1 Wesen des Software-Produktmanagements

Ziele und Aufbau

Im Wegweiser durch dieses Buch wurde bereits angesprochen, dass es große Unterschiede zwischen der Erstentwicklung eines Produkts einerseits und seiner Erhaltung und Weiterentwicklung andererseits gibt. Diese Thematik wird im vorliegenden Kapitel vertieft. Dazu werden

- verschiedene, einander ergänzende Blickwinkel des Software-Produktmanagements dargestellt,
- der Lebenszyklus und die Evolutionsgesetze eines Softwareprodukts von der Erstentwicklung bis zur Außerbetriebnahme näher beleuchtet und
- die Haupttätigkeiten der Systemerhaltung identifiziert.

Am Ende des Kapitels hat der Leser ein vertieftes Verständnis für die Problematik des Software-Produktmanagements und damit auch für die Notwendigkeit geregelter Prozesse, wie sie in den Folgekapiteln vorgestellt werden.

1.1 Das Besondere am Software-Produktmanagement

Der Sinn und Zweck der Informationstechnologie kann nicht darin bestehen, immer wieder neue Anwendungssysteme zu entwickeln, bloß weil die technischen Rahmenbedingungen sich alle paar Jahre verändern. Die wirklich geschäftskritischen Anwendungen sind viel zu komplex und zu eng mit der Organisation und den Geschäftsprozessen verwoben, als dass sie sich so einfach gegen ein neues Modell austauschen ließen. Die Funktionalität, die in solchen Systemen steckt, ist das Ergebnis einer mehrjährigen Evolution und lässt sich nicht auf Anhieb wiederherstellen. Es dauert wieder Jahre, bis das neue System den gleichen fachlichen Reifegrad erreicht wie das alte.

Deshalb ist es völlig rational, so lange wie möglich an bestehenden Systemen festzuhalten, auch wenn sie technologisch überholt sind. Es kommt weniger auf die technologische Verpackung als auf den fachlichen Inhalt an. Solange dieser noch stimmt und die Systemerhaltungskosten noch vertretbar sind, besteht kein Anlass, ein solches System abzulösen. Bestehende Softwaresysteme verlieren keinen Wert. Im Gegenteil, ihr Wert nimmt mit der Zeit zu, denn sie werden ständig angepasst und ergänzt. Ablösereif sind sie erst dann, wenn sie sich nicht mehr oder nur teuer fortentwickeln lassen.

Die Erhaltung und Weiterentwicklung solcher Legacy-Systeme ist das Ziel des Produktmanagements. Hier geht es um die Aufrechterhaltung und Verbesserung einer Dienstleistung. Produktmanagement und Projektmanagement sind zwar keine Gegensätze, aber sie unterscheiden sich gravierend in einigen wichtigen Grundzügen, ebenso wie Produkte sich von Projekten unterscheiden. Das Produktmanagement muss vom bestehenden Produkt, seiner Architektur und seiner Umgebung ausgehen. Ein Methoden- oder Technologiewechsel ist nur mit großem Aufwand möglich. Dem Produktmanager[1] sind in vielerlei Hinsicht die Hände gebunden. Er kann nur in kleinen Schritten vorgehen, ohne die Kontinuität des Produkts zu gefährden.

Softwareprodukte sind Gebrauchsgüter, die von ihren Benutzern benötigt werden, um bestimmte Bedürfnisse zu erfüllen. Software-Entwicklungsprojekte erfordern hingegen Aktivitäten, die dazu führen, dass ein Produkt zur Erfüllung der Benutzeranforderungen geschaffen wird. Den Benutzer interessiert letztendlich nur das Produkt, denn nur das Produkt kann seine Wünsche erfüllen. Aus der Sicht der Benutzer sind Entwicklungsprojekte nur ein Mittel zum Zweck. Sie sind unberechenbar, geraten leicht aus der Bahn, verursachen unvorhergesehene Kosten, dauern eine unbestimmte Zeit und führen – falls sie überhaupt jemals zum Abschluss kommen – allzu oft zu einem Produkt, das der Anwender gar nicht haben wollte. Es ist allemal sicherer und in den meisten Fällen günstiger, neue Bedürfnisse durch bestehende Systeme abzudecken. Statt den großen Sprung nach vorne zu wagen, geht der Vorsichtige in kleinen Schritten vor und führt nicht ein großes, sondern viele kleine Projekte durch. Dies ist die Strategie des Produktmanagements, die auf die Evolution bestehender Systeme zielt [Lehm80].

1.1.1 Produktmanagement als Lebenszyklusmanagement

Projekte sind einmalige, zeitlich und räumlich begrenzte Vorhaben, um ein vorgegebenes Ziel zu erreichen. Dieses Ziel ist im Falle eines Entwicklungsprojekts die Bereitstellung eines Anwendungssystems mit bestimmten, genau spezifizierten Funktionen und nicht-funktionalen Eigenschaften. So gesehen ist das Projektma-

---

1. Der Produktmanager ist Vorgesetzter der an der Erhaltung und Weiterentwicklung des Produkts Mitwirkenden (Produktmannschaft). Die Aufbauorganisation wird in Kapitel 4 näher vorgestellt.

nagement auf einen bestimmten Zeitabschnitt fixiert. Der Projektmanager wählt einen geeigneten Weg zum Ziel, zum Beispiel über den Rational Unified Process, plant die Wegbegehung, führt seine Mannschaft über den Weg zum Ziel und feiert, wenn sie ankommt. Produktmanagement ist anders, denn der Weg ist das Ziel. Man beginnt mit einem Zustand und wandelt ihn immer wieder in einen anderen, angeblich besseren um (Abb. 1–1). Einen vordefinierbaren Endzustand gibt es nicht, und auch wenn es ihn gäbe, würde man nicht wissen, wann man ihn erreicht hat. Die Wahrscheinlichkeit, durch die Lösung eines Problems neue zu schaffen, ist sehr hoch. Daher müssen Änderungen mit allen Konsequenzen durchdacht werden.

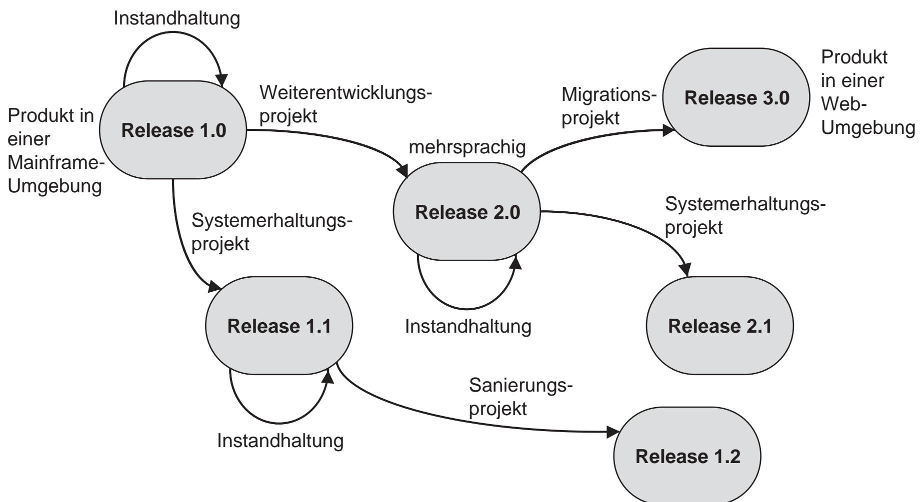

Abb. 1–1 *Produktzustandsübergänge*

Im Gegensatz zum Projektmanagement ist Produktmanagement nicht zeitlich begrenzt. Es beginnt mit der Geburt eines Produkts und endet, wenn dieses für tot erklärt wird. Somit ist Produktmanagement gleich Lebenszyklusmanagement [Zveg87].

1.1.2 Produktmanagement als Projekt der Projekte

In diesem unvorhersehbaren Zeitabschnitt zwischen Geburt und Tod eines Softwareprodukts finden viele Projekte statt. Jeder Zustandswechsel des Produkts ist das Resultat eines Projekts. Je nachdem wie viele Zustandsübergänge man zulässt, gibt es mehr oder weniger viele Projekte. Sie können hintereinander und auch nebeneinander ablaufen. Das Produktmanagement plant, organisiert und steuert sie. So gesehen ist Produktmanagement gleich einem Projekt der Projekte. Es setzt die Ziele der untergeordneten Projekte, stößt sie an und kontrolliert, ob sie am Ziel ankommen [Grad87].

1.1.3 Produktmanagement als Management der Softwareerhaltung und -weiterentwicklung

Laut einer Entscheidung des amerikanischen Rechnungshofes aus dem Jahre 1983 gelten alle Arbeiten an einem Softwareprodukt nach dem ersten Einsatz des Produkts als Software Maintenance [GAO81]. Das National Bureau of Standards verfeinerte diese Definition in einer Special Publication aus dem Jahre 1985. Dort heißt es: »*Software Maintenance is the performance of those activities required to keep a software system operational and responsive after it has been accepted and placed into production. It is the set of activities which result in changes to the originally accepted – baseline – product. These changes consist of modifications created by correcting, inserting, deleting, extending and enhancing the baseline system.*« Maintenance darf hier nicht mit dem deutschen Begriff »Wartung« gleichgesetzt werden, sondern bedeutet mehr. Maintenance umfasst alle Aktivitäten, die erforderlich sind, um ein System im Betrieb zu halten. Das NBS-Dokument 500-106 geht weiter und beschreibt, wie ein Software- Maintenance-Betrieb aufzubauen ist und wie er zu funktionieren hat [NBS85]. Die wichtigsten Maintenance-Prozesse sind Änderungsmanagement, Konfigurationsmanagement, Testmanagement und Release-Management. Sie beziehen sich alle auf das Softwareprodukt, und der Produktmanager ist verantwortlich für sie. Er hat dafür zu sorgen, dass neue Versionen der Software in geregelten Intervallen herauskommen, ohne die laufenden Benutzertätigkeiten zu stören.

Außer für die Erhaltung der Software ist der Produktmanager auch für deren Weiterentwicklung verantwortlich. Softwareprodukte sind selten wirklich fertig, wenn sie freigegeben werden (Abb. 1–2). Das wissen die Anwender, und das wissen die Entwickler erst recht. Es bleiben je nach Größe und Komplexität des Produkts viele Anforderungen offen. Die bereits bekannten werden durch weitere, unvorhergesehene erweitert, die durch die Nutzung des Produkts entstehen. Deshalb wird neben der Erhaltung der Software (Maintenance) auch die ständige Weiterentwicklung (Evolution[2]) betrieben [vHor79]. Der Produktmanager hat dafür zu sorgen, dass diese beiden Prozesse sich nicht gegenseitig behindern. Sie haben im Gleichschritt nebeneinander herzulaufen.

So gesehen ist Produktmanagement gleich dem Management der Softwareerhaltung und -weiterentwicklung. Es ist die Aufgabe des Produktmanagers, die Erhaltungs- und Weiterentwicklungsaktivitäten zu planen, zu überwachen, zu

---

2. Der Begriff »Evolution« ist im Zusammenhang mit Softwareprodukten leider mit unterschiedlichen Bedeutungen belegt:
   1. Weiterentwicklung (zum Beispiel im Titel des *Journal of Maintenance and Evolution*)
   2. Jene Phase im Produktlebenszyklus, in der noch weiterentwickelt wird (zum Beispiel bei Rajlich und Bennett).
   3. Der gesamte Produktlebenszyklus (zum Beispiel bei Lehman und Belady)

   In diesem Buch wird der Begriff im ersten Sinne verwendet. Auf Ausnahmen wird vor Ort hingewiesen.

messen und zu verbessern. Dafür ist ein ständiger, übergeordneter Prozess mit vielen vorübergehenden, untergeordneten Prozessen erforderlich.

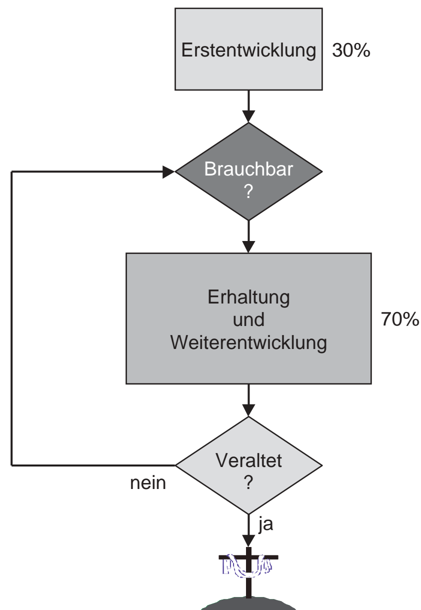

Abb. 1–2 *Verhältnis der Softwareerhaltung und Weiterentwicklung zur Erstentwicklung*

1.1.4 Produktmanagement als Konfigurationsmanagement

Produktmanagement hat etwas mit Produktverwaltung zu tun. Das Produkt selbst – einschließlich der ausführbaren Objekte, des Quellcodes, der Kontrollprozeduren, der Steuerungsdaten und der dazugehörigen Dokumente – muss in einem konsistenten und kompletten Zustand erhalten werden [Bers84]. Man darf nicht die Masken der letzten Version mit den Programmen der neuen Version vermischen, falls Masken geändert wurden. Die Versionen müssen getrennt geführt und unabhängig voneinander verwaltet werden. Dies gilt auch für die Trennung von Entwicklungs-, Test- und Produktionszuständen. Außerdem müssen die einzelnen Konfigurationselemente (Abb. 1–3) gut gesichert werden, denn es darf nichts verloren gehen. So gesehen ist Konfigurationsmanagement ein ganz wesentlicher Aspekt des Produktmanagements. Der Produktmanager ist für die Konsistenz, Vollständigkeit und Sicherheit der Produktversionen verantwortlich, auch dann, wenn er diese Tätigkeit an andere delegiert.

Abb. 1–3 *Objekte des Konfigurationsmanagements*

1.1.5 Produktmanagement als Produktvertrieb

Ein Softwareprodukt lebt nur so lange, wie es von mindestens einem Anwender benutzt wird. In dem Moment, wo es keiner mehr einsetzt, ist es am Ende seines Lebenszyklus angekommen. Der Produktmanager müsste Interesse daran haben, diesen Zeitpunkt möglichst lange hinauszuschieben. Das erreicht er,

1. indem er die Anwender vom Wert seines Produkts überzeugt. Sie müssen sehen, dass sie davon einen nachweisbaren, betriebswirtschaftlichen Vorteil haben;
2. indem er alle Probleme, die als Folge der Produktnutzung entstehen, möglichst schnell beseitigt;
3. indem er dafür sorgt, dass das Produkt alle Wandlungen in der Umgebung des Benutzers nachvollzieht;
4. indem er ihre neuen Bedürfnisse möglichst schnell und preiswert befriedigt und
5. indem er gute persönliche Beziehungen zu den Anwendern pflegt.

Viele dieser Tätigkeiten lassen sich an andere delegieren, aber nicht die erste und auch nicht die letzte. Der Produktmanager muss sich im Gegensatz zum Projektmanager, der sein festes Ziel hat, ständig um die Benutzer des Produkts kümmern, damit es noch Ziele für weitere Projekte gibt. So gesehen ist der Produktmanager der Vertreter des Produkts gegenüber den Anwendern. Seine Person steht für die Kontinuität des Produkts und der damit verbundenen Dienstleistung.

In einem Artikel in der Fachzeitschrift »*Software Testing und Quality Engineering*« werden die Eigenschaften eines idealen Produktmanagers so beschrieben:

- Er kennt alle Anwender des Produkts.
- Er ist nicht mit einem einzigen Anwender zu eng verbunden.
- Er hat reichlich Erfahrung auf dem Anwendungsgebiet des Produkts.
- Er kennt sich mit der Dienstleistung des Produkts aus.
- Er kann in Konfliktfällen vermitteln.
- Er kann sowohl mit den Anwendern als auch mit den Lieferanten gut umgehen.
- Er trägt die volle Verantwortung für das Produkt.
- Er verteidigt die Produktmannschaft gegen Kritik von außen.
- Er behandelt die Produktmannschaft als kompetenten Partner.
- Er ist bereit, unpopuläre Entscheidungen zugunsten des Produkts zu treffen.
- Er setzt erreichbare technische Ziele.
- Er kommuniziert regelmäßig mit der Produktmannschaft.
- Er überschätzt sein Fachwissen nicht.
- Er verfolgt die Entwicklungen auf dem Markt.
- Er mischt sich nicht in technische Angelegenheiten ein.
- Er ist bereit, unpassende Anforderungen der Anwender abzulehnen.

Die Autorin betont, wie wichtig es sei, eine einzige Person als Repräsentanten des Produkts zu haben: »*The Product Manager is the one person officially responsible for a product – its delivery and its continuity. This person is both the champion of the product stakeholders and the representative of the product developers. It is the product manager who keeps the vision of the product alive and who unites the interests of the users with those of the owners in pursuing a common goal.*« [Risi03]

1.1.6 Produktmanagement als Software Engineering im großen Stil

Software-Produktmanagement ist ohne Software Engineering nicht denkbar. Es steht und fällt mit wohl definierten Prozessen für das Änderungsmanagement, das Mängelmanagement, die Konfigurationsverwaltung und den Regressionstest sowie mit Werkzeugen, die diese Prozesse unterstützen – Analysatoren, Monitore, Repositories, Überprüfungswerkzeuge und viele andere mehr. Mit gutem Willen, Entschlossenheit und Fähigkeiten allein lässt sich kein Softwareprodukt erhalten. Dies ist auch einer der Hauptbeweggründe für das Outsourcing und für den Einsatz von Standardprodukten. Kleine und mittlere Anwenderbetriebe können sich richtiges Software Engineering gar nicht leisten. Die dafür erforderliche Infrastruktur ist ihnen zu teuer. Deshalb können sie zwar Anwendungssysteme schaffen, aber nicht erhalten. Die Erhaltungskosten steigen ihnen über den Kopf. Dies trifft übrigens auch für viele Großbetriebe zu, die erst spät erkannt haben,

dass sie die »*Total Cost of Ownership*« und die Kosten des Produktmanagements total unterschätzt haben.

1.2 Die vier P im Software Engineering

Vier Worte mit P als erstem Buchstaben spielen eine zentrale Rolle im Software Engineering (Abb. 1–4). Es sind die Begriffe:

- Prototyp
- Produkt
- Projekt
- Prozess

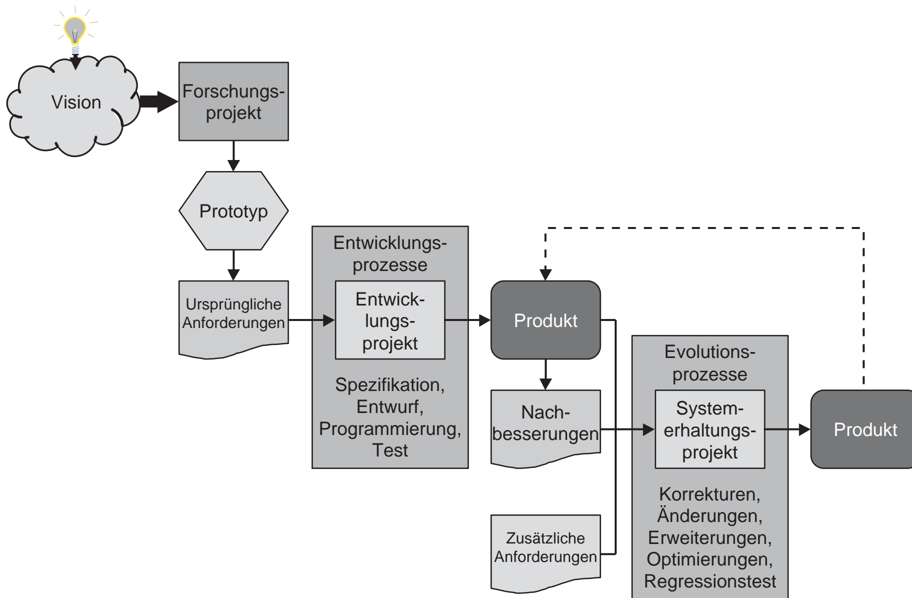

Abb. 1–4 *Prototypen, Projekte, Prozesse und Produkte*

1.2.1 Softwareprototypen

Ein Prototyp ist laut ANSI/IEEE 610 »*a preliminary type, form, or instance of a system that serves as a model for later stages or for the final, complete version of the system*« [IEEE90]. Man baut ihn, um die Anforderungen des potenziellen Anwenders auszuloten und zu analysieren. Ein Prototyp dient also nur einem vorübergehenden Zweck, nämlich herauszufinden, was der Anwender wirklich haben will. Er ist daher von vorübergehendem, begrenztem Nutzen und nicht für die Produktion vorgesehen.

Andererseits können die Kosten eines Prototyps sehr hoch werden. Im Flugzeugbau oder im Hausbau ist es möglich, ein Miniaturexemplar zu erstellen, das zu Testzwecken verwendet werden kann. Dieses Modell kann dazu dienen, Daten über die Performance und die Standfestigkeit des späteren Produkts zu sammeln. Hinzu kommt, dass die Blaupausen für das geplante Produkt von den Anwendern des Produkts weitgehend verstanden werden. Man erkennt die Raumeinteilung, die Fassade des Hauses, die Fensterplatzierung und andere Eigenschaften. Im Zusammenhang mit dem Modell hat der potenzielle Anwender ein recht gutes, wenn auch nicht vollständiges Bild seines geplanten Hauses. Deshalb kann das endgültige Produkt nicht allzu weit von seiner Vorstellung abweichen, und es dürften seine späteren Änderungs- und Erweiterungswünsche im Rahmen bleiben.

Von diesem Stand der Modellierung sind wir in der Softwarebranche leider noch weit entfernt. Wir haben zwar verschiedene Modellierungssprachen, zum Beispiel die Unified Modeling Language (UML), aber die wenigsten Anwender sind in der Lage, sie zu verstehen – auch wenn sie vollständig wären, was selten der Fall ist. Um die Vorstellungen des Anwenders zu entdecken oder überhaupt erst zu erwecken, bleibt uns nur übrig, einen Prototyp zu bauen. Da Software ein immaterielles Produkt ist, zählt nur seine Leistung. Wir müssen es also in allen Variationen simulieren. Nur dann ist der Anwender in der Lage zu beurteilen, ob das Produkt sein Problem löst, denn er sucht nach einer fachlichen Lösung, und die technische Umsetzung ist nur ein Mittel dazu.

Es darf niemanden wundern, wenn der Softwareprototypbau so teuer wird. Er ist wahrlich eine Expedition ins Ungewisse, und hier sind agile Vorgehensmodelle und Extreme Programming durchaus angebracht. Zum Schluss kommt man zu lauffähigen Programmen, die der Benutzer bedienen kann und die die geforderte Leistung in einer halbwegs akzeptablen Weise erbringen. Vielleicht sind sie in Visual Basic oder JavaScript implementiert. Sie sind jedoch keinesfalls als endgültiges Produkt gedacht, und dementsprechend werden sie auch nicht getestet oder getunt. Es reicht, wenn der Benutzer damit arbeiten kann.

Es ist schon teuer genug, einen Prototyp zu einzelnen Prozessen zu bauen. Es wird besonders teuer, wenn ganze Anwendungssysteme simuliert werden. Denn der Umfang des Prototyps liegt im Gegensatz zum Flugzeug- oder Hausbau nahe am Umfang des endgültigen Produkts. Was den Prototyp noch vom Produkt unterscheidet, ist weniger die Funktionalität als die Qualität. Der Prototyp wird billig gebaut, sozusagen mit Pressholzplatten statt Ziegeln.

In Anbetracht der Kosten des Softwareprototypbaus haben die meisten Anwender bisher darauf verzichtet. Sie haben es vorgezogen, das Produkt gleich zu bauen. Leider sind diese vermeintlichen Produkte in Wirklichkeit nur Prototypen gewesen. Sie wurden wie Prototypen aus Pressholzplatten konstruiert und dienten hauptsächlich dazu, die wahren Anforderungen der Anwender zu entdecken. Die Folge davon waren noch höhere Nachbesserungskosten. Es wurde so lange an diesen angeblichen Produkten herumgebastelt, bis sie endlich einen Stand erreichten, den man tatsächlich als Produkt bezeichnen konnte. Dieser Pro-

zess dauerte oft Jahre. Es wäre besser gewesen, die erste Version gleich als Prototyp zu deklarieren und als solchen zu verkaufen. Bisher sind weder die Anwender noch die Entwickler bereit gewesen, die wahren Kosten eines Softwareprodukts zu akzeptieren. Stattdessen hat man sich mit der produktiven Nutzung von Prototypen abgefunden, ist zunächst in das Pressholzhaus eingezogen und hat während des Wohnens die Pressholzwände Stück für Stück durch Steinmauern ersetzt. Dieser Nachbesserungsprozess hat auch einen Namen bekommen: Er heißt iterative Entwicklung oder Softwareevolution [KeSl99].

1.2.2 Softwareprodukte

ANSI/IEEE 610.12 definiert ein Softwareprodukt als »*the complete set of computer programs, procedures and associated documentation and data designated for delivery to a user*«[IEEE90].

Ein Softwareprodukt besteht also aus:

- Programmen
- Prozeduren
- Dokumenten
- Daten

Hinzu kommen noch die Testfälle.

Die Programme existieren als Quellcode, Objektcode oder als Bytecode – ein Zwischenstadium zwischen Quellcode und Objektcode. Es ist eine Frage der Vereinbarung, in welcher Form der Code ausgeliefert wird. Möglicherweise wird er auch gar nicht ausgeliefert, sondern nur auf einem Anwendungsserver bereitgestellt, so wie das bei Webservices der Fall ist. Dadurch bleibt der Code im Besitz des Produktverantwortlichen und – was für die Erhaltung und Weiterentwicklung sehr günstig ist – nur in einer Kopie auf einer Anlage.

Die Prozeduren sind die Ablaufsteuerungskomponenten. In der Hostwelt sind das die Kontrollprozeduren, in der Unix- und Linux-Welt die Skripte oder Workflow Procedures. Die Ablaufsteuerungskomponenten sind parametrisiert und in der Regel anwenderspezifisch. Sie werden daher im Gegensatz zum Programmcode lokal gepflegt und weiterentwickelt. Es könnte sogar die Aufgabe des Anwenders sein, dies zu tun.

Die Dokumente können technischer, fachlicher oder nutzungsbezogener Art sein. Eine Bedienungsanleitung oder ein Benutzerhandbuch bezieht sich auf die Nutzung des Produkts. Sie müssen auf jeden Fall als Buch, CD oder Online-Webseiten bereitgestellt werden. Die fachliche Dokumentation ist das Fachkonzept eines Produkts. Sie sollte den Anwendern in irgendeiner Form zugänglich gemacht werden, damit sie das spezifizierte Verhalten des Produkts mit dem tatsächlichen vergleichen können. Sie ist also in eine Form zu verpacken, die Benutzer verstehen. Schließlich gibt es die technische Dokumentation, also den Entwurf der Produktarchitektur, der einzelnen Programme und der Schnittstellen.

Sofern man nicht in einer Open-Source-Umgebung operiert, muss sie nicht zwingend an die Anwender ausgeliefert werden.

Die Daten, die zum Produkt gehören, sind die Readonly-Tabellen, Steuerungsdateien und Datenbanken. Sie enthalten die Parameter, Schlüssel und Texte, die Programme benötigen, um korrekt zu laufen. Diese Daten können lokal oder zentral geführt werden. Für die Systemerhaltung und Weiterentwicklung ist es einfacher, sie zentral zu führen. Wenn die Programme aber verteilt sind, kann dies zu Performance-Engpässen führen.

Ein weiterer Bestandteil eines Softwareprodukts sind die Testfälle. Sie haben mittlerweile die gleiche Bedeutung wie die Prozeduren und Dokumente. Sie können an die Anwender ausgeliefert werden. Es dürfte kein Softwareprodukt ohne Testware geben.

Ein Softwareprodukt unterscheidet sich von einem Softwareprototyp dadurch, dass es eine ganz andere Qualität und damit einen anderen Reifegrad hat. An einen Prototyp kann man keinerlei Ansprüche bezüglich Zuverlässigkeit, Benutzbarkeit, Performance, Portierbarkeit, Wiederverwendung oder Wartbarkeit stellen. Das sind alles Eigenschaften eines Produkts. Mit einem Prototyp kann man zufrieden sein, wenn er die Funktionalität bei beschränkter Nutzung und häufigen Abbrüchen halbwegs erfüllt. Denn gerade das ist es, was einen Prototyp von einem Produkt unterscheidet. Ein Produkt muss auch die nichtfunktionalen und insbesondere die qualitativen Anforderungen erfüllen. Ein Prototyp braucht das nicht zu tun, sondern erfüllt lediglich die funktionalen Anforderungen, und auch diese nur zum Teil.

Anwender und leider auch viele Softwareentwickler neigen dazu, den Kostenunterschied zwischen Prototypen und Produkten zu unterschätzen. Er beträgt laut Frederick Brooks, ehemaliger Leiter des IBM OS-Projekts, mindestens 1 zu 3 [Broo75].

1.2.3 Softwareprojekte

Es ist interessant, dass im Software Engineering Glossary der ANSI/IEEE der Begriff Projekt nicht vorkommt. Das zeigt, wie schwierig es ist, mit diesem Begriff umzugehen. Im Wörterbuch wird das Wort »Projekt« in Verbindung mit Projektierung gebraucht. Im Englischen heißt es »*a proposal of something to be done – a schema, a plan for an undertaking*« [Webs56]. Im Deutschen spricht man von einem »Vorhaben«. Also ist ein Projekt ein Vorhaben, um etwas zu schaffen, das es bisher nicht gab, oder etwas zu verändern, das es bisher gab. Auf jeden Fall geht es darum, einen Zustandswechsel herbeizuführen.

Bezogen auf Software ist ein Projekt ein Vorhaben, um eine Software zu schaffen, die bisher nicht existierte, oder eine Software zu verändern, die bereits existierte. »Verändern« heißt, von einem Zustand in einen anderen zu verwandeln. Also ist ein Softwareprojekt ein Softwarezustandsübergang.

Das Objekt dieses Übergangs könnte ein Softwareprototyp oder ein Softwareprodukt sein. Falls das Produkt oder der Prototyp noch nicht existiert, handelt es sich um ein Entwicklungsprojekt. Falls es aber bereits existiert und in einen anderen Zustand versetzt wird, handelt es sich um ein Weiterentwicklungs- oder Systemerhaltungsprojekt. Ein Weiterentwicklungsprojekt ist es dann, wenn die Funktionalität, also die Größe der Software zunimmt. Ein Systemerhaltungsprojekt ist es dann, wenn die Software sich nur ändert, aber in ihrer Funktionalität nicht zunimmt.

Es gibt also drei Projekttypen:

- Entwicklungsprojekte,
- Weiterentwicklungsprojekte und
- Systemerhaltungsprojekte,

wobei sich jeder davon auf einen Prototyp oder ein Produkt beziehen kann.

Ein Projekt ist ein Vorhaben, und Vorhaben haben einen Anfang und ein Ende. Also sind Projekte zeitlich begrenzt. Damit sie nicht ewig laufen, wird ein Endtermin gesetzt. Jedes Vorhaben und damit jedes Projekt benutzt Ressourcen. Ein Softwareprojekt benutzt Hardware-, Software-, Raum- und Personalressourcen. Ressourcen kosten Geld, egal ob man sie besitzt oder ob man sie nur ausleiht. Daher verursacht ein Projekt Kosten. Damit sie nicht ins Endlose steigen, wird zu Beginn eine Kostengrenze gesetzt. Es dürfen nur bis zu dieser Grenze Ressourcen benutzt und damit Kosten verursacht werden. Wir werden hier die Ressourcennutzung als den Raum eines Projekts bezeichnen.

Deshalb ist jedes Softwareprojekt zeitlich und ressourcenbezogen begrenzt.

Dieses Buch beschränkt sich auf bestehende Softwareprodukte. Es kommen darin nur Projekttypen vor, die sich auf ein bestehendes Produkt beziehen. Dazu gehören:

- Weiterentwicklungsprojekte
- Systemerhaltungsprojekte

Systemerhaltungsprojekte umfassen die reinen Korrektur- und Änderungsarbeiten. Weiterentwicklungsprojekte umfassen außer Erweiterungen auch Sanierungs- und Optimierungsarbeiten, Migrationen und Integrationsarbeiten [BeSa87].

1.2.4 Softwareprozesse

Nach ANSI/IEEE 610.12 ist ein Prozess »*a sequence of steps performed to achieve a given purpose; for example, the software development process*« [IEEE90].

In Anlehnung an diese Definition ist ein Prozess eine Art Fertigungsstraße. Gefertigt werden entweder Softwareprodukte oder Prototypen. Die Fertigungsschritte sind vorgegeben. Einige davon sind vollautomatisiert, andere nur teilautomatisiert, wieder andere manuell. Die Schritte haben eine konstante oder variable Reihenfolge. Im ersten Fall haben wir es mit einem statischen Prozess, im

zweiten Fall mit einem dynamischen Prozess zu tun. Dynamische Prozesse lassen sich der jeweiligen Situation anpassen. Das alte Wasserfallmodell der Softwareentwicklung ist ein Beispiel für einen statischen Prozess. Der Rational Unified Process ist ein Beispiel für einen dynamischen Prozess, denn der Anwender muss ihn konfigurieren.

Prozesse stehen im Mittelpunkt des Software Engineering, denn ohne Prozesse ist Software Engineering kein Engineering. Der Begriff Engineering impliziert, dass Produkte oder Prototypen nach genormten Prozessen hergestellt oder verarbeitet werden. Allerdings kommen im Software Engineering viele Prozessarten vor – darunter Entwicklungsprozesse, Testprozesse und Systemerhaltungsprozesse. Ein Entwicklungsprozess regelt, wie ein Softwareprototyp oder ein Softwareprodukt zu entwickeln ist. Ein Systemerhaltungsprozess regelt, wie ein existierendes Produkt zu erhalten und weiterzuentwickeln ist. ANSI/IEEE 1219 aus dem Jahre 1998 gibt vor, welche Schritte in welcher Reihenfolge in einem Softwareerhaltungsprojekt auszuführen sind [IEEE98].

Das bringt uns zum Verhältnis zwischen Prozessen und Projekten. Es ist eins zu viele. Durch einen Prozess laufen viele Projekte. Der gemeinsame Prozess ist eben das, was die Projekte verbindet. In der Welt der Softwareerhaltung und -weiterentwicklung ist ein Projekt gleich einer Auslieferung. Ein Projekt wird durchgeführt, um das Produkt von einem Istzustand in einen neuen Sollzustand zu versetzen. Die Dauer des Projekts wird vom Lieferintervall bestimmt. Sie variiert von einem Tag bis zu einem Jahr. Deshalb gibt es in der Systemerhaltung so viele Projekte. Im Grunde genommen könnte man jede Veränderung des Codes als Projekt bezeichnen. Dies würde jedoch zu einer unbeherrschbaren Anzahl kleiner Projekte führen, die mit Bestimmtheit nicht nach einem vorgeschriebenen Prozess ablaufen.

Um standardisierten, ingenieurmäßigen Systemerhaltungsprozessen wie denen der ANSI/IEEE gerecht zu werden, sind wir angehalten, die vielen einzelnen Mängelmeldungen und Änderungsanträge zu einem Systemerhaltungsprojekt zusammenzubündeln, um sie dann alle auf einmal nach den vorgeschriebenen Prozessen zu spezifizieren, zu implementieren und zu verifizieren. Erst dann kann von Software Engineering die Rede sein [Schn87].

1.3 Maintenance und Evolution von Softwaresystemen

1.3.1 Software Maintenance als Systemerhaltung

Ein Großteil der Verwirrung, die in der Softwaretechnologie herrscht, ist auf die unterschiedliche Nutzung der Begriffe zurückzuführen. Gerade was »Software Maintenance« anbetrifft, gibt es viele Interpretationen, und sie sind von Kulturkreis zu Kulturkreis unterschiedlich. Im Deutschen wird das Wort Wartung mit Reparatur assoziiert. Man stellt einen Zustand wieder her. Im Englischen wird

Maintenance im Zusammenhang mit »Erhaltung« gebraucht. »*To maintain order*« heißt die Ordnung erhalten. Im Deutschen würde keiner davon sprechen, dass die Ordnung gewartet werden müsse. Wenn Deutsche also von Software- oder Systemwartung reden, dann haben sie Reparaturarbeiten am System vor Augen. Das hat einen negativen Beigeschmack, denn die nächste Frage liegt nahe: Warum ist das System reparaturbedürftig? Was hat man falsch gemacht? Und prompt sind wir bei der Schuldzuweisung.

Wenn Engländer und Amerikaner von »Maintenance« sprechen, dann denken sie zunächst nicht an Reparatur, obwohl die auch dazugehört, sondern an Erhaltung. Für sie ist »*System Maintenance*« Systemerhaltung und alles, was dazugehört: Anpassung, Veränderung, Verbesserung und natürlich auch Reparatur. So gesehen ist der deutsche Begriff »Softwarewartung« eine falsche Übersetzung aus dem Englischen, weil er dem Begriff »*to maintain*« nicht gerecht wird. Er sollte »*Softwaresystemerhaltung*« heißen, und so wird er auch in diesem Buch verwendet.

Für das »IEEE Standard Glossary of Software Engineering Terminology« ist Software Maintenance »*the modification of a software product after delivery to correct faults, to improve product performance and other attributes of the product, or to adapt the product to a changing environment*« [IEEE90] und [LiSw80].

Es gibt also drei Haupttätigkeiten, für die auch eigene Begriffe geschaffen wurden (Abb. 1–5):

- »to correct faults« (*corrective maintenance*)
- »to improve performance and other attributes« (*perfective maintenance*)
- »to adapt the product to a changing environment« (*adaptive maintenance*)

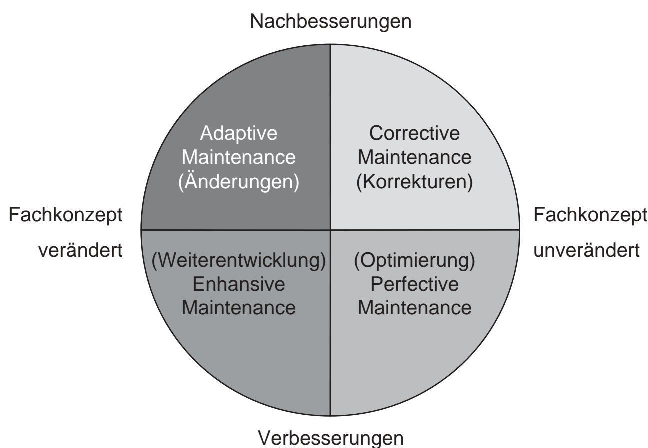

Abb. 1–5 *Hauptaktivitäten der Softwareerhaltung und Weiterentwicklung*

Dies entspricht der von Parikh zitierten Dreieinigkeitslehre, wonach Brahma das Leben schafft, Vishnu das Leben pflegt und Shiva das Leben auflöst. Softwaresysteme veralten in dem Maße, wie sie geändert werden. Sie werden sozusagen zu Tode gepflegt [Brow80]. Je öfter sie geflickt und gestreckt werden, desto brüchiger werden sie. Irgendwann erreichen sie einen Punkt, wo sie sich nicht mehr flicken und strecken lassen. Dies ist der Punkt, an dem sie abgelöst werden müssen. Nur Systeme, die keiner oder nur wenigen Anpassungen unterworfen sind, können lange leben. Je mehr Anpassungen ein System durchmacht, also je höher seine Änderungsrate ist, desto kürzer wird seine Lebensdauer sein, außer es wird im Sinne der hinduistischen Lehre der Wiedergeburt regeneriert oder einem Reengineering unterworfen (Abb. 1–6).

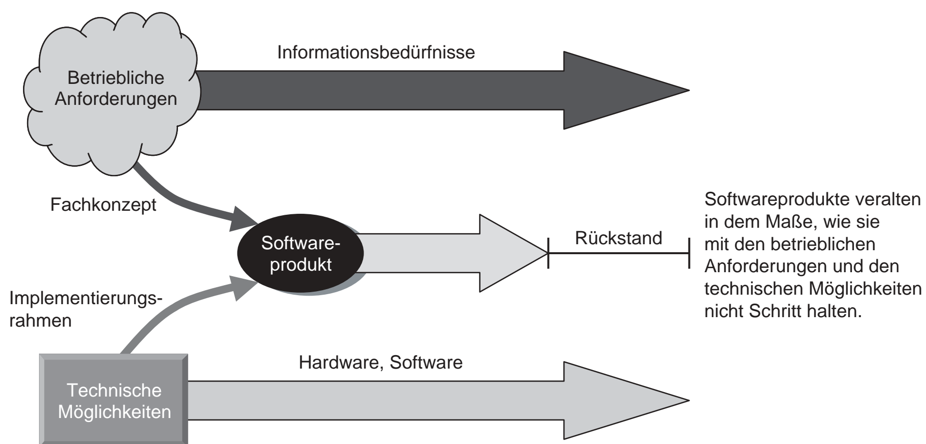

Abb. 1–6 *Der Software-Alterungsprozess*

Corrective Maintenance

Corrective Maintenance (korrektive Systemerhaltung) ist eindeutig Reparatur. Aber Reparatur wovon? Es gibt kleine und große Mängel. Wenn das Kühlsystem in einem Auto nicht funktioniert, überhitzt sich der Antrieb, und es kommt zu einem Motorschaden. Das Auto ist nicht mehr zu benutzen. Dies ist natürlich ein schwerer Mangel. Wenn die Benzinanzeige nicht richtig funktioniert, ist der Fahrer verwirrt und weiß nicht, wann er tanken soll. Dies ist zwar ärgerlich, aber es verhindert die Nutzung des Autos nicht, denn man kann trotzdem damit fahren. Dies könnte man als mittelschweren Mangel einstufen. Wenn der Lack einen Kratzer hat, vermindert dies zwar die Schönheit des Autos und vielleicht auch noch den Wiederverkaufswert, aber dies beeinträchtigt die Nutzung in keiner Weise. Dies wäre als leichter Mangel einzustufen.

In einem Softwaresystem kann ein Adressierungsfehler dazu führen, dass wichtige Daten überschrieben werden. Auch wenn das System weiter läuft, sind die folgenden Ergebnisse alle verfälscht. Das System ist in diesem Zustand nicht

benutzbar, also liegt ein schwerer Mangel vor. Wenn eine Summe in einer Liste falsch gebildet wird, ist der Benutzer zunächst verwirrt und weiß nicht, was er glauben soll. Wenn er jedoch weiß, dass diese bestimmte Summe falsch ist, kann er sie ignorieren und die restlichen Ergebnisse heranziehen. Das verhindert nicht die Nutzung des Systems. Dieser Mangel ist deshalb als mittelschwer einzustufen. Wenn ein Text auf der Oberfläche falsch geschrieben ist, ist dies zwar ärgerlich, aber dieser Schreibfehler beeinträchtigt in keiner Weise die Nutzung des Systems. Dies ist eindeutig als leichter Mangel zu betrachten.

Wir sehen daran, was für eine Bandbreite allein die korrektive Systemerhaltung abdeckt. Es ist ein großer Unterschied, ob jemand mit überhitztem Motor auf der Autobahn steht oder ob er sich auf dem Parkplatz über einen Lackkratzer ärgert. Es ist ein ebenso großer Unterschied, ob ein IT-System ausfällt und hundert Benutzer nicht arbeiten können oder ob in einer Liste eine Zeile verschoben ist. Die Umstände und auch die finanziellen Konsequenzen sind völlig anders. Aus diesen Gründen kann ein schwerer Mangel hundert- oder tausendfach schwerer wiegen als ein leichter Mangel. Mangel ist nicht gleich Mangel. Deshalb teilt sich die korrektive Systemerhaltung in die

- **Instandhaltung**, also die Behebung schwerer Mängel, und
- die **Mängelkorrektur**, also die Beseitigung mittelschwerer und leichter Mängel.

Die Systeminstandhaltung ist ereignisgetrieben. Sie muss sofort reagieren, so wie der ADAC-Notdienst auf der Autobahn. Weil man nie wissen kann, wann der nächste Notfall eintritt, muss dafür ein Notdienst eingerichtet sein. Er wird von Leuten betrieben, die wie die Feuerwehr nur darauf warten, den nächsten Notfall zu behandeln.

Mängelkorrektur ist im Gegensatz dazu planbar. Der Autofahrer kann mit dem falschen Benzinanzeiger noch eine Weile herumfahren. Dieser Mangel und auch der mit dem Lackkratzer wird bei der nächsten Inspektion behoben. Deshalb soll der Autobesitzer eine Liste aller ihm bekannten Mängel führen und sie in regelmäßigen Abständen alle auf einmal beheben lassen. Beim Softwaresystem ist dies der nächste Release-Wechsel.

Perfective Maintenance

Perfective Maintenance (perfektionierende Systemerhaltung) hat zwei Ziele: »*to improve performance*« und »*to improve other attributes of the product*«. Performance ist Schnelligkeit, aber auch Effizienz bei der Nutzung vorhandener Ressourcen. Um beim Beispiel des Autos zu bleiben, wäre dies die Steigerung der Geschwindigkeit und der Beschleunigung oder die Reduzierung des Benzinverbrauchs. Im Falle eines Softwaresystems bedeutet Performance-Verbesserung die Reduktion der Antwortzeiten und Durchlaufzeiten sowie die Reduzierung der Systemauslastung und des Speichergebrauchs. In beiden Fällen wird das Ziel über Tuningmaßnahmen erreicht.

»*To improve other attributes of the product*« lässt viele Interpretationen zu. Es könnte bedeuten, das Auto neu zu lackieren oder den Kofferraum zu verkürzen, um mehr Sitzplatz zu gewinnen. Bei einem Softwaresystem könnte man die Bildschirmanzeigen grafisch gestalten, zum Beispiel durch die Screen-Scratching-Technik, oder den Programmcode zu restrukturieren, um den Einbau zusätzlicher Funktionen zu erleichtern.

Die Performance-Verbesserungen werden im Englischen als »*optimizing actions*« bezeichnet – zu Deutsch »Optimierung«. »*The improvement of other attributes*« ist das, was im Englischen als »Reengineering« bezeichnet wird. Im Einzelnen gibt es:

- User Interface Reengineering
- Data Reengineering
- Program Reengineering
- Architectural Reengineering [Snee91]

Reengineering bedeutet in diesem Sinne die technische Verbesserung des Systems, ohne die fachliche Leistung zu beeinflussen. Es soll leichter werden, das System zu bedienen, zu pflegen und auch auszubauen. Im Deutschen könnte man den Begriff »Sanierung« dafür verwenden.

Bei Optimierungen und Sanierungen handelt es sich um geplante Eingriffe in das bestehende System, um es besser zu machen. Was besser ist, muss man in beiden Fällen genau definieren und den Aufwand dafür kalkulieren, um klären zu können, ob es sich wirklich lohnt.

Adaptive Maintenance

Adaptive Maintenance (adaptive Systemerhaltung) ist »*to adapt the product to a changing environment*«. Wichtig ist hier der Begriff »*changing environment*«. Auch beim Auto haben wir es mit einer veränderlichen Umgebung zu tun. Im Winter brauchen wir andere Reifen als im Sommer. Der Reifenwechsel ist ein gutes Beispiel für adaptive Systemerhaltung. Es ist auch üblich, im Winter Antifrostmittel in das Kühlwasser zu gießen. Auch dies ist eine adaptive Systemerhaltungsmaßnahme.

Bei Softwaresystemen gibt es gleich zwei Umgebungen: eine fachliche und eine technische. In der fachlichen Umgebung ändern sich die Gesetze, die Vorschriften, die Geschäftsregeln und die Arbeitsabläufe. In der technischen Umgebung ändern sich die Datenbanksysteme, die Transaktionsmonitore, die Betriebssysteme und die Rechner, auf denen die Anwendungssysteme laufen. Das Umfeld eines Systems ist ständig in Bewegung, und die Software muss ständig daran angepasst werden. Es ändern sich die Datenstrukturen, die Datentypen, die Verarbeitungslogik und die Programmschnittstellen.

Früher, als die Welt sich nicht so schnell wandelte, genügte es, die Systeme im Jahresrhythmus zu verändern. Man sammelte die Änderungsanträge und baute

sie alle auf einmal in das System ein. Dann gab es eine neue Version der Software. Heute, da sich alles viel schneller bewegt – die fachliche wie auch die technische Umgebung –, sehen wir uns gezwungen, die Änderungen alle drei Monate vorzunehmen und den Anwendern zur Verfügung zu stellen. Die Änderungen häufen sich, und die Lieferintervalle werden immer kürzer.

Gerade hier in diesem Punkt unterscheiden sich Softwareprodukte stark von anderen Gebrauchsgütern wie Autos. Der Begriff »Software« (»Weichware«) impliziert, dass das Produkt von Anfang an dazu bestimmt ist, leicht änderbar zu sein. Ein Softwaresystem ist die Abbildung eines anderen, zum Beispiel technischen, sozio-ökonomischen oder betriebswirtschaftlichen Systems. In dem Maße, wie sich das abgebildete System ändert, muss das Softwareprodukt nachziehen. Software, die sich nicht wandelt, ist bald unbrauchbar, weil sie keine adäquate Abbildung der Realität ist. Adaptive Systemerhaltung ist also etwas ganz Wichtiges. Ohne permanente Fortschreibung stirbt die Software. Software braucht Anpassung wie Pflanzen das Wasser.

Wenn Deutsche von Wartung sprechen, dann sehen sie darin ein notwendiges Übel. Dies ist aber beim englischen Begriff »*Maintenance*« keineswegs der Fall. Maintenance heißt »Erhaltung«, und dazu gehört in der Softwaretechnik vor allem die Anpassung. Dies ist der wichtigste Teil der Systemerhaltung – wichtiger sogar als die Reparatur und die Verbesserung.

Maintenance Services

In Anlehnung an die offizielle IEEE-Definition gehören drei Dienste zum reinen Softwareerhaltungsbetrieb:

- ein **Instandhaltungsdienst** für die sofortige Behebung schwerer Mängel
- ein **Nachbesserungsdienst** für die periodische Anpassung des Systems (adaptive Systemerhaltung) und die Beseitigung von mittelschweren und leichten Mängeln (Mängelkorrektur)
- ein **Reengineering-Dienst** für die gelegentliche Überarbeitung des Systems (perfektionierende Systemerhaltung) [Long90]

Da die Mängelkorrektur und die adaptive Systemerhaltung zur gleichen Zeit am gleichen Objekt arbeiten, werden sie hier zum Nachbesserungsdienst zusammengefasst. Damit kommen wir dem englischen Begriff »*improvement*« entgegen. Software Maintenance ist, abgesehen vom ereignisgesteuerten Instandhaltungsdienst und den gelegentlichen Sanierungsprojekten, ein ständiger »*improvement service*« und sollte von den Anwendern dementsprechend honoriert werden.

1.3.2 Softwareevolution als Weiterentwicklung

Der Untertitel dieses Buches ist nicht ohne Grund »Wartung und Weiterentwicklung bestehender Anwendungssysteme«. Diese beiden Tätigkeiten sind nicht

gleichartig. Wir haben uns im letzten Abschnitt mit dem Begriff Software Maintenance auseinander gesetzt. Wir wissen, dass es *corrective, perfective* und *adaptive maintenance* gibt. Im Englischen gibt es neben dem Begriff »Maintenance« auch den Begriff »Evolution«. Evolution ist nicht gleich Maintenance, obwohl sie meistens von den gleichen Leuten zur gleichen Zeit am gleichen Objekt ausgeführt wird. Aus dieser Erkenntnis heraus wurde die wichtigste Fachzeitschrift zu diesem Themengebiet im Jahre 1995 von »*Journal of Software Maintenance*« in »*Journal of Software Maintenance and Evolution*« umbenannt. Die führenden Fachleute auf diesem Gebiet haben entschieden, dass Evolution etwas anderes als Systemerhaltung ist, nämlich die Vermehrung der Produktleistung durch Weiterentwicklung [ChHa01].

Natürlich kommt in diesem Zusammenhang die Frage auf, was Evolution von adaptiver Systemerhaltung unterscheidet. Die Antwort hat handfeste finanzielle Auswirkungen: Adaptive Systemerhaltung wird vom Anwender als Teil des normalen Nachbesserungsdienstes erwartet. Er ist auch bereit, dafür zu bezahlen, aber nur im Rahmen der jährlichen Wartungsgebühren. Das System bietet ihm ja nicht mehr Möglichkeiten als vorher, sondern wurde bloß einer sich ändernden Umwelt angepasst. Statt eines sechsstelligen Datums gibt es ein achtstelliges Datum. Statt der Deutschen Mark gibt es den Euro und statt einer Mehrwertsteuer von 16 % eine von 20 %.

Dagegen führt eine Weiterentwicklung zu einer Steigerung der Systemdienstleistung. Das Produkt bietet – gemessen an seiner Umwelt – mehr Möglichkeiten, was seinen Wert erhöht. Dieser Wertzuwachs hat sich in einer direkten Finanzierung durch die Anwender niederzuschlagen.

Die Differenzierung zwischen Weiterentwicklung und adaptiver Systemerhaltung hat also starke kommerzielle Auswirkungen. Ihr Dreh- und Angelpunkt ist der Ausdruck »sich ändernde Umwelt«, der unterschiedlich weit ausgelegt werden kann: Geht es nur um die technische Umwelt, also die Anpassung an neue Hardware, Betriebs- und Datenbanksysteme? Oder auch um die organisatorische, also die Berücksichtigung neuer betrieblicher Abläufe? Sind neue Gesetze Teil einer »sich ändernden Umwelt«? Was ist mit Änderungen an öffentlichen Schnittstellen, zum Beispiel zum Finanzamt, zu Meldebehörden oder zur Börse? Was als Systemumwelt gilt und welche Änderungen davon im Rahmen der adaptiven Systemerhaltung berücksichtigt werden, muss im Wartungsvertrag genau spezifiziert werden. Wer glaubt, hier Zeit sparen zu können, wird später ein Vielfaches davon in Konflikten um die Klassifizierung von Anwenderanträgen verlieren.

Bei der Vertragsgestaltung sind vorhersagbare Kosten das oberste Ziel des Produktmanagers. Allgemeinklauseln wie »Das System wird im Rahmen der Wartungsgebühren an alle steuerlichen Änderungen angepasst« können leicht zur Kostenfalle werden. Hier empfiehlt sich das Einziehen einer Aufwandsobergrenze, ab der der Anwender in die Pflicht genommen wird.

1.3.3 Zur Notwendigkeit der Unterscheidung

Es ist die Aufgabe des Produktmanagements, die Weiterentwicklungsanträge von den Änderungsanträgen sorgfältig zu trennen. Weiterentwicklungsanträge bilden eine separate Kostenart und werden getrennt abgerechnet. Das ausführende Personal ist angehalten, seine Arbeitsstunden möglichst genau gegen die einzelnen Anträge zu buchen. Die Mängelkorrekturen und Änderungen fallen üblicherweise unter die Aufgaben, die von den Wartungsgebühren gedeckt werden. Die Erweiterungen bewirken dagegen Weiterentwicklungskosten und werden zusätzlich zu den Wartungsgebühren abgerechnet.

Wie andere Betriebe muss auch ein Erhaltungsbetrieb nach betriebswirtschaftlichen Grundsätzen geführt werden. Er leistet Dienste und schafft damit einen Wert für die Anwender, die diesen Wert bezahlen müssen. Die meisten Verträge sehen vor, dass die Kosten der Systeminstandhaltung vom Produktverantwortlichen übernommen werden müssen. Ein anderer Teil der Dienste – die Nachbesserung – ist planbar und kontrollierbar und wird durch fest vereinbarte jährliche Zahlungen gedeckt. Der dritte Teil der Dienste – die Weiterentwicklung – ist weder planbar noch kontrollierbar. Er geschieht als Folge nicht vorhersehbarer Anforderungen, die zu einer Erweiterung der bestehenden Funktionalität führen. Schon allein aus steuerlichen Gründen muss dieser Produktwertzuwachs durch Zusatzeinnahmen gedeckt werden.

Daraus folgt, dass die Weiterentwicklung in einer parallelen Schiene anders abzulaufen hat als die Nachbesserung. Weiterentwicklungsarbeiten müssen kalkuliert und dem Anwender als verbindliches Angebot unterbreitet werden. So hat der Anwender die Freiheit zu entscheiden, ob er bereit ist, die Entwicklung zu bezahlen oder nicht. In »Nice to have«-Fällen, wie der Erstellung einer zusätzlichen Statistik, kann er notfalls darauf verzichten, wenn die Kosten zu hoch sind.

Auf jeden Fall ist zu vermeiden, dass der Erhaltungsbetrieb die Weiterentwicklung eines Produkts über die Wartungsgebühren finanziert. Die Wartungsgebühren sind für die Grunderhaltung gedacht und nicht für die Evolution. Im Falle von Standardsoftware soll die Weiterentwicklung über die neuen Lizenzeinnahmen finanziert werden. Wenn es keine neue Lizenz gibt, dann gibt es auch keine Weiterentwicklung. Bei eigenen Anwendungssystemen sollen die Fachabteilungen für die Weiterentwicklung bezahlen. Dieses Thema wird im nächsten Kapitel ausführlicher behandelt.

1.4 Ein Produktlebenszyklusmodell für Software

Systemerhaltung und Weiterentwicklung sind Aktivitäten im Rahmen des Lebenszyklus eines Softwaresystems. Die Begriffe »*lifecycle*«, »*product lifecycle*« und »*lifecycle management*« haben im Software Engineering eine zentrale Rolle und eine entsprechend lange Vorgeschichte. Software Engineering als Disziplin wurde auf der NATO-Tagung im Jahre 1969 in Garmisch ins Leben gerufen, und

zwar als Antwort auf die damaligen chaotischen Verhältnisse in der Softwareentwicklung. Der Vater des Begriffs war Frederik Bauer von der TU München [NaRa69].

Die chaotischen Verhältnisse sind vielerorts geblieben, aber daneben ist ein umfangreiches Wissensgebiet entstanden, das es ermöglicht hat, mancherorts große und komplexe Softwaresysteme systematisch zu entwickeln und über mehrere Jahre oder gar Jahrzehnte zu erhalten und weiterzuentwickeln. Das erste Lebenszyklusmodell für Software wurde bereits ein Jahr nach der entsprechenden NATO-Tagung von Winston Royce von Boeing Aircraft ins Leben gerufen. Royce schlug vor, die Softwareentwicklungsprozesse in mehrere aufeinander folgende Phasen zu zerlegen, wobei Maintenance als eigenständige Phase hinter der Einführung am Ende des Entwicklungszyklus stand. Aufgrund der Darstellung des Phasenmodells wurde es als »das Wasserfallmodell« (Abb. 1–7) bezeichnet [Royc70].

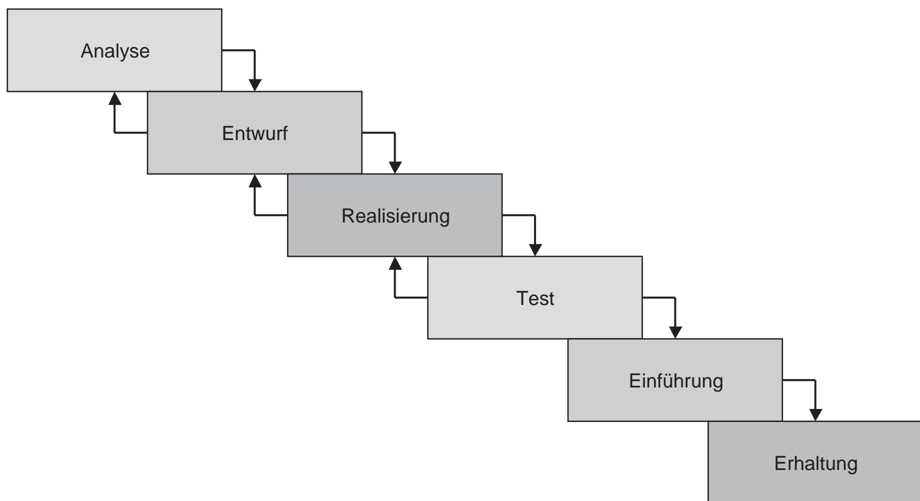

Abb. 1–7 *Das klassische Wasserfallmodell*

Seitdem ist viel Wasser über die Phasen gelaufen, und viele Alternativmodelle sind vorgeschlagen worden, unter anderem das Spiralmodell von Boehm [Boeh88], das V-Modell von Hughes Aircraft [Deut82], das evolutionäre Modell von Gilb [Gilb88], das zyklische Modell von Henderson [HeEd90], das iterative Modell von Jacobson [Jaco92] und zuletzt der Rational Unified Process von Kruchten [Vers00]. Allen diesen Modellen ist die Schwierigkeit gemeinsam, die sie mit den Aktivitäten Maintenance und Evolution haben. Das liegt vor allem daran, dass es sich um Entwicklungsmodelle handelt. Sie betrachten die korrektiven, adaptiven und perfektionierenden Aufgaben als Teil der Weiterentwicklung und die Weiterentwicklung als Fortsetzung der Entwicklung unter anderem Vorzeichen. Wer diese Logik aber zu Ende denkt, kommt zum Schluss, dass eine Entwicklung nie zu Ende ist, was im gewissen Sinne auch wahr ist [TrBa99].

Diese Schlussfolgerung der unendlichen Entwicklung mag theoretisch korrekt sein und darf auch für Forschungsprojekte gelten. In der Wirtschaft und erst recht vor Gericht ist sie nicht haltbar. Ein Betrieb, der ein Softwareprodukt einsetzt, muss davon ausgehen, dass das Produkt fertig entwickelt ist, sonst hat er keine Regressansprüche, wenn Mängel auftreten. Der Hersteller des Produkts könnte behaupten, es sei noch in Entwicklung, und er könne deshalb für nichts garantieren. Spätestens hier wird die schöne offene Welt der endlosen Softwareentwicklung von der harten Wirklichkeit der Wirtschaft eingeholt. Der Anwender erwartet von Produkten, die er einsetzt, einen gewissen Fertigstellungsgrad, und nach den geltenden Gesetzen hat er auch ein Recht darauf.

Vom Standpunkt des Richters aus gesehen liegt der amerikanische Rechnungshof ganz richtig, wenn er behauptet, ab der ersten Freigabe herrschen andere Regeln. Was vorher als Entwicklungsfehltritt angesehen wurde, gilt ab diesem Punkt als Absturz mit schwerwiegenden finanziellen Folgen. Aus ist es mit der Spielerei der Softwareentwicklung, den iterativen oder zyklischen Phasen. Ab hier wird es ernst. Die Software muss mindestens das leisten, was dem Anwender versprochen wurde, sei es ein Standardprodukt oder ein speziell für ihn entwickeltes System. Wenn sie es nicht tut, gibt es finanzielle Einbußen für den Hersteller.

In den Fällen, in denen der Betrieb seine Anwendungssysteme selbst entwickelt, sind die Konsequenzen der Produktionsausfälle und der falschen Ergebnisse nicht so schwerwiegend wie bei gekauften oder nach Auftrag entwickelten Produkten. Aber auch hier wird es Folgen geben. Der IT-Leiter, der Gruppenleiter, der Projektleiter oder alle drei werden zur Verantwortung gezogen.

Es ist also rein akademisch zu behaupten, eine Softwareentwicklung sei niemals abgeschlossen. Um den Anforderungen der Wirtschaft und des Gesetzes zu genügen, muss eine Trennlinie gezogen werden. Diesseits dieser Grenze ist alles Entwicklung und daher zu verzeihen. Aber jenseits von ihr werden die Verursacher von Schäden zur Verantwortung gezogen. Genau dies ist die Trennlinie zwischen reiner Entwicklung und Erhaltung mit Weiterentwicklung.

Ein Lebenszyklusmodell, das dieser harten Realität entspricht, wurde auf der *International Conference on Software Engineering* im Jahre 2000 präsentiert und ein Jahr später im *IEEE Software* Magazin veröffentlicht. Die Väter des Konzepts sind beide führende Wissenschaftler auf dem Gebiet der Systemerhaltung: Keith Bennett von der Universität Durham in England und Vaclav Rajlich von der Wayne State Universität in Amerika [RaBe00]. Sie schlagen ein fünfteiliges Phasenkonzept vor (Abb. 1–8). Die Phasen werden hier im Deutschen die »5 E« genannt:

- **Entstehungsphase**
- **Entwicklungsphase**
- **Evolutionsphase**
- **Erhaltungsphase**
- **Entsorgungsphase**

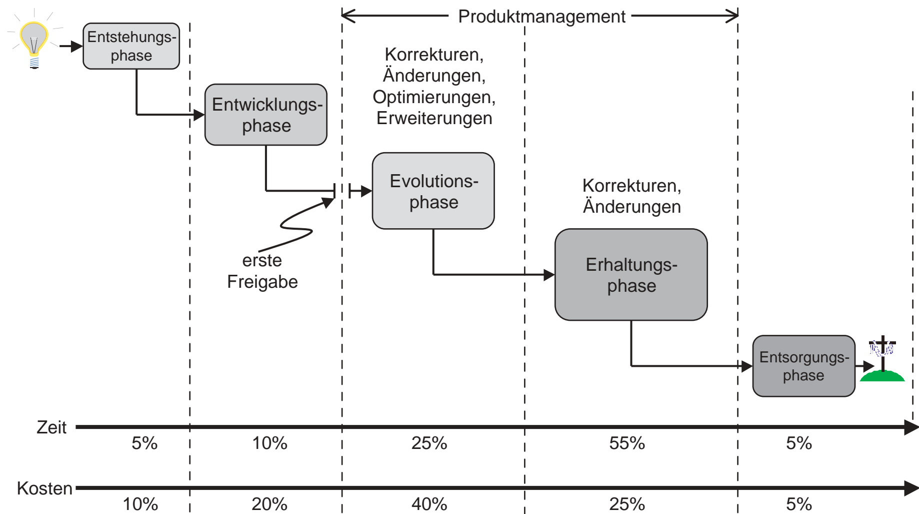

Abb. 1–8 *Produktlebenszyklusmodell*

1.4.1 Entstehungsphase

Die Entstehungsphase ist die Zeit, in der ein Softwaresystem konzipiert wird. Entweder ist man in der Lage, die Anforderungen an das neue System auf Anhieb zu erkennen und in einem Fachkonzept niederzuschreiben, was bei der zunehmenden Komplexität der Anwendungen immer seltener der Fall ist, oder man baut einen Prototyp, um die Anforderungen zu sammeln. Mit Hilfe des Prototyps wird ein Fachkonzept erstellt, in dem die funktionalen und nicht funktionalen Anforderungen spezifiziert sind.

Zu den funktionalen Anforderungen gehören:

- das **Prozessmodell**, also die Beschreibung der zu unterstützenden Geschäftsprozesse,
- das **Objektmodell**, also die Beschreibung der zu implementierenden Geschäftsobjekte,
- das **Schnittstellenmodell**, also die Beschreibung der Interaktionen zwischen den Geschäftsobjekten und Geschäftsprozessen,
- das **Datenmodell**, also die Beschreibung der logischen Datenentitäten und deren Beziehungen,
- das **Funktionsmodell**, also die Beschreibung der einzelnen Anwendungsfälle, und
- das **Entscheidungsmodell**, also die Beschreibung der Geschäftsregeln.

Hinzu kommen die fachlichen Testfälle, die aus dem Prototyp oder aus der Überlagerung der Daten-, Funktions- und Entscheidungsmodelle abgeleitet werden.

Zu den nichtfunktionalen Anforderungen gehören:

- die **Qualitätskriterien**, die zu erfüllen sind, mit dem geforderten Grad der Erfüllung,
- die **Systemeinschränkungen** bezüglich der Nutzung von Hardware, Software und sonstiger Betriebsmittel,
- die **Risiken**, die beim Einsatz des Systems zu berücksichtigen sind,
- die **Stufen für die Auslieferung** des Systems oder der Paketierungsplan und
- die **Kriterien für die Abnahme** und für die Freigabe der ersten produktiven Version.

Es ist wichtig, die Ausarbeitung eines Konzepts einschließlich des Baus des Prototyps als eigenständiges Projekt mit eigenem Budget zu betrachten. Das Ergebnis dieses Projekts ist ein ausführlicher Plan vom System, das in der nächsten Phase zu entwickeln ist. Diese erste Phase kann je nach Größe und Komplexität der Anwendung von wenigen Monaten bis zu zwei Jahren dauern [RoRo99].

1.4.2 Entwicklungsphase

In der Entwicklungsphase wird das eigentliche Produkt gebaut. Hier erfolgt der Entwurf der Systemarchitektur, der Datenbanken, der Komponenten, der Klassen und der Schnittstellen gefolgt von ihrer Implementierung. Parallel dazu werden die Testarchitektur, die Testprozeduren und die technischen Testfälle konzipiert und die passenden Testdaten bereitgestellt. Die Entwicklungsarbeit besteht aus den üblichen Teilphasen:

- Design,
- Programmierung und
- Test,

die selbstverständlich mehrfach wiederholt werden und zu Änderungen am Fachkonzept führen können. Die Korrektur, Anpassung und Erweiterung des Fachkonzepts während der Entwicklungsphase können sogar als die ersten Systemerhaltungsarbeiten angesehen werden, wenn sie auch längst nicht jene Konsequenzen wie in der Produktion haben. In dieser glücklichen Phase hat man noch die Freiheit, alles auf den Kopf zu stellen oder sogar wieder von vorn, mit einer neuen Technologie anzufangen, da noch keiner vom System abhängig ist.

Auch diese Phase kann von wenigen Monaten bis zu zwei Jahren dauern, je nachdem, wie groß die Anwendung ist und wie oft man die Entwicklungsphasen wiederholen muss. Am Ende erfolgt ein harter Schnitt. Das Produkt wird ausgeliefert, und ab diesem Zeitpunkt gelten andere Regeln. Das System gehört jetzt den Anwendern. Die Entwickler müssen sicherstellen, dass die Anforderungen des Anwenders erfüllt bleiben, und können sich nur noch auf einem schmalen Grat fortbewegen [ErPe00].

1.4.3 Evolutionsphase

Während der Evolutionsphase haben die Entwickler zwar wesentlich weniger Spielraum, aber sie haben immer noch die Möglichkeit, das System zu erweitern. Denn neben den klassischen Systemerhaltungstätigkeiten findet noch eine Weiterentwicklung statt. Das System kann sogar bis zu 20 % jährlich wachsen, ohne seinen Zusammenhalt zu verlieren.

In dieser Phase ist die Arbeit in mindestens drei parallel laufende Pakete geteilt. Die Aufbauorganisation folgt dieser Unterteilung. An erster Stelle steht die Instandhaltung der in der Produktion eingesetzten Version. Die Arbeit hier ist kritisch und beansprucht die höchste Priorität. Daneben gibt es die Nachbesserungsarbeiten – Mängelkorrektur und Änderungsdienst. Als Drittes findet die Weiterentwicklung statt. Hier werden neue Funktionen, Daten und Schnittstellen in das Produkt eingebaut. Diese Tätigkeit ist das, was die Bezeichnung »Evolution« rechtfertigt.

Während der Evolution hat man auch noch die Möglichkeit, auf einem vierten Feld tätig zu werden. Dort werden Optimierungs- und Sanierungsprojekte durchgeführt. All das, was zur technischen Perfektion des Produkts beiträgt, zum Beispiel Restrukturierung, Refaktorierung, Bereinigung, Kapselung und Konvertierung, gehört hierher. Die Zahl und der Umfang dieser Projekte hängt davon ab, wie viel Kapazität von den anderen drei Feldern übrig bleibt, und diese ist meistens minimal. Derartige Reengineering-Aktivitäten werden zwar von der Wissenschaft stark überbetont, finden jedoch in der Praxis aus Kosten- und Kapazitätsgründen meist nur in Notfällen statt.

Eine wichtige Eigenschaft der Evolutionsphase ist die Fortschreibung des Konzepts. Solange das System weiterentwickelt wird, wird das Fachkonzept aus der Entstehungsphase korrigiert und fortgeschrieben. Wenn sich eine Funktion oder Datenstruktur im Code verändert, wird die entsprechende Funktion oder Datenstruktur im Konzept entweder schon vorher verändert oder nachgezogen. Eigentlich sollten Konzept und Code parallel zueinander im Gleichschritt fortgeschrieben werden. Dieser zusätzliche Aufwand ist deshalb notwendig, weil man eine Referenz für den Test braucht. Um den Code zu verifizieren, müssen die Testfälle aus dem Konzept abgeleitet werden. Das geht nur, wenn das Konzept aktuell ist. Daneben werden auch einige andere Zwecke erfüllt, zum Beispiel die Erstellung der Dokumentation.

Es ist umstritten, wie lange die Evolutionsphase dauern kann. Gestützt auf die Evolutionsstudien von Belady und Lehman behaupten Bennett und Rajlich, dass sie höchstens fünf Jahre dauern kann. Da mit jeder Erweiterung die Komplexität der Software zunimmt, wird es zunehmend schwieriger, die Software zu ändern. Allein bei einer jährlichen Wachstumsrate von 10 % wäre das System nach fünf Jahren über 60 % größer als bei der Erstauslieferung. Wenn man die vielen Änderungen und Mängelkorrekturen dazurechnet, erkennt man, dass der Code sich in fünf Jahren um drei Viertel ändern kann, was so gravierend ist, dass

ein Wachstumsstopp die Folge sein kann. Wenn man andererseits das Geld und die Kapazität dazu hat, kann man durch ständige Reengineering-Maßnahmen die Komplexität bekämpfen und die Dauer der Evolutionsphase dadurch verlängern.

Allerdings stößt jedes Softwaresystem irgendwann an seine Grenzen und erreicht einen Stand, der sich nicht mehr erweitern lässt. In diesem Moment ist es am Ende der Evolutionsphase angekommen und geht in die Erhaltungsphase über [Lieb93].

1.4.4 Erhaltungsphase

In der Erhaltungsphase findet nur noch Instandhaltung und Nachbesserung statt. Mit eingeschränkter Kapazität erhält man das System am Leben.

Wie lange ein Softwaresystem in dieser Phase bleiben kann, hängt von der Umgebung ab. In der IBM-Hostwelt sind jede Menge Assembler- und COBOL-Systeme aus den späten 60er- und frühen 70er- Jahren zu finden. Sie haben 30 Jahre überlebt, sind meist Teil einer komplexen IT-Landschaft und leisten immer noch einen Dienst für die Anwender. Im Prinzip können solche Systeme so lange leben, wie die IBM-Betriebssysteme sie noch unterstützen.

Andere Systeme, die später in der 4GL-Generation entstanden sind, wurden schon aufgegeben. Der Grund ist nicht, dass sie nicht lauffähig wären. Vielmehr werden die Umgebungen, in denen sie realisiert wurden, nicht mehr unterstützt, oder es lassen sich keine Programmierer dafür finden. Es sind also hauptsächlich technische Gründe, die zur Aufgabe eines Anwendungssystems führen. Die hohe Anzahl von Anwendungssystemen in dieser Phase zeugt weniger davon, wie stabil sie sind, als davon, wie starr die IT-Welt ist. Anwender neigen dazu, an technologisch schon lange überholten Systemen festzuhalten.

Dennoch kann die Erhaltungsphase so lange dauern, wie die technische Umgebung unterstützt wird und sich Personal finden lässt, das bereit ist, das alte System zu pflegen. Durch neue Kapselungstechniken ist es sogar möglich, solche Relikte aus vergangenen Zeiten in moderne Webarchitekturen einzubinden. Damit wird ihre Lebenszeit nochmals verlängert [WiHu92].

1.4.5 Entsorgungsphase

Eine Vielzahl von Ereignissen kann das Ende des Systems einläuten: Die technische Basis wird nicht länger unterstützt. Das Datenbanksystem wird abgelöst. Der Compiler ist nicht mehr funktionsfähig. Das Systemerhaltungspersonal geht in Pension. Die Funktionalität der Software passt überhaupt nicht mehr zur Anwendung, oder die Anwendung selbst stirbt aus. Wenn eine oder mehrere dieser Bedingungen auftreten, ist es Zeit, das System auszumustern. Bennett und Rajlich benutzen den Begriff »*Outphasing*«, um diesen Prozess des Herunterfahrens des Systemerhaltungsbetriebs zu benennen.

Das System selbst kann noch eine Weile in der Produktion bleiben, solange ein Benutzer es noch braucht. Aber es wird nicht mehr gepflegt. Spätestens beim nächsten Umgebungswechsel oder bei der nächsten fachlichen Veränderung wird auch der letzte Anwender gezwungen, eine andere Lösung zu suchen [Lewi98].

1.5 Zur Problematik der Systemerhaltung und -weiterentwicklung

Der Aufgabenbereich des Produktmanagements ist in Anbetracht der Vielfalt des Themas ziemlich breit. Er umfasst alle Tätigkeiten, die für die Erhaltung und Evolution von Softwaresystemen erforderlich sind, einschließlich der üblichen Managementtätigkeiten wie Budgetplanung, Aufbauorganisation, Ablauforganisation, Projektmanagement, Änderungsmanagement, Testmanagement und Qualitätsmanagement. Bei größeren Systemen wird es notwendig sein, diese Aufgaben auf mehrere Köpfe zu verteilen. In diesem Fall gibt es nicht nur einen Produktmanager, sondern einen Produktmanagementstab.

In den einzelnen Kapiteln wird auf die Teilgebiete des Produktmanagements eingegangen. Im folgenden Abschnitt wird die Problematik der Systemerhaltung und Systemevolution aus industrieller und wissenschaftlicher Sicht zusammengefasst.

1.5.1 Systemerhaltung aus industrieller Sicht

Systemerhaltung stellte sich bereits in den 70er-Jahren als ein unübersehbarer Kostenfaktor in der betrieblichen Datenverarbeitung heraus. In der Anfangszeit der Datenverarbeitung hatten die ersten Entwickler genug zu tun, die Programme zum Laufen zu bringen. Jeder war froh, wenn die Programme nur halbwegs das leisteten, was man von ihnen erwartet hatte. Umso größer war die Enttäuschung, als man entdeckte, dass man mit der Inbetriebnahme noch lange nicht fertig war. Es gab immer wieder so genannte »*Bugs*« zu entfernen, und die Benutzer kamen ständig mit neuen Änderungsanträgen. Bald entdeckten die Programmierer, dass sie Gefangene ihrer Schöpfung waren. Je komplexer und nützlicher ihre Programme waren, umso mehr wollten die Benutzer von ihnen zusätzlich haben.

Dieses Phänomen hatte schon Girish Parikh in den 70er-Jahren erkannt und Ende der 70er-Jahre zusammen mit Nicholas Zvegintzov in einem Tutorial on Software Maintenance geschildert [PaZv79]. Er und Gerald Weinberg gelten als die Entdecker der speziellen Problemstellungen der Systemerhaltung. Parikh stellte fest, »*maintenance is a process with its own rules and techniques*«, und er prägte den Satz: »*Any fool with a knife can remove an appendix from a patient, but only a trained person can remove it without any undesired side effects.*« Parikh und Weinberg verfassten 1978 den berühmten Artikel »*Mid-City Triangle*«, und vier Jahre später folgte das Buch von Parikh zum Thema »*Tech-*

niques of Program and System Maintenance« [Pari82]. »*Mid-City Triangle*« behandelt das Schicksal von Systemerhaltungsprogrammierern in der Metropole Chicago, die versuchen, durch ständige Jobwechsel auszubrechen. Am Ende landen sie dort, wo sie angefangen haben, nämlich bei der Pflege der Programme, die sie vor zehn Jahren selbst geschrieben hatten [WePa82].

Als Inder hatte Parikh ein besonderes Verhältnis zur Systemerhaltung, weil, wie er behauptet, die Idee des Lebenszyklus in der Hindu-Religion verankert ist. Dort herrscht nicht ein Gott – der Gott der Schöpfung –, sondern gleich drei: Brahma, der Gott der Schöpfung, Vishnu, der Gott der Erhaltung, also Maintenance, und Shiva, der Gott der Wiedergeburt, also des Reengineering [Pari86]. Inder wissen also, dass Schöpfung (also Entwicklung) nur der Beginn eines sich wiederholenden Lebenszyklus ist, dass Erhaltung im Leben eines Menschen wie auch im Leben eines Systems eine noch wichtigere Rolle spielt und dass der Tod eines Systems am Ende des Lebenszyklus nicht wirklich ein Ende ist, sondern ein neuer Anfang (Abb. 1–9). Mit dieser Einstellung haben es die Inder mit der Systemerhaltung, die sie als etwas ganz Natürliches akzeptieren, einfacher. Es darf also nicht wundern, dass es ein Inder war, der »*Software Maintenance*« als Beschäftigungsfeld entdeckt hat und dass es auch Inder sind, die heute damit das größte Geschäft machen [Rama00].

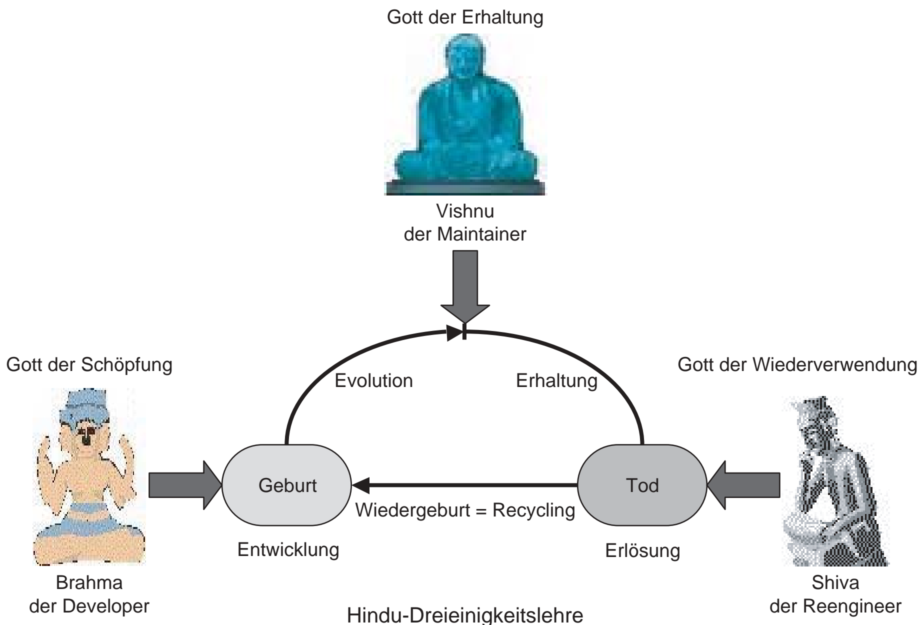

Abb. 1–9 *Das Urmodell für den Softwarelebenszyklus*

Wenn also Systemerhaltung unvermeidlich ist und überhaupt den Hauptteil der Softwarearbeit ausmacht, dann müsste die Software von Anfang an danach ausgerichtet sein. Die ersten Schriften über Software Maintenance befassten sich des-

halb nicht nur mit der Technik der Systemerhaltung, sondern vor allem damit, wie die Software zu konstruieren ist, um ihre Erhaltung möglichst leicht zu machen. Kein Geringerer als David Parnas hat sich sehr früh darüber Gedanken gemacht, wie man den Erhaltungs- und Weiterentwicklungsaufwand in Grenzen halten könnte. Sein Artikel »*Designing Software for Ease of Extension and Contraction*« hat die Forschung zum Thema »*Maintainability*« ausgelöst [Parn79].

In dem im Jahre 1983 erschienenen Buch »*Software Maintenance – The Problem and its Solution*« schreibt James Martin, der Guru der 80er- Jahre: »*If all programs to be maintained were well documented and cleanly structured, and used third normal form data models and data dictionaries for generating the programmer's data, the task of the maintainer would be much easier[...]. The problem for most maintainers is that they have to maintain ill-documented code that is covered with patches, with no comprehensible structure and that has data representations buried in the program code ...*« [MaCl83].

Solche Kritiken führten Mitte der 80er-Jahre zu einem Boom an Software Reengineering Tools, die versprachen, die Software pflegeleicht zu machen. Leider haben sie nur selten das gebracht, was versprochen wurde. Aber sie dienten immerhin dazu, die Fachwelt für die Bedeutung der Systemerhaltung zu sensibilisieren. Jetzt wusste jeder, dass monolithische, unstrukturierte und unsaubere Programme die Erhaltungskosten in die Höhe treiben. Einer der Hauptgründe für die zu jener Zeit aufkommende Objekttechnologie war, dass sie die Fortschreibungsfähigkeit steigern soll. Objektorientierte Programme mit ihrer Klassenstruktur sollten viel leichter zu ändern und zu erweitern sein [Meye88]. Diese Betonung der Wartbarkeit erweckte vielerorts den Eindruck, man könne die Systemerhaltungskosten durch neue und bessere Konstruktionsmethoden fast auf null reduzieren. Dieser Eindruck ist natürlich falsch. Erhaltungskosten entstehen aus der Notwendigkeit, Systeme zu korrigieren, anzupassen und zu erweitern. Auch die bestdurchdachten und bestkonstruierten Systeme müssen fortgeschrieben werden. Durch Konstruktionsmethoden wie die strukturierte Programmierung und die objektorientierte Programmierung konnte man zwar den Aufwand für Programmänderungen vermindern, aber nicht die Programmänderungen vermeiden. Sie ergeben sich aus der Veränderung der fachlichen und technischen Umgebung und sind unabhängig von der Konstruktion der Software selbst. Leider ist es noch nicht einmal bewiesen, dass Objektorientierung wirklich den Erhaltungsaufwand reduziert. Einige Studien haben sogar das Gegenteil gezeigt [WiMa93].

Die Technik der Programmänderung selbst wurde in zahlreichen Vorschriften und Handbüchern niedergelegt. Auf das erste Handbuch von Parikh folgte die Richtlinie des U.S. National Bureau of Standards mit dem Titel »*Guidance on Software Maintenance*« [Osbo85]. Darin wird Schritt für Schritt vorgegeben, wie ein Softwaresystem zu ändern ist. Ein weiteres Handbuch von Arthur unterschied zwischen Mängelkorrekturen, Änderungen und Erweiterungen und gab eine Vorgehensweise für jeden dieser Typen vor [Arth87]. Bis Ende der 80er- Jahre war die Erhaltungsproblematik in der Industrie allgemein bekannt und der Erhaltungs-

prozess mehrfach dokumentiert. Es wurden diverse Erhaltungstechniken in Industrieseminaren gelehrt, darunter die Top-down-, Bottom-up- und Inside-Out-Methode, und es waren bereits zahlreiche Werkzeuge auf dem Markt wie Code-Auditoren, statische Analysatoren und automatische Dokumentatoren, um den Erhaltungsprozess zu unterstützen [BaMi82]. Während die einen Maintenance als das ständige Feilen an einem Produkt sahen, definierten sie andere wie Victor Basili als die Schaffung immer neuer Nachfolgeprodukte aus dem gleichen Ursprungsprodukt [Basi90].

In den USA gab es zu dieser Zeit sogar eine *National Association of Software Maintenance* als Dachverband aller am Thema Interessierten und ein monatlicher Newsletter speziell für Systemerhaltungsprogrammierer mit Berichten aus der Praxis. Es enthielt auch Tipps, wie sie ihre Arbeit bessern verrichten könnten, denn die Abneigung der Programmierer gegen Systemerhaltung jeglicher Art stellte sich bald als Herausforderung fürs Management heraus [MeHa89]. In deutscher Sprache erschien das erste Buch zum Thema »Softwarewartung« erst im Jahre 1987 [WiBa87]. Drei Jahre später folgten gleich zwei Bücher mit dem Titel »Softwarewartung«. Das eine stammt von Franz Lehner [Lehn91], das andere von Harry Sneed [Snee90].

1.5.2 Systemerhaltung aus wissenschaftlicher Sicht

Wie oft in der Informatik hat die Wissenschaft das Thema »*Software Maintenance*« erst später entdeckt. Im Jahre 1983 veranstaltete die IEEE die erste »*International Conference on Software Maintenance*« (ICSM) an der U.S. Marine Post Graduate University in Monterey [Arno83]. Als die ersten Maintenance-Konferenzen in den 80er-Jahren stattfanden, kamen die meisten Beiträge entweder aus der Industrie oder der Bundesverwaltung in den USA. Die Universitäten waren kaum vertreten. Erst im Laufe der 80er-Jahre begannen die Universitäten und Forschungsinstitute sich für das Thema zu interessieren. Die ersten Artikel erschienen in den *Communications of the ACM* und im Magazin *IEEE Software*. Im Mai 1986 brachte die *IEEE Software* eine Sonderausgabe unter dem Titel »*Software Maintenance – Keeping the System Going*« heraus. Der Untertitel unterstrich, worum es ging: »das System im Betrieb erhalten« [ArMa86].

Ab 1988 nahm die Forschungsintensität rapide zu. Die *International Conference on Software Maintenance* begann, jährlich stattzufinden. An der Universität Maryland wurden die ersten Dissertationen zum Thema »*Software Maintenance*« unter der Leitung von Victor Basili verfasst [BaRo88]. Die U.S. Bundesverwaltung begann, etliche Studien auf diesem Gebiet zu finanzieren, und im Jahre 1989 wurde das »*Journal of Software Maintenance*« vom Wiley-Verlag in London herausgebracht.

In Europa gab es zunächst zwei Forschungsschwerpunkte – der eine an der Universität Neapel in Italien und der andere an der Universität Durham in England. In Durham fand 1988 der erste europäische *Workshop on Software Main-*

tenance statt, und bald darauf wurde das Institut von Keith Bennett zum »*National Centre of Software Maintenance*« umbenannt. Im Rahmen des europäischen IT-Forschungsprogramms ESPRIT wurden mehrere einschlägige Projekte gefördert. Die ICSM kam 1991 zum ersten Mal nach Europa – in Sorrento. Seitdem findet die Konferenz jedes zweite Jahr in Europa statt, und 1997 wurde die europäische *Conference for Software Maintenance and Reengineering* von einer Gruppe deutscher, schweizerischer und italienischer Professoren ins Leben gerufen [Rich97].

Seitdem hat die Forschung zur Softwaresystemerhaltung weltweit ein explosives Wachstum erlebt. Neue Bücher erschienen jährlich. Die Fachzeitschriften zum Software Engineering sind voll mit Artikeln über »*Maintenance*«. Beiträge zu den Maintenance-Konferenzen kommen aus aller Welt – insbesondere aus Indien, der geistigen Geburtsstätte der Erhaltungstheorie.

Auch im deutschsprachigen Raum ist »Softwarewartung« mittlerweile ein etabliertes Fachgebiet im Rahmen der Kern- und der Wirtschaftsinformatik. Maintenance wird explizit erforscht und gelehrt an den Universitäten Karlsruhe, Koblenz, Stuttgart, Essen, Wien, Zürich und Regensburg. In der Gesellschaft für Informatik gibt es mindestens zwei Fachgruppen, die sich mit Systemerhaltung und Reengineering befassen – eine aus der Managementperspektive und die andere aus der technischen Perspektive. Außerdem tragen deutschsprachige Wissenschaftler jetzt in zunehmendem Maße zu den internationalen und europäischen Fachkonferenzen bei. Was noch fehlt, ist eine systematische Ausbildung in Softwareerhaltungsverfahren und -techniken, aber auch das wird nicht lange auf sich warten lassen. Der Bedarf seitens der Industrie wird immer stärker.

1.5.3 Softwareevolutionslehre

Die Lehre der Softwareevolution hat einen anderen Ursprung und eine etwas andere Entwicklungsgeschichte als die der Systemerhaltung. Beide wurden von Indern begründet. Es war der indische Professor Ramamorthy an der Universität California, der bereits 1977 den Softwarelebenszyklus mit dem Lebenszyklus biologischer Systeme verglich. Er beschrieb, wie Softwaresysteme aus einer Idee geboren werden, langsam heranwachsen, ihre Kinderkrankheiten überwinden, irgendwann einmal stabil und leistungsfähig werden, dann aber durch die Last der vielen Anpassungen immer komplexer und schwerfälliger werden, bis sie eines Tages ihre Aufgaben nicht länger erfüllen können [RaUs90].

Damit sind wir bei den Evolutionsgesetzen von Belady und Lehman angelangt [LeBe85]. Durch ihre Untersuchung der OS/360-Entwicklung und einiger anderer Großsysteme sind sie zu dem Schluss gekommen, dass die Evolution solcher Systeme gewissen Gesetzmäßigkeiten unterliegt. Es hat sich nämlich gezeigt, dass große Sprünge mit komplexen Softwaresystemen nicht möglich sind. Ein komplexes, mehrschichtiges System lässt sich nur stufenweise in kleinen Schritten weiterentwickeln. Jede Stufe bringt einen neuen Kenntnisstand im gemeinsamen

Lernprozess der Entwickler und Anwender. Jeder neue Kenntnisstand dient als Basis für die nächste Entwicklungsstufe. Infolgedessen müssen die Stufen klein und gleichmäßig sein, und es ist erforderlich, die letzte abzusichern, ehe man die nächste angeht. Ein überstürztes Vorgehen kann zum Zusammenbruch aller bisher erreichten Stufen führen.

Aus der Studie der Evolution großer Softwaresysteme sind die folgenden fünf Gesetze hervorgegangen:

Das Gesetz der fortdauernden Änderung

Nutzbare Software spiegelt Abläufe und Objekte der realen Welt wider. Sobald diese sich verändern, muss sich auch die Software verändern, um mit ihnen im Einklang zu bleiben. Software, die stehen bleibt oder sich langsamer wandelt als die reale Welt, wird zunehmend nutzloser.

Das Gesetz der zunehmenden Komplexität

Am Anfang stimmt der Inhalt eines Softwaresystems, also seine Funktionalität mit seiner Form, also Architektur überein. Form und Inhalt passen zueinander. Jede Veränderung der Funktionalität vergrößert den Abstand zwischen Form und Inhalt. Mit zunehmendem Inhalt und gleich bleibender Form nimmt die Komplexität des Systems zu. Mit jeder neuen Entwicklungsstufe wird das Erreichen der nächsten deshalb schwieriger.

Das Gesetz der abnehmenden Qualität

Da jede neue Entwicklungsstufe die Kluft zwischen der Form und dem Inhalt vergrößert, sinkt die Qualität des Systems. Es entstehen immer mehr Widersprüche zwischen den alten und den neuen Anforderungen, und es werden immer mehr Kompromisse eingegangen. Dies führt zur Systemerosion und damit zu Qualitätsproblemen.

Das Gesetz des gebremsten Wachstums

Die zunehmende Komplexität, gekoppelt mit der abnehmenden Qualität, bremst die Weiterentwicklung und bringt sie irgendwann zum Stehen. Das heißt, dass die Weiterentwicklungsstufen immer kleiner und die Release-Intervalle immer größer werden, um mit der steigenden Komplexität und der sinkenden Qualität fertig zu werden.

Das Gesetz der abnehmenden Produktivität

Um nach mehreren Entwicklungsstufen überhaupt noch weiterentwickeln zu können, muss die inhaltliche Weiterentwicklung gestoppt werden, um die Form wieder anzupassen. Das System muss dafür einem Reengineering unterworfen

werden. Je mehr Kapazität durch Reengineeringmaßnahmen gebunden ist, umso weniger bleibt für die inhaltliche Weiterentwicklung. Also nimmt die Produktivität ab. Zum Schluss ist man nur noch mit der Anpassung der Form beschäftigt und verliert den Anschluss an die abzubildende Fachlichkeit. Die Nützlichkeit des Systems nimmt ab, und der Bedarf nach Ablösung wird immer stärker.

Nach diesen Gesetzen ist die Änderungsrate dafür ausschlaggebend, wie lange ein Anwendungssystem leben kann. Das hat Lehman dazu gebracht, Systeme in drei Kategorien einzuteilen:

- **S-Systeme** (Wegwerfsysteme)
- **P-Systeme** (statische Systeme)
- **E-Systeme** (dynamische Systeme) [BeLe76]

S-Systeme (Abb. 1–10) sind Systeme, die sich kaum ändern. Sie werden spezifiziert, um ein bestimmtes Problem in Zeit und Raum zu lösen. Wenn die Lösung nicht mehr zum Problem passt, wird sie weggeworfen und eine neue ausgearbeitet. Man spricht deshalb von Wegwerfsoftware. Für kleine Systeme ist es oft billiger, sie wegzuwerfen und neu zu bauen, als sie zu erhalten und weiterzuentwickeln. Für große Systeme kommt dieser Ansatz wegen der Höhe der Investition nicht in Frage. Da Entwickler von kleinen Systemen in diesem Modus arbeiten und in dieser Kategorie denken, haben sie für die Problematik der Systemerhaltung und -evolution wenig Verständnis [Demi81].

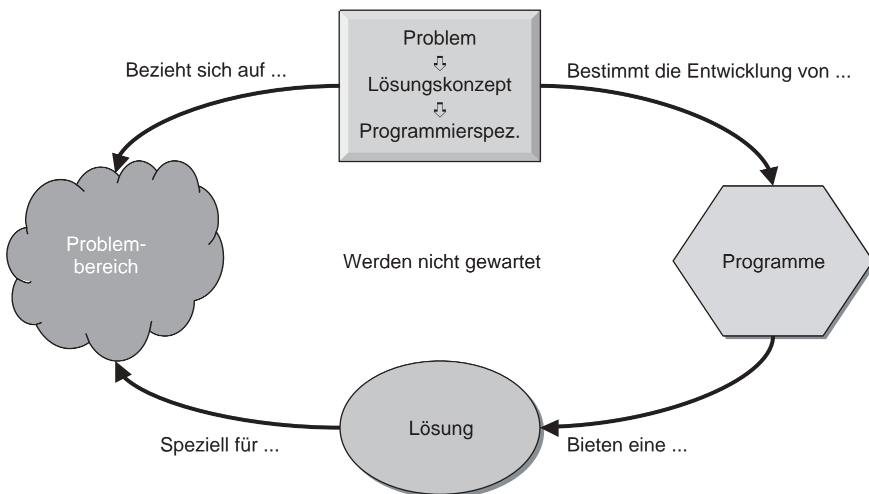

Abb. 1–10 *Wegwerfsysteme*

P-Systeme (Abb. 1–11) sind Systeme mit einer geringen Änderungsrate. Sie wachsen weniger als 10 % jährlich [Lehm80A]. Da sie sich wenig ändern, sind auch die Release-Intervalle länger und die Entwicklungsstufen kleiner. Das lässt Zeit für die Anpassung der Form an die Funktion. Auch wenn dies nicht gemacht wird, ist

der Erosionsprozess langsamer, so dass solche Systeme länger leben. Man spricht hier auch von statischen Systemen. Statische Systeme bilden statische Prozesse ab. In der betriebswirtschaftlichen Welt sind das in der Regel die Back-Office-Anwendungen. Solche Anwendungen sind deshalb die besten Kandidaten für Standardlösungen.

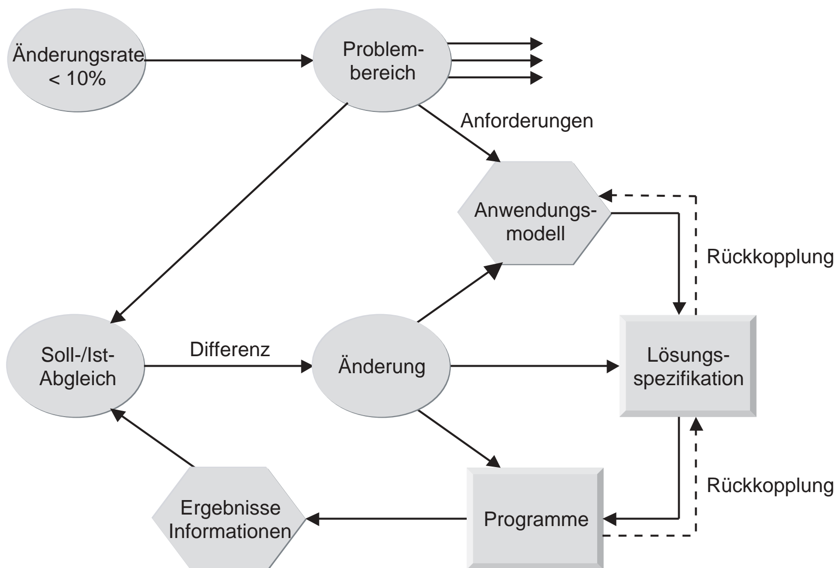

Abb. 1–11 *Statische Systeme*

E-Systeme (Abb. 1–12) sind Systeme mit einer hohen Änderungsrate. Sie wachsen mehr als 10 % jährlich [Lehm80A], weshalb die Release-Intervalle kürzer sein müssen. Außerdem ist es schwer, die Weiterentwicklung zu planen, weil auch die Ziele sich verschieben. Man läuft der sich fortbewegenden Umgebung ständig nach. Dies beschleunigt natürlich den Erosionsprozess und verkürzt die Lebensdauer der Software. Um dagegen zu wirken, ist man gezwungen, die Form dieser Systeme ständig anzupassen, was viel Kapazität verschlingt. Solche Systeme werden als dynamische Systeme bezeichnet. Ihre Erhaltung und Evolution setzt eine umfangreiche Infrastruktur voraus und bindet viele Ressourcen. Nur große Anwender mit viel Kapital können sich solche Systeme leisten. In der Betriebswirtschaft fallen die Front-Office-Anwendungssysteme meistens in diese Kategorie.

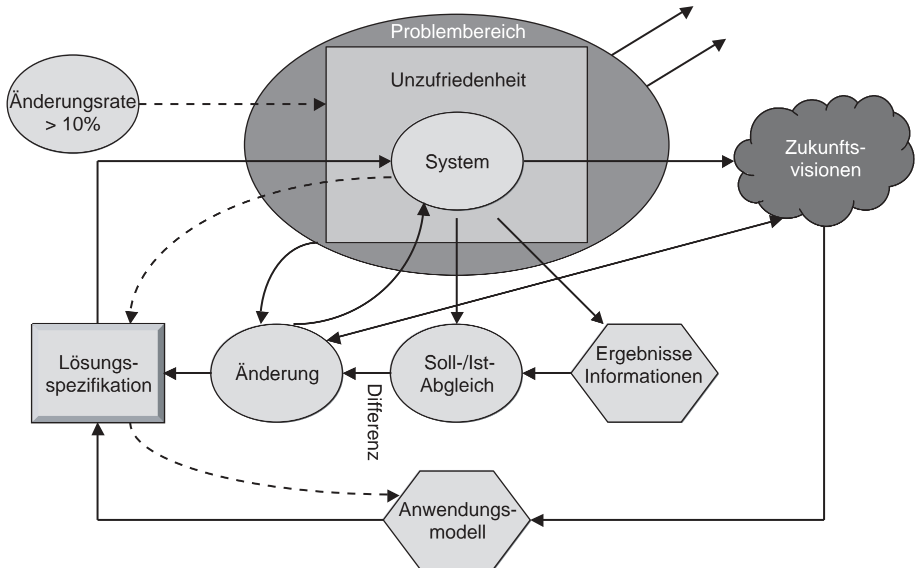

Abb. 1–12 *Dynamische Systeme*

1.5.4 Die Verbindung von Systemerhaltung und Evolution

Es wurde sehr früh erkannt, dass es schwierig ist, Systemerhaltung und Systemevolution auseinander zu halten. Deshalb wurde das *Journal of Software Maintenance* in *Journal of Software Maintenance and Evolution* umgetauft. Andererseits ist es aus betriebswirtschaftlichen Gründen wichtig, zwischen Erhaltung und Erweiterung zu unterscheiden. Erhaltung impliziert die Beibehaltung des Status quo [Guim83]. Erweiterung deutet auf einen Wertzuwachs hin. Ein System, das 1 Million Euro gekostet hat, müsste nach einer Erweiterung um 10 % 1,1 Millionen Euro kosten. Die Erhaltung des Systems kostet wiederum einen proportionalen Anteil des gegenwärtigen Wertes, zum Beispiel 15 % jährlich vom augenblicklichen Verkaufswert. Ein Produktmanager muss wissen, welche Kosten auf das Konto der Systemerhaltung und welche auf das Konto der Systemerweiterung zu buchen sind.

Ansonsten sind die Arbeiten am System selbst, ob Mängelkorrektur, Änderung oder funktionale Erweiterung, alle ähnlich. Sie unterscheiden sich nur in Nuancen. Es ist sogar schwierig, zwischen einer Änderung und einer Mängelkorrektur zu unterscheiden. Gemeinsam haben sie, dass die Arbeit stets von der bestehenden Software – von Code, Dokumenten und Testfällen – ausgeht und durch sie eingeschränkt ist. Es wird etwas geändert, gelöscht oder hinzugefügt. Danach wird die geänderte Software mit den alten sowie den neuen Anforderungen abgeglichen. Nur größere Erweiterungen, die zu zusätzlichen Komponenten,

Schnittstellen oder Datenbanktabellen führen, können eindeutig als Weiterentwicklung gekennzeichnet werden.

In Anbetracht dieser engen Verflechtung zwischen Systemerhaltungsmaßnahmen und Systemerweiterungsmaßnahmen wurde der Untertitel »Wartung und Weiterentwicklung bestehender Anwendungssyteme« gewählt. Das Produktmanagement ist für beide Aktivitäten gleichermaßen verantwortlich. Es ist das Produkt, also das Objekt der Arbeit, das sie miteinander verbindet, und es ist das Produktmanagement, das sie organisiert, plant, steuert und kontrolliert.