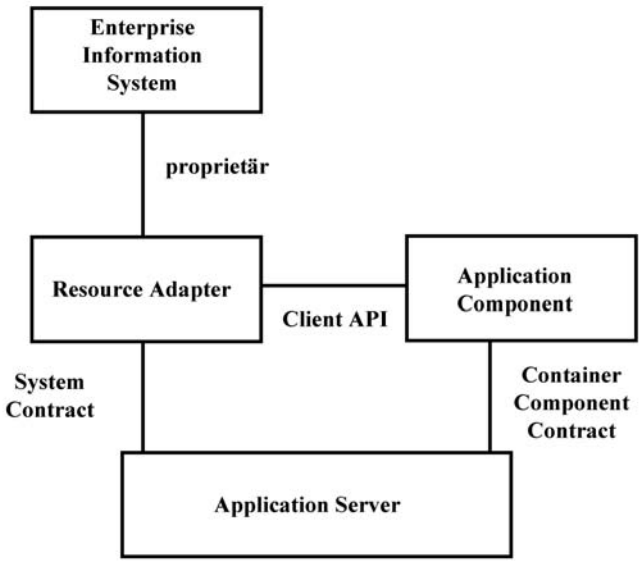

Abb. 12.20: Die Java Connector Architecture

JMS Queue, erhält, wird diese an das zugehörige Message Bean weitergeleitet und anschließend verarbeitet. Ein Message Bean ist demnach ein einfacher Empfänger von Informationen. Ist die Nachricht eingetroffen, wird die Methode onMessage() aufgerufen. Das Messaging ist auf Grund der EJB-Zugehörigkeit asynchron und parallel. Die Lebensdauer eines Message Bean ist so lange, wie die Ausführung der Methode onMessage() dauert.

Der Java Message Service bietet eine einfachere und flexiblere Alternative für den asynchronen Austausch von Nachrichten zu der in Java präferierten Remote Method Invocation, RMI, an. Er verbraucht weniger Ressourcen und ist weniger restriktiv. Der Java Message Service stellt zwei Nachrichtenservicemechanismen, publish-and-subscribe und point-to-point, zur Verfügung. Beide Mechanismen können von den Message-Driven Beans verwendet werden. Ein weiterer wichtiger Aspekt der Message-Driven Beans ist zudem, dass sie in der Lage sind, gleichzeitig Nachrichten zu erzeugen und zu konsumieren. Alle Angaben zu den Message-Driven Beans sowie deren Eigenschaften werden, wie für Session und Entity Beans, in dem Deploymentdescriptor festgelegt.

12.4.8 Java Connector Architecture

Die Java Connector Architecture wurde entworfen, um damit die Verbindungen von anderen „Applikationen“ zu den J2EE-Komponenten zu vereinfachen. Solche „Applikationen“ rangieren von Datenbanksystemen über Enterprise

Resource Planning Software bis hin zu Programmen, welche in Transaktionsmonitoren ablaufen, daher ist die Java Connector Architecture ideal für die Ankoppelung eines Legacysystems geeignet.

Die Java Connector Architecture definiert eine Menge von Mechanismen, so genannte Contracts, so dass die Applikationen einfach in die Application Server integriert werden können. Diese Mechanismen sind so entworfen worden, dass sie Skalierbarkeit, Sicherheit und Transaktionalität sicherstellen. Der Contract existiert zwischen dem J2EE Application Server und den jeweiligen Applikationen.

Die Java Connector Architecture definiert ein clientseitiges Interface, welches den J2EE-Applikationskomponenten, wiederum Enterprise Java Beans, erlaubt, auf andere Applikationen zuzugreifen. Dieses Interface wird Common Client Interface genannt. Umgekehrt muss auch die Applikation ihre Seite des Contracts erfüllen. Mit diesem Contract ist eine Applikation in der Lage, von ihrer Seite mit dem J2EE Application Server zu kommunizieren. Da die Java Connector Architecture den Ressourceadaptor festlegt, kann dieser in jedem J2EE-kompatiblen Application Server genutzt werden.

12.4.9 Java und XML

Innerhalb von Java existieren mehrere Schnittstellen zur Nutzung von XML s. Abschn. 12.3.1, welche die entsprechenden Funktionalitäten innerhalb von Java zur Verfügung stellen. Diese einzelnen Schnittstellen für XML sind:

- JAXM – Die Java API for XML Messaging, JAXM, ist eine Standardschnittstelle, um damit XML-Nachrichten zu empfangen und zu versenden, daher wird sie hauptsächlich für asynchrone Koppelungen eingesetzt. Sie ermöglicht es auch, neue Standards wie ebXML, electronic business XML, und andere zu nutzen.
- JAXP – Die Schnittstelle für das XML-Processing, JAXP, ermöglicht es, die Standards DOM, SAX und XSLT direkt zu nutzen, ohne die konkrete XML-Parser-Implementierung zu kennen.
- JAXB – Die Übersetzung zwischen XML nach Java und umgekehrt wird durch das Java API for XML Bindings, JAXB, vorgenommen. Diese Schnittstelle ermöglicht es, XML direkt als Javaobjekt zu instanziieren, aber auch ein Javaobjekt nach XML abzubilden, was einem einfachen Marshalling entspricht.
- JAXR – Das Java API for XML Registries ermöglicht es, auf ein UDDI-Repository, s. Abschn. 12.9.4, zuzugreifen.

Diese in ihrer Gesamtheit sehr mächtige Bibliothek von XML-Funktionalitäten macht es relativ einfach und effizient innerhalb einer Javaumgebung XML-Operationen auszuführen.

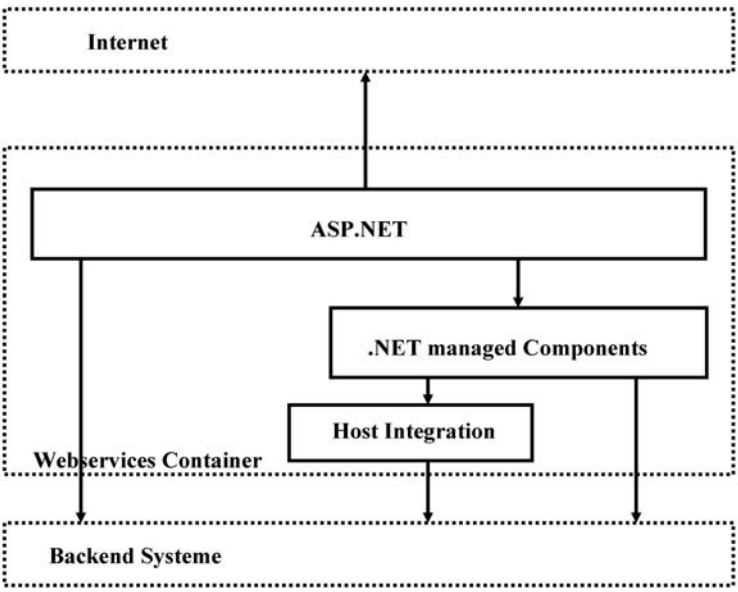

Abb. 12.21: Webservices des .NET-Frameworks

12.5 .NET

Die .NET-Strategie wurde von Microsoft Mitte des Jahres 2000 bekannt gegeben. Die Zielsetzung hinter .NET ist der Versuch, größere Anteile auf dem Servermarkt zu gewinnen, nachdem Microsoft eindeutig den heutigen Client-Markt dominiert. Microsoft erklärt auf seiner Homepage[23]:

> *.NET ist die ... Microsoft Plattform für XML-Web Services, die Informationen, Geräte und Endanwender in einer einheitlichen und personalisierten Weise miteinander verbindet.*

Der Sprachstandard XML, s. Abschn. 12.3.1, ist bei .NET die Schlüsseltechnologie, die zur Datenübertragung und Datenspeicherung genutzt wird. Microsoft konnte dabei auf die mit der Windows-DNA-Plattform gesammelten Erkenntnisse aufbauen. Bei der Kommunikation zwischen den Komponenten betritt Microsoft neue Wege. Die Webservices kommunizieren nicht mehr länger wie herkömmliche Komponenten über DCOM miteinander, sondern benutzen SOAP, welches nur HTTP und XML verwendet und somit keine homogene Infrastruktur auf dem Server und dem Client voraussetzt, s. Abb. 12.21.

Die gesamte .NET-Plattform befindet sich innerhalb eines Containers, des Webservice-Containers, welcher Qualities of Service sowie Transaktions-,

[23] *www.microsoft.de*

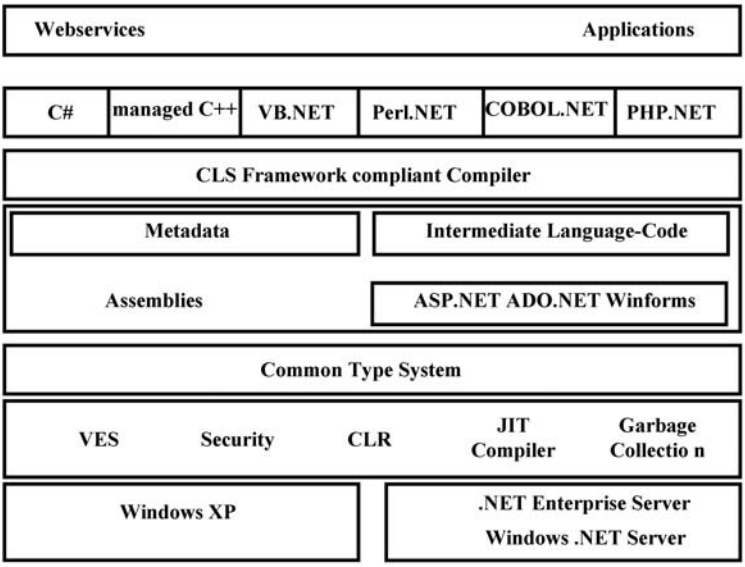

Abb. 12.22: Architektur des .NET-Frameworks

Sicherheits- und Nachrichten-Services für die Enterprise-Applikationen unterstützt, insofern ähnelt dies der Idee der Enterprise Java Beans.

Das erklärte Ziel des .NET-Frameworks ist es, aus verschiedenen Produktgebieten, welche Microsoft in den Jahren zuvor getrennt angeboten hat, in ein „Gesamtprodukt“ zu bündeln. Zu den Produktkomponenten zählen:

- Database Access – Die Persistenz ist ein fundamentaler Teil jeder größeren Applikation, gleichgültig ob diese Applikation datenbank- oder auch dateizentrisch ist. Das .NET-Framework liefert die Active Data Objects, ADO, als zentralen Bestandteil der Persistenz.
- Directory Services – Wie schon bei J2EE, s. Abschn. 12.4, deutlich wurde, ist das Auffinden von Services oder Objekten elementar wichtig. Innerhalb von .NET wird dies über das so genannte Active Directory sichergestellt. Das Active Directory stammt ursprünglich von den Fileservices ab, was auch seine gute Unterstützung beim Auffinden von URLs und ähnlichen Ressourcen erklärt.
- Messaging – Das Messaging wird durch die Microsoft Message Queue, MSMQ, gewährleistet, ein Produkt, das dem MQ-Series von IBM, s. Abschn. 12.7, sehr ähnlich ist.
- Mailing – Das Mailing wird durch Microsoft Exchange Server gewährleistet. Im Gegensatz zu den klassischen SMTP und POP3-Protokollen liegt hier der Schwerpunkt auf IMAP.

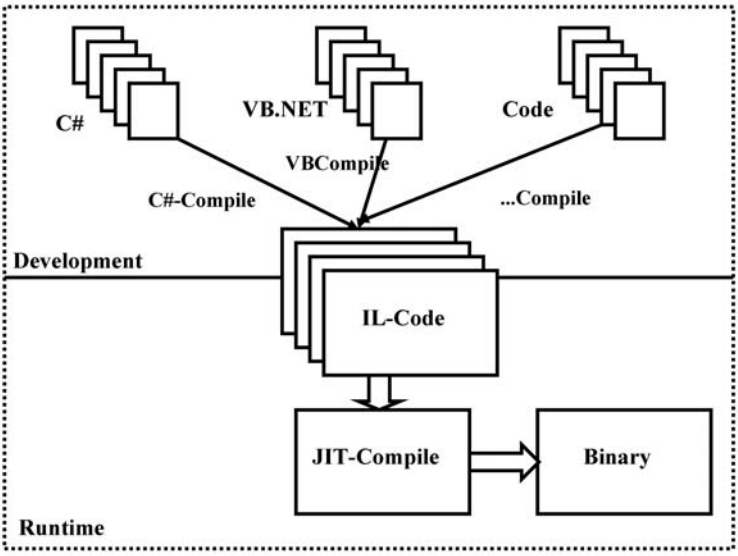

Abb. 12.23: Der Intermediate Language Code

- Business Process Automation – Die BPA erlaubt es Microsoft, eine Reihe von Funktionen ereignisgesteuert automatisch abzuarbeiten. Diese Automatismen sind eine wichtige Funktion für ein System, welches aus einem Einzelarbeitsplatzsystem heraus entstanden ist. Traditionelle Batch- oder Mehrplatzsysteme, wie z.B. MVS oder Unix, sehen solche Funktionalitäten als trivial an.
- Application Server – Mit dem Enterprise Server liefert Microsoft einen Application Server sowie ein System, das Lastverteilung erlaubt.

Alle .NET-Applikationen werden in der Common Language Runtime, CLR, ausgeführt. Die CLR-Programme bestehen aus dem Intermediate Language Code, ILC, welcher von den einzelnen Compilern der jeweiligen .NET-Sprachen erzeugt wird, s. Abb. 12.23. Die Common Language Runtime besteht aus den Bestandteilen:

- Common Type System,
- Virtual Execution System,
- Security Manager,
- Just-In-Time-Compiler,
- Garbage Collection.

Das .NET-Framework stellt ein einheitliches Typsystem für Programmiersprachen, das Common Type System, CTS,[24] zur Verfügung. Es stellt zwangsläufig den kleinsten gemeinsamen Nenner der einzelnen Sprachen dar, ist jedoch so mächtig, dass es für alle .NET-Sprachen eine einheitliche Basis bildet und somit das Arbeiten ohne Typkonvertierung über die Sprachgrenzen der .NET-Sprachen hinweg ermöglicht. Es ist konsequent objektorientiert, enthält z.B. Klassen, Interfaces, Mehrfachvererbung, Polymorphie und virtuelle Methoden. Es können auch eigene Value-Types definiert werden. Das Common Type System stellt die Interoperabilität sicher, indem es eine Untermenge definiert, die alle Sprachen implementieren müssen. Diese Untermenge ist das so genannte Common Language Subsystem.

Weiterhin ist die Integration der Garbage Collection in die Common Language Runtime wichtig. Diese steht allen Sprachen, auch jenen, die bisher nur eine manuelle Speicherverwaltung unterstützten, zur Verfügung.

Der Programmquellcode der .NET Sprachen wird in die so genannte Microsoft Intermediate Language, auch als MSIL bezeichnet, übersetzt. Diese ist streng typisiert und maschinenunabhängig. Zur Laufzeit wird der MSIL-Code durch den Just-In-Time-Compiler in den proprietären Maschinencode übersetzt. .NET-Applikationen sind folglich interoperabel, da sie alle in derselben Sprache vorliegen, als so genannte Assemblies. Der MSIL-Code enthält, neben dem ausführbaren Programmcode, auch die kompletten Metadaten der Programme. Als Ergebnis dieses Ansatzes benötigen .NET-Programme keine externen Ressourcen, wie z.B. die Registry. Die so entstandenen Applikationen lassen sich daher sehr einfach im Rahmen des Deployments einsetzen.

Bei der Ausführung der Assemblies wird ein kurzer Maschinencode ausgeführt, welcher die Common Language Runtime zur Ausführung des Programms lädt. Die Common Language Runtime übernimmt die Kontrolle über das jeweilige Assembly, welches dann Managed Code genannt wird. Außerdem steuert die Common Language Runtime die Garbage Collection und die Interoperabilität mit Nicht-.NET-Applikationen. Die Garbage Collection erfordert von einigen Sprachen wie z.B. C/C++, eine Einschränkung ihrer Eigenschaften. Die .NET Version von C++ wird deshalb auch managed C++ genannt. Bedingt durch die .NET-typische Garbage-Collection-Implementierung muss man bei dem Managed-C++ auf die gewohnte Pointerarithmetik verzichten.[25]

Betrachtet man die riesigen Mengen herkömmlichen Sourcecodes, welche noch immer im Einsatz sind, verwundert es nicht, dass .NET-Applikationen auch mit Unmanaged-Code zusammenarbeiten können. Unmanaged-Code ist ein Code, welcher nicht unter der Kontrolle der Common Language Runtime steht. Dieser Code wird zwar trotzdem von der Common Language Runtime ausgeführt, er bietet aber nicht mehr alle Vorteile wie beispielsweise die des Common Type System und die der automatischen Garbage Collection.

[24] In der Sprache Java existiert dieses Problem nicht, da es hier nur eine Sprache, aber diverse virtuelle Maschinen gibt.

[25] Die Garbage Collection kann die verwendeten Pointer nicht berücksichtigen.

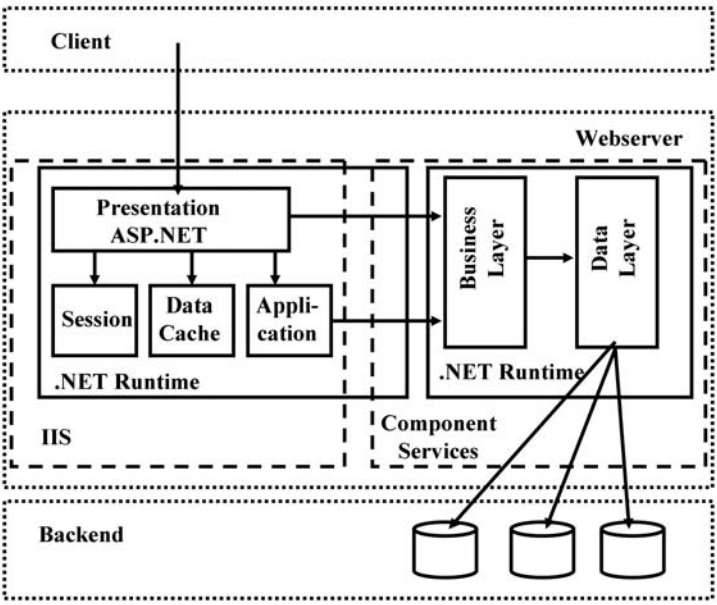

Abb. 12.24: Der .NET Applikation Server

Im Gegensatz zu J2EE ist COBOL, eine der traditionellen Legacysoftwaresprachen, eine Programmiersprache der .NET-Architektur, was dazu führt, dass COBOL-Sourcecode von Legacysystemen aus direkt in eine .NET-Umgebung portiert werden kann.

12.6 Enterprise Application Integration

Die Enterprise Application Integration, kurz EAI genannt, ist im elektronischen Handel sehr wichtig, da die Integration bestehender Applikationen, im Vergleich zur Neuentwicklung, die preisgünstigere Variante ist. Generell gilt, dass Enterprise Application Integration immer dann zum Einsatz kommt, wenn Legacysysteme integriert oder ausgeweitet werden müssen. Die Zielsetzung hinter der Enterprise Application Integration ist es stets, die beteiligten Systeme möglichst lose und idealerweise ohne gegenseitige Abhängigkeiten miteinander zu verbinden. Enterprise Application Integration beschäftigt sich somit stets mit den Integrationsprozessen von bestehenden Softwaresystemen. Diese sind oft sehr heterogen aufgebaut und es bedarf spezieller Techniken, um gemeinsame Integration zu bewerkstelligen. Das bekannteste Beispiel für diese Form der Integration ist EDIFACT, bei dem die Informationen nicht nur elektronisch übermittelt werden, sondern automatisch bestimmte Aktionen innerhalb einzelner Systeme auslösen können.

Das Problem der Enterprise Application Integration lässt sich in die drei Ebenen aufteilen:

- Syntax
- Semantik
- Geschäftsprozesse

Die meisten traditionellen Ansätze, wie z.B. EDIFACT oder DTA, definieren nur die Syntax und die Datentypen. Während die Integration auf technischer und syntaktischer Ebene heute weitgehend durch allgemein akzeptierte Standards gelöst ist, so z.B. CORBA oder MQ-Series, ist die semantische[26] und geschäftsprozessurale[27] Integration zum Teil noch ungelöst. Die drei möglichen Integrationsebenen definieren wiederum drei verschiedene Formen der Enterprise Application Integration:

- Data Integration – Unter der Datenintegration wird sowohl die Verbindung von verschiedenen Applikationen durch Datenaustausch, als auch durch die gemeinsame Nutzung von Datenbanken durch verschiedene Applikationen verstanden. Primär beschäftigt sich die Datenintegration mit den strukturierten Daten, die formalisiert erfasst, verwaltet und verarbeitet werden. Das Kernproblem einer Datenintegration besteht darin, unterschiedliche Datenmodelle für unterschiedliche Applikationen transparent zu machen, dabei können unterschiedliche Typsysteme oder Beschränkungen in den Datentypen sehr hinderlich sein, von dem Problem der Datenqualität ganz zu schweigen.
  Im Bereich der Datenintegration gibt es zwei Grundmechanismen:
  – Database-to-Database,
  – Federated Database.
  Im ersten Fall werden Synchronisationsmechanismen zwischen den beteiligten Datenbanken aufgebaut, auch Replikation genannt. Der größte Aufwand bei der Replikation ist der Aufbau des notwendigen Regelwerkes zur Konfliktbewältigung, da gleiche Daten an unterschiedlichen Orten entstehen können. Oft folgen solche Konfliktlösungsstrategien den Hierarchien in Unternehmen, s. Abschn. 13.14. Im zweiten Fall werden die Datenbanken zu einer großen zusammengezogen, indem die einzelne Applikation nur noch eine virtuelle Datenbank sieht, welche dann physisch wieder in einzelne zerfällt. Die Folge hiervon ist, dass Federation zu einer losen und Database-to-Database zu einer engeren Koppelung führt. Der Einsatz einer Datenintegration führt zu einer schnellen und, im Regelfall, recht preiswerten Lösung, da die einzelne Applikation, im Normalfall, von der Integration unberührt bleibt. Umgekehrt liefert diese Integrationsform keine „echten“ Objekte, da das gesamte Konsistenzregelwerk für Objekte umgangen wird.

[26] XML ist hier der erste Schritt in diese Richtung.

[27] Die Taxonomien der Webservices sind ein möglicher Ansatz für eine pragmatische Integration.

- Business Object Integration – Eine Integration über Geschäftsobjekte ist zwar wesentlich aufwändiger als eine reine Datenintegration, dafür aber sehr viel effektiver. Die Geschäftsobjektintegration erzwingt die Definition einheitlicher Objektmodelle für einzelne Geschäftsobjekte auf fachlicher Ebene. Ein Geschäftsobjekt wird dabei durch die Klasse, Interfaces und Exceptions beschrieben. Die zugrunde liegende Kommunikation ist meist ein höherwertiges Protokoll, wie z.B. CORBA oder IDOC von SAP.
- Business Process Integration – Die Integration von Geschäftsprozessen gestaltet den Workflow auf Geschäftsprozessebene. Die Steuerung der übergeordneten Geschäftsprozesslogik ist hier ein integrativer Bestandteil. Durch die Business Process Integration werden völlig separate, heterogene Applikationen für einen konkreten Geschäftsprozess integriert, indem Geschäftsprozessregeln für die Abwicklung der Geschäftsprozesse definiert werden. Diese Form der Integration verspricht den höchsten Mehrwert, da jetzt völlig neue Geschäftsprozesse und Abläufe auf Basis der vorhandenen Prozessfragmente möglich werden.

Die Enterprise-Application-Integration-Systeme bieten eine große Zahl von Services an, welche komplementär zu den Middlewareprodukten sind. Diese Services sind, individuell betrachtet, üblicherweise relativ einfach. Erst durch ihr „synergistisches“ Zusammenspiel[28] entsteht die Mächtigkeit eines Enterprise-Application-Integration-Systems. Zu diesen Services zählen:

- Interface Services – Die Interface Services werden in den Enterprise-Application-Integration-Systemen durch vorgefertigte Adaptoren geliefert. Die Nutzung solcher Adaptoren kann die Kosten für die Realisierung einzelner Interfaces drastisch reduzieren. Die Java Connector Architecture ist eine Weiterentwicklung der Adaptorenidee. Die Adaptoren mappen die Interfaces der einzelnen zu integrierenden Applikationen auf das generische Interface des Enterprise-Application-Integration-Systems. Die Komplexität des Interfaces mit seinen Daten und Funktionen wird in einer eigenen Softwareschicht gekapselt.
- Transformation Services – Die Transformationsservices erleichtern die Entwicklung von Regeln zur Transformation von Schnittstellen bzw. Datenformaten, basierend auf der Semantik. Hier wird die Semantik einer Applikation in die Syntax einer anderen Applikation umgeformt. Die Transformationsservices ermöglichen die Definition und das Mapping von Geschäftsobjekten.
- Business Process Services – Die Business Process Services ermöglichen es, Schnittstellen und Daten unterschiedlicher Applikationen miteinander zu verbinden und dabei eventuell zu transformieren. Die Business Process Services stellen dabei stets sicher, dass der Geschäftsprozess in seiner Abwicklung den definierten Geschäftsprozessregeln folgt.

[28] Dies ist ein technisches Beispiel für die Emergenz in Systemen, s. S. 268.

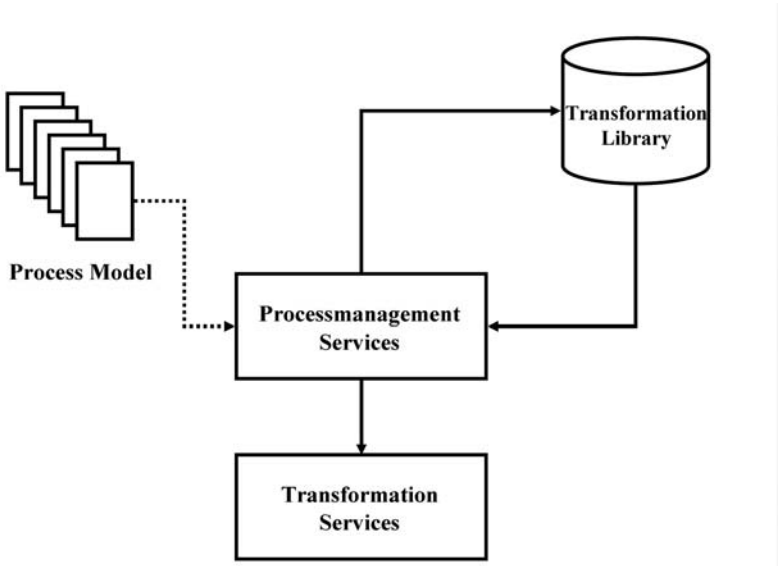

Abb. 12.25: Business Process Integration via Transformation

Alle Enterprise-Application-Integration-Systeme können durch ihre generischen Funktionalitäten beschrieben werden. Diese verschiedenen Funktionalitäten adressieren zugleich auch die erwähnten Integrationskonzepte. Während die Integrations- und Interface-Services eine Voraussetzung zur Datenintegration bilden, ermöglichen Transformationsservices eine Integration durch die Geschäftsobjekte, und die Business Process Services ermöglichen eine Geschäftsprozessintegration in einem Unternehmen.

- Integration Services – Die Integration Services ermöglichen es, zusammen mit den Interface Services eine Integration auf Daten- oder Interface-Ebene bereitzustellen. Dabei können die Schnittstellen oder die Daten auf unterschiedlichsten Applikationen, Betriebssystemen oder Hardwareplattformen implementiert worden sein. Alle Enterprise-Application-Integration-Systeme benutzen für solche Operationen die „klassischen“ Middleware-Systeme, so z.B. CORBA, MQ-Series von IBM oder Tuxedo von BEA. Solche Integration Services stellen meist eine Reihe von Funktionalitäten zur Verfügung, welche stark auf der technischen bzw. infrastrukturellen Ebene angesiedelt sind:
  - Kommunikationsservices – Die Kommunikationsfunktionalität ermöglicht den physischen Transport von Daten zwischen den verteilten Applikationen. Dabei gibt es Unterschiede darin, ob die Kommunikation synchron, d.h. durch direkten Aufruf, oder asynchron via Messages abläuft. Der synchrone Aufruf impliziert stets eine starke und

der asynchrone eine lose Koppelung. Innerhalb der meisten Enterprise-Application-Integration-Systeme wird beim Anbinden von Legacysoftware die lose Koppelung bevorzugt. Dies hat historische Gründe, da ein großer Teil der Legacysysteme in ihrem Kern batchorientiert entwickelt wurde und sich daher nur bedingt für Synchronität eignet.
  - Naming Services – Die Naming Services innerhalb der Enterprise Application Integration haben zwei Funktionen. Zum einen geht es um das Auffinden von Diensten, zum anderen aber auch darum, unterschiedliche Adressierungsformen für den Aufruf umwandeln zu können. Die Modelle an dieser Stelle sind denen aus dem Bereich Messaging eng verwandt.
  - Security – Mit diesem Service wird die Autorisierung, sowie die Authentisierung geregelt.
- Interface Services – Die einzelnen Applikationen, unabhängig davon, ob es sich um Individual- oder COTS-Software handelt, verwenden zur Kommunikation mit anderen Applikationen zumeist unterschiedliche Interface-Technologien. Die Interface Services stellen nun eine Reihe von Funktionen zur Verfügung, welche zur Lösung dieser Aufgabenstellung dienen.
  - Transformation Services – Diese Services unterstützen die Datenintegration durch die Abbildung der Enterprise-Application-Integration-internen Interfaces auf die Schnittstellen der jeweiligen Applikation. Während die Schnittstellen von COTS-Software, wie z.B. SAP R/3, auf Interfaces wie IDOCS oder BAPIs aufbauen, ist die Integration von Individualsoftware sehr viel problematischer. Da Individualsoftware in der Regel nicht für die Integration in eine Gesamtsystemlandschaft konzipiert wurde, fehlen hier die entsprechenden Mechanismen, dies zu bewerkstelligen. Gegenüber dem Enterprise-Application-Integration-System erweist sich die Individualsoftware als stark gekapselt. Die Integration von solchen Applikationen findet entweder über eine Datenbankkoppelung, indem direkt in den Datenhaushalt der zu integrierenden Applikation eingegriffen wird, oder über die Implementierung einer neuen Funktionalität statt.
  - Meta Data Services – Metadatenservices dienen den Geschäftsobjekten, wie auch der Business Process Integration. Sie registrieren, welche Datenstrukturen die einzelnen Applikationen benötigen bzw. bereitstellen. Diese Metadaten bilden somit strukturelle Informationen über die Daten und ihre Verwendung ab.
- Transformation Services – Die Summe der Transformation Services sind der eigentliche Kern jedes Enterprise-Application-Integration-Systems. Sie übernehmen die Daten von den Softwareobjekten, transformieren sie auf andere Interfaces und rufen damit auch andere Softwareobjekte auf. Für diese Transformation Services sind die Metadaten zur Steuerung als auch der eigentlichen Transformation notwendig, diese werden von den Meta Data Services geliefert. Damit dies bewerkstelligt werden kann, sind folgende Bestandteile nötig:

- Ein Metadatenobjektmodell aller zu integrierenden Systeme,
- eine Transformationsbibliothek, welche Informationen über die Datenstrukturen der auszutauschenden Nachrichten enthält.

Als langfristig besonders effektiv hat es sich herausgestellt, innerhalb des Enterprise-Application-Integration-Systems ein kanonisches Format für die einzelnen Geschäftsobjekte zu definieren; dies ist analog der Multichannelarchitektur, s. Abschn. 12.3.2. Die Größe der Transformationsbibliotheken wächst in diesem Fall mit

$$n_{Transformationen} = \mathcal{O}\left(n_{Applikation}\right).$$

Im Vergleich hierzu steigt die Zahl der direkten Verknüpfungen zwischen den einzelnen Softwareobjekten mit:

$$n_{Transformationen} = \mathcal{O}\left(n^2_{Applikation}\right).$$

Bei einer großen Anzahl von Applikationen n erweist sich die Koppelung über ein kanonisches Format folglich am günstigsten. Ausnahme ist hier die Koppelung zwischen genau zwei Softwareobjekten, in diesem Fall lohnt sich der Overhead für die Einrichtung des kanonischen Formats nicht.

- Identification Services – Die Identification Services ermöglichen es, eingehende Message zu validieren und ihre Adressaten zu identifzieren.
- Routing Services – Sehr häufig kann eine Transformation mehrere Empfänger besitzen, oder es wird abhängig vom Inhalt der Message ein bestimmter Empfänger gesucht. Dies taucht vor allem bei der Applikation von Geschäftsprozessregeln auf. Ein Enterprise-Application-Integration-System benötigt daher entsprechende Services, die in der Lage sind, Prozeduren auszuführen, Zugriff auf den Inhalt eingehender Daten zu gewährleisten und ein dynamisches Routing an verschiedene Applikationen, je nach dem Ergebnis, zu unterstützen.
- Transaction Services – Neben einfachen Transaktionen ermöglichen die Transaction Services auch den Einsatz von verteilten Transaktionen. Allerdings setzt dies den Einsatz eines Transaktionsmonitors mit einer 2-Phase-Commit-Funktionalität voraus.

- Business Process Management Services – Völlig analog der Art und Weise, wie die Transformation Services Strukturen von Daten integrieren, koordinieren Business Process Management Services die Strukturen von Transformationen. Sie führen Transformationsroutinen aus, die zuvor in einem Geschäftsprozessmodell definiert wurden, s. Abb. 12.25. Theoretisch lassen sich dadurch verteilte Applikationen integrieren, um aus deren Prozeduren neue Applikationen zu realisieren.
- Runtime Environment – Enterprise-Application-Integration-Systeme bilden eine Abstraktionsschicht zusätzlich zu den sowieso vorhandenen Applikationen. Damit diese Komplexität beherrscht werden kann und das System sich während der Laufzeit durch Performanz, Skalierbarkeit, Verfügbarkeit und Zuverlässigkeit auszeichnen kann, müssen in dieser Umgebung folgende Funktionalitäten vorhanden sein:

- Scaling – In den Bereich des Skalierens gehört das Verfahren zum Load Balancing. Hierdurch wird die Arbeitslast auf mehrere Server verteilt. Das Load Balancing führt durch eine Verteilung der Transformationen auf verschiedene Server zu einer besseren Auslastung der einzelnen Server, da die Transformationen in der Regel sehr CPU-intensiv sind.
- Availability – Das Failoververfahren dient zur höheren Verfügbarkeit des Gesamtsystems durch verbesserte Verfügbarkeit der einzelnen Teile. Erreicht wird dies durch ein automatisches Routing von Anfragen an Backup-Server.
- Distribution – Für die Implementierung eines verteilten Enterprise-Application-Integration-Systems existieren zwei grundsätzliche Möglichkeiten: Zum einen alle Transformationen zentral an einer Stelle anzusiedeln, und zum anderen die Transformation auf den Systemen der jeweiligen Applikation zu belassen. Die Konsequenz aus der Transformationsverteilung ist eine Verteilung der Metadateninformationen, welche von den Transformationen benötigt werden.
- Monitoring – Analog zu den „klassischen“ Middlewaresystemen muss ein Enterprise-Application-Integration-System auch eine Komponente zum Monitoring der Bestandteile enthalten.

Die hier angesprochenen Enterprise-Application-Integration-Systeme und die später besprochenen Webservices, s. Abschn. 12.9, unterscheiden sich, oberflächlich betrachtet, zunächst kaum voneinander. Der Hauptunterschied liegt in dem Einsatz von Standardprotokollen für die Kommunikation zwischen den einzelnen Applikationen. Im Fall der Webservices werden WSDL und SOAP genutzt, während ansonsten proprietäre Protokolle vorzufinden sind. Für den praktischen Einsatz sind diese beide Ansätze jedoch nicht konkurrierender Natur, im Gegenteil, sie ergänzen sich. Die Enterprise-Application-Integration-Systeme decken mit ihrer Fähigkeit zu Transaktionen und komplexen Objekt- bzw. Prozesszuständen die Anforderungen der internen Services sehr gut ab, während die Webservices sich auf Grund ihrer hohen Standardisierung und Flexibilität, sowie der Einfachheit ihrer Interfaces, für den überbetrieblichen Einsatz vorzüglich eignen.

Große Systeme, welche aus einer Enterprise Application Integration heraus entstehen, zeichnen sich meistens durch eine sternförmige Topologie aus. Die Komplexität der Enterprise-Application-Integration-Systeme wird primär durch die Tatsache bestimmt, dass sie dazu dienen, Legacysysteme zu integrieren mit der Folge, dass die Gesamtentropie sich zu:

$$S_{EAI} \geq \sum_{Applikationen} S_i$$

ergibt. Folglich ist die Gesamtentropie mit einer Enterprise Application Integration stets größer als die Summe der Teilentropien.

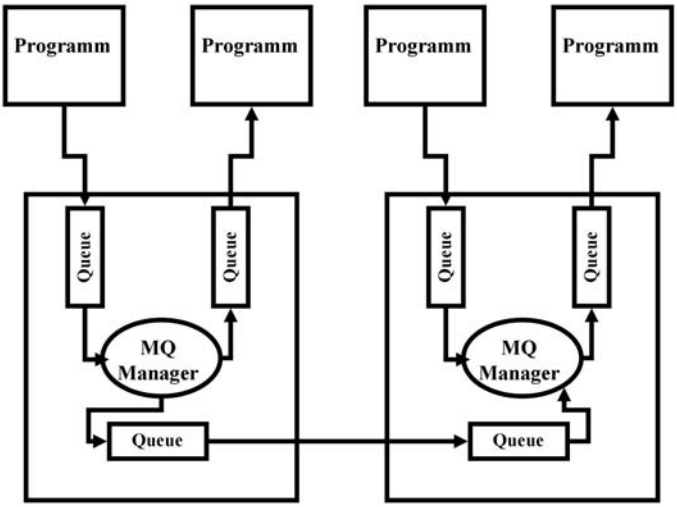

Abb. 12.26: Die MQ-Series-Architektur

12.7 MQ-Series

Einen gewissen Sonderfall stellt das Produkt MQ-Series von IBM dar. Es wurde spezifisch für den Einsatz von Message-Queueing und Transactionqueues entwickelt, MQ-Series kann jedoch auch als eine Infrastruktur zur Enterprise Application Integration genutzt werden. Es existieren auch Konkurrenzprodukte, wie die Microsoft Message Queue oder TIBCO Rendezvous, aber diese sind strukturell sehr ähnlich.

Bei einer solchen Message Oriented Middleware, s. Abb. 12.26, werden die Nachrichten an andere Programme nicht diesem Programm direkt, sondern an den Message Queue Manager gesendet, welcher dann anhand eines internen Regelwerkes entscheidet, in welche Queues diese Nachricht transportiert wird. Das gerufene Programm wiederum hat die Aufgabe, die Nachricht aus seiner eigenen Inputqueue abzuholen und sie dann subsequent zu verarbeiten. Durch diese Technik lassen sich beliebig viele andere Programme anbinden, aber es lässt sich auch die Asynchronität gut unterstützen. Wird eine solche Message Oriented Middleware im Zusammenhang mit XML-Nachrichten konsequent eingesetzt, entsteht ein SOA-ähnliches System, welches intern einen Bus-Charakter besitzt.

Das Produkt MQ-Series wird in Client-Server- oder in beliebig verteilten Umgebungen eingesetzt. Die zu einer Applikation gehörenden Programme können in unterschiedlichen Rechnerarchitekturen auf verschiedensten Plattformen ausgeführt werden. Die Applikationen sind von einem System oder

einer Plattform zu einem anderen System oder einer anderen Plattform übertragbar. Die Programme werden in verschiedenen Programmiersprachen einschließlich Java und COBOL geschrieben. Für alle Plattformen ist derselbe Queuing-Mechanismus gültig.

MQ-Series kann aufgrund der Nutzung von Queuing als eine indirekte Kommunikation betrachtet werden. Der einzelne Softwareentwickler ist nicht in der Lage, den Namen der Zielapplikation, zu der eine Message gesendet wird, zu spezifizieren. Es werden ausschließlich logische Queue-Namen als Ziele genutzt. Es können ein oder mehrere Eingangsqueues und verschiedene Ausgangsqueues für eine Applikation existieren. Die Ausgangsqueues enthalten Informationen, die in anderen Applikationen verarbeitet werden sollen, oder Antworten für Applikationen, welche die Transaktion initiiert haben. Bei einer Applikation ist es nicht erforderlich, dass der Softwareentwickler die Zielapplikation detailliert kennt. Es ist belanglos, ob die Applikation momentan im Betrieb ist oder keine Verbindung zu ihr besteht. Die Applikation sendet ihre Nachricht an die Queue, welche mit der Applikation verbunden ist. Letztere kann zur Zeit des Requests verfügbar sein oder nicht. MQ-Series beobachtet den Transport zur Zielapplikation und startet diese, wenn es eventuell notwendig ist. Wenn die Zielapplikation nicht verfügbar ist, steht die Message in der Queue und wird später verarbeitet. Abhängig davon, ob die Verbindung zwischen zwei Systemen hergestellt ist oder nicht, befindet sich die Queue entweder in der Zielmaschine oder in dem sendenden Rechner. Eine Applikation kann prinzipiell über mehrere Tage laufen, oder sie wird getriggert, d.h. sie wird automatisch gestartet, wenn genau eine oder eine spezifische Anzahl von Nachrichten in der Queue sind.

In MQ-Series werden verschiedene Queue-Arten angeboten. Diese sind:

- Lokale Queue – Eine Queue heißt lokal, wenn sie sich unter der Kontrolle des Queue-Managers befindet, mit dem die sendende Applikation verbunden ist. Sie wird zur Speicherung von Nachrichten für Applikationen benutzt, welche denselben Queue-Manager verwenden, beispielsweise haben zwei Applikationen je eine Queue für eingehende und ausgehende Messages. Da der Queue-Manager beide Programme bedient, sind alle vier Queues lokal. Beide Applikationen müssen nicht physisch auf demselben Rechner laufen, in diesem Fall wird die Queue einer Servermaschine genutzt.
- Cluster-Queue – Eine Cluster-Queue ist eine lokale Queue, welche überall in einem Cluster von Queue-Managern bekannt ist, folglich kann jeder Queue-Manager, der zu dem Cluster gehört, direkt Nachrichten an sie senden.
- Remote-Queue – Eine Queue heißt remote, wenn sie zu einem anderen Queue-Manager gehört. Die Remote-Queue stellt keine reale Queue dar. Die Remote-Queue kann genauso wie eine beliebig andere lokale Queue benutzt werden. Der MQ-Series-Administrator definiert, wo diese Queue sich aktuell befindet. Remote-Queues werden mit einer Transmission-Queue

verknüpft. Keine Applikation ist in der Lage, Nachrichten direkt aus einer Remote-Queue zu lesen.

- Transmission-Queue – Bei der Transmission-Queue handelt es sich um eine spezielle lokale Queue. Die Transmission-Queues werden als Zwischenschritt benutzt, wenn Nachrichten an Queues gesendet werden, welche zu unterschiedlichen Queue-Managern gehören. Typischerweise wird nur eine Transmission-Queue für jeden Remote-Queue-Manager genutzt. Für alle Nachrichten, die auf Queues mit einem Remote-Queue-Manager als Besitzer abgelegt werden, erfolgt die Abspeicherung zunächst in einer Transmission-Queue dieses Remote-Queue-Managers. Die Nachrichten werden dann von der Transmission-Queue gelesen und an den Remote-Queue-Manager weitergeleitet. Bei der Benutzung von MQ-Series-Clusters gibt es nur eine Transmission-Queue für alle Nachrichten, welche an alle anderen Queue-Manager in dem Cluster gesendet werden. Die Transmission-Queues werden nur intern von dem Queue-Manager genutzt. Wenn eine Applikation eine Remote-Queue öffnet, erhält diese die Queue-Attribute von der Transmission-Queue, folglich wird das Schreiben von Nachrichten in eine Queue durch eine Applikation durch die Transmission-Queue-Charakteristiken ermöglicht.
- Dynamic Queue – Eine Dynamic-Queue wird spontan definiert, wenn sie von einer Applikation benötigt wird. Dynamic-Queues können vom Queue-Manager automatisch gelöscht werden, sobald die Applikation endet. Es handelt sich hierbei um lokale Queues. Diese werden oft zur Speicherung von Zwischenergebnissen benutzt, eine Art Scratchdatei.
- Alias-Queue – Alias-Queues sind keine realen Queues sondern nur Definitionen. Sie werden verwendet, um derselben physikalischen Queue unterschiedliche Namen zuzuweisen.

Das Schreiben und Lesen in eine Queue ist völlig analog der Benutzung einer Datei. Aus diesem Blickwinkel betrachtet sind viele MQ-Programme dateizentrisch organisiert.

12.8 Service Oriented Architecture

Eine Service Oriented Architecture, kurz SOA genannt, modelliert das gesamte Unternehmen als eine Ansammlung von Services, welche über das Unternehmen verteilt und jedem zugänglich sind. Dieser Gedanke stellt eine radikale Abkehr von den traditionellen stove-pipe-Architekturen dar, s. Kap. 6.

Große monolithische Systeme werden in kleinere Teilsysteme zerlegt. Diese Teilsysteme besitzen ihrerseits wiederum Komponentencharakter. Folglicherweise sind sie in gewissem Sinne autark. Der Aufruf dieser Komponenten innerhalb der SOA geschieht ausschließlich über öffentlich bekannte Standardprotokolle. Eine der bisher am weitesten beachteten Implementierungsformen solcher Service Oriented Architectures sind die Webservices, s. Abschn. 12.9,

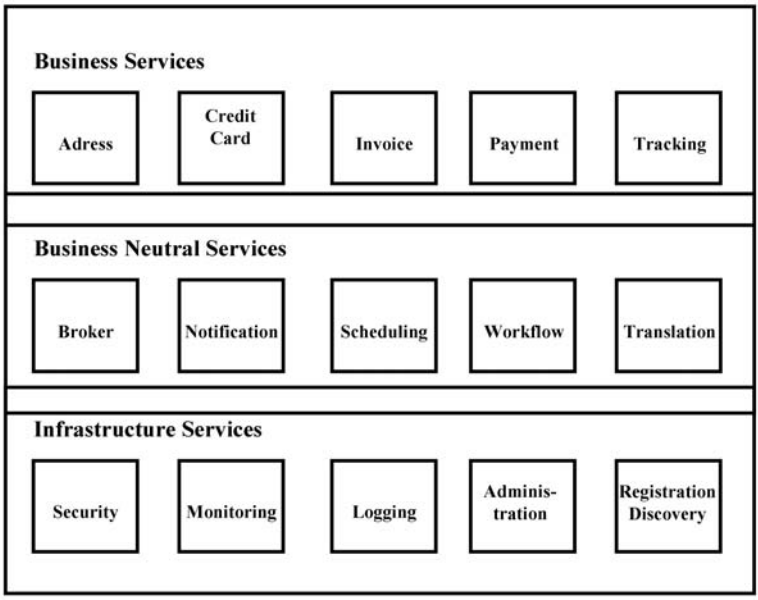

Abb. 12.27: Basis-SOA-Komponenten

andere mögliche Implementierungsformen sind Enterprise Java Beans, s. Abschn. 12.4, oder auch CORBA, es existieren aber auch .NET Implementierungen, s. Abschn. 12.5.

Die allgemeine Service Oriented Architecture, s. Abb. 12.27, kann als eine 3-Tier-Architektur angesehen werden. In diesem Kontext werden die Schichten nicht als Schichten, sondern als Service Layers bezeichnet.

Neben diesen statischen Funktionalitäten haben Service Oriented Architectures ein zweites Charakteristikum, ihre Dynamik. Innerhalb der SOA werden alle Services nicht statisch, sondern ausschließlich dynamisch gebunden, die Folge hiervon ist die Notwendigkeit, folgende Konzepte zu etablieren:

- Publishing – Die Fähigkeiten und Existenz eines neuen Services, bzw. die geänderten Eigenschaften eines bestehenden Services müssen dem gesamten Unternehmen bekannt sein, damit sie überhaupt genutzt werden können. Diese Bekanntmachung bezeichnet man als Publishing.
- Finding oder Discovery – Wie wird ein bestehender Service gefunden? So einfach dies klingt, es ist recht komplex, da die Auffindung des „richtigen“ Services eine hohe semantische Leistung darstellt.
- Binding – Der aufgefundene Service muss aufgerufen und sein Interface genutzt werden, diesen Vorgang nennt man Binding. Das Binden an den bestehenden Service ist vermutlich der einfachste Teil einer Service Oriented Architecture.

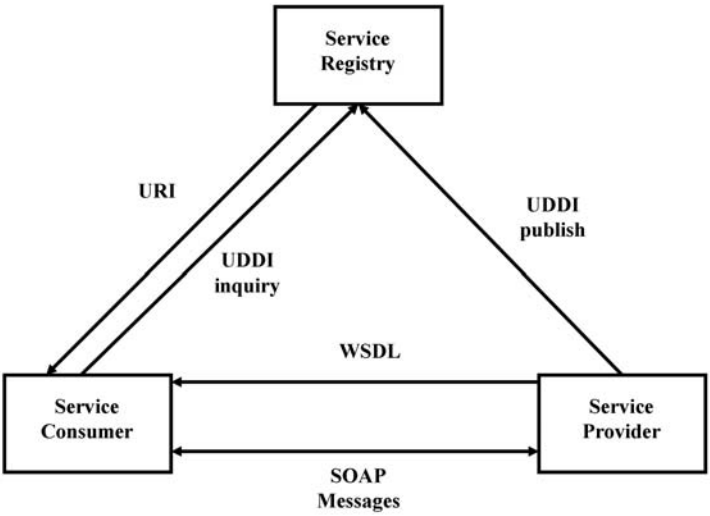

Abb. 12.28: Die drei Rollen bei den Webservices

12.9 Webservices

Eine spezielle Form der Service Oriented Architectures sind die Webservices. Da diese auf öffentlich verbreiteten und häufig genutzten Protokollen aufbauen, sind sie recht einfach und quasi universell zu integrieren. Bei den Webservices handelt es sich um eine der möglichen Implementierungsformen einer SOA.

12.9.1 Modell

Die Webservice-Architektur basiert wie jede SOA-Architektur auf den Wechselwirkungen zwischen drei verschiedenen Beteiligten, dem Service Provider, auch Server genannt, dem Service Requester, auch Client genannt, und dem Service Registry, s. Abb. 12.28.

Der Service Provider stellt die einzelnen Webservices zur Verfügung und publiziert sie via UDDI, s. Abschn. 12.9.4, und WSDL, s. Abschn. 12.9.6, im Service-Registry. Der Service Requester wiederum findet seine gesuchten Services mit Hilfe der Sprache WSDL und der UDDI im Service Registry und nutzt die dortigen Interface-Definitionen, um sich gegen den Service Provider zu binden.

Die konkrete Nutzung der Webservices läuft dann transparent über das Netzwerk mit Hilfe von SOAP, s. Abschn. 12.9.3, zwischen dem Service Requester und dem Service Provider, s. Abb. 12.29.

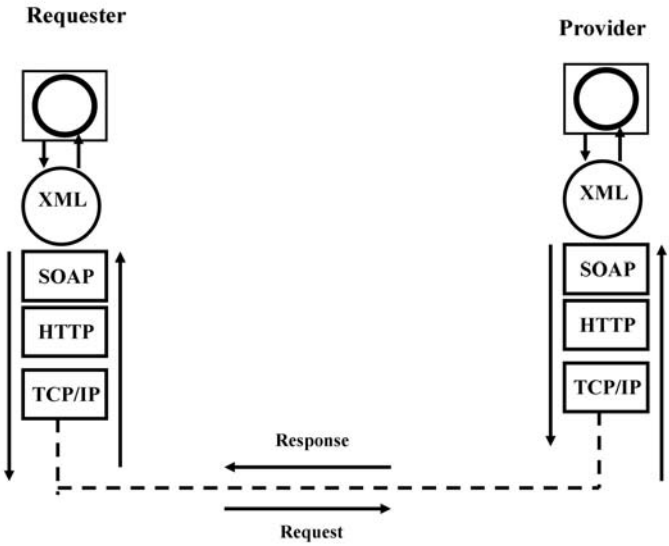

Abb. 12.29: Kommunikation der Webservices via SOAP

Im Rahmen von Legacysystemen ist eine der Herausforderungen, die Gemeinsamkeiten verschiedener Anforderungen auf der einen Seite und die Gemeinsamkeiten der jeweiligen vorhandenen Implementation auf der anderen Seite zu isolieren. Neben diesen „Core Assets“, sind noch die Variationspunkte für die entstehenden Webservices zu bestimmen. Das Vorgehen entspricht daher einer Produktlinienstrategie, s. Kap. 9. So werden sukzessiv Funktionalitäten isoliert und dann, meistens per Wrapper, als Webservices allen zur Verfügung gestellt. So entsteht, langfristig gesehen, ein lose gekoppeltes System aus Services. Obwohl, im Vergleich zum originären Legacysystem, der einzelne Service relativ klein ist, hat das Gesamtsystem eine recht hohe Entropie und Komplexität.

Wenn die Entropie S_{total} betrachtet wird, so setzt sie sich aus zwei Teilen zusammen, der Entropie der einzelnen Services und der Entropie der Koppelungen, s. Gl. 12.3. Die Komplexität des Gesamtsystems wird in fast allen Fällen erhöht, so dass ein solcher Schritt nur eine Zwischenlösung für ein Legacysystem darstellen kann.

$$\begin{aligned} S_{total}^{WS} &= \sum_{i \in Service} S_i^{WS} + \sum_{j \in Coupling} S_j \\ &\approx S_{total}^{Legacy} + \sum_{j \in Coupling} S_j \end{aligned} \tag{12.3}$$

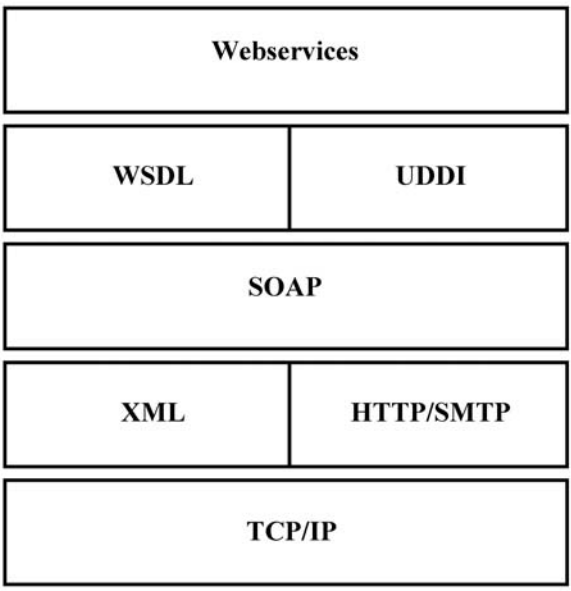

Abb. 12.30: Der Webservice-Protokollstack

$$\gg S_{total}^{Legacy}.$$

Die Koppelungen enthalten jetzt eine Form der „externalisierten“ Komplexität, da für die Gesamtentropie das gesamte System betrachtet werden muss. Aber nicht nur die Software als solche wird davon betroffen sein, auch organisatorische Fragen werden durch die Webservices berührt:

- Wem gehören die Daten?
- Wem gehört der Code?
- Wer hat die Verantwortung für welchen Service?
- Wie funktionieren die Service Level Agreements?

Insgesamt betrachtet ist auch der technische Overhead zur Zeit nicht besonders gering, die Performanz dürfte etwa einen Faktor 10 bis 100 langsamer sein als eine vergleichbare direkte Koppelung, bei einem Transport über öffentliche Netze käme noch ein zusätzlicher Overhead auf Grund der Verschlüsselung hinzu.

12.9.2 Services

Im Vergleich zu den mehr allgemein orientierten Komponenten sind die Webservices ein spezieller Fall, denn hierunter versteht man lose gekoppelte ausführbare Applikationen, welche dynamisch über ein TCP/IP-Protokoll

eingebunden werden. Aus einer anderen Perspektive beurteilt, sind Webservices eine mögliche Implementierungsform von einer Service Oriented Architecture, s. Abschn. 12.8. Um die Definition zu vervollständigen, beschränken wir unsere Webservices auf folgende Konstellation:

Ein Webservice ist eine Server-Applikation, die über das XML-Protokoll SOAP mit seinen Clients kommuniziert.

Die offizielle Definition von Webservices ist laut dem World Wide Web Consortium:

> *... software application identified by a URI, whose interfaces and binding are capable of being defined, described and discovered by XML artifacts and supports direct interactions with other software applications using XML based messages via Internet based protocols.*

Welche Voraussetzungen an Technik auf der Protokollebene sind für die Webservices notwendig?

Obwohl Webservices auf Standardprotokollen aufbauen, brauchen sie eine gewisse Menge von Voraussetzungen. Diese Voraussetzungen bauen systematisch aufeinander auf; dies ist der so genannte Webservice-Protokollstack, s. Abb. 12.30. Der Webservice-Protokollstack braucht von unten nach oben betrachtet:

- TCP/IP – Diese logische Basisverbindung stellt das Rückgrat jeder Kommunikation im Webservice-Umfeld dar.
- XML – Die Protokollsprache XML dient für den Nachrichtenaustausch der einzelnen Webservice-Aufrufe, s. Abschn. 12.3.1.
- HTTP – Das HTTP nutzt das TCP/IP als darunter liegendes Transportprotokoll. Durch SOAP wird HTTP sowohl für den Aufruf, als auch den Austausch der XML-Dateien bzw. XML-Datenströme genutzt.
- SOAP – s. Abschn. 12.9.3
- UDDI – s. Abschn. 12.9.4
- WSDL – s. Abschn. 12.9.6

Aufbauend auf diesen Protokollstack werden die Webservices implementiert. Interessanterweise sind die Teile TCP/IP, HTTP, XML und SOAP so weit verbreitet, dass sie in vielen Bereichen den heutigen De-facto-Standard darstellen.

Obwohl komponentenbasierte Architekturen schon ein gewisses Alter haben, ist das wirklich Neue an den Webservices ihre lose Koppelung. Hierin unterscheiden sie sich drastisch von traditionellen Komponentenarchitekturen wie beispielsweise CORBA. Die zweite interessante Linie, die es zu betrachten lohnt, ist die der Enterprise Application Integration, s. Abschn. 12.6. Diese ist im Vergleich zu Webservices nicht so erfolgreich, da die Investitionsvoraussetzungen für eine Enterprise Application Integration sehr viel höher sind als für die Webservices. Aus technischer Sicht erzeugt eine Enterprise Application Integration keine flexiblen generischen Interfaces, welche eine ideale Voraussetzung für Wiederverwendung darstellen. Mittelfristig wird es zu einer

gewissen Koexistenz zwischen beiden Techniken kommen, mit der Enterprise Application Integration innerhalb eines Subsystems und den Webservices innerhalb des Intra- und Internets.

Auch die großen monolithischen Legacysysteme können von den Webservices genutzt werden. In diesem Fall erhalten die Legacysysteme zusätzliche Interfaces, die Teile ihrer monolithischen Funktionalität als Webservices zur Verfügung stellen. Dieses Vorgehen ist recht kostengünstig und schnell zu bewerkstelligen. Dieser Trend lässt sich gut daran ablesen, dass heute alle großen ERP- und CRM-Hersteller schon SOAP-Interfaces für ihre Software besitzen oder zumindest angekündigt haben.

In diesem Sinne sind Webservices Geschäftsprozessimplementierungen, welche im Internet über wohldefinierte Interfaces sowie über standardisierte Internetprotokolle zur Verfügung stehen. Diese Webservices erlauben es einem Unternehmen, seine Dienstleistungen einer großen Anzahl von Nutzern zur Verfügung zu stellen. Durch die Nutzung von standardisierten Internetprotokollen geschieht dies in einer einfachen und effektiven Art und Weise.

Obwohl eine Reihe von verschiedenen Internetprotokollen existieren, hat sich XML als der De-facto-Standard für Webservices herauskristallisiert. Das XML spielt die zentrale Rolle bei der Definition, Implementierung und Ausführung der Webservices, s. Abschn. 12.3.1.

Historisch gesehen ist die Idee der Webservices nicht neu, schon CORBA hat ähnliche Mechanismen unterstützt. Was aber hierbei neu ist, ist die Einfachheit und Effektivität der Nutzung und der Gebrauch der standardisierten Internetprotokolle. Webservices basieren heute auf XML und dem plattformunabhängigen SOAP-Protokoll, Abschn. 12.9.3. Genauso wie HTML sind Webservices sehr einfach zu nutzen und hochflexibel. Sie sind einfach und gleichzeitig universell genug, damit sie von einer großen Gemeinde genutzt werden können. Diese Einfachheit hat sie so populär gemacht, dass sie schneller ein De-facto-Standard wurden, als es ihrem eigentlichen Reifegrad entsprach.

Die Erstellung von Webservices folgt zwei verschiedenen Implementierungsstilen:

- Dokumentenbasiert: bei dieser Variante steht der Austausch großer XML-Dokumente im Vordergrund. Starke Asynchronität und ein hoher Anteil an statischer Information mit großen Bytemengen pro Service-Aufruf kennzeichnen diesen Stil.
- RPC-basiert: der RPC-basierte Stil ist sehr ähnlich dem CORBA bzw. DCOM und RMI. Hier ist die einzelne Bytemenge je Service-Aufruf eher klein und die Funktionalität steht im Vordergrund, was zu einer feinen Granularität bei den Webservices führt.

Ganze Bereiche, wie beispielsweise Transaktionssicherheit oder 2-Phase-Commit, sind im Rahmen von Webservices heute noch ungeklärt. Die Einführung von Transaktionen in Webservices haben spezielle Charakteristika. Webservices brauchen langlebige, zum Teil beliebig lange andauernde, und

komplexe Transaktionen. Die Ausführung einer Webservice-Transaktion kann Tage, ja sogar Wochen dauern. Solche Typen von Transaktionen werden oft als Geschäftsprozesstransaktionen oder als Transactional Workflow bezeichnet. Das Ergebnis der Transaktion muss innerhalb eines Workflows nicht beendet sein, was den Service recht komplex macht. Im Bereich der Transaktionen zeigt sich die Reife von CORBA, da hier die einzelnen Infrastrukturteile schon vorhanden sind. Gleichzeitig wird aber der Nachteil der Proprietät von CORBA offensichtlich.

Im Vergleich zu den Enterprise-Application-Integration-Systemen, s. Abschn. 12.6, fehlen den heutigen Webservices Funktionalitäten in den Infrastruktur- und Managementbereichen.

Die Einführung eines Webservice-Registers ermöglicht die rasche Nutzung jenseits eines simplen RPC-Services. Im Gegensatz zu Komponenten in einem Application Server sind die Webservices relativ grob-granular[29] und self-contained, d.h. sie nutzen keine klassischen Interfaceerweiterungsmechanismen. In diesem Umfeld ist der Ablauf einer Sitzung relativ einfach:

1 Durchsuchen des Webservice-Registers
2 Einbinden des gesuchten Webservices
3 Aufruf des Webservices
4 Antwort des Webservices auswerten
5 Ende der Verbindung

Bei dieser einfachen Struktur ist es unwahrscheinlich, dass ein komplexer Client einzig aus Webservices aufgebaut werden kann, obwohl es durchaus sinnvoll sein kann, mehrere Webservices zu nutzen. Die recht lose Koppelung der Webservices untereinander behindert die Entwicklung völlig neuer Geschäftsprozesse auf Grundlage der Webservices.

12.9.3 SOAP

Das Simple Object Access Protocol, kurz SOAP genannt, ist ein einfaches Protokoll für den Austausch von Informationen in einer dezentralisierten, verteilten Softwareumgebung. Das SOAP-Protokoll basiert auf XML und wurde vom W3C-Konsortium verabschiedet.

In der heutigen Praxis existieren noch einige Probleme:

- viele SOAP-Toolkits implementieren nur eine Untermenge der SOAP-Spezifikation bzw. der XML-Spezifikation.
- ein Teil der SOAP-Spezifikation ist optional, beispielsweise die Typeninformation für die encodierten Parameter. Dieser optionale Unterschied zwischen unterschiedlichen Implementierungen kann sich zu einer großen Inkompatibilität ausweiten.

[29] Die bisher implementierten Webservices bieten keinerlei komplexe fachliche Funktionalitäten an. Sie stellen in den meisten Fällen eher eine Form der Machbarkeitsstudie dar.

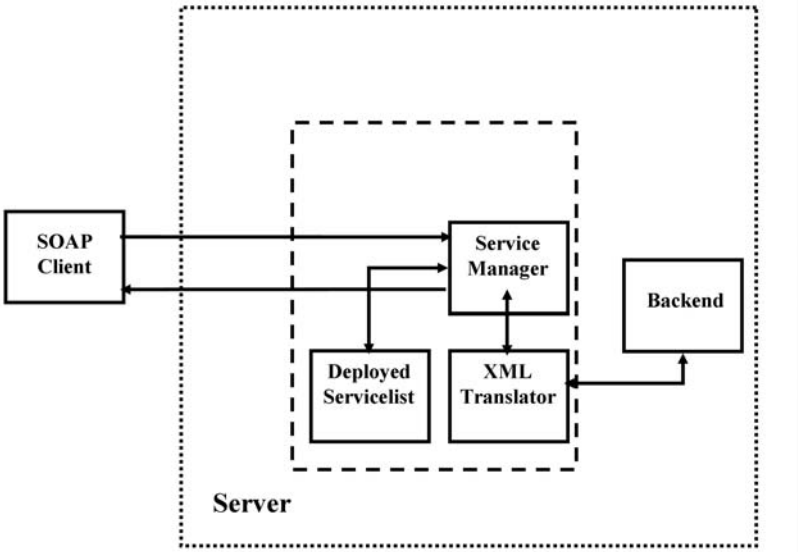

Abb. 12.31: Kommunikation mit SOAP

Damit, speziell im heterogenen Umfeld, die Interoperabilität sichergestellt werden kann, ist für die Integration von einer Applikation zu einer anderen ein Protokoll nötig, welches Implementierungsdetails und Plattformabhängigkeiten negieren kann. Das zur Zeit einfachste Protokoll für Integration und Interoperabilität ist SOAP. In der B2B-Kommunikation braucht jedes Unternehmen nur eine Seite des SOAP-Channels, welcher die Verbindung aufrechterhält, zu implementieren.

Eine typische SOAP-Kommunikation, s. Abb. 12.31, besteht aus folgenden Teilen:

- der eigentliche Webservice
- SOAP-Client – Der SOAP-Client ist eine Applikation, welche in der Lage ist, einen SOAP-Request an einen SOAP-Server via HTTP zu senden. Der SOAP-Request ist eine mögliche Form der Nachrichten, die andere Form, der SOAP-Response, wird vom SOAP-Server zurückgesandt.
- SOAP-Server – Der SOAP-Server ist auch eine Applikation, welche in der Lage ist, auf einen Request zu reagieren. Auf den ursprünglichen Request schickt der Server den SOAP-Response. Der SOAP-Server braucht drei verschiedene Teile:
  - Service-Manager – Der Service-Manager ist verantwortlich für das Management der Services gegen die Requests. Er liest den Request und ermittelt, ob der Service bei ihm vorhanden ist. Hierzu wird die Deployed Service List benötigt. Falls der Service vom Server tatsächlich

zur Verfügung gestellt wird, nutzt der Server den XML-Translator, um die Nachricht für die konkrete Applikation zugänglich zu machen. Die Antwort des Services wird wiederum vom XML-Translator gekapselt und im Rahmen einer SOAP-Response dem Client als XML-Dokument übermittelt.
- Deployed Service List – Diese Liste enthält die momentanen Services, die zur Verfügung stehen.
- XML-Translator

Obwohl dieses Protokoll relativ einfach strukturiert ist, ist es gerade diese Einfachheit, welche SOAP so erfolgreich macht. Das SOAP-Protokoll hat noch eine zweite interessante Eigenschaft, es lässt sich auch asynchron, so z.B. via E-Mail, nutzen.

Die Eigenschaft des SOAP-Protokolls, zustandslos zu sein, hat zur Konsequenz, dass es einfacher zu verwenden und schneller zu implementieren ist als ein vergleichbares zustandsbehaftetes Protokoll. Das weit verbreitete Vorurteil, dass zustandslose Services besser skalieren als zustandsbehaftete Services, ist, wenn überhaupt, nur für sehr einfache Services gültig. Services wie Time Server oder ähnlich gelagerte, welche memoryresident ohne Plattenzugriff oder Transaktionen auskommen, sind skalierbar. Bei allen anderen, und das sind im Allgemeinen alle geschäftsrelevanten Vorgänge, spielen komplexe Algorithmen oder Datenbankzugriffe eine Rolle, so dass die vorgebliche Zustandslosigkeit der SOAP-Implementierung irrelevant geworden, bzw. die Skalierbarkeit aufgehoben ist.

Aber aus dieser Zustandslosigkeit erwachsen auch einige Nachteile. Auf Dauer ist ein zustandsloses Protokoll sehr unpraktisch, da wichtige Elemente wie Transaktionsverhalten oder Parallelität nur sehr schwierig in einem zustandslosen Protokoll implementiert werden können.

Die Aktualität und Zugkraft von SOAP lässt sich auch daraus ableiten, dass es mittlerweile einige Unternehmen gibt, welche ihre CORBA-Applikationen auf Webservices via SOAP abbilden. Dies ist relativ einfach, da die dem CORBA zugrunde liegende IDL, die Interface Definition Language, sich recht gut nach WSDL abbilden lässt. Das so entstehende System kanalisiert mit Hilfe eines Gateways die SOAP-Aufrufe in das CORBA-Legacysystem.

12.9.4 UDDI

Die Abkürzung UDDI steht für Universal Description, Discovery and Integration. Das UDDI-Projekt versucht, die Interoperabilität und Verfügbarkeit von Webservices zu verstärken. Das UDDI adressiert spezifisch das Problem des Auffindens von Services, wie werden nämlich Services im Internet gefunden und genutzt, wobei das konkrete Interface des jeweiligen Services durchaus sehr unterschiedlich aussehen kann. Auf Grund des dynamischen Charakters von UDDI werden Services für alle zum jeweiligen erwünschten Zeitpunkt zur

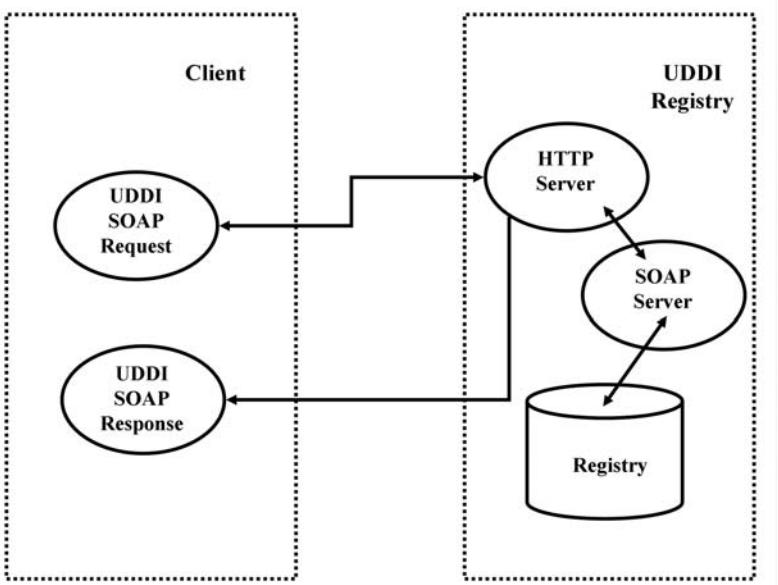

Abb. 12.32: Nachrichtenfluss in UDDI

Verfügung gestellt. Außerdem ermöglicht UDDI die Vergleichbarkeit konkurrierender Webservices, beispielsweise in Bezug auf Preis und Leistungsfähigkeit.

Das UDDI basiert inhaltlich auf SOAP und XML. Genauer gesagt, baut UDDI auf einer Netzwerktransportschicht und einer SOAP-basierten XML-Nachrichtenschicht auf. Die Nutzung von WSDL als Interface-Definitionssprache für die einzelnen Webservices, welche mit Hilfe von UDDI publiziert werden, ermöglicht einen hohen Grad an Austauschbarkeit.

Das UDDI ist eine Registry, welche die zugänglichen Definitionen von Unternehmen und deren Webservices besitzt. Außerdem sind branchenspezifische Informationen in Form einer Taxonomie enthalten. Das zusätzliche Business Identification System macht die Auffindung der einzelnen Unternehmen leichter. Das UDDI gibt ein Programmiermodell und Schema für die Kommunikation mit der Registry vor, s. Abb. 12.32. Alle Schnittstellen in der UDDI-Spezifikation sind in XML definiert, welche in eine SOAP-Envelope eingebettet ist und als Basistransportschicht das HTTP nutzt.

Bevor die Registry von einzelnen Webservice-Providern programmatisch beschickt werden kann, muss sie mit den branchenspezifischen, technischen Modellen, den so genannten tModels, bestückt werden. Die tModels enthalten die branchenspezifische Semantik der Datenelemente und bilden jeweils eine Taxonomie. Wenn die tModels vorhanden sind, kann ein Webservice-Provider sein Unternehmen und die Webservices, welche es anbietet, innerhalb eines technischen Modells registrieren. Jeder Webservice erhält einen Unique Uni-

versal Identifier, UUID, der während der gesamten Lebenszeit des Webservices konstant bleibt. Die Webservice-Clients durchsuchen nun die Registry nach bestimmten gewünschten Webservices. Jeder UDDI-Webservice-Eintrag enthält Informationen über:

- technische Modelle, denen der Webservice zugeordnet ist,
- den jeweiligen Provider mit Name, Adresse, Kontaktadresse und neutralen Merkmalen,
- die Services des jeweiligen Providers, die nach verschiedenen Taxonomien zugeordnet werden können, so beispielsweise ISO-3166-2 für geographische Taxonomien,
- die Webservice-Bindings, die Frage danach, wie der Webservice genutzt werden kann, d.h. die technische Spezifikation zu Aufruf und Nutzung des Webservices inklusive der URL, die anzusteuern ist.

12.9.5 Taxonomie

Die so entstehenden Taxonomien bedürfen einer speziellen Betrachtung, denn sie müssen gleichzeitig als Bindeglied zwischen Menschen und den beteiligten IT-Systemen dienen. Diese neu entstehenden Taxonomien müssen hierarchisch organisiert sein, da sie die großen Komplexitäten des Geschäftslebens abbilden müssen. Analog zu den biologischen Taxonomien werden sich hier mehrere Taxonomien parallel ausbilden. Idealerweise sind diese orthogonal zueinander. Diese unterschiedlichen Taxonomien werden verschiedene Aspekte des Service-Verhaltens klassifizieren. Die Folge hiervon ist, dass die einzelnen Webservices sich mehrfach an unterschiedlichen Stellen registrieren lassen müssen, um ein hohes Maß an Aufrufbarkeit zu erreichen. Zwar wird auf Dauer eine gewisse Konvergenz der Taxonomien entstehen, aber da sich die Webservices selbst relativ rasch ändern dürften, existiert in der Taxonomie bzw. der Registry eine permanente Fluktuation.

Solche Taxonomien sind nur dann sinnvoll verwendbar, wenn sie von einem Menschen verwaltet und aufgesetzt werden, um den semantischen Kontext der Webservices reflektieren zu können. Jede entstehende Kategorie muss die Semantik ihrer enthaltenen Webservices definieren und jeder Service innerhalb einer Kategorie muss dieselbe Semantik implementieren; die Services werden sich letztendlich nach folgenden Größen differenzieren:

- Geschwindigkeit,
- Preis,
- Zuverlässigkeit.

Eine solche semantische Beschreibung muss für einen Menschen verständlich und gleichzeitig für einen Parser syntaktisch interpretierbar sein. Die einzelnen Kategorien müssen ein einfaches oder multiples Vererbungsschema besitzen, welches wiederum ein Spiegelbild der Geschäftswelt ist. Die darin enthaltenen Services werden in der Regel zu mehreren Kategorien gehören.

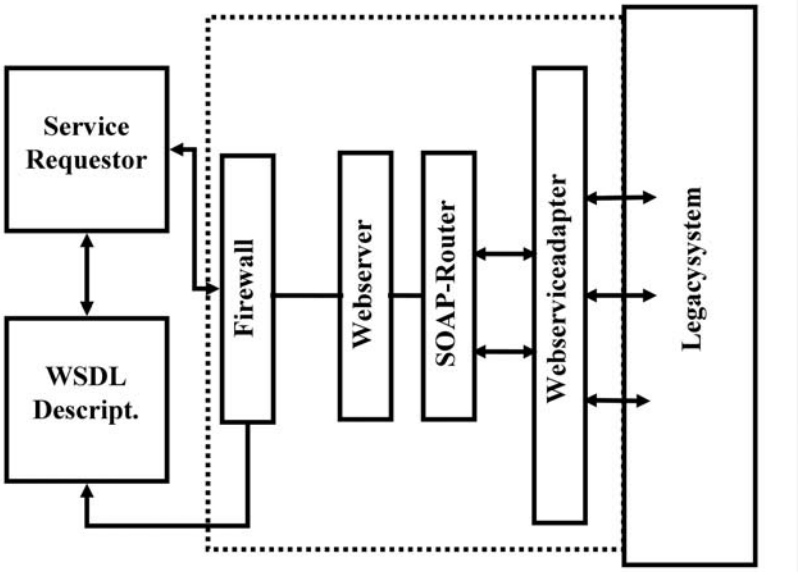

Abb. 12.33: Einbau einer Legacysoftware in eine Webserviceumgebung

12.9.6 WSDL

Die Web Services Definition Language, WSDL, ist eine Spezifikation für netzwerkbasierte XML-Services. Es existiert ein Microsoft-Vorschlag mit dem Namen Discovery of Web Services, DISCO, welcher aber nie an Gewicht außerhalb von Microsoft gewann und interessanterweise nicht in der .NET Strategie, s. Abschn. 12.5, enthalten ist. Die WSDL ist ein notwendiger Bestandteil für die Infrastruktur von Webservices. Sie ermöglicht es Webservice-Providern, ihre eigenen Services unabhängig von dem darunter liegenden Protokoll zu beschreiben. De facto handelt es sich immer um SOAP als Protokoll. WSDL ist eines der Schlüsselelemente für UDDI, da ohne eine klare Interface Definition Language Webservices nicht sinnvoll zu publizieren sind.

Die WSDL benutzt XML, um seine Definitionen des Services an potentielle Clients zu übertragen. Neben dem Namen der Datentypen wird auch der Portname, d.h. der logische Port des Servers, an den Client übertragen.

12.9.7 Legacysoftwareintegration

Für das Webservicesystem ist eine Legacysoftware eine Software unter vielen, die es gilt, als Service zur Verfügung zu stellen. Die Schwierigkeit liegt darin, die beteiligten Serviceadaptoren möglichst effizient und änderbar zu gestalten. Zwar ist die konzeptionelle Form, s. Abb. 12.33, recht einleuchtend, wird aber in der Praxis fast gar nicht eingesetzt, denn die Daten des

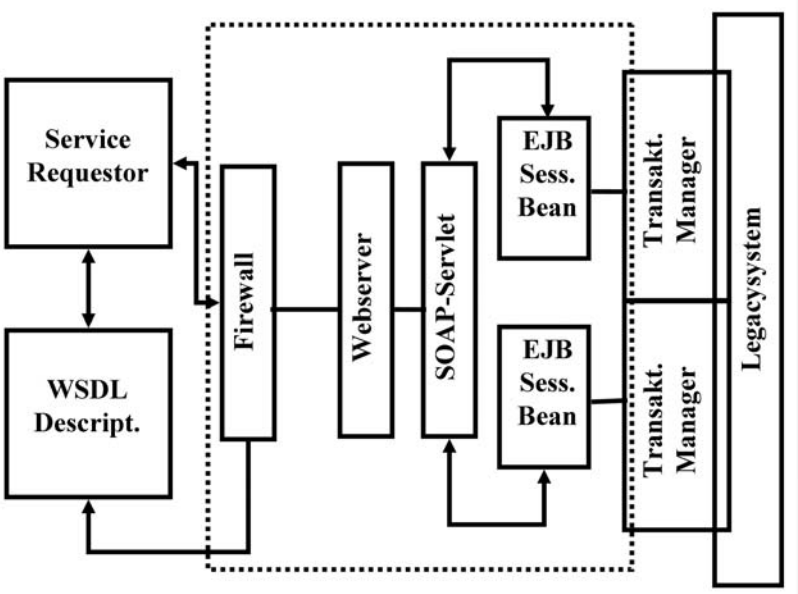

Abb. 12.34: Einbau einer Legacysoftware in eine Webserviceumgebung unter Berücksichtigung von Transaktionen

Legacysystems müssen in geordneter Art und Weise verändert werden. Dies kann nur über die dafür vorgesehenen Transaktionen geschehen. Solche Transaktionssysteme werden innerhalb von Transaktionsmanagern wie CICS oder IMS ausgeführt. Die Verbindung zu den Transaktionen auf der Legacyseite geschieht über EJB Session Beans, was durch den Einsatz von Message-Beans recht einfach vor sich geht. Es gibt allerdings auch die Möglichkeit, direkt auf den Transaktionsmonitor, via IMS-Connect oder ein ähnliches Protokoll, zuzugreifen. Der Zugriff über den Transaktionsmonitor ist meistens besser, da jetzt das originäre Legacysystem parallel weiter existieren kann und gleichzeitig die Datenqualität erhalten bleibt.

12.10 Systemintegration

Die Legacysysteme sind in den meisten Fällen entweder daten- oder serviceorientiert. Speziell die Programme, welche unter TP-Monitoren lauffähig sind, zeigen eindeutig Servicecharakteristika. Da ein vollständiger Ersatz durch ein COTS-Software-System nicht sinnvoll oder ein komplettes Reengineering nicht möglich ist, stellt sich die Frage, mit welchen Architekturen das bestehende Legacysystem erweitert werden kann, oder im umgekehrten Fall, wie das Legacysystem eingebunden werden kann.

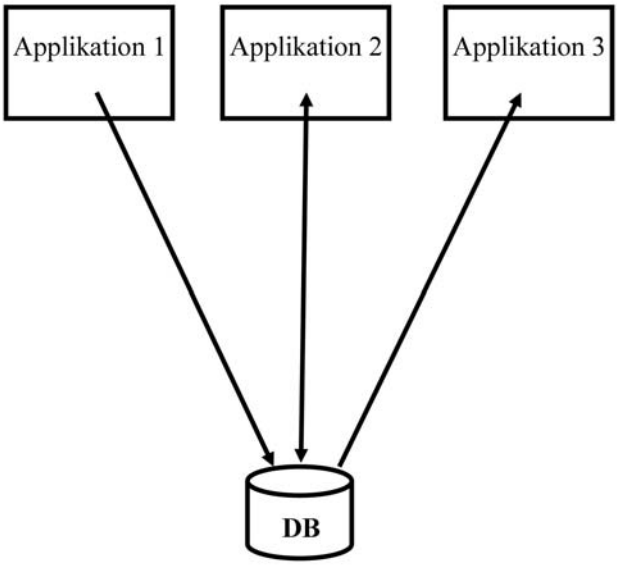

Abb. 12.35: Integration via Datenhaltung in Legacysoftware

Um diese Frage zu klären, ist es hilfreich, sich die Integrationsmechanismen zu betrachten, welche im Kontext von Legacysoftware sinnvoll sind.

12.10.1 Datenintegration

Viele Legacysysteme nutzen ihre Datenhaltung zum Austausch bzw. zur Koppelung von Applikationen, s. Abb. 12.35. Diese datenzentrierten Softwaresysteme resultieren in einer sehr engen Koppelung zwischen der Datenhaltung, meist einer Datenbank, und den Applikationen. Aber auch die Applikationen sind über diesen Mechanismus sehr eng miteinander verknüpft. Besonders unangenehm ist die Tatsache, dass diese enge Koppelung zu starken Abhängigkeiten der Daten und Datenmodelle, sowie zu verteilten Integritätsregeln führt. Oft lässt sich nicht mehr ausmachen, warum welche Applikation etwas prüft oder welche impliziten Voraussetzungen eine bestimmte Applikation hat.

In einer solchen Situation kann eine Modernisierung nur durch eine Entkoppelung geschehen, s. Abb. 12.36. Die vorhandene enge Koppelung zwischen den Datenmodellen wird durch ein Metadatenmapping für die „neue“ Applikation aufgelöst, so dass diese ihr eigenes Datenmodell und, eventuell, auch ihre eigene Datenhaltung besitzen kann. Diese Kapselung über Middleware und Metadatenschema eignet sich ideal für eine inkrementelle Erneuerung des gesamten Legacysystems. Außerdem wird hierdurch der Einsatz von Enterprise Application Integration entscheidend unterstützt.

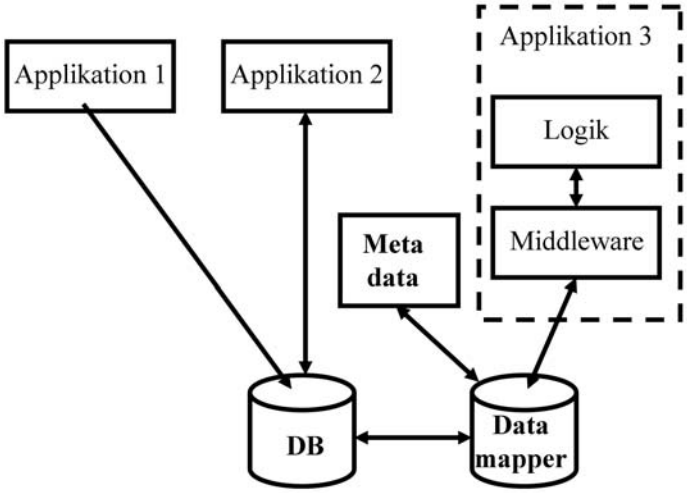

Abb. 12.36: Datenintegration einer modernisierten Applikation

12.10.2 Serviceintegration

Vielfach ist die vorhandene Legacysoftware serviceorientiert, in diesem Fall ist die Koppelung zwischen den einzelnen Programmen der Legacysoftware durch Funktionsaufrufe realisiert worden. Ein solches Modell kann nach einer Modernisierung ein ähnliches Verhalten fortsetzen. Allerdings müssen dabei die Interfaces sauber restrukturiert und dokumentiert werden.

Eine andere Möglichkeit besteht in einer Koppelung mit Hilfe von Message-Queues, s. Abb. 12.38. Ein solches Vorgehen ermöglicht eine einfache Integration in ein Enterprise-Application-Integration-Umfeld und kann die weitere Nutzung bestehender Transaktionsmonitore sicherstellen. Dies hat zur Folge, dass eine Modernisierung über Message-Queues auch inkrementell vorgenommen werden kann, mit der Folge eines risikominimierten Migrationspfades.

12.10.3 Präsentationsintegration

Neben den daten- und serviceorientierten Wegen existiert auch die Möglichkeit, eine für den Endbenutzer scheinbare Integration der Legacysoftware durch das Screenscraping durchzuführen. Zwar erscheint es aus systemtechnischer Sicht als eine Sackgasse, der Endbenutzer jedoch empfindet es als eine nahtlose Integration, welche sogar soweit gehen kann, dass er Drag&Drop oder auch Cut&Paste in der neuen Oberfläche nutzen kann. Technisch gesehen, s. Abb. 12.39, wird die bestehende Oberfläche neu interpretiert und graphisch

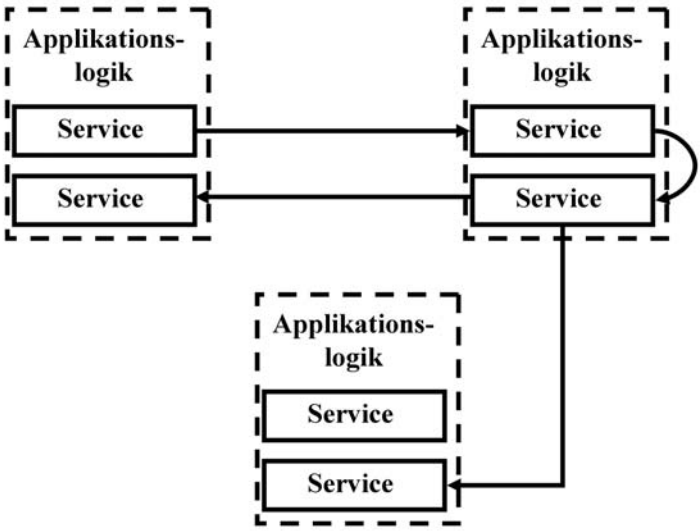

Abb. 12.37: Serviceintegration via direktem Aufruf in Legacysoftware

aufbereitet. Einer der großen Vorteile dieses Verfahrens ist, dass die bisherige Oberfläche, meist eine charakter-orientierte CICS-BMS- oder IMS-MFS-Oberfläche, noch weiterhin betrieben werden kann. Der große Nachteil ist die sehr enge Koppelung mit der Oberfläche der Legacysoftware, da jetzt auf jede Veränderung dieser sofort reagiert werden muss. Da der Screenscraper den protokollspezifischen Datenstrom ausliest, in der Regel ein 3270-Datenstrom, ist jede Veränderung der Oberfläche direkt merkbar.

Eine Abschwächung der engen Koppelung ist der Weg, über einen Screenscraper zu gehen, welcher die Metainformationen nutzt und gleichzeitig einen XML-Datenstrom zur Verfügung stellt. Hierdurch gelingt eine zeitweise Entkoppelung der Änderungen.

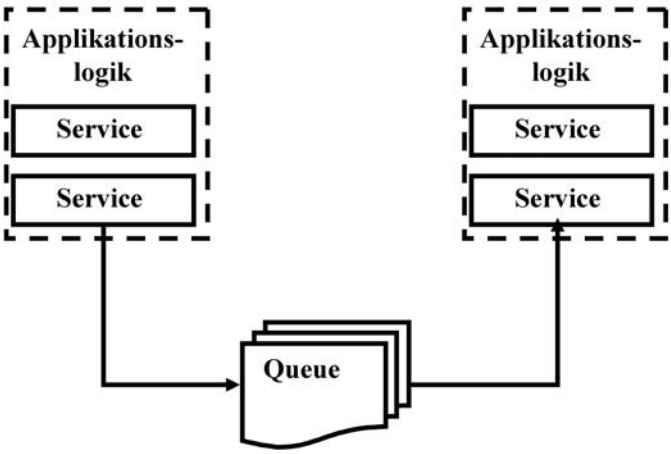

Abb. 12.38: Serviceintegration via Message-Queue in Legacysoftware

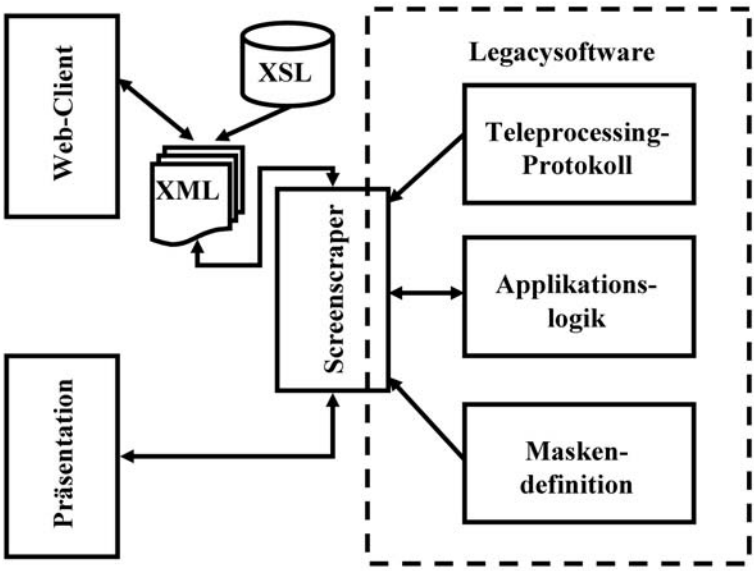

Abb. 12.39: Präsentationsintegration via Screenscraper in Legacysoftware

13

Patterns und Antipatterns

He who the sword of heaven will bear
Should be as holy as severe;
Pattern in himself to know,
Grace to stand, and virtue go;
More nor less to others paying
Than by self-offences weighing.
Shame to him whose cruel striking
Kills for faults of his own liking!
Twice treble shame on Angelo,
To weed my vice and let his grow!
O, what may man within him hide,
Though angel on the outward side!
How may likeness made in crimes,
Making practise on the times,
To draw with idle spiders' strings
Most ponderous and substantial things!
Craft against vice I must apply:
With Angelo to-night shall lie
His old betrothed but despised;
So disguise shall, by the disguised,
Pay with falsehood false exacting,
And perform an old contracting.

Measure for Measure,
William Shakespeare

Die Patterns, welche auch Entwurfsmuster genannt werden, kamen in den neunziger Jahren in Mode. Mittlerweile sind die Patterns, speziell im akademischen Umfeld, als Beschreibungsmittel für Vorgänge und Strukturen in Systemen allgemein akzeptiert worden.

Die ursprüngliche Definition des Begriffs Pattern stammt von Alexander:

> *Each pattern describes a problem which occurs over and over again in our environment, and then describes the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice. Each pattern is a three part rule, which expresses a relation between a certain context, a problem and a solution.*

Ziel hinter den Patterns ist es, eine Erfahrung, ohne Anspruch auf Vollständigkeit oder Kausalität, darzustellen und dabei einen Stil zu wählen, so dass diese Erfahrung auch Nichtexperten offensichtlich vermittelt werden kann. Obwohl diese Entwurfsmuster an sich keine Probleme direkt lösen können, da sie ja nur Wege aufzeigen, geben sie doch dem Leser ein „Aha“-Erlebnis und erhöhen die Lernkurve.

Die Patterns sind also hinreichend abstrakte Entwurfsmuster, um bestimmte Probleme adressieren zu können. Die Antipatterns sind, im Gegensatz dazu, Beschreibungen von Problemfällen. Bei den Antipatterns steht zunächst die Symptomatik, gefolgt von der Lösung des Problems, im Vordergrund.

Die hier aufgeführten Patterns und Antipatterns stellen eine mehr oder minder subjektive Auswahl im Umfeld von Legacysystemen dar.

13.1 Softwaredarwinismus

Wird die gesamte Software eines Unternehmens als ein Pool von potentiell wiederverwendbaren Softwareobjekten betrachtet, so lassen sich Betrachtungsweisen ähnlich den Evolutionsbetrachtungen von Darwin[1] anstellen. Die Softwareentwickler müssen sich aus diesem Pool von Softwareobjekten einige zur Wiederverwendung aussuchen und andere dabei vernachlässigen. Ein so wiederverwendetes Softwareobjekt kann sich selbst „weitervererben“ und damit seine Chancen auf zukünftige Wiederverwendung erhöhen.

Die Folge dieses Softwaredarwinismus ist, dass Softwareobjekte, welche der Softwareentwickler nicht attraktiv findet, nicht genutzt werden und in der Versenkung verschwinden. Zu den darunter liegenden Kräften hinter diesem Darwinismus gehören:

- Verfügbarkeit – Wenn ein Softwareobjekt nicht verfügbar ist, wird es nie genutzt. Genutzte Softwareobjekte haben stets eine höhere Verfügbarkeit, weil sie offensichtlich vorhanden sind. Die mentale Verfügbarkeit bzw. die häufige Verwendung eines Softwareobjekts erhöht die Chancen, das Objekt wiederzufinden und damit auch die Chancen des Objektes, sich weiter im Gedächtnis zu erhalten.
- Verständlichkeit – Wenn ein Softwareobjekt unverständlich ist, wird es auch nicht verwendet werden. Insofern ist Verständlichkeit eine stark treibende Kraft im Softwaredarwinismus.

Im Umkehrschluss aus dieser darwinistischen Betrachtung muss Wiederverwendung explizit den Softwaredarwinismus in Betracht ziehen. Die Softwareobjekte müssen so entwickelt werden, dass Softwareentwickler diese wiederverwenden wollen!

[1] Moderne Biohistoriker heben hervor, dass der „Darwinismus“ ursprünglich von Alfred R. Wallace stammt. Anscheinend war Charles Darwin bekannter als Wallace und man verbannt seinen Namen mit der Theorie.

Die Folge dieses Mechanismus ist, dass wiederverwendbare Softwareobjekte so konzipiert sein müssen, dass sie sofort und ohne Veränderung einsetzbar sind, auch „works out of the box“ genannt. In Bezug auf Legacysoftware bedeutet dies, die enthaltenen Softwareobjekte leichter zugänglich zu machen.

13.2 Kleine Oberfläche

Obwohl das Information Hiding schon während der Zeit der Structured Analysis stark propagiert wurde, wird es auch heute noch erstaunlich oft verletzt. Aus Sicht eines Softwareobjektes lässt sich das Information Hiding auch anders formulieren:

Eine Komponente sollte stets eine minimale Oberfläche in Bezug auf ihr Volumen haben.

Das Volumen eines Softwareobjektes ist seine Komplexität und die Oberfläche die Breite und Komplexität seiner Interfaces. Je mehr ein einzelnes Softwareobjekt leistet, desto größer wird sein Volumen, allerdings ist es fatal, wenn seine Oberfläche im gleichen Maß ansteigt. Denn es gilt im Softwaredarwinismus: Je komplizierter ein Interface, desto weniger Softwareentwickler wollen es nutzen! Der Einsatz eines Wrappers, damit die Komplexität eines vorhandenen Interfaces gemildert werden kann, ist oft ein Schritt in die richtige Richtung.

In dieselbe Richtung zielt die Idee der Incremental Relevation, bei der die Komplexität eines Interfaces durch die Verwendung von Objekten im Interface reduziert wird. Allerdings müssen diese Objekte eine Defaultinitialisierung besitzen. Dadurch kann die subjektive Komplexität für Anfänger gemindert werden, da ja die Defaultinitialisierungen nicht überschrieben werden müssen. So bleibt das Interface, trotz seiner hohen Komplexität, für einfache Fälle relativ simpel, was wiederum die Chancen auf Wiederverwendung erhöht.

13.3 Service Layer

Die meisten Applikationen innerhalb eines großen Unternehmens benötigen unterschiedliche Zugriffsmechanismen auf ihre Daten. Meist liegt der Unterschied darin begründet, dass das Reporting und die regulären Applikationen voneinander getrennt sind oder ein Datawarehouse existiert, welches ein Client für die entsprechenden Daten ist. Obwohl diese Interfaces unterschiedlichen Zwecken dienen und sich eventuell unterscheiden, brauchen sie doch eine gemeinsame Querschnittsfunktionalität im Bereich der korrekten semantischen Darstellung bzw. Operation. Solche Funktionalitäten können zum Teil hochkomplex sein und nur durch eine Reihe von Transaktionen implementiert werden. Würde man für jedes Interface die Implementierung wiederholen, so würde eine große Menge redundanter Sourcecode entstehen, was die bekannten Folgeprobleme, wie z.B. Bloating, nach sich zieht.

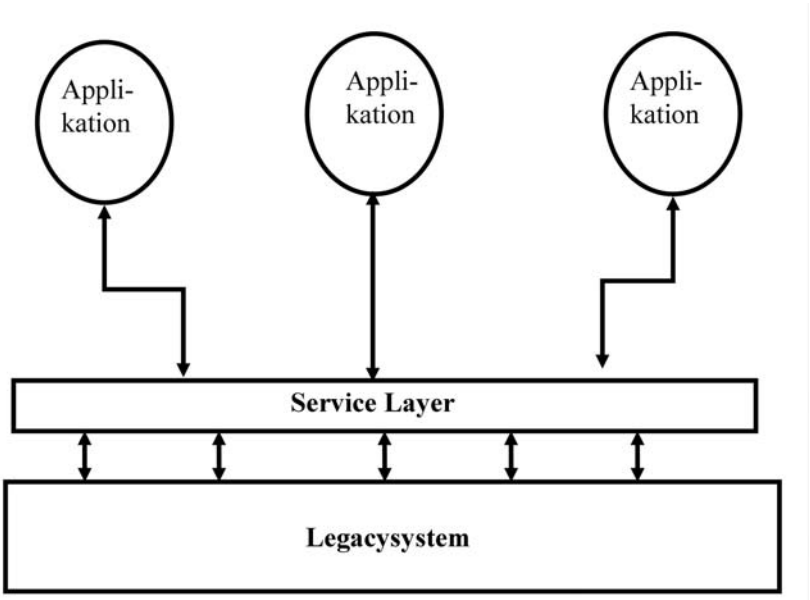

Abb. 13.1: Der Service Layer

Ein Pattern, um dieses Problem zu adressieren, ist das Service Layer Pattern. Es definiert die „Systemgrenze“ der Applikation und die möglichen Operationen, welche diese Grenze bietet, aus Sicht der anderen Clients, welche die Interfaces nutzen wollen. Auf Grund der möglichen Client-Operationen auf dem Service Layer ist dessen Funktionalität wohlbekannt. Hieraus lassen sich im Folgeschritt Interfaces ableiten, welche diese Funktionalitäten bündeln und geeignet abstrahieren.

Die Schaffung eines Service Layers kann mit ganz unterschiedlichen Implementierungstechniken vollzogen werden. Unabhängig davon, resultiert das Pattern in einer Service Oriented Architecture, s. Abschn. 12.8. Mögliche Webservices wiederum sind ein Spezialfall, d.h. eine Implementierung mit einer speziellen Form des Interfaceprotokolls dieses Entwurfsmusters.

13.4 Gateway

Ein Standardproblem ist der Zugriff auf Ressourcen außerhalb der eigentlichen Applikation. Hierbei ist es durchaus möglich, dass mehrere externe Ressourcen vorhanden sind und angesprochen werden müssen. Die einzelnen Schnittstellen der externen Ressourcen an jeder Stelle der Applikation bzw. für jede Applikation neu zu programmieren, führt zu erhöhtem Aufwand und einem sehr unflexiblen System.

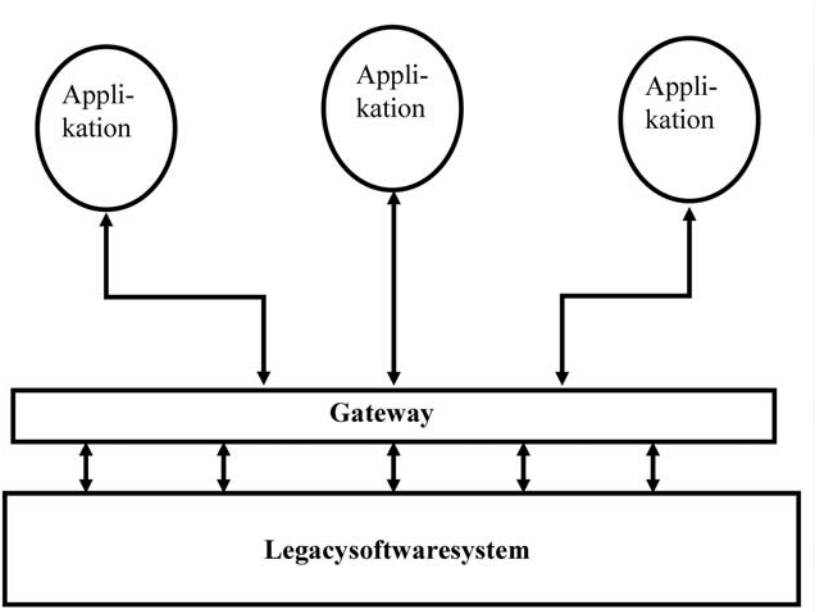

Abb. 13.2: Das Gateway

Die Lösung hierfür ist ein Gateway-Pattern. Dabei werden die verschiedenen Schnittstellen zu einer logisch sinnvollen Einheit gebündelt und den Applikationen als eine Schnittstelle zur Verfügung gestellt. Auf diese Weise entsteht eine Architektur, welche der Java Connector Architecture, s. Abschn. 12.4.8, ähnlich ist, wobei die Gateway-Lösung zu weniger granularen Interfaces führt als eine simple Wrapper-Lösung, da sie ja ganze Schnittstellen bündelt.

13.5 Teile und Modernisiere!

Der Name dieses Entwurfsmusters entstand nach der bekannten „divide and conquer"-Philosophie. Die vorliegende Problematik ist, dass ein Legacysystem obsolet wird und Teile davon noch gebraucht werden, aber nicht kritisch für den Erfolg sind. Klassische Vorgehensweisen, wie die Veränderung des Legacysystems, sind nicht sinnvoll, da es ja obsolet ist; das gleiche gilt für den Einsatz eines Wrappers. Die einzig sinnvolle Lösung ist eine Übertragung des Sourcecodes auf das neue System. Zunächst einmal geschieht die Übertragung automatisch und es muss nur die Sicherstellung der Lauffähigkeit erreicht werden, ohne dass hierbei der Inhalt verstanden werden muss. Insofern ist es kein Reengineering, sondern eine Art Sourcecodesalvaging. Durch diesen Schritt kann das Legacysystem abgelöst werden. Allerdings besitzt das neue System jetzt eine inakzeptabel hohe Entropie, aber, und das ist der Vorteil, es ist jetzt

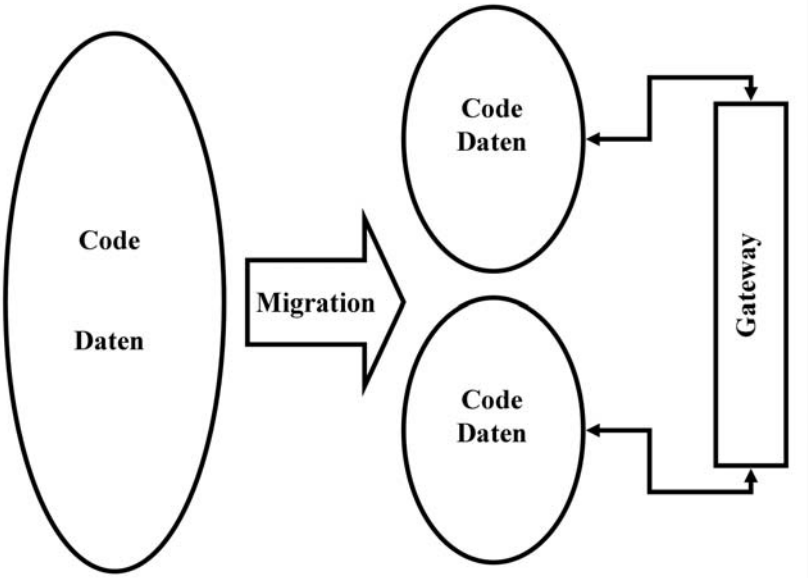

Abb. 13.3: Die „Teile und Modernisiere“-Migration

Zeit gewonnen worden, diese hohe Entropie durch nachfolgendes Refaktoring, s. Kap. 5, zu senken.

Der Vorteil dieses Vorgehens beruht darauf, dass der Aufwand kontrollierbar und phasenorientiert bleibt, das große Risiko ist jedoch, dass der finale Refaktoring-Schritt aus diversen Gründen unterlassen wird. Diese Unterlassung geschieht, trotz bester Absichten, gar nicht so selten, da IT-Abteilungen das Refaktoring nur schwer gegenüber den Fachabteilungen legitimieren können, schließlich läuft die Software doch ...

13.6 Externalisierung

Viele heutige Legacysysteme sind aus viel älteren, vollständig batchorientierten Systemen entstanden. Solche Batchsysteme bestehen meist nicht aus einem einzelnen Programm, sondern aus einer ganzen Abfolge von Programmen, welche erst in ihrem gesamten Ablauf ein fachlich sinnvolles Ergebnis liefern. Meistens sind diese unterschiedlichen Phasen eines Ablaufes nicht gut gekapselt, sondern die nachfolgenden Programme setzen bestimmte Reihenfolgen voraus. Dieses interne Interface setzt dann sehr oft auch eine große Anzahl von ungeprüften Vorbedingungen voraus.

Die Lösung zu dieser unflexiblen losen Koppelung ist die Externalisierung des internen Interfaces. Es wird inkrementell durch ein externes Interface ersetzt, damit es auch von anderen Programmen genutzt werden kann. Im ersten

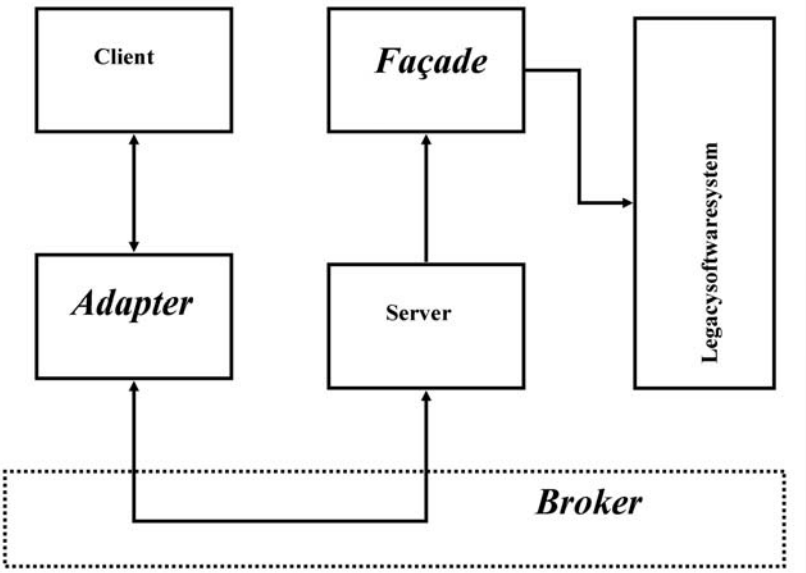

Abb. 13.4: Das Zusammenspiel von Broker, Adaptor und Façade

Schritt werden die Vorprogramme so abgeändert, dass sie das neue Interface optional unterstützen. Das Programm wird jetzt um das neue Interface erweitert und wenn dieses erfolgreich läuft, kann das alte Programm langsam abgeschaltet werden.

13.7 Legacysoftwareintegration

Die eigentliche Legacysoftwareintegration bedarf des Zusammenspiels mehrerer Patterns, um sinnvoll agieren zu können. Eine solche Kombination, welche beim Einsatz von Legacysoftware oft anzutreffen ist, wird aus der Kombination von Adaptor, Broker und Façade gebildet, s. Abb. 13.4, wobei die Aufgabenteilung zwischen diesen drei Patterns klar ist. Das Broker-Pattern dient dazu, die Heterogenität der verschiedenen Systeme zu überbrücken, quasi eine Infrastrukturplattform für Aufrufe und Datenaustausch zur Verfügung zu stellen. Der Adaptor ist ein Konstrukt, das die Interfaces umwandelt von einer gegebenen Repräsentation in eine, die vom Client erwartet wird, während die Façade die diversen Interfaces zu einem verknüpft und so das Legacysystem gegenüber den Komplexitäten der Client-Welt abschirmt. Selbstverständlich wirkt das Façade-Pattern in beide Richtungen, d.h. der Client wird auch vor den Komplexitäten des Legacysystems beschützt.

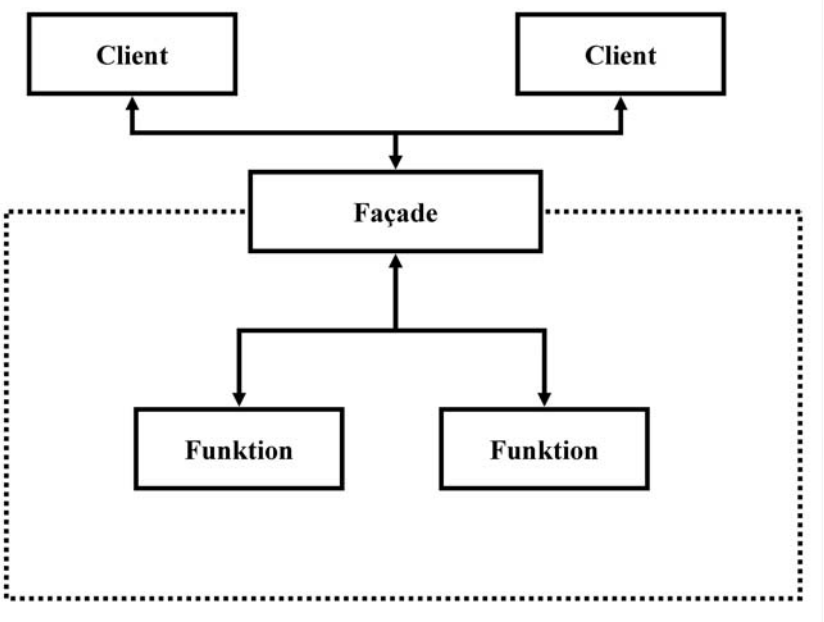

Abb. 13.5: Die Façade

13.8 Façade

Eine Façade kapselt ein komplettes Subsystem gegenüber der Außenwelt ab, s. Abb. 13.5, indem sie die vollständige Komplexität des Subsystems gegenüber dem Client abkapselt. Hierdurch wird eine Entkoppelung möglich, da der Client jetzt nichts mehr über die Implementierung des Subsystems wissen muss. Einzig die Façade ist ihm noch zugänglich und bleibt, auch bei der nachfolgenden Maintenance, stabil. Letztendlich steckt hinter der Façade die Idee, dass die Wiederverwendung einer Blackbox-Komponente einfacher ist, als wenn das Wissen über die konkrete Implementierung vorhanden sein muss. Eine Façade ist eine spezielle Form des Wrappers, wenn ein komplettes Subsystem mit seiner Hilfe als Façade isoliert wird.

13.9 Adaptor

Ein Adaptor wird immer dann verwandt, wenn das gerufene und das rufende System schon fertig sind. Er dient zur Abbildung der jeweiligen Interfaces zwischen den beiden Systemen, und zwar nur genau zwischen jeweils zwei Systemen. Der Adaptor erlaubt es eigentlich inkompatiblen Systemen, zusammenzuarbeiten. Im Gegensatz zur Façade, welche stets ein neues Interface definiert, benutzt der Adaptor ein bestehendes Interface. Man könnte ihn salopp als einen Interfacewrapper bezeichnen.

13.10 Schichten

Das vermutlich älteste und „natürlichste“ Entwurfsmuster ist das Schichtenpattern. Dieses Entwurfsmuster versucht, die Software in Schichten, Layers, aufzuteilen, s. Abb. 13.6. Jede Schicht i nutzt die Eigenschaften und Funktionalitäten der darunter liegenden Schicht $i + 1$, um Funktionen zu realisieren. Die Schicht i wiederum stellt alle ihre Funktionen der Schicht $i - 1$ zur Verfügung. Üblicherweise steigt mit den Schichten auch das Abstraktionsniveau an, d.h. $i - 1$ ist sehr viel abstrakter als die Schicht $i + 1$.

Dieses Pattern kann erfolgreich eingesetzt werden, um ein System zu strukturieren, wenn zwei Richtlinien befolgt werden. Zum einen darf die Kommunikation zwischen zwei Schichten nur unter Zuhilfenahme aller Zwischenschichten passieren und zum anderen muss jede Schicht von der hierarchisch darüber liegenden Schicht vollständig entkoppelt sein. Die erste Richtlinie führt zu einer expliziten Beschränkung der Kommunikation auf den nächsten Nachbarn und die zweite auf eine rein unidirektionale Koppelung zwischen den Schichten.

Typischerweise beschäftigen sich die unteren Schichten mit hardwarenäheren Aspekten als die oberen Schichten. Dies führt zu relativ einfachen Portierungen auf andere Plattformen, da jetzt nur die unteren Schichten ausgetauscht werden müssen. Dieses Entwurfsmuster ist so erfolgreich gewesen, dass es heute faktisch in allen Netzwerkprotokollen und Betriebssystemen wiederzufinden ist.

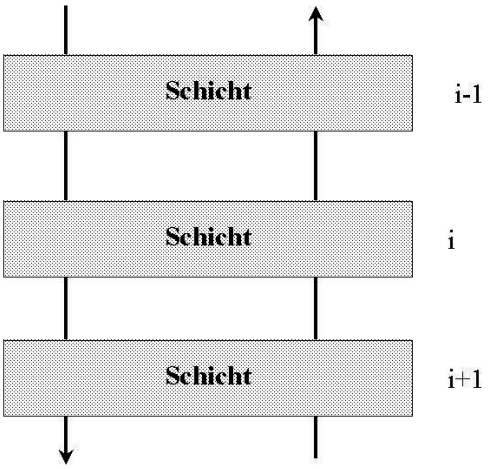

Abb. 13.6: Die Schichten

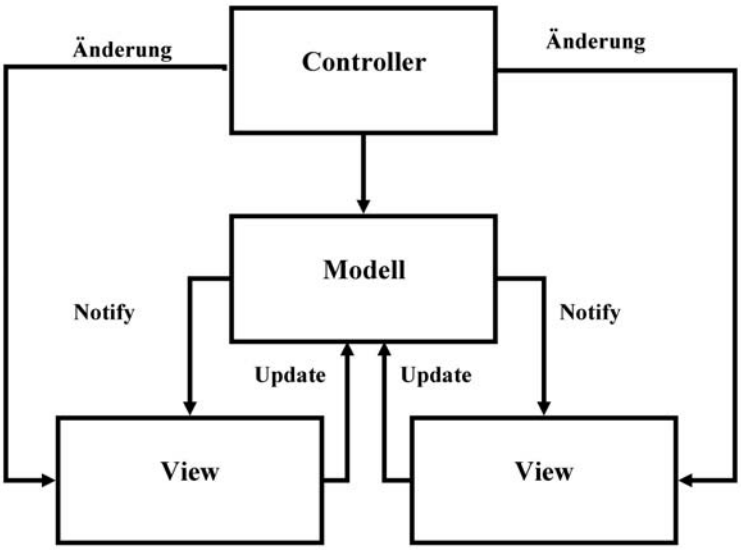

Abb. 13.7: Das Model View Controller Pattern

Eine bekannte Variante des Entwurfsmusters ist die Relaxed Layered Architecture, bei der es auch direkte Kommunikation zwischen nicht benachbarten Schichten geben darf. Diese Varianten wurden hauptsächlich aus Performanzgründen eingeführt, obwohl sich die niedrigere Wartbarkeit einer Relaxed Layered Architecture als problematisch erweist.

13.11 Model View Controller

Das Model View Controller Pattern, auch MVC genannt, ist das verbreitetste Pattern für den Aufbau von Benutzerschnittstellen, s. Abb. 13.7. Dieses Entwurfsmuster besteht aus drei getrennten Teilen, welche jeweils dedizierte Aufgaben erfüllen und über eine vorbestimmte Art und Weise zusammenarbeiten.

- Model – Das Modell enthält die Applikationsdaten, implementiert das Verhalten der Daten und reagiert auf Anfragen bezüglich des Zustandes der Daten. Es weiß aber nichts über die Repräsentation der Daten.
- View – Der View ist die graphische Repräsentation der Daten. Ein View ermöglicht die graphische und textuelle Ausgabe an den Endanwender.
- Controller – Der Controller empfängt die Benutzeraktionen über den Eingangskanal, in der Regel eine Maus oder eine Tastatureingabe, und steuert sowohl den View als auch das Model entsprechend.

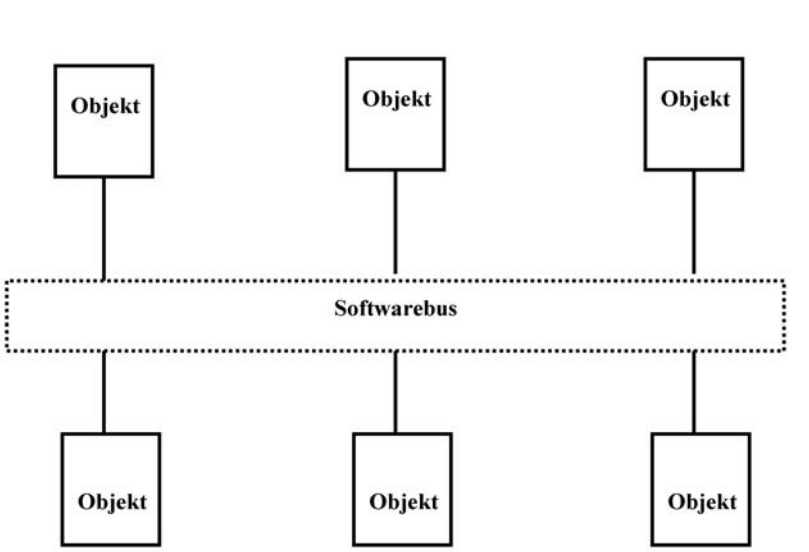

Abb. 13.8: Das Distributed Object Pattern

Innerhalb von Legacysoftware sind diese drei verschiedenen Aufgaben oft völlig vermischt in einem einzigen Programm aufzufinden. Die Nutzung des MVC-Patterns ermöglicht eine flexible Entkoppelung dieser einzelnen Teile. Einer der Gründe für die Forderung, die Repräsentation – View – von der Fachlichkeit – Model – zu trennen, liegt in der unterschiedlichen Änderungshäufigkeit der beiden. Windowsoberflächen, beispielsweise, ändern sich alle 2 Jahre, aber die Fachlichkeit sehr viel langsamer, in der Größenordnung von 10-15 Jahren. Von daher erlaubt das MVC-Pattern den Austausch der Oberfläche als eine Modernisierungsmaßnahme.

Damit das MVC-Pattern sinnvoll eingesetzt werden kann, muss es ein Subscribe-Notify-Protokoll geben, damit die Views aktuell bleiben können. Wenn die Daten sich verändern, wird – über ein Notify – jeder View über die Änderung informiert und kann entsprechend reagieren.

13.12 Distributed Object

Das einfachste der Distributed Object Patterns ist das Client-Server-Entwurfsmuster. Hier wird ein Gesamtsystem aus einer Reihe von dedizierten Servern mit den jeweiligen Clients aufgebaut. Die Rollenteilung in Client und Server bleibt während der gesamten Lebensdauer der Applikation stabil, obwohl die konkrete Rollenbeziehung nur innerhalb eines Subsystems Sinn macht. So

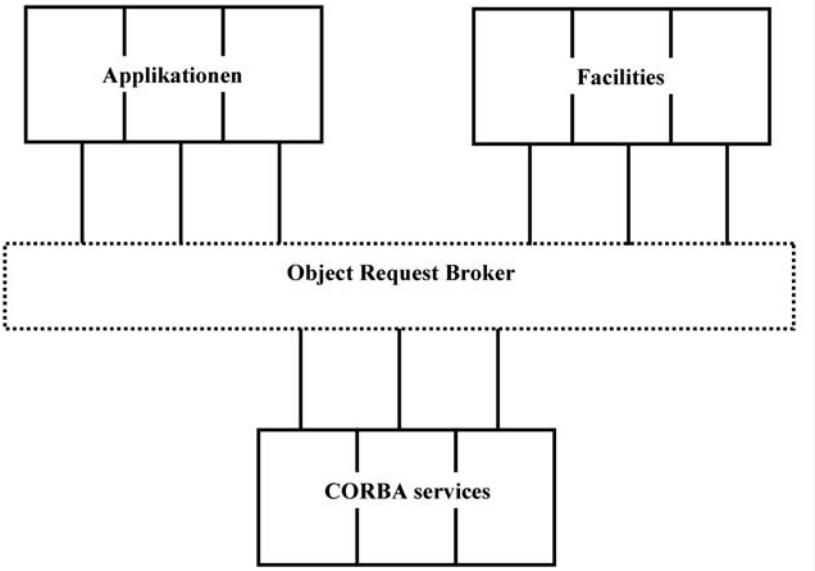

Abb. 13.9: Das Broker Pattern, am CORBA-Beispiel

kann z.B. ein Client im Subsystem $\mathcal{A}$ die Rolle eines Servers im Subsystem $\mathcal{B}$ wahrnehmen.

Damit die Services eines Servers genutzt werden können, muss dem Client das Protokoll, der Ort des Servers und die Ausprägung der Services ab initio bekannt sein, umgekehrt nicht, d.h. der Server weiß relativ wenig über den Client. Die Idee der Entkoppelung von Client und Server über ein Netzwerk hinweg führte zum Distributed-Object-Entwurfsmuster, bei dem die traditionelle Serverrolle aufgehoben ist und der Server durchaus andere Server zwecks Bearbeitung und Zur-Verfügung-Stellung von Services nutzen kann. Bekannte Vertreter dieses Entwurfsmusters sind das Java RMI oder, partiell, CORBA. Die wohl verbreitetste Applikation, die diesem Pattern folgt, ist das World Wide Web, wo Millionen von Webservern ihre Services den Clients, korrekterweise deren Browsern, zur Verfügung stellen.

13.13 Broker

Eine Broker-Architektur bzw. das Broker-Entwurfsmuster ist in Abb. 13.9 dargestellt. Der Schwerpunkt des Broker-Patterns liegt im Aufbau der Infrastruktur zur Verteilung und Kommunikation der Distributed Objects, s. Abbildungen 13.9 und 13.8. Das Herz des Entwurfsmusters ist die Kommunikationsinfrastruktur, welche Broker genannt wird.

Dieser Broker versteckt die Details des Netzwerkes, des Betriebssystems und der exakten Lokation der einzelnen verteilten Objekte so, dass deren konkrete Implementierung transparent wird.

Die Rollen von Client und Server sind in einer Broker-Architektur aufgehoben, da diese Rollen dynamisch, je nach Aufrufverhältnis, zugeordnet werden können. Der bekannteste Vertreter dieser Architektur ist der CORBA-Standard. Eine hohe Skalierbarkeit und Portierbarkeit zeichnet dieses Entwurfsmuster aus, allerdings weist es sowohl Performanzprobleme als auch eine eingeschränkte Effizienz auf.

13.14 Conway's Law

Bereits 1968 erkannte Conway den Zusammenhang zwischen der Organisationsform eines Softwareentwicklungsprojekts und dem dabei entstehenden Softwareprodukt. Alle Entwurfsentscheidungen werden durch die vorherrschenden Organisationsstrukturen beeinflusst und somit nicht ausschließlich durch den individuellen Designer gefällt. Beim näheren Hinsehen erkennt man in den Strukturen des Softwaresystems die Struktur der Organisation wieder. Die Software, welche ein Unternehmen erstellt, spiegelt auch immer die Kommunikationsstruktur des Unternehmens wider, die ihrerseits aus der Organisationsstruktur abgeleitet werden kann.

Ist die Organisationsstruktur eines Unternehmens und seiner Projekte groß und unflexibel, wirkt sich dies nachteilig auf die Struktur und die Flexibilität der Softwareprodukte aus. Alteingesessene Firmen haben daher häufig nicht nur mit einem übernatürlich aufgeblasenen Management zu kämpfen, das sich krampfhaft selbst am Leben erhält, sondern auch mit Programmen, welche mit ähnlichen Problemen behaftet sind. Dieses Phänomen zeigt sich gut innerhalb der Legacysoftware, da diese als Bestandteil eines soziotechnischen Systems, des Legacysystems, über die Jahre hinweg alle veränderten Organisationsformen abgebildet hat. Insofern kann die Legacysoftware auch als Historie eines Unternehmens verstanden werden. Die Korrelation zwischen der Software und der Organisation zeigt sich noch an anderer Stelle: Zu einem gegebenen Zeitpunkt existierte eine klare Verantwortlichkeit für Daten und Funktionen innerhalb der Legacysoftware. Mit zunehmender organisatorischer Entwicklung verschob sich dies, mit der Folge, dass es heute große Unsicherheiten bezüglich der Verantwortlichkeiten von Daten und Funktionen in der Software gibt.

Interessanterweise hat Conway's Law auch Auswirkungen auf alle Restrukturierungsprojekte. Solche können nur dann erfolgreich sein, wenn die Kommunikationsstruktur des Unternehmens in Betracht gezogen bzw. bekannt ist.

13.15 Silver Bullet

Das Silver Bullet-Antipattern ist sehr weit verbreitet, dahinter steht die Ansicht, dass das richtige Werkzeug die Probleme sofort lösen kann. Diese An-

nahme ist immer falsch! In Organisationen haben Probleme immer organisatorische Ursachen, erst nachdem diese Ursachen beseitigt wurden, kann ein Werkzeug zur Unterstützung des Prozesses, nicht jedoch zur Problemlösung[2], eingesetzt werden. Die Verkäufer von COTS-Software stellen gerne ihre Software als das Silver Bullet dar, welches alle Probleme lösen kann.

Moderne Codeparser oder auch automatisierte Konverter werden oft als Lösungen aller Probleme im Legacysystem angepriesen, obwohl klar ist, dass ohne ein Domänenwissen ein Legacysystem nicht beherrschbar ist. Außerdem senken solche Werkzeuge die Entropie eines Legacysystems nicht ab.

Parallel zu dem Silver Bullet existiert auch der „Golden Hammer". Dieser Ausdruck wird verwandt, wenn mit einem Werkzeug oder einer Lösung schon Erfahrungen gewonnen wurden und diese Erfahrungen positiv waren; dann wird versucht, jedes Problem mit diesem Werkzeug oder Verfahren zu lösen: „*If you got a hammer, everything looks like a nail!*"

13.16 Batteries not included

Dieses Antipattern ist typisch für eine COTS-Software-Umgebung, oft wird es auch als Vendor Lock In bezeichnet. Es existiert in zwei Varianten, zum einen für die Softwareentwicklungsumgebung und zum anderen für die Laufzeitumgebung. Bei der zweiten Variante handelt es sich darum, dass Infrastruktursoftware, welche durch einen COTS-Hersteller geliefert wurde, obsolet werden kann. Dies gefährdet das Legacysystem immens. Diese Form des Antipatterns ist durchaus nicht ungewöhnlich, fast jedes Legacysystem kennt dies: Plötzlich existiert eine bestimmte Datenbank nicht mehr, oder für die vorhandene Hardware werden keine Upgrades mehr geliefert. Die Folge ist fast immer eine explizite Migration zu einer anderen COTS-Software als Ersatz, mit den typischen Konsequenzen des kleineren Maintainability Index.

Die erste Form dieses Antipatterns taucht meist in den Softwareentwicklungsumgebungen auf, welche vom Hersteller nicht mehr unterstützt werden – dies scheint der Lebensweg fast aller 4GL-Systeme[3] zu sein. In diesem Falle wird die Umgebung nicht mehr angepasst, was dazu führt, dass die Legacysoftware nicht mehr weiterentwickelbar ist. Als Maßnahme wird in der Regel die Umgebung umgestellt, was aber implizit zu einer Portierung des Sourcecodes der Legacysoftware führt.

[2] „*A fool with a tool is still a fool.*"

[3] Auch Visual-J von Microsoft ist hierfür ein Beispiel, oder Werkzeuge, welche nur auf OS/2 lauffähig sind.

14

Epilog

The law hath not been dead, though it hath slept:
Those many had not dared to do that evil,
If the first that did the edict infringe
Had answer'd for his deed: now 'tis awake
Takes note of what is done; and, like a prophet,
Looks in a glass, that shows what future evils,
Either new, or by remissness new-conceived,
And so in progress to be hatch'd and born,
Are now to have no successive degrees,
But, ere they live, to end.

Measure for Measure
William Shakespeare

Die Zukunft der Softwareentwicklung im Allgemeinen und die der Legacysysteme im Besonderen wird sich in den kommenden Jahrzehnten drastisch verändern, denn Softwaresysteme werden immer stärker als inhärent komplexe Systeme verstanden. Die Akzeptanz dieser Komplexität führt zu völlig anderen Zugängen bei der Entwicklung der Systeme.

In der Vergangenheit wurde und auch noch heute wird Software so gebaut, dass sie ein spezifisches, vorhersagbares und streng deterministisches Verhalten zeigt. Dieses Verhalten wird auf allen Ebenen, angefangen von einer kleinen Subroutine bis hin zu einem kompletten Programmpaket, verlangt. Aber in der Zukunft wird die IT-Welt durch autonome unabhängige Komponenten bestimmt werden. Diese können in einer offenen und flexiblen Welt kein deterministisches Verhalten zeigen. Die Tradition, mechanistische Modelle für Software zu bilden, hat ausgedient. Die Herausforderung, ein System zu bauen, wird nicht sein, es vollständig zu kontrollieren, sondern die einzelnen Komponenten unabhängig zu lassen, solange sie ihren jeweiligen Service Level Agreements gehorchen, und nur zu verlangen, dass das Gesamtsystem ein vernünftiges, d.h. adäquates Verhalten zeigt, also ein Gesamtsystem zu bauen, ohne das Mikrowissen über jede einzelne Komponente zu besitzen. Ökosysteme und physikalische Systeme, speziell der Übergang von der Quantenmechanik in die klassische Physik, zeigen, dass es trotz großer Unsicherheiten auf der Mikroebene möglich ist, ein sehr stabiles Gesamtsystem zu bauen. Solche komplexen Systeme gewinnen ihre Stabilität nicht aus dem determini-

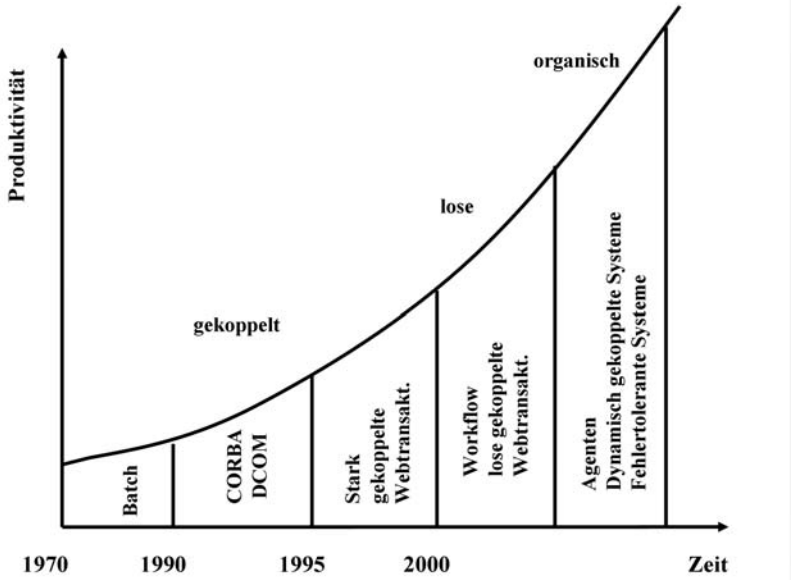

Abb. 14.1: Die Zukunft der Software

stischen Verhalten einzelner Teile, sondern aus thermodynamischen Größen, welche das Gesamtsystem betreffen.

Die Systeme der Zukunft müssen explizit in Betracht ziehen, dass alle Systeme in einer offenen und vor allen Dingen veränderlichen Welt existieren werden. Das heutige Internet ist ein konkretes Beispiel für ein komplexes selbstregulierendes System, welches offen ist. Die Folge hinter dieser Offenheit ist die Forderung nach Adaptivität und nicht nach Stabilität der einzelnen Komponenten.

Aber auch die Maintenance wird sich in Zukunft drastisch verändern. Autonomie und Offenheit bedingen, dass kein Softwaresystem stabil sein kann und die Akzeptanz des Wertes der Veränderbarkeit wird den Wert und das Ansehen von Maintenance drastisch erhöhen. Außerdem werden kontinuierliche Evolutionsmechanismen dominant werden.

Erstaunlicherweise werden Fertigkeiten von Softwareentwicklern, welche zu einem Teil der heutigen Strukturen in der Legacysoftware geführt haben, in Zukunft wieder eine große Rolle spielen. Der interne Aufbau besonders alter Legacysysteme war immer durch zwei, von den sechziger bis zu den achtziger Jahren, knappen Ressourcen bestimmt: die CPU-Taktraten und die Größe des Hauptspeichers. Fast jede Software, welche nach den achtziger Jahren entstanden ist, ignoriert diese beiden limitierenden Größen. Der Grund hierfür liegt darin, dass sich die Taktraten und der verfügbare Hauptspeicher über lange Jahre permanent sehr stark erhöht haben. Der Hintergrund hierfür ist das so genannte *Moore's Law*, welches aussagt, dass sich die Zahl der Transistoren

pro Flächeneinheit alle 18 Monate verdoppelt, mit der Folge der schnelleren CPU und des größeren Hauptspeichers. Aber nach 2020 wird diese Steigerung nicht mehr möglich sein. Etwa im Jahr 2022 werden die Größen für einen einzelnen Transistor im atomaren Bereich erreicht, dies stellt ein physikalisches Limit dar und wird vermutlich nicht mehr zu unterschreiten sein. Vermutlich wird es aber schon vor Erreichung der physikalischen Grenze von *Moore's Law* zu Problemen kommen: Das Wirthsche Gesetz[1] besagt, dass die Verarbeitungsgeschwindigkeit von Software schneller fällt als die Geschwindigkeit der Hardware zunimmt.[2] Der Hauptgrund hinter der Verlangsamung ist das *„feature bloating“*, s. S. 10. Folglich dürfte der Punkt an dem Software nicht mehr nutzbar ist schon früher erreicht werden.

In der Folge dieser physikalischen Limitierung[3] werden altbekannte Werte, wie geringer Hauptspeicherbedarf oder effiziente Ausnutzung der Taktraten, wieder stärker in den Vordergrund gerückt werden. Die hierfür notwendigen Techniken sind identisch mit den in den sechziger und siebziger Jahren entwickelten: Die Geschichte wiederholt sich!

Omnia sic transeunt, ut revertantur.
Nihil novi facio, nihil novi video ...

Seneca
4 v. - 65 n. Chr.

[1] Benannt nach Niklaus Wirth, dem Erfinder von Pascal.
[2] Dieses Beobachtung wir oft, nach Hiob **1**,21, als: *„Intel giveth and Microsoft taketh away“* formuliert.
[3] Ein möglicher Ausweg wäre hier ein Quantencomputer.

Literaturverzeichnis

[1] G. Abramson, M. Kuperman: *Small world effect in an epidemological model*, Phys. Rev. Lett. 86, S. 2587, 2001

[2] A. Abran, M. Maya: *A Sizing Measure for Adaptive Maintenance Work Products*, International Conference on Software Maintenance, IEEE Computer Society, Nizza, Frankreich, S. 286-294, 1995

[3] A. Abran, P. N. Robilliard: *Function Point Analysis: An Empirical Study of Its Measurement Processes*, IEEE Transactions on Software Engineering Vol. 22(12), S. 895-910, 1996

[4] J.R. Abrial: *Assigning Programs to Meaning*, Prentice Hall, 1993

[5] J. Adams et al.:*Patterns for e-Business*, IBM Press, 2002

[6] W. S. Adolph: *Cash cow in a tar pit: Reengineering a legacy system*, IEEE Software, Vol. 13(3), S. 41-47, 1996

[7] *Manifesto for Agile Software Development*, http://www.agilealliance.org

[8] R. Agraval et al.: *Vinci: A Service-Oriented Architecture for Rapid Development of Web Applications*, http://www.almaden.ibm.com/cs/people/bayardo/ps/www2001.pdf, 2001

[9] A. Alderson, H. Shah: *Viewpoints on Legacy Systems*, Communications of the ACM, Vol. 42(3), S. 115-117, 1999

[10] C. Alexander et al.: *A Pattern Language*, Oxford University Press, 1977

[11] C. Alexander: *The Timeless Way of Building*, Oxford University Press, 1979

[12] D. Alur et al.: *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall, 2001

[13] S.W. Ambler: *Agile Modeling: Best Practices for the Unified Process and Extreme Programming*, Wiley Computer Press, 2002

[14] A. I. Antón, C. Potts: *Functional Paleontology: System Evolution as the User Sees It*, Proc. 23rd (IEEE) International Conference on Software Engineering, Toronto, Canada, S. 421-430, 2001

[15] R. Arnold:*Impact Analysis of Software Systems*, McGraw-Hill, 1990
[16] R. Arnold:*Software Reengineering*, IEEE Computer Society Press, 1993
[17] E. C. Arranga, F. P. Coyle: *COBOL: Perceptions and reality*, Computer, Vol. 30(3), S. 126-128, 1997
[18] C. Ashworth, M. Goodland: *SSADM: A Practical Approach*, McGraw Hill, 1990
[19] T. Augustine, C. Schroeder: *An Effective Metrics Process Model*, CrossTalk, Vol. 12(6), S. 4-7, 1999
[20] C. Axton et al.: *Web Services for the Enterprise: Opportunities and Challenges*, Ovum, 2002
[21] N. T. J. Bailey: *The mathematical theory of infectious diseases*, Griffin, 1975
[22] T. Baker: *Lessons Learned Integrating COTS-Software into Systems*, in: Proceedings of the First International Conference on COTS-Based Software Systems, Lecture Notes in Computer Science, Springer, 2002
[23] M.J. Balcer, S.J. Mellor, *Executable UML: A Foundation for Model Driven Architecture*, Addison Wesley, 2002
[24] A. L. Barbasi: *Linked: How Everything is connected to Everything Else and What it Means for Business, Science and Everyday Life*, Plume, 2003
[25] A. L. Barbasi, E. Bonabeau: *Scale-free networks*, Scientific American, May Ausgabe, S. 60-69, 2003
[26] V. Basili, D. Weiss:*A Methodology for Collecting Valid Software Engineering Data*, IEEE Transactions on Software Engineering, Vol. 10(6), S. 728-738, 1984
[27] V. Basili et al.:*How Reuse Influences Productivity in Object Oriented Systems*, Communications of the ACM, Vol. 39(10), S. 104-116, 1996
[28] L. Bass et al.: *Software Arcitecture in Practice*, Addison Wesley, 1997
[29] S. Bayer, J. Highsmith: *RADical software development*, American Programmer, Vol. 7, S. 35-42, 1994
[30] H. Bauer:*Unternehmensportale: Geschäftsmodelle, Design, Technologien*, Galileo, 2001
[31] K. Beck: *Extreme Programming Explained. Embrace Change*, Addison Wesley, 2000
[32] S.G. Beckner, S.T. Norman: *Air Force Development Guide*, MITRE Technical Report 98B00000074, 1998
[33] M. Beedle: *Pattern Based Reengineering*, Object Magazine, 1997
[34] B.L. Belady, M.M. Lehman: *A model of large program development*, IBM Systems Journal, 15(3), 1976
[35] S. Bendifallah, R. Scacchi: *Understanding Software Maintenance Work*, IEEE Transactions on Software Engineering, Vol. 13(3), S. 311-334, 1987
[36] K. Bennett: *Legacy Systems: Coping with Success*, IEEE Software, Vol. 12(1), S. 19-24, 1995

[37] P. Benyon-Davies: *Information Systems Development: An Introduction to Information Systems Engineering*, The Macmillan Press, 1993
[38] S. C. Berczuk, B. Appleton: *Software Configuration Management Patterns*, Addison Wesley, 2002
[39] P. Bernus et al.:*Handbook of Architectures of Information Systems*, Springer, 1998
[40] Bianchi, A. et al.: *Evaluating Software Degradation through Entropy.* in: *Pro. of the Seventh International Software Metrics Symposium METRICS 2001*, S. 210-220, London 2001
[41] J. Bisbal et al.: *Legacy Information Systems: Issues and Directions*, IEEE Software, September/October, 1999
[42] T. J. Biggerstaff: *Design Recovery for Maintenance and Reuse*, IEEE Computer, Vol. 22(7), S. 36-49, 1989
[43] B. I. Blum: *Software Engineering – A Holistic View*, Oxford University Press, 1992
[44] B. W. Boehm: *Software Engineering Economics*, Prentice Hall, 1981
[45] B. Boehm: *A spiral model of software development and enhancement*, IEEE Computer, Vol. 21, S. 61-72, 1988
[46] B. W. Boehm: *Software Risk Management*, IEEE Computer Society Press, 1989
[47] B. Boehm: *Anchoring the Software Process*, IEEE Software, Vol. 13(7), S. 73-82, 1996
[48] B. Boehm, D. Port: *Escaping the Software Tar Pit: Model Clashes and How to Avoid Them*, ACM Software Engineering Notes, Vol. 24(1), S. 36-48, 1999
[49] S. Bollig, D. Xiao: *Throwing of the shackles of a legacy system*, IEEE Computer, Vol. 31(6), S. 104-109, 1998
[50] T. B. Bollinger, C. McGowan: *A Critical Look at Software Capability Evaluations*, IEEE Software, Vol. 8(4), S. 25-41, 1991
[51] P. Bollobais: *Random Graphs*, Academic Press, 1985
[52] G. Booch et al.: *The Unified Modelling Language User Guide*, Addison Wesley, 1999
[53] J. Bosch: *Design and Use of Software Architectures*, Addison Wesley, 2000
[54] J. Bosch, M. Högström: *Product Instantiation in Software Product Lines: A Case Study* in: *GCSE '00: Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers*, Springer, 2001
[55] A. Bouguettaya et al.: *Interconnecting Heterogeneous Information Systems*, Kluwer Academic Press, 1998
[56] D. Box: *Essential COM*, Addison Wesley, 1998
[57] D. Box et al.: Simple Object Access Protocol (SOAP) 1.1, W3C Note, http://www.w3c.org/TR/SOAP/, 2000

[58] K. Braa et al.:*Organizing the Redesign Process in System Development*, The Journal of Systems and Software, Vol. 33(2), S. 133-140, 1996
[59] M.H. Brackett:*Data Sharing Using a Common Data Architecture*, Wiley Computer Press, 1994
[60] J. Bradshaw (Hrsg.): *Software Agents*, AAAI Press, 1997
[61] B. Brehmer: *In one word: Not from experience*, Acta Psychologica, Vol. 45, S. 223-241, 1980
[62] L. Briand et al.: *Q-MOPP: qualitative evaluation of maintenance organizations, processes and products*, Journal of Software Maintenance: Research and Practice, Vol. 10(4), S. 249-278, 1998
[63] P. Bristow et al.: *Enterprise Portals: Business Application and Technologies*, Butler Group, 2001
[64] C. Britton: *IT Architecture and Middleware: Strategies for Building Large, Integrated Systems*, Prentice Hall, 2001
[65] M. Brodie, M. Stonebraker: *Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach*, Morgan Kaufmann, 1995
[66] R. Brooks: *Towards a theory of the comprehension of computer programs*, International Journal of Man-Machine Studies, Vol. 18, S. 543-554, 1983
[67] W. J. Brown et al.: *AntiPatterns: Refactoring Software, Architectures and Projects in Crisis*, John Wiley & Sons, 1998
[68] L. Brownsword et al.: *Developing New Processes for COTS-based Systems*, IEEE Software, Vol. 17(4), S. 48-55, 2000
[69] R. Budde te al.: *Prototyping: An Approach to Evolutionary System Development*, Springer, 1992
[70] J. Buckley et al.: *Towards a taxonomy of software changes*, Journal of Software Maintenance and Evolution: Research and Practice 2003, S. 1-7, 2003
[71] W. J. Buffam: *E-Business and IS Solutions: An Architectural Approach to Business Problems and Opportunities*, Addison Wesley, 2000
[72] B. Burton et al.:*The Reusable Software Library*, IEEE Software 4(4), S. 25-33, 1987
[73] F. Buschmann et al.: *Pattern-Oriented Software Architecture – A System of Patterns*, John Wiley & Sons, 1996
[74] D. N. Card, R. L. Glass: *Measuring Software Design Quality*, Prentice Hall, 1990
[75] D. Carney et al.: *Complex COTS-Based Software Systems: Practical Steps for Their Maintenance*, Journal of Software Maintenance: Research and Practice, Vol. 12(6), S. 357-376, 2001
[76] N. Chapin et al.: *Types of software evolution and software maintenance*, Journal of Software Maintenance and Evolution, Vol. 13, S. 3-30, 2001

[77] G. Chastek et al.: *Product Line Analysis: A Practical Introduction*, Carnegie Mellon University, 2001
[78] P. Checkland, J. Scholes: *Soft Systems Methodology in Action*, John Wiley & Sons, 1990
[79] J. Chew: *Making ERP Work*, Forrester Research Inc., 2001
[80] E. Chikofsky, J. Cross II: *Reverse Engineering and Design Recovery: A Taxonomy*, IEEE Software, Vol. 7(1), S. 13-17, 1990
[81] W. C. Chu et al.:*Pattern based software re-engineering: a case study*, Journal of Software Maintenance: Research and Practice, Vol. 12(2), S. 121-141, 2000
[82] L. Chung et al.: *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publisher, 2000
[83] A. Cimitile, G. Visaggio: *Software salvaging and the call dominance tree*, Journal of Systems Software, Vol. 28, S. 117-127, 1995
[84] A. Cimitile et al.:*Identifying objects in legacy systems*, 5th International Workshop on Program Comprehension, IWPC'97, S. 138-147, 1997
[85] W. J. Clancey: *Heuristic Classifications*, Artificial Intelligence, Vol. 27, S. 289-350, 1985
[86] W. J. Clancey: *Model Construction Operators*, Artificial Intelligence, Vol. 27, S. 289-350, 1985
[87] P. Clements, L. Northrop: *Software Production Lines: Practices and Patterns*, Addison Wesley, 2001
[88] P. Clements et al.: *Evaluating Software Architectures, Methods and Case Studies*, Addison Wesley, 2002
[89] P. Coad et al.: *Object Models: Strategies, Patterns and Applications*, Prentice Hall, 1997
[90] A. Cockburn: *Goals and Use Cases*, Journal of Object Oriented Programming, Vol. 10(5), S. 35-40 , 1997
[91] A. Cockburn: *Surviving Object Oriented Projects*, Addison Wesley, 1998
[92] A. Cockburn: *Agile Software Development*, Addison Wesley, 2002
[93] M.A. Cook: *Building Enterprise Information Architectures: Reengineering Information Systems*, Prentice Hall, 1996
[94] D. Coleman et al.: *Using Metrics to Evaluate Software System Maintainability*, Computer, Vol. 27(8), S. 44-49, 1994
[95] D. Coleman et al.: *The Application of Software Maintainability Models in Industrial Software Systems*, Journal of Systems Software, Vol. 29(1), S. 3-16, 1995
[96] P. Comer, J. Chard: *A measurement maturity model*, Software Quality Journal, Vol. 2(4), S. 277-289, 1993
[97] L. Constantine, L. Lockwood: *Software for Use*, ACM Press, 1999
[98] S. D. Conte et al.: *Software Engineering Metrics and Models*, Benjamin/Cummings Pub., 1986

[99] J. Coplien, D. Schmidt: *Pattern Languages of Program Design*, Addison Wesley, 1995
[100] J. Coplien: *Software Patterns*, SIGS Publication, 1996
[101] G. Coulouris, J. Dollimore: *Distributed systems – concepts and design*, Addison Wesley, 1994
[102] F. Coyle: *XML, Web Services and The Data Revolution*, Addison Wesley, 2002
[103] J. Crichlow: *The Essence of Distributed Systems*, Prentice Hall, 2000
[104] M. A. Cusumano, R. Selby: *Microsoft Secrets*, Addison Wesley, 1995
[105] M. A. Cusumano, R. Selby: *How Microsoft builds software*, Communication of the ACM, Vol. 40, S. 53-61, 1997
[106] M. A. Cusumano, D. B. Yoffe: *Software development on Internet time*, IEEE Computer, Vol. 32, S. 60-69, 1999
[107] C. Date: *Selected Readings in Database Systems*, Addison Wesley, 1987
[108] A. M. Davies: *Software Requirements: Objects, Functions and States*, Prentice Hall, 1993
[109] A. M. Davies: *Fifteen principles of software engineering*, IEEE Software, Vol. 11(6), S. 94-97, 1994
[110] M. M. Davydov: *Corporate Portals and e-Business Integration*, McGraw-Hill, 2001
[111] V. K. Decyk, C. D. Norton: *Modernizing Fortran 77 Legacy Code*, NASA Tech Briefs, Vol. 27(9), S. 72-74, 2003
[112] G. Dedene, J. P. De Vreese: *Realities of off-shore reengineering*, IEEE Software, Vol. 7(1), S. 35-45, 1995
[113] S. M. Deklava: *The Influence of the Information System Development Approach on Maintenance*, Management Information Systems Quarterly, Sept., 1992
[114] T. DeMarco: *Controlling Software Projects*, Dorset House, 1982
[115] M. Diaz, J. Sligo: *How Software Process Improvement Helped Motorola*, IEEE Software, Vol. 14(5), S. 75-81, 1997
[116] O. Diekmann, J. A. P. Heesterbeek: *Mathematical epidemology of infectious diseases: model building, analysis and interpretation*, John Wiley & Sons, 2000
[117] E. W. Dijkstra: *The End of Computing Science*, Communication of the ACM, Vol. 44(3), S. 92, 2001
[118] C. Domd, M. Green: *Phase Transition and Critical Phenomena*, Academic Press, 1972
[119] J. B. Dreger: *Function Point Analysis*, Prentice Hall, 1989
[120] M.J. Earl: *Information Management: The Strategic Dimension*, Oxford University Press, 1988
[121] C. Ebert: *Dealing with Nonfunctional Requirements in Large Software Systems*, Annals of Software Engineering, Vol. 3(9), S. 367-395, 1997
[122] J. Edwards, D. DeVoe: *3-Tier Client/Server At Work*, John Wiley & Sons, 1997

[123] N. Eldredge: *The Pattern of Evolution*, W. H. Freeman and Company, 1999
[124] K. El Emam et al.: *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination*, IEEE Computer Society, 1998
[125] R. Ellison et al.: *Survivable Network System Analysis: A Case Study*, IEEE Software, Vol. 16(4), S. 70-77, 1999
[126] J. Elster: *Explaining Technical Change*, Cambridge University Press, 1983
[127] W. Emmerich:*Engineering Distributed Objects*, John Wiley & Sons, 2000
[128] Expert Choice Inc.: *Expert Choice 2000 Quick Start Guide and Tutorials*, Expert Choice Inc., 2000
[129] D. Faust, C. Verhoef: *Software Product Line Migration and Deployment*, Software – Practice & Experience, John Wiley & Sons, 2003
[130] L. Fejis et al.: *A Relational Approach to Software Architecture Analysis*, Software Practice and Experience, Vol. 28(4), S. 371-400, 1998
[131] M. Felici: *Taxonomy of Evolution and Dependability*, Proceedings for the Second International Workshop on Unanticipated Software Evolution, USE 2003, Warschau, Polen, S. 95-104, 2003
[132] N. Fenton, S.L. Pfleeger: *Software Metrics: A Rigorous Approach*, Chapman & Hall, 1997
[133] S. Focardi et al.: *A Stochastic Model of Software Maintenance and Its Implications in Extreme Programming Processes*, in: G. Succi, M. Marchesi (Hrsg): *Extreme Programming Examined*, Addison Wesley, 2001
[134] M. Foucault: *Überwachen und Strafen: Die Geburt des Gefängnisses*, Suhrkamp 1976
[135] M. Fowler: *Analysis Pattern: Reusable Object Models*, Addison Wesley, 1997
[136] M. Fowler: *Refactoring: Improving the Design of Existing Code*, Addison Wesley, 1999
[137] W. B. Frakes, C. J. Fox: *Sixteen Questions About Software Reuse*, Communications of the ACM, Vol. 38(6), S. 75-87, 1995
[138] D. Frankel: *Model Driven Architecture: Applying MDA to Enterprise Computing*, John Wiley & Sons, 2003
[139] P. Freeman (Hrsg.): *Software Reusability*, Computer Science Press, 1987
[140] G. W. Furnas et al.: *The vocabulary problem in human-system communication* Communications of the ACM, Vol. 30(11), S. 964-971, 1987

[141] K. B. Gallagher, J. R. Lyle: *Using Program Slicing in Software Maintenance*, IEEE Transactions on Software Engineering, Vol. 17(8), S. 751-761, 1991
[142] E. Gamma et al.: *Design Patterns, Elements of Reusable Object Oriented Software*, Addison Wesley, 1995
[143] N. Ganti, W. Brayman: *The Transition of Legacy Systems to a Distributed Architecture*, John Wiley and Sons, 1995
[144] D. Garlan, M. Shaw: *An Introduction to Software Architecture, Advances in Software Engineering*, 1, World Scientific, 1993
[145] D. Garmus, D. Herron: *Function Point Analysis – Measurement Practices for Successful Software Projects*, Addison Wesley, 2001
[146] P. R: Garvey: *Probability Methods for Cost Uncertainty Analysis – A Systems Engineering Perspective*, Marcel Dekker Inc., 2000
[147] J. Gharajedaghi: *Systems Thinking. Managing Chaos and Complexity. A Platform for Designing Business Architecture*, Butterworth-Heinemann, 1999
[148] W. W. Gibbs: *Software's Chronic Crisis*, Scientific American, September, S. 86-95, 1994
[149] T. Gilb: *Principles of Software Engineering Management*, Addison Wesley, 1988
[150] R. L. Glass: *Business applications: what should a programming language offer?*, The Software Practitioner, Septemberausgabe, 1996
[151] R. L. Glass: *Cobol – a contradiction and an engima*, Communications of the ACM, Vol. 40(9), S. 11-13, 1997
[152] R. L. Glass: *Computing Calamities – Lessons Learned from Products, Projects and Companies that Failed*, Prentice Hall, 1998
[153] R. L. Glass: *Software Runaways – Lessons Learned from Massive Software Projects Failures*, Prentice Hall, 1998
[154] R. L. Glass: *ComputingFailures.com – War Stories from the Electronic Revolution*, Prentice Hall, 2001
[155] J. Gleick: *Chaos: Making a New Science*, Penguin Books, 1987
[156] B. Goldfedder: *The Joy of Patterns: Using Patterns for Enterprise Development*, Addison Wesley, 2002
[157] M. Goodyear: *Enterprise System Architectures*, CRC Press, 2001
[158] S. Gossain: *Accessing Legacy Systems*, in: ObjectExpert, 1997(3)
[159] R. B. Grady: *Practical software metrics for project management an process improvement*, Prentice-Hall, 1992
[160] I. Graham: *Migrating to Object Technology*, Addison Wesley, 1994
[161] I. Graham: *Reuse: A Key to Successful Migration*, Object Magazine, Vol. 5(6), S. 82-83, 1995
[162] S. Graham et al.:*Building Web Services with Java*, Sams Publishing, 2002
[163] E. Gray, W. Smith: *On the limitations of software process assessment and the recognition of a required re-orientation fpr global process improvement*, Software Quality Journal, Vol. 7(1), S. 21-34, 1998

[164] R. Grimes: *DCOM Programming*, Wrox, 1997
[165] T. J. Grose et al.: *Mastering XMI: Java Programming with XMI, XML and UML*, John Wiley & Sons, 2002
[166] V. Gurbaxani: *The new world of Information Technology outsourcing*, Communications of the ACM, Vol. 39(7), S. 45-47, 1996
[167] E. Hall: *Managing Risk, Methods for Software Systems Developers*, Addison Wesley 1997
[168] T. Hall, N. Fenton: *Implementing Effective Software Metrics Programs*, IEEE Software, Vol 14(2), S. 55-65, 1997
[169] M. Halstead: *Elements of Software Science*, Elsevier North Holland, 1977
[170] M. Hammer, J. Champy: *Reengineering the Corporation*, New York, 1993
[171] D. Harel, M. Politi: *Modeling Reaction Systems with Statecharts*, McGraw-Hill, 1998
[172] E. Harrold, W. Means: *XML in a Nutshell*, O'Reilly, 2001
[173] P. Harmon et al.: *Developing E-Business Systems and Architectures*, Morgan Kaufman, 2001
[174] W. Harrison: *An Entropy-Based Measure of Software Complexity*, IEEE Transactions on Software Engineering, 18(11), 1992
[175] D.C. Hay: *Data Model Patterns: Conventions of Thought*, Dorset House, 2003
[176] D.C. Hay: *Requirement Analysis: From Business Views to Architecture*, Prentice Hall, 2003
[177] M. Henning, S. Vinoski: *Advanced CORBA Programming with C++*, Addison Wesley, 1999
[178] S. Henry: *Software metrics based on information flow*, IEEE Transactions on Software Engineering, Vol. 7(5), S. 481, 1981
[179] B. Henderson-Sellers: *Object-oriented metrics: measures of complexity*, Prentice Hall 1996
[180] Th. Hess, L. Brecht: *State of the Art des Business Process Redesign*, Gabler Verlag, 1995
[181] A. Hevner: *Phase containment metrics for software quality improvement*, Information and Software Technology, Vol. 39(13), S. 867-877, 1997
[182] D. Higgins: *Data Structured Software Maintenance*, Dorset House Publishing, 1986
[183] J. Highsmith: *Adaptive Software Development – A Collaborative Approach to Managing Complex Systems*, Dorset House, 2000
[184] J. Highsmith: *Agile Software Development Ecosystems*, Addison Wesley, 2002
[185] S. Hissam, D. Carney: *Isolating Faults in Complex COTS-Based Systems*, Journal of Software Maintenance: Research and Practice, Vol. 11(3), S. 183-199, 1998

[186] H. F. Hofmann, F. Lehner: *Requirements Engineering as a success factor in software projects*, IEEE Software, Juli/August 2001
[187] C. Hofmeister et al.: *Applied Software Architecture*, Addison Wesley 1999
[188] J. Holland: *Hidden Order: How Adaptation Builds Complexity*, Addison Wesley, 1995
[189] J. Holland: *Emergence: From Chaos to Order*, Addison Wesley, 1998
[190] I. F. Hooks, K. A. Farry: *Customer Centered Products: Creating Successful Products Through Smart Requirements Management*, Amacom, 2001
[191] F. Hoque: *e-Enterprise: Business Models, Architecture and Components*, Cambridge University Press, 2000
[192] A. Hunt, D. Thomas: *The Pragmatic Programmer*, Addison Wesley, 2000
[193] H.H. Husmann: *Re-engineering Economics*, Eden Systems, 1990
[194] D.W. Hybertson et al.: *Maintenance of COTS-intensive Software Systems*, Journal of Software Maintenance, Vol. 9(4), S.203-216, 1997
[195] I. Jacobsen et al.: *Software Reuse – Architecture, Process and Organization for Business Success*, Addison Wesley, 1997
[196] R. Jain: *The Art of Computer Systems Performance Analysis*, Wiley-Interscience 1991
[197] P. Jalote: *Fault Tolerance in Distributed Systems*, Prentice Hall, 1994
[198] M. Jazayeri et al.: *Software Architecture for Product Families*, Addison Wesley, 2000
[199] T. Jennings: *Intelligent Integration*, Butler Group, 2002
[200] R. Jeffrey, J. Stathis: *Function Point Sizing: Structure, Validity and Applicability*, Empirical Software Engineering: An International Journal, Vol. 1(1), S. 11-30, 1996
[201] C. Jones: *Assessment and Control of Software Risks*, Prentice-Hall, 1994
[202] C. Jones: *Software Metrics: Good, Bad, and Missing*, Computer Vol.27(9), S. 98-100, 1994
[203] C. Jones: *Applied Software Measurement: Assuring Productivity and Quality*, McGraw-Hill, 1996
[204] C. Jones: *Patterns of Software Systems Failure and Success*, International Thomson Computer Press, 1996
[205] C. Jones: *Estimating Software Costs*, McGraw-Hill, 1998
[206] C. Jones: *The Year 2000 Software Problem – Quantifying the Costs and Assessing the Consequences*, Addison Wesley, 1998
[207] M. Jørgensen, *An Empirical Study of Software Maintenance Tasks*, Journal of Software Maintenance: Research and Practice, Vol.7(1), S. 27-48, 1995
[208] M. Jørgensen, D. I. K. Sjøberg, *Impact of experience on maintenance skills*, Journal of Software Maintenance: Research and Practice, Vol.14(2), 2002

[209] E. A. Karlsson: *Software Reuse – A Holistic Approach*, John Wiley & Sons, 1995
[210] R. Kazman et al.: *Scenario Based Analysis of Software Architecture*, IEEE Software, Vol. 13(11), S. 47-55, 1996
[211] J. J. Kassbøll: *How evolution of information systems may fail: many improvements adding up to negative effects*, European Journal of Information Systems, Vol. 6, S. 172-180, 1997
[212] S. Kauffman, W. MacReady; *Technological Evolution and Adaptive Organisation*, Complexity Journal, Vol. 1(2), S. 26-43, 1995
[213] K. E. Kendall, J. E. Kendall: *Systems Analysis and Design*, Prentice-Hall, 1995
[214] B. Kernighan, P. J. Plauger: *Elements of Programming Style*, Mc Graw-Hill, 1978
[215] G. Kiczales et al.: *Getting Started with AspectJ*, Communications of the ACM, Vol. 44(10), S. 59-65, 2001
[216] R. Klösch, H. Gall: *Objektorientiertes Reverse Engineering*, Springer, 1995
[217] S. T. Knox: *Modeling the Cost of Software Quality*, Digital Technical Journal, Vol. 5(4), S. 9-17, 1993
[218] D.E. Knuth: *Literate Programming*, The Computer Journal, Vol. 27, S. 97-111, 1984
[219] W. Kobitzsch et al.: *Outsourcing in India*, IEEE Software, Vol 12(2), S. 78-86, 2001
[220] J. Kontio: *A Case Study in Applying a Systematic Method for COTS-Software Selection*, in: Proceedings of the International Conference on Software Engineering, Berlin March 25-29, 1996, IEEE Computer Society, 1996
[221] P.B. Kruchten: *The Rational Unified Process*, Addison Wesley, 1999
[222] M. Lacity, R. Hirschheim: *Information Systems Outsourcing: Myths, Metaphors and Realities*, John Wiley & Sons, 1993
[223] M. Lacity, R. Hirschheim: *Beyond the Information Systems Outsourcing Bandwagon: The Insourcing Response*, John Wiley & Sons, 1995
[224] T. Langenohl: *Systemarchitekturen elektronischer Märkte*, Rosch-Buch, 1995
[225] F. Lanubile, G. Vissagio: *Decision-driven Maintenance*, Journal of Software Maintenance, Vol. 7(7), S. 91-116, 1999
[226] K. Lee et al.: *Feature Based Approach to Object-Oriented Engineering of Applications for Reuse*, Software– Practice and Experience, Vol. 30(9), S. 1025-1046, 2000
[227] M. M. Lehman: *On understanding Laws, evolution and conversation in the large program lifecycle*, Journal of Software & Systems,. Vol. 1, S. 213-221, 1980

[228] M. M. Lehman: *Programs, Life Cycles and Laws of Software Evolution*, Proceedings of IEEE on Software Engineering,. Vol. 68(9), S. 1060-1076, 1980
[229] M. M. Lehman: *Program evolution*, Academic Press, 1985
[230] M. M. Lehman et al.:*Metrics and laws of Software Evolution*, In: *Proceedings of the Fourth International Software Metrics Symposium*, Albuquerque, 1997
[231] F. Lehner: *Softwarewartung, Management und methodische Unterstützung*, Hanser Verlag, 1991
[232] S. Letovsky, E. Soloway: *Delocalized plans and program comprehension*, IEEE Software, Vol. 19(3), S. 41-48, 1986
[233] B. Lienz, E. Swanson: *Software Maintenance Management*, Addison Wesley, 1980
[234] W. C. Lim: *Effects of reuse on quality, productivity and economics*, IEEE Software, Vol. 17(1), S. 23-30, 1994
[235] W. C. Lim: *Managing Software Reuse*, Prentice Hall, 1997
[236] D. S. Linthicum: *David Linthicum's Guide to Client/Server and Intranet Development*, John Wiley & Sons, 1997
[237] D. S. Linthicum: *Enterprise Application Integration*, Addison Wesley, 1999
[238] D. S. Linthicum: *B2B Application Integration – e-Businesss-Enable Your Enterprise*, Addison Wesley, 2000
[239] S. B. Lippman, J. Lajoie: *C++ Primer*, Addison Wesley, 1998
[240] M. Lippert et al.: *EXtreme Programming in Action*, John Wiley & Sons, 2002
[241] M. Lorenz, J. Kidd: *Object-Oriented Software Metrics. A Practical Guide*, Prentice Hall, 1994
[242] C. Lovelock et al.: *Services Marketing*, Prentice Hall, 1996
[243] R. Malveau, T.J. Mowbray: *Software Architect Bootcamp*, Prentice Hall, 2001
[244] J. J. Marciniak (Hrsg.): *Encyclopedia of Software Engineering*, John Wiley & Sons, 1994
[245] F. Marinescu: *EJB Design Patterns – Advanced Patterns, Processes and Idioms*, John Wiley & Sons, 2002
[246] J. Martin, J. Leben: *Client/Server Database Enterprise Computing*, Prentice Hall, 1995
[247] J. Martin, C. McClure: *Software Maintenance: The Problem and Its Solutions*, Prentive Hall, 1983
[248] R. C. Martin:*Agile Software Development, Principles, Patterns and Practices*, Prentice Hall, 2002
[249] D. Masak: *Moderne Enterprise Architekturen*, Springer, 2004
[250] A. von Mayrhauser, A. M. Vans: *Identification of dynamic comprehension processes during large scale maintenance*, IEEE Transactions on Software Engineering, Vol. 22(6), S. 424-438, 1996
[251] P. McBreen: *Software Craftsmanship*, Addison Wesley, 2002

[252] T. J. McCabe, A. H. Watson: *Software Complexity*, Journal of Defense Software Engineering Vol. 7(12), S. 5-9, 1994
[253] L. A. McCauley: *Requirements Engineering*, Springer Verlag, 1996
[254] C. McClure: *The three R's of software automation: re-engineering, repository, reusability*, Prentice Hall, 1992
[255] J. McGovern et al.:*Java Web Services Architecture*, Morgan Kaufmann, 2003
[256] J. D. McGregor, D. A. Sykes: *A Practical Guide to Testing Object Oriented Software*, Addison Wesley, 2001
[257] *OMG Model Driven Architecture Home Page*, http://www.omg.org/mda/index.htm
[258] N. Mead et al.:*Managing Software Development for Survivable Systems*, Annals of Software Engineering, Vol. 11(11), S. 45-78, 2000
[259] N. Mead et al.:*Towards Survivable COTS-Based Systems*, Cutter IT-Journal, Vol. 14(2), S. 4-11, 2001
[260] S. Mellor, M. Balcer: *Executable UML: A Foundation for Model Driven Architecture*, Addison Wesley, 2002
[261] A. Melton: *Software Measurement*, International Thomson Computer Press 1996
[262] S. J. Metsker: *Design Patterns in C#*, Addison Wesley, 2004
[263] S. Meyers: *Effective C++ - 50 Specific Ways to Improve Your Programs and Designs*, Addison Wesley, 1992
[264] B. Meyers, P. A. Oberndorf: *Managing Software Acquisition: Open Systems and COTS-Software Products*, Addison Wesley, 2001
[265] H. F. Mili et al.:*Reusing Software: Issues abd Research Directions*, IEEE Transactions on Software Engineering, Vol. 21(6), S. 528-562, 1995
[266] M. Minsky: *The Psychology of Computer Vision*, McGraw-Hill, 1975
[267] M. E. Modell: *A Professional's Guide to Systems Analysis*, McGraw Hill, 1996
[268] *OMG Meta Object Facility Specification*, http://www.omg.org
[269] R. Monson-Haefel: *Enterprise Java Beans*, O'Reilly, 2000
[270] D. Montgommery: *Introduction to Statistical Quality Control*, John Wiley & Sons, 1985
[271] C. Moore, M. E. J. Newman: *Epidemics and percolation in small-world networks*, Phys. Rev. E61, S. 56-78, 2000
[272] J.P. Morgenthal: *Enterprise Application Integration with XML and Java*, Prentice Hall, 2001
[273] J. Moubray: *Reliability-Centered Maintenance*, Industrial Press, 2001
[274] H.A. Müller et al.: *A Reverse Engineering Approach To Subsystem Structure Identification*, Software Maintenance Research and Practice, Vol. 5, S. 181-204, 1993
[275] G. Myers: *Reliable Software through Composite Design*, Petrocelli/Charter, 1975

[276] E. Newcomer: *Understanding Web Services: XML, WSDL, SOAP and UDDI*, Addison Wesley, 2002
[277] F. Niessink, H, van Vliet: *Software Maintenance from a Service Perspective*, Journal of Software Maintenance, Vol. 12(2), S. 103-120, 2000
[278] G. Norris et al.: *E-Business and ERP: Transforming the Enterprise*, John Wiley & Sons, 2000
[279] P. Nowak: *Structures and Interactions – Characterising Object Oriented Software Architectures*, Dissertation, University of Southern Denmark, 1999
[280] D. Oberle: *Mythologie der Informatik*, preprint, Universität Karlsruhe, 2001
[281] R. J. Offen, R. Jeffery: *Establishing Software Measurement Programs*, IEEE Software, Vol. 14(2), S, 45-53, 1997
[282] P. W. Oman, J. Hagemeister: *Constructing and Testing of Polynomials Predicting Software Maintainability*, Journal of Systems and Software Vol. 24(3), S. 251-266, 1994
[283] P. W. Oman, C. Cook: *A taxonomy for programming style*, 18th ACM Computer Science Conference Proceedings, S. 244-247, 1990
[284] A. Onoma et al.: *Software Maintenance – an Industrial Experience*, Journal of Software Maintenance: Research and Practice, Vol. 7, S. 333-375, 1995
[285] R. Orfali et al.:*Instant CORBA*, John Wiley & Sons, 1997
[286] R. Otte et al.: *Understanding CORBA*, Prentice Hall, 1996
[287] M. A. Ould: *CMM and ISO 9001*, Software Process – Improvement and Practice, Vol. 2(4), S. 281-289, 1996
[288] S.R. Palmer, J.M. Felsing: *A Practical Guide to Feature-Driven Development*, Prentice Hall, 2002
[289] G. Parikh(Hrsg.): *Techniques of Program and System Maintenance*, Q.E.D. Information Science, 1988
[290] R. Pastor-Satorras et al.: *Dynamical and correlation properties of the Internet*, Phys. Rev. Lett. 87, S. 258, 2001
[291] M. C. Paulk: *How ISO 9001 compares to CMM*, IEEE Software Vol. 12(1), S. 74-83, 1995
[292] W. E. Perry: *Managing System Maintenance*, Q.E.D. Information Science, 1981
[293] D. Perry, A.L. Wolf: *Foundations for the Study of Software Architecture*, ACM Software Engineering Notes, 17(4), 1992
[294] T. Pfarr, J. E. Reis: *The Integration of COTS/GOTS Within NASA's HST Command and Control System*, Proceedings of the First International Conference on COTS-Based Software Systems, Lecture Notes in Computer Science, Springer, 2002
[295] S.L. Pfleeger: *Lessons Learned in Building a Corporate Metrics Program*, IEEE Software, Vol. 10(3), S. 67-74, 1993

[296] S.L. Pfleeger: *Software Engineering – Theory and Practice*, Prentice Hall, 1998
[297] T.M. Pigoski: *Practical Software Maintenance: Best Practices for Managing Your Software Investment*, Wiley 1997
[298] H. Plattner, *Der Einfluß der Client-Server-Architektur*, Gabler, 1991
[299] J. Poole et al.: *Common Warehouse Metamodel*, John Wiley & Sons, 2003
[300] M. Poppendieck, T. Poppendieck: *Lean Development – An Agile Toolkit*, Addison Wesley, 2003
[301] J. S. Poulin: *Managing Software Reuse*, Addison Wesley, 1997
[302] R.S. Pressman: *Software Engineering – A Practicioner's Approach*, McGraw-Hill, 1997
[303] I. Prigogine, I. Steingers: *The End of Certainty: Time, Chaos and the New Laws of Nature*, Free Press, 1991
[304] K. Pulford et al.: *A Quantitative Approach To Software Management – The Ami Handbook*, Addison Wesley, 1996
[305] R. Purushothaman, D. E. Perry: *Towards Understanding Software Evolution: One-Line Changes*, International Workshop on Mining Software Repositories, 2004
[306] L. H. Putnam, W. Myers: *Measures for Excellence – Reliable Software on Time, Within Budget*, Yourdan Press Computing Series, 1992
[307] J. R. Putnam: *Architecting with RM-ODP*, Prentice Hall, 2001
[308] R. Rajkumar: *Synchronization in Real-Time Systems: A Priority Inheritance Approach*, Kluwer Academic Publishers, 1991
[309] W. E. Rajput: *E-Commerce Systems Architecture and Applications*, Artech House, 2000
[310] E. Ray: *Learning XML*, O'Reilly, 2001
[311] E. Rechtin, M.W. Maier: *The Art of Systems Architecting*, CRC Press, 1997
[312] D. Reisberg: *Cognition*, W. W. Norton & Company, 1997
[313] R. Riehm: *Integration von heterogenen Applikationen*, Difo-Druck Bamberg, 1997
[314] L. Rising; *The Patterns Handbook: Techniques, Strategies and Applications*, Cambridge University Press, 1998
[315] R. Rock-Evans: *DCOM Explained*, Digital Press, 1998
[316] D. Rogerson: *Inside COM*, Microsoft Press, 1997
[317] T. P. Rout: *SPICE: A Framework for Software Process Assessment*, Software Process – Improvement and Practice, Vol. 1(1), S. 57-66, 1995
[318] W. Royce: *Software Project Management: A Unified Framework*, Addison Wesley, 1998
[319] W. Rubin, M. Brain: *Understanding DCOM*, Prentice Hall, 1999
[320] S. Rugaber, J. White: *Restoring a Legacy: Lessons Learned*, IEEE Software, Vol. 15(4), S. 28-33, 1998

[321] W. Ruh et al.: *Enterprise Application Integration: A Wiley Tech Brief*, John Wiley & Sons, 2001
[322] J. Rumbaugh et al.: *The Unified Modeling Language Reference Manual*, Addison Wesley, 1998
[323] W. Schaefer et al. (Hrsg.): *Software Reusability*, Ellis Horwood, 1994
[324] A.W. Scheer: *Architecture of Integrated Information Systems, Foundations of Enterprise Modelling*, Springer, 1992
[325] A.W. Scheer: *Business Process Engineering: Reference Models for Industrial Enterprises* Springer, 1994
[326] C. H. Schmauch: *ISO 9000 for Software Developers*, ASQC Quality Press, 1995
[327] D.C: Schmidt et al.: *Pattern-Oriented Software Architecture, Vol.2 : Patterns for Concurrent and Networked Objects*, John Wiley & Sons, 2000
[328] N. F. Schneidewind: *The State of Software Maintenance*, IEEE Transactions on Software Engineering, Vol. 13(3), S. 303-310, 1987
[329] S. L. Schneberger: *Client/Server software maintenance*, McGraw Hill, 1997
[330] P. Schuh: *Recovery, Redemption and Extreme Programming*, IEEE Software, Vol. 18, S. 34-41, 2001
[331] K. Schwaber, M. Beedle: *Agile Software Development with Scrum*, Prentice Hall, 2002
[332] K. Schwaber: *Agile Project Management with Scrum*, Microsoft Press, 2003
[333] R. C. Seacord et al.: *Modernizing Legacy Systems*, Addison Wesley, 2003
[334] D. Serain: *Middleware*, Springer, 1998
[335] M. Shaw, D. Garlan: *Software Architecture: Perspectives of an Emerging Discipline*, Prentice Hall, 1996
[336] G. Shegalov et al.: *XML-enabled workflow management for e-service across heterogenous platforms*, Springer, 2001
[337] M. Shepperd: *Fundamentals of software measurement*, Prentice Hall, 1995
[338] C. Shirky: *Web services and context horizons*, IEEE Computer, Vol. 35(9), S. 98-100, 2002
[339] J. Siegel, *CORBA fundamentals and programming*, John Wiley & Sons, 1996
[340] A. R. Simon: *Systems Migration – A Complete Reference*, Van Nostrand Reinhold, 1992
[341] D. Simon, T. Eisenbarth: *Evolutionary Introduction of Software Product Lines*, in: SPLC 2: Proceedings of the Second International Conference on Software Product Lines, S.272-282, Springer, 2002
[342] O. Sims: *Business Objects – Delivering Cooperative Objects for Client/Server*, McGraw-Hill, 1994

[343] M. Sipper: *The Emergence of Cellular Computing*, IEEE Computer, Vol. 37(7), S. 18-26, 1999
[344] C. Smith: *Performance Engineering of Software Systems*, Addison Wesley, 1990
[345] H. M. Sneed: *Software-Wartung*, Rudolf Miller Verlag, 1990
[346] H. M. Sneed: *Planning the Reengineering of Legacy Systems*, IEEE Software, 12(1), S. 24-34, 1995
[347] H. M. Sneed: *Objektorientierte Softwaremigration*, Addison Wesley, 1998
[348] H. M. Sneed: *Encapsulation of legacy software: A technique for reusing legacy software components.*, Annals of Software Engineering, Vol.9, S. 293-313, 2000
[349] I. Sommerville, P. Sawyer: *Requirements Engineering: A Good Practice Guide*, John Wiley & Sons, 1997
[350] I. Sommerville: *Software Engineering*, Addison Wesley, 2000
[351] J. Soukup: *Taming C++ – Pattern Classes and Persistence for Large Projects*, Addison Wesley, 1994
[352] J.F. Sowa, J.A. Zachman:*Extending and Formalizing the Framework for Information Systems Architecture*, IBM Journal 31(3), 1992
[353] S. Spewak: *Enterprise Architecture Planing*, John Wiley and Sons, 1992
[354] R. D. Stacey: *Complexity and Creativity in Organizations*, Berret-Koehler, 1996
[355] T. A. Standish: *An essay on software reuse*, IEEE Transactions on Software Engineering, Vol. 10(5), S. 494-497, 1984
[356] G. Stark et al.:*An Examination of the Effects of Requirement Changes on Software Maintenance Releases*, Journal of Software Maintenance: Research and Practice, Vol. 11, S. 293-309, 1999
[357] A. M. Stavely: *Toward Zero Defect Programming*, Addison Wesley, 1998
[358] T. D. Steiner, D. B. Teixera: *Technology in Banking – Creating Value and Destroying Profits*, McGraw-Hill, 1992
[359] N. Stern et al.:*COBOL for the* 21^{st} *Century*, John Wiley and Sons, 2003
[360] Stanford University: *Enterprise Architecture Home Page*, http://www.stanford.edu/group/APS/arch/index.html
[361] M. Svahnberg, J. Bosch: *Evolution in Software Product Lines: Two Cases*, Journal of Software Maintenance, Vol. 11(6), S. 391-422, 1999
[362] E. B. Swanson, C. M. Beath: *Maintaining information systems in organizations*, John Wiley & Sons, 1989
[363] C. Symons: *Function point analysis: difficulties and improvements*, IEEE Transactions on Software Engineering, Vol. 14(1), S. 2-11, 1988
[364] C. Szyperski: Component Software: Beyond Object Oriented Programming, Addison Wesley, 1997

[365] A. A. Takang, P. A. Grubb: *Software Maintenance Concepts and Practice*, Thomson Computer Press, 1996
[366] J. Taramaa et al.: *From Software Configuration to Application Management*, Journal of Software Maintenance and Evolution: Research and Practice, Vol. 8(1), S. 26-38, 1996
[367] A. Tannenbaum: *Metadata Solutions*, Addison Wesley, 2001
[368] T. Thai, H. Q. Lam: *Net Framework Essentials*, O'Reilly, 2001
[369] R. H. Thayer: *Software Engineering Project Management*, Edwards Brothers Inc., 2003
[370] N. Thomas: *Building Scalabale Web Applications / Web Services Using JCACHE, JMS and XML*, SpiritSoft, 2002
[371] S. Thatte: *XLANG: Web Services for Business Process Design*, Microsoft Corp., 2001
[372] F. Tip: *A survey of program slicing techniques*, Journal of Programming Languages, Vol. 3, S. 121-189, 1995
[373] The Open Group Architecture Framework: *Technical Reference Model*, http://www.opengroup.org/togaf
[374] K. S. Trivedi: *Probability ans Statistics with Reliability, Queuing and Computer Science Applications*, Prentice-Hall, 1982
[375] V. F. Turchin: *The Phenomenon of Science*, Columbia University Press, 1977
[376] V. F. Turchin: *A constructive interpretation of the full set theory*, Journal of Symbolic Logic, Vol. 52(1), S. 172-173, 1987
[377] W. M. Turski: *Reference Model for Smooth Growth of Software Systems*, IEEE Transactions ons Software Engineering, Vol. 22(8), 1996
[378] R. Tuver, M. Munroe: *An early Impact Analysis Technique for Software Maintenance*, Journal of Software Maintenance, Vol. 6(1), S. 35-52, 1994
[379] S. Valverde et al.:*Scale-Free Networks from Optimal Design*, Europhysics Letters, Vol. 60(4), S. 512-517, 2002
[380] C. Verhoef: *Quantitative IT Portfolio Management*, Science of Computer Programming, Vol. 45(1), S. 1-96, 2002
[381] C. Verhoef. *Quantitative Aspects of Outsourcing Deals.*, Science of Computer Programming, 2004. To Appear.
[382] H. van Vliet: *Software Engineering: principles and practice*, John Wiley & Sons, 2000
[383] J. Vlissides: *Pattern Hatching: Design Patterns Applied*, Addison Wesley, 1998
[384] J.M. Voas: *Disposable Information Systems: The Future of Maintenance?*, Journal of Software Maintenance, Vol. 11(2), S.143-150, 1999
[385] W. Völter et al.: *Server Component Patterns*, John Wiley & Sons 2002
[386] N. Wallace: *COM/DCOM*, The Coriolis Group, 1999

[387] K. Wallnau et al.:*Building Systems from Commercial Components*, Addison Wesley, 2001
[388] J. Warmer, A. Kleppe: *The Object Constraint Language*, Addison Wesley, 1999
[389] I. Warren: *The Renaissance of Legacy Systems*, Springer, 1998
[390] D. J. Watts, S. H. Strogatz: *Collective Dynamics of Small-World Networks*, Nature, Vol. 393, S. 440, 1998
[391] D. J. Watts: *Small Worlds: the Dynamics of Networks between Order and Randomness*, Princeton University Press, 1999
[392] L. Wayne: *Managing Software Reuse*, Prentice Hall, 1998
[393] B. F. Webster: *Pitfalls of Object-Oriented Development*, M& T Books, 1995
[394] K. D. Welker, P. W. Oman: *Software Maintainability Metrics Models in Practice*, Journal of Defense Software Engineering Vol. 8(11), S. 19-23, 1995
[395] D. Weiss, C. T. R. Lai: *Software Product-Line Engineering: A Family-Based Software Development Process*, Addison Wesley, 1999
[396] S.A. Whitmire: *Object-Oriented Design Measurement*, John Wiley & Sons, 1997
[397] L. Willcocks, G. Fitzgerald: *A Business Guide to Outsourcing Information Technology*, Business Intelligence, 1994
[398] H. Williamson: *XML: The Complete Reference*, Osborne, 2001
[399] W.E.Wong et al.: *Quantifying the Closeness between Program Components and Features*, The Journal of Systems and Software, Vol. 54, S. 87-98, 2000
[400] D. Woods: *Enterprise Services Architecture*, O'Reilly, 2003
[401] H. Willke: *Systemtheorie I: Eine Einführung in die Grundprobleme der Theorie sozialer Systeme*, Lucis& Lucius, 2001
[402] R. K. Yin: *Case Study Research Design an Methods*, Sage Publications, 1994
[403] E. Yourdan, L.L. Constantine: *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice Hall, 1986
[404] E. Yourdan, L.L. Constantine: *Decline and Fall of the American Programmer*, Prentice Hall, 1993
[405] E. Yourdan: *Death March – The Complete Software Developer's Guide to Surviving „Mission Impossible" Projects*, Prentice Hall, 1997
[406] S. Zahran: *Software Process Improvement: Practical Guidelines for Business Success*, Addison Wesley, 1997
[407] U. Zdun: *Dynamically generating web application fragments from page templates*, In: Proceedings of the Symposium of Applied Computing, Madrid, 2002
[408] V. A. Zeithaml, M. J. Bitner: *Service Marketing*, McGraw Hill, 1996
[409] M. V. Zelkowitz et al.: *Principles of Software Engineering and Design*, Prentice Hall, 1979

[410] J. Zimmermann, G. Benecken: *Verteilte Komponenten und Datenbankanbindung*, Addison Wesley, 2000
[411] H. Zuse: *Software Complexity Measure and Methods*, de Gruyter, 1991
[412] H. Zuse: *A Framework for Software Measurement*, de Gruyter, 1991

Sachverzeichnis