1 Einführung in die Softwareevolution

1.1 Wartung und Evolution – eine Begriffsbestimmung

Im Jahre 1981 definierte der US-amerikanische Rechnungshof Softwarewartung als alle Arbeiten an einem Softwareprodukt nach dem ersten produktiven Einsatz [GAO81]. Der Rechnungshof sah sich aus budgetären Gründen genötigt, eine Trennlinie zwischen Entwicklung und Wartung zu ziehen. Entwicklungskosten wurden als einmalige Projektkosten eingestuft. Wartungskosten wurden hingegen als operative, fortdauernde Kosten im Rahmen des Jahresbudgets des Anwenders angesehen. Es ging also um eine buchhalterische Unterscheidung zwischen Festkosten und einmaligen Kosten. Entwicklungskosten konnten jederzeit zurückgestellt werden. Wartungskosten mussten jedes Jahr eingeplant werden. So gesehen war die Unterscheidung zwischen Entwicklung und Wartung von Anfang an eine Frage der Kosteneinteilung.

Etwas später hat das U.S. National Bureau of Standards diese Definition in einer Special Publication aus dem Jahre 1983 verfeinert. Dort heißt es: »*Software maintenance is the performance of all 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 …*« [NBS83].

Der englische Begriff »Maintenance« dürfte in diesem Sinne mit dem deutschen Begriff »Wartung« nicht gleichgesetzt werden. »To maintain« heißt im Deutschen »erhalten« bzw. »in Stand halten«. Der Begriff Wartung impliziert Reparieren, was weniger ist als Erhalten. Erhaltung beinhaltet Wartung, ist aber nicht darauf beschränkt. Der englische Begriff »Maintenance« umfasst alle Aktivitäten, die erforderlich sind, um ein System in Betrieb zu halten, einschließlich Änderungen und Erweiterungen. Ein anderes NBS-Dokument geht noch weiter und beschreibt genau, welche Einzelaktivitäten unter dem Begriff »Maintenance« fallen – darunter Fehlerbehebung, Änderung, Erweiterung, Sanierung, Optimierung und die Managementtätigkeiten Change Management, Configuration Management, Testmanagement und Releasemanagement [NBS85] (siehe Abb. 1–1).

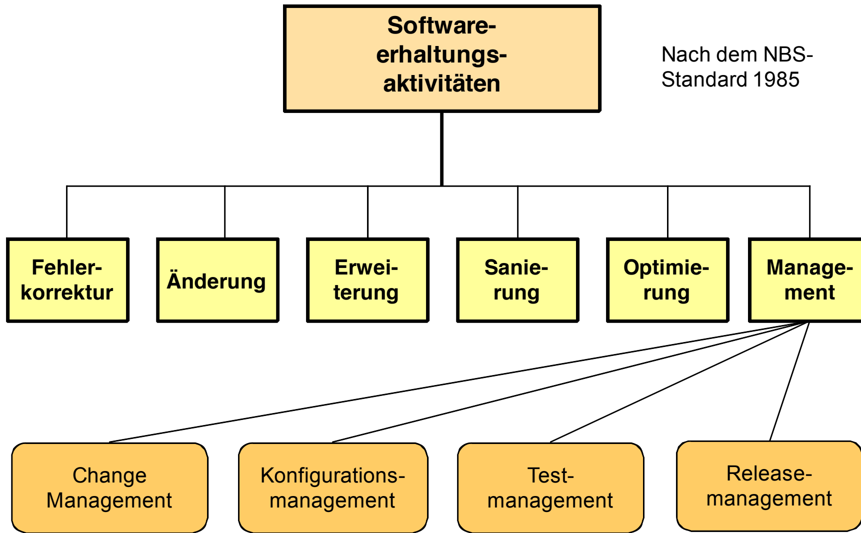

Abb. 1–1 *Softwareerhaltungsaktivitäten*

1.1.1 Zum Ursprung des Begriffes »Maintenance«

Der Begriff »*Software Maintenance*« geht zurück auf eine Studie der US-Luftwaffe aus dem Jahre 1970. Die Studie mit dem imposanten Titel »*A Study of Fundamental Factors underlying Software Maintenance Problems*« befasste sich mit der damals neuen Herausforderung, fertige Softwaresysteme im Betrieb zu erhalten. Zum ersten Mal wurden Projektauftragnehmer aufgefordert, die von ihnen gelieferten Programme so zu schreiben, dass fremde Programmierer mit ihnen umgehen konnten [Good71]. Es tauchten Begriffe wie Strukturierung, Modularisierung und Kommentierung auf, die den Erhalt der Software erleichtern sollten. Richard Canning, der Herausgeber der Fachzeitschrift »EDP Analyzer« nahm das Thema auf und – gestützt auf drei weiteren Berichten aus der US-Industrie – veröffentlichte er im Oktober 1972 einen Leitartikel zum Thema »The Maintenance Iceberg«. Darin stellte er Folgendes fest: »*Most business users of computers have come to recognize that about 50% of their programming expenditures goes to maintaining programs already in operation. This maintenance includes not only the correcting of errors in the programs but also enhancements and extensions. Since this is a rather steady type of expenditure, few people seem to get excited about it. But maintenance involves a lot more than just 50% of the programming expense. Data processing management is measured on how well and how rapidly user change requests are implemented …*« Daran ist zu erkennen, dass die Erhaltung bestehender Software recht früh als die Kehrseite der Softwareentwicklung erkannt wurde [Cann72].

1.1.2 Zum Unterschied zwischen Erhaltung und Entwicklung

Die Unterscheidung zwischen Entwicklung und Erhaltung folgte aus der Entscheidung des amerikanischen Rechnungshofs. Der Übergang von Entwicklung zu Erhaltung ist fließend. Irgendwann entscheidet der Abnehmer des Produkts, dass es möglich ist, das Produkt produktiv einzusetzen, auch wenn es nicht alle seine Wünsche abdeckt. Ab diesem Zeitpunkt hat das Produkt einen anderen Status. Es wird gewartet und weiterentwickelt [Schn87]. Ausschlaggebend ist die Erhaltung der Dienstleistung. Zuallererst muss das Produkt seine produktiven Aufgaben wahrnehmen. Nur an zweiter Stelle geht es um die Erweiterung des Produkts durch zusätzliche Aufgaben. Hierin liegt der Unterschied zur agilen Entwicklung, bei der es an erster Stelle um die Entwicklung neuer Eigenschaften geht, auch wenn Teile des Produkts bereits benutzt werden.

Die Entscheidung des Benutzers, ein Softwareprodukt abzunehmen, hängt vom Fertigstellungsgrad bzw. von der Qualität des Produkts ab. Beide müssen zumindest aus der Sicht des Benutzers klar definiert sein. In Projekten, bei denen die Produktentwicklung im Auftrag vergeben wird, müssen die Kundenanforderungen genau spezifiziert und dokumentiert sein. Inwieweit sie dann erfüllt sind, ist der Stoff zahlreicher Gerichtsprozesse. Es hat sich nämlich herausgestellt, dass es keineswegs einfach und oft sogar unmöglich ist, ein komplexes Softwaresystem im Voraus vollständig zu spezifizieren. Dies wird als Grund für die agile Entwicklung angeführt und ist auch der Grund, warum Gerichtsprozesse in der IT fast immer zuungunsten der Auftraggeber ausgehen. Der Auftragnehmer kann leicht nachweisen, dass die Spezifikation unzulänglich ist. Juristisch gesehen sind Softwaresysteme so gut wie nie fertig. Weil es so schwer festzustellen ist, wann ein Softwareprodukt ausreichend fertig ist, wird die Fortentwicklung auch nach der Freigabe im Rahmen der sogenannten Wartung fortgesetzt, allerdings unter anderen Vorzeichen als zuvor. Die Kontinuität der Dienstleistung hat den absoluten Vorrang. In dem ersten IEEE-Tutorial zum Thema Software Maintenance beschreibt Nicholas Zwegintzov die Systemerhaltung als eine Schleife nach der Erstentwicklung, die so lange wiederholt wird, wie das System noch brauchbar ist [PaZv79] (siehe Abb. 1–2).

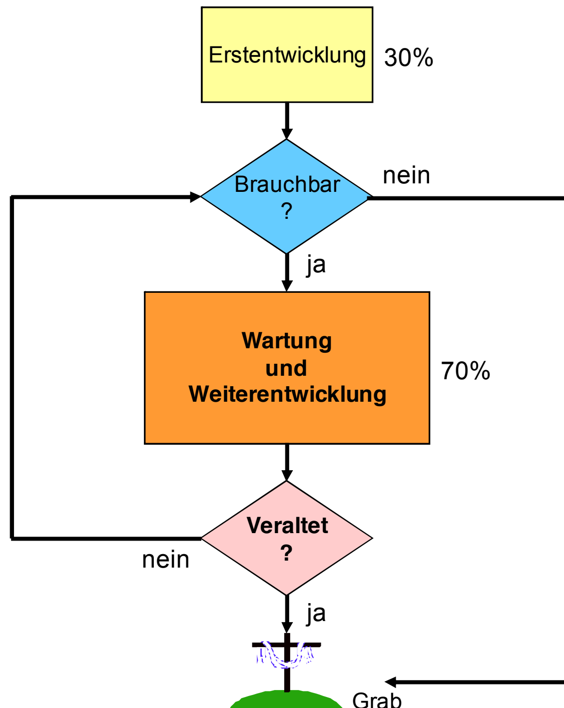

Abb. 1–2 *Das Leben eines Softwareprodukts*

1.1.3 Zum Unterschied zwischen Erhaltung und Evolution

Die Schwierigkeit, Fortentwicklung und Wartung voneinander zu trennen, hat die Informatikwissenschaft veranlasst, einen neuen Begriff einzuführen – Evolution. Dieser Begriff wurde schon von Belady und Lehman in den 70er-Jahren benutzt, um die Entwicklung des OS-370-Betriebssystems bei IBM zu beschreiben. Die beiden Forscher haben damals festgestellt, dass ein komplexes Softwareprodukt wie ein Betriebssystem nie wirklich abgeschlossen ist. Es wird fortdauernd entwickelt. Lehman benutzte – in Anlehnung an die Darwin'schen Evolutionslehre – den Begriff »Evolution«, um diesen Zustand zu bezeichnen [Bela79]. Etwas später, in den 90er-Jahren, als auch die Anwendungssysteme immer komplexer wurden, entschied der Herausgeber der Fachzeitschrift »Journal of Software Maintenance«, die Zeitschrift in »Journal of Software Maintenance and Evolution« umzubenennen [ChCi01]. Er wollte damit den Unterschied zwischen Evolution und Maintenance unterstreichen. Maintenance sei die Erhaltung eines Produkts, Evolution bedeutet die Weiterentwicklung. Demnach sind Fehlerbehebungen und Änderungen zu der bestehenden Funktionalität als »Maintenance« zu bezeichnen, während die Erweiterung der Funktionalität und die Steigerung der Qualität in

Form von Reengineering bzw. Refactoring-Maßnahmen als »Evolution« gelten. Softwareevolution kann man in diesem Sinne als permanente Nachbesserung betrachten [Chap01] (siehe Abb. 1–3).

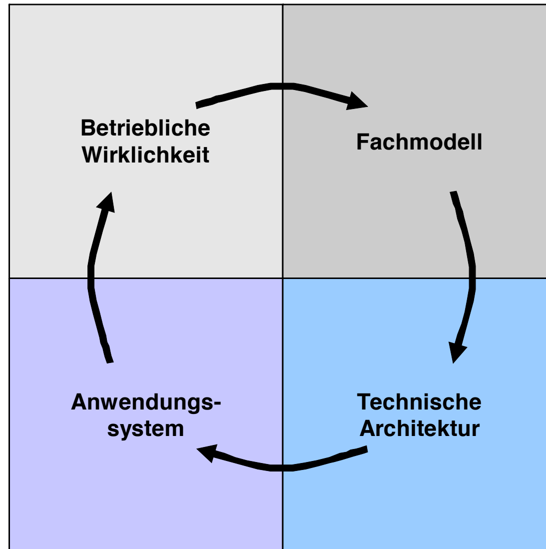

Abb. 1–3 *Evolution als permanente Nachbesserung*

Das erste Wartungszyklusmodell hat Girish Parikh vorgeschlagen. Parikh, der aus Indien Anfang der 70er-Jahre nach Amerika kam, wurde sofort mit der Wartung alter COBOL-Programme beauftragt. Parikh machte das Beste daraus und begann darüber zu schreiben. Sein erstes Buch »Handbook of Software Maintenance« erschien im Jahre 1982 [Pari82]. Als Inder hatte Parikh ein besonderes Verhältnis zur Software Maintenance, weil – wie er behauptet – die Idee des Lebenszyklus in der Hindu-Religion verankert ist. Dort herrscht nicht nur ein Gott – der Gott der Schöpfung –, sondern gleich drei: Brahma, der Gott der Schöpfung, Vishnu, der Gott der Erhaltung, und Shiva, der Gott der Wiedergeburt [Pari82]. Evolution, sprich Veränderung, gehört zur Systemerhaltung, denn ohne Veränderung kann weder ein Mensch noch ein Softwaresystem überleben. Für Parikh war der Begriff »Evolution« unzertrennlich mit dem Begriff »Maintenance« verbunden [Pari87] (siehe Abb. 1–4).

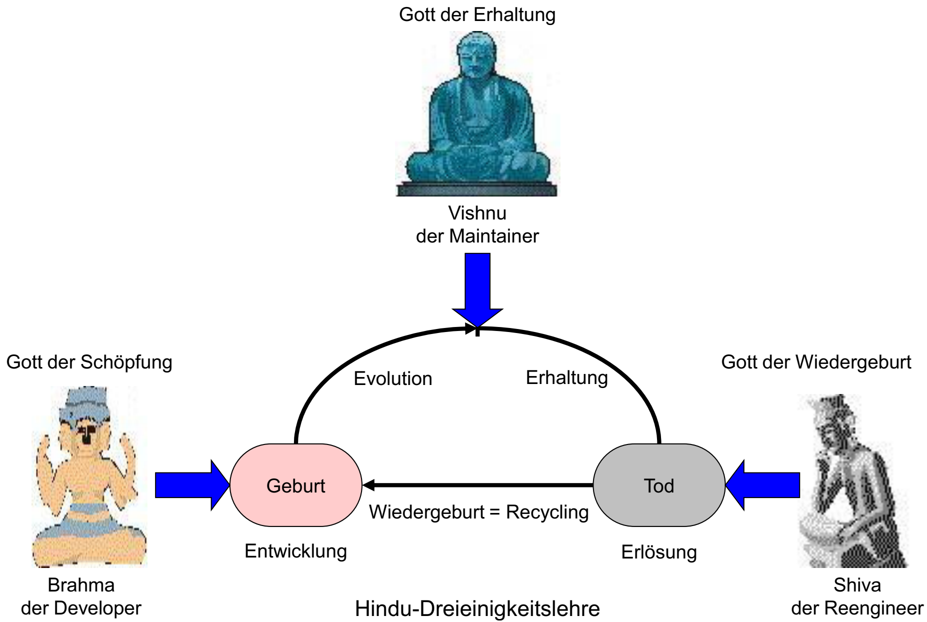

Abb. 1–4 *Das Urmodell des Softwarelebenszyklus*

Keith Bennett und Vaclav Rajlich haben im Jahre 2000 ein Phasenmodell veröffentlicht, in dem sie Evolution von Entwicklung und Erhaltung trennen [BeRa00]. Zunächst wird ein Produkt bis zu einer gewissen Reife entwickelt, dann geht es in die Phase der Evolution (Weiterentwicklung) über. Erst wenn es sein Wachstum beendet hat, tritt das Produkt in die Phase der Erhaltung ein. In der Erhaltung wird es nur noch korrigiert und geändert (siehe Abb. 1–6). Irgendwann einmal geht das auch nicht mehr und das Produkt wird abgelöst bzw. ausgemustert. Problematisch in diesem Phasenkonzept sind die Übergänge von einer Phase in die andere. Hier stellen sich die Fragen, wann ist ein Produkt reif genug, um aus der Entwicklung in die Evolution zu treten, und wann ist es stabil genug, um aus der Evolution in die Erhaltung überzugehen. Diese Fragen konnten bisher nur andeutungsweise beantwortet werden [Lehm98].

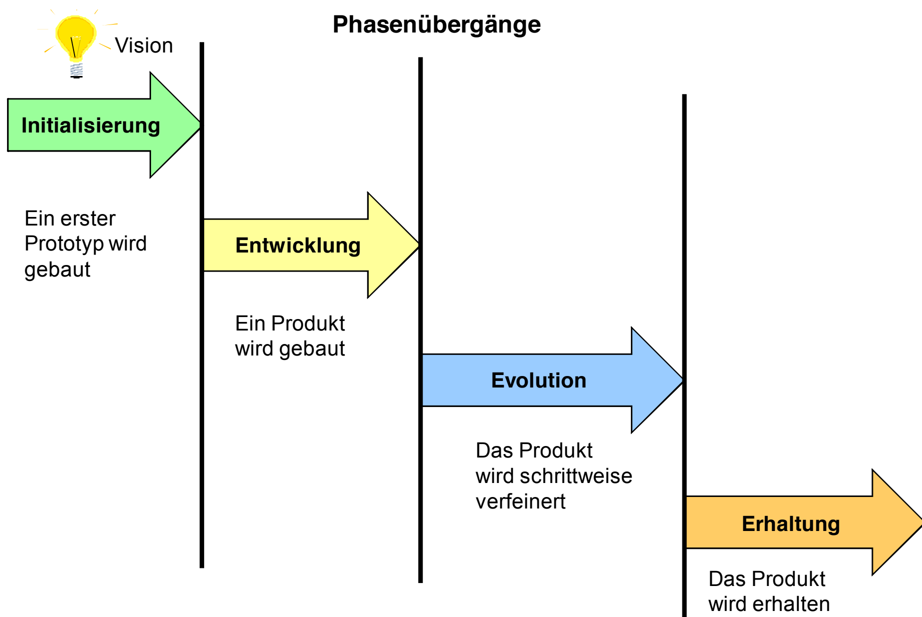

Abb. 1–5 *Phasen im Leben eines Softwareprodukts*

Bei der »*International Conference on Software Maintenance*« kam es immer wieder zur Diskussion um den Titel der Konferenz. Einige Mitglieder des Steuerungsausschusses schlugen vor, die Konferenz in »*Conference on Software Maintenance and Evolution*« umzubenennen. Andere haben sich dagegengestellt. Ihr Argument war, dass Maintenance schon immer Evolution mit beinhaltet hatte. Die Unterscheidung zwischen Arbeiten vor und nach der ersten Freigabe diente dazu, Maintenance und Evolution als eine gemeinsame Phase zu definieren. Es bestehe daher kein Anlass, die zwei Begriffe zu trennen.

Die Amerikaner, allen voran der Präsident der »American Maintenance Association« Nikolaus Zwegintzov, hatten kein Problem mit dem Begriff »Maintenance«. Es waren hauptsächlich die Europäer, die darauf drängten, zwischen Maintenance und Evolution zu unterscheiden. Ihrer Meinung nach bezieht sich Maintenance auf die Erhaltung des gegenwärtigen Werts eines Systems, Evolution deutet hingegen auf eine Wertsteigerung hin. Durch zusätzliche Funktionalität und/oder Qualität steigt der Wert eines Systems. Demnach gehören Korrekturen und Änderungen zur Erhaltung, Erweiterungen und Verbesserungen bzw. Sanierungen zur Evolution [RaWB01]. In diesem Buch werden beide Tätigkeiten unter dem umfassenderen Begriff »Evolution« zusammengefasst.

1.1.4 Zum Unterschied zwischen Änderung und Erweiterung

Der Unterschied zwischen Änderung und Erweiterung ist für den Laien nicht ohne Weiteres zu erkennen. Ist eine Erweiterung nicht auch eine Änderung? Die Antwort ist Ja und Nein. Es hängt davon ab, wie man ein Softwareprodukt betrachtet. Wenn man das Produkt als Ganzes sieht, ist eine Erweiterung bzw. ein neuer Codebaustein eine Veränderung des Ganzen. Wenn man das Produkt jedoch als eine Aggregation einzelner Bausteine betrachtet, dann ist das Hinzufügen eines neuen Bausteins etwas anderes als die Änderung eines bestehenden Bausteins. Außerdem, mit dem Hinzufügen neuer Bausteine, sprich Funktionalität, steigt der Wert eines Produkts. Mit 1.000 Modulen hat die Software vielleicht einen Wert von 100.000€. Mit 1.100 Modulen steigt der Wert um 10% auf 110.000€. Jedes neue Modul bringt neue Funktionalität und Funktionalität hat einen monetären Wert. Erweiterung erhöht den Wert des Produkts. Deshalb müssten in Verträgen mit externen Partnern Erweiterungen extra bezahlt werden, während Änderungen neben Korrekturen durch die sogenannte Wartungsgebühr abgedeckt sind [KeSl99].

1.1.5 Zum Unterschied zwischen Korrektur und Sanierung

Die gleiche Unterscheidung wie zwischen Änderung und Erweiterung gilt auch zwischen Korrektur und Sanierung. Eine Korrektur bezieht sich auf die Beseitigung einer Abweichung zwischen Ist und Soll. Das System sollte einen Preis von 150€ berechnen, kommt aber auf einen Preis von 155€. Ergo handelt es ich um einen Fehler. Das System verhält sich nicht so, wie es sich verhalten sollte. Anders ist es, wenn der Code des Systems schwer änderbar ist. Ein Wartungsprogrammierer braucht zwei Tage, um ein paar neue Anweisungen in einen alten Codebaustein einzufügen. Durch eine Überarbeitung des besagten Codebausteins kann dieser Änderungsaufwand auf einen Tag reduziert werden. Man könnte behaupten, der Code hätte vom Anfang an besser geschrieben werden sollen und sei deshalb fehlerhaft. Dennoch, mit diesem Fehler kann der Anwender leben, mit dem fehlerhaften Preis jedoch nicht. Bei der Korrektur werden Fehler im Verhalten der Software behoben. Bei der Sanierung werden Fehler in der Konstruktion behoben [Snee84]. Das System ist zu langsam, zu unsicher oder zu schwer zu pflegen, aber es liefert richtige Ergebnisse. Demnach ist Sanierung die Beseitigung von Konstruktionsmängeln, während Korrektur (Fehlerbehebung) die Beseitigung von Funktionsmängeln ist. Korrektur ist als Muss, Sanierung als Kann zu betrachten. Mit einer Sanierung steigt die Qualität des Systems. Mit einer Korrektur erlangt es den Wert, den es schon immer haben sollte. Ergo gehört eine Sanierung zur gleichen Kategorie wie eine Erweiterung, nämlich unter den Begriff »Evolution«. Korrekturen und Änderungen gehören hingegen unter den Begriff »Wartung«. Die Unterscheidung zwischen diesen beiden Begriffen ist letztendlich eine Frage der Wirtschaftlichkeitsbetrachtung [Hube97].

1.2 Iterative und evolutionäre Softwareentwicklung

Manche sehen in Softwareevolution eine Fortsetzung der evolutionären Softwareentwicklung. Der Begriff evolutionäre Entwicklung wurde von Tom Gilb in den 80er-Jahren geprägt [Gilb88]. Danach werden komplexe Softwareentwicklungsprojekte in mehrere aufeinander folgende Projekte aufgeteilt. In dem ersten Projekt werden die Kernanforderungen des Benutzers umgesetzt und ein Rumpfprodukt übergeben. In den darauffolgenden Projekten werden weitere Anforderungen, die sich aus der Nutzung des Produkts ergeben, erfüllt. Gleichzeitig wird das Produkt durch Nachbesserungen stabilisiert und optimiert. Nach einiger Zeit wird es auch saniert. Das heißt, die Zahl der unterschiedlichen Tätigkeiten nimmt immer weiter zu (siehe Abb. 1–6). Da Anwender immer neue Anforderungen stellen, ist die evolutionäre Entwicklung eines komplexen Systems im Prinzip nie zu Ende, aber die Benutzer haben immer ein Produkt, mit dem sie arbeiten können [Wood99]. In dieser Hinsicht ist die evolutionäre Softwareentwicklung als Vorgänger der agilen Entwicklung zu betrachten.

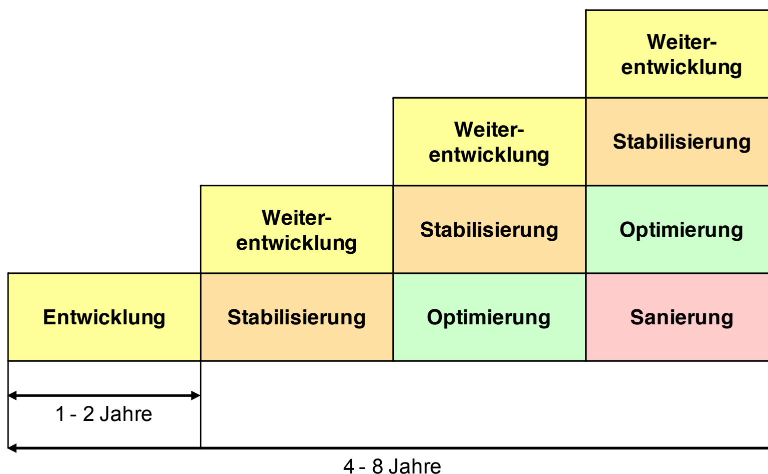

Abb. 1–6 *Evolutionäre Softwareentwicklung nach Tom Gilb [Gilb85]*

Etwa zur gleichen Zeit, als die evolutionäre Softwareentwicklung vom Gilb propagiert wurde, entstand bei der IBM die iterative Softwareentwicklung [Soti01]. Danach werden zu Beginn eines Projekts sämtliche Benutzeranforderungen erhoben und dokumentiert. In den darauffolgenden Teilprojekten werden Teilmengen dieser Anforderungen so lange implementiert, bis alle Anforderungen abgearbeitet sind. Da keine neuen Anforderungen angenommen werden, ist das Gesamtprojekt irgendwann einmal zu Ende. Das heißt, es wird zuerst der komplette Funktionsumfang festgelegt und danach Stück für Stück implementiert. Das Problem mit diesem Ansatz ist, dass er davon ausgeht, ein Anwender sei am Anfang

in der Lage, alle seine Anforderungen zu erkennen – dies ist aber nur bei trivialen Projekten der Fall. In der heutigen komplexen IT-Welt wissen die meisten Anwender kaum, was sie verlangen sollen. Sie müssen deshalb beginnen, ein System zu bauen, um zu entdecken, welches System sie überhaupt bauen sollen. Die iterative Entwicklung drückt sich in dem spiralen Modell von Boehm aus dem Jahre 1987 aus, wobei das System ursprünglich im vollen Umfang spezifiziert wird und anschließend nach und nach implementiert wird [Boeh88] (siehe Abb. 1–7).

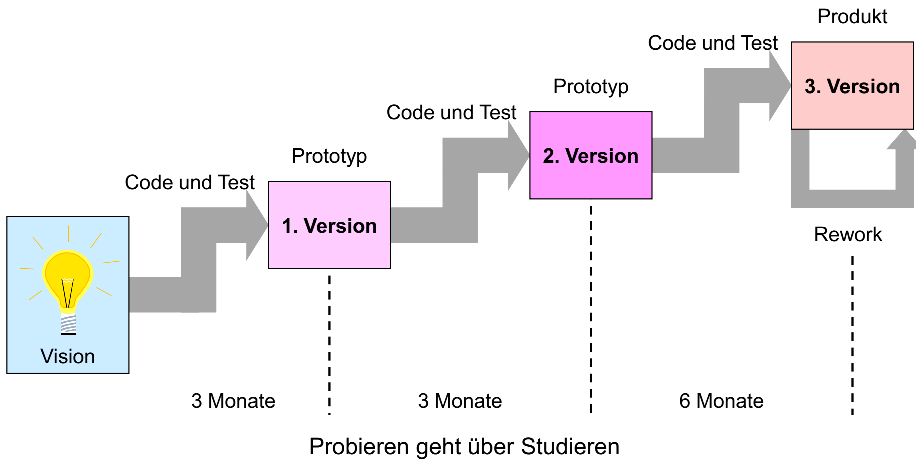

Abb. 1–7 *Inkrementelle Softwareentwicklung*

1.3 Softwareevolution und agile Softwareentwicklung

Die volle Funktionalität und die dazu passende Qualität eines komplexen IT-Systems vorauszuplanen ist in der Tat schwierig, wenn nicht gar unmöglich. Daher der Ruf nach einer agilen Entwicklung, die es den Anwendern erlaubt, ihre Anforderungen ständig zu verändern. Wichtig ist, dass sie stets ein laufähiges Produkt haben, das zumindest einen Teil ihrer bisherigen Anforderungen abdeckt [ZhPa11]. Wenn ein Produkt auf diese Art entwickelt wird, ist es auch schwer, den Zeitpunkt zu bestimmen, an dem die Entwicklung aufhört und die Evolution beginnt. Der amerikanische Rechnungshof legte fest, dass dieser Zeitpunkt die erste Freigabe eines Teilprodukts sei. Davor liegt die Entwicklung, danach die Wartung bzw. Evolution. Demnach wären die heutigen agilen Entwicklungsprojekte bis auf die ersten Monate alle Evolutionsprojekte. Der Anwender bekommt gleich zu Beginn des Projekts ein funktionsfähiges Teilprodukt und kann damit arbeiten. Das, was schon im Einsatz ist, wird gewartet und weiterentwickelt, während neue Teile des gleichen Produkts erst entwickelt werden. So gesehen ist ein agiles Projekt drei Projekte in einem:

- ein Entwicklungsprojekt, bei dem neue Komponente gebaut werden,
- ein Evolutionsprojekt, bei dem bestehende Komponente weiter ausgebaut und saniert werden, und
- ein Wartungsprojekt, bei dem bestehende Komponenten korrigiert und angepasst werden [PoPo06].

Es wird immer schwieriger, zwischen den verschiedenen Tätigkeiten zu unterscheiden. Man ist versucht, wie das bei agilen Projekten der Fall ist, alles zusammenzufassen und als eine endlose Entwicklung zu bezeichnen. Dies hat aber weitreichende Folgen für die Budgetierung von IT-Projekten. Wer wagt schon vorherzusagen, was ein solches Entwicklungsprojekt insgesamt kosten wird, wenn der Projektplaner allenfalls die Anforderungen für die nächste Entwicklungsstufe zu sehen bekommt. Der Rest des Projekts liegt verborgen hinter dem Horizont. Es liegt nahe, nicht das Ganze zu kalkulieren, sondern immer nur das nächste Release. In diesem Fall ist ein agiles Entwicklungsprojekt gleich einem Evolutionsprojekt. Entwicklung und Evolution werden auf die gleiche Art und Weise budgetiert. Nur für die Wartung bzw. Erhaltung der bereits im Einsatz befindlichen Komponenten gibt es ein separates Budget und eventuell ein separates Team [OGPM12].

Der auffälligste Unterschied zwischen Softwareevolution und agiler Entwicklung ist das Release-Intervall. Iterationen in einer agilen Entwicklung dauern zwischen zwei und sechs Wochen. In der Softwareevolution liegen die Release-Intervalle zwischen einem Monat und einem Jahr. Das Tempo der Veränderung ist ausschlaggebend. Im Laufe einer agilen Entwicklung wächst der Umfang des Produkts gleichmäßig von Release zu Release. Irgendwann wird aber das Ausmaß der Produktänderung geringer, und zwar in dem Maße, wie die Benutzeranforderungen erfüllt sind. Das Wachstum verlangsamt sich, die Release-Intervalle werden länger und die agile Entwicklung wird immer mehr zu einer Softwareevolution. Irgendwann geht sie ganz in die Softwareevolution über. Man kann hier von einer Konvergenz von Entwicklung und Evolution sprechen (siehe Abb. 1–8).

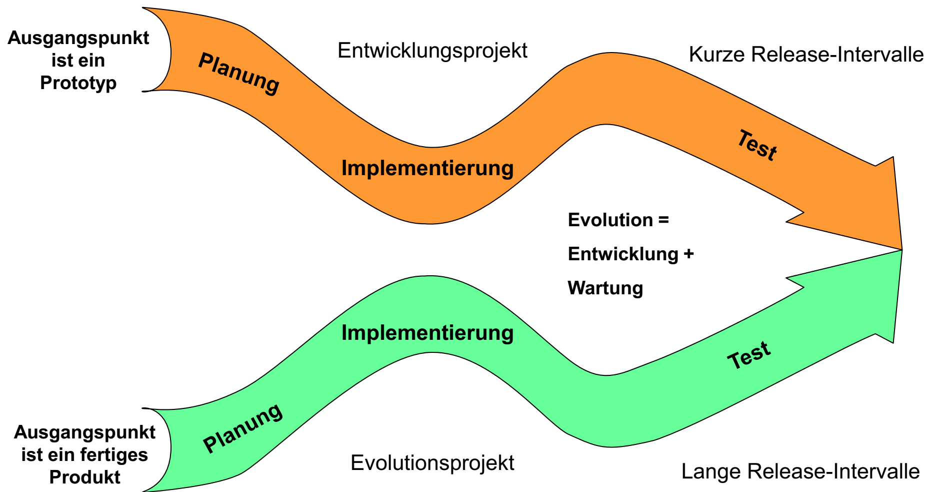

Abb. 1–8 *Konvergenz von Entwicklung und Evolution*

Wer zwischen Entwicklung, Evolution und Wartung unterscheiden will, muss letztendlich auf das Maß für den Grad der Größenänderung zurückgreifen. Im Falle der Wartung, sprich Korrektur und Änderung, dürfen sich die Anzahl der Systemelemente nicht und die Größe der Bausteine nur geringfügig ändern. Der Funktionsumfang bleibt konstant. Wartung ist Werterhaltung, nicht Wertsteigerung. Im Falle der Evolution darf sich der Funktionsumfang bzw. die Anzahl der Anwendungsfälle und deren Schritte in einem Release nur geringfügig ändern, d.h. unter 10 % bleiben. Sollte die Größe eines Systems mehr als 10 % in einem Release steigen, dann handelt es ich um eine evolutionäre Entwicklung bzw. um ein agiles Entwicklungsprojekt. Diese Unterscheidung wird manchen als Haarspalterei erscheinen, aber für die Kostenträger eines Produkts ist es wichtig. Wartung und Evolution werden als operative Festkosten erfasst, während eine Entwicklung einmalige Projektkosten verursacht [MGRH10].

Bei der Entwicklung neuer Systeme spielen agile Entwicklungsmethoden eine zunehmend große Rolle. Man hat erkannt, dass komplexe Softwaresysteme auf Anhieb nicht zu verwirklichen sind. Das Produkt entsteht stufenweise aufgrund immer neuer oder anderer Anforderungen. Ein agiles Projekt ist so gesehen ein Lernprozess. Der Benutzer, der das Projekt steuert, lernt aus der Erfahrung mit den bisherigen Produktstufen, was er bei den nächsten Stufen zu verlangen hat. Man könnte dies als Weiterentwicklung bezeichnen und die Vorgehensweise auf die Softwareevolution übertragen [Pfle98].

Es gibt jedoch wichtige Gründe, dies nicht zu tun. Zum Ersten ist bei einem agilen Entwicklungsprojekt das Produkt noch nicht voll im Einsatz [Mack00]. Es wird zwar als Prototyp getestet, gilt aber noch nicht als vollwertiges Produkt. Das

Projekt trägt noch keine Verantwortung für die Kontinuität der Dienstleistung. Zum Zweiten ist das Produkt die meiste Zeit nur bruchteilmäßig benutzbar. Der größte Teil ist noch im Bau. In einem Evolutionsprojekt handelt es sich um ein weitgehend fertiges Produkt, das zwar weiterentwickelt wird, aber nur geringfügig, d.h. < 10 %. Zum Dritten unterliegt ein Produkt in der Wartung einem strengen Change Management. Es geht also nicht, dass ein Vertreter der Anwender einfach eine Story erzählt und die Entwickler springen auf, sie zu implementieren. Die Kosten unterliegen einer strengen Kontrolle. Anwender müssen formale Änderungsanträge – Change Requests – stellen und diese müssen von einem »Change Control Board« nach einer sorgfältigen Wirtschaftlichkeitsprüfung genehmigt werden. Die Softwareevolution wird also strenger geführt [Snee00].

Schließlich liegt in einem agilen Entwicklungsprojekt der Fokus auch auf der Kreativität des Teams. Alles ist darauf ausgerichtet, die Kreativität der Beteiligten auszuschöpfen. Innovation ist das Ziel. In einem Evolutionsprojekt herrscht Ordnung und Disziplin. Alles dient der Erhaltung der bestehenden Ordnung. Das Ziel ist Stabilität. Vorschriften müssen verabschiedet und auch eingehalten werden, um das drohende Chaos zu verhindern. In dieser Welt herrschen die als typisch deutsch bezeichneten, ordnungstragenden Tugenden wie Disziplin, Ordnungsliebe und Aufmerksamkeit zum Detail, während in der agilen Entwicklung eher die ordnungsspringenden Tugenden wie Freiheit, Kreativität und Improvisation vorherrschen [Snee76].

In der Tat geht es hier um zwei entgegengesetzte Welten mit einer jeweils anderen Grundhaltung. Es wäre abwegig zu behaupten, die eine Welt sei der anderen Welt überlegen. Beide Welten haben ihre Berechtigung. Die Welt der agilen Entwicklung dient dazu, innovative Produkte in die Welt zu setzen. Die Welt der Softwareevolution dient dazu, diese Produkte in einer sich stets wandelnden Welt zu erhalten.

1.4 Wartung und Evolution in einer serviceorientierten IT-Welt

Beim Erscheinen dieses Buches steht die IT-Welt vor einem größeren Paradigmenwechsel. Neben dem Übergang von einer Silo-ähnlichen Applikationslandschaft in eine serviceorientierte Systemarchitektur findet der Einzug des Cloud Computing statt. Mit der Serviceorientierung wird die Nutzung vorgefertigter Softwarebausteine gefördert, die nicht statisch in der benutzereigenen Software eingebunden sind, sondern von einem zentralen Service-Repository über das Internet aufgerufen werden. Sie werden also dynamisch in das System des Anwenders eingebunden. Wichtig ist, dass die Entwickler einer neuen Applikation ihre Bausteine nicht selbst herstellen. Sie verwenden vorgefertigte Bausteine in Form von Webservices auf einem Servernetz [CoDe10]. Der Preis dafür ist, dass sie ihre Anwendungen um diese Bausteine herum bauen müssen. Ihre Freiheit ist dadurch eingeschränkt.

Die Bausteine, sprich Services, reichen von kleinen Subroutinen wie Kalenderfunktionen und Preisberechnungsalgorithmen bis hin zu kompletten betriebswirtschaftlichen Anwendungen wie Fertigungssteuerung und Lagerhaltung. Man spricht hier von der Granularitätsstufe. Je kleiner die Granularität, desto größer die Freiheit der Entwickler bei der Gestaltung ihrer Software, aber auch umso mehr müssen sie in die Integration und den Test der Bausteine investieren. Je größer die Granularität, desto geringer ist die Freiheit der Entwickler, die Systeme so zu gestalten, wie sie es wollen, aber umso weniger Aufwand müssen sie betreiben, die Bausteine zusammenzusetzen [Snee11].

In einer serviceorientierten Architektur sind die Bausteine bzw. Services im Besitz des Anwenderbetriebs. Der Anwender kann sie beliebig anpassen und ausbauen, ist aber auch für deren Wartung und Weiterentwicklung zuständig. Das kann teuer werden und die meisten Anwender sind damit überfordert. Sie haben weder das Know-how noch die Kapazität, solche allgemeingültigen, wiederverwendbaren Softwarebausteine zu konstruieren und zu erhalten. Die Zahl der Entwickler, die dazu in der Lage sind, ist sehr beschränkt. Deshalb werden die meisten Anwender es vorziehen, ihre Bausteine aus der Cloud zu beziehen. Dort werden sie von Serviceprovidern gegen eine fest vereinbarte Nutzungsgebühr bereitgestellt – also »Software on Demand« bzw. »Software as a Service«. Der große Vorteil des Cloud-Angebots ist, dass die Anwender ihre Softwarebausteine bzw. Services nicht selbst entwickeln müssen, und was noch wichtiger ist, sie nicht selbst warten und weiterentwickeln müssen. Die Verantwortung für die Evolution der Services liegt beim Serviceprovider. Dafür gibt es bereits entsprechende Vorschläge seitens des amerikanischen Software Engineering Institute [SmLe07].

Der Nachteil des Cloud-Angebots ist die Abhängigkeit. Der Anwender ist abhängig vom Provider, dass seine Services immer verfügbar sind und dass sie korrekt und effizient funktionieren. Eine weitere Abhängigkeit entsteht, wenn der Anwender nicht will, dass er bei jeder Nutzung eines Service seine Daten an den Server überträgt. Um dies zu vermeiden, muss er seine persistenten Daten an den Serviceprovider abgeben. Damit gerät er in eine doppelte Abhängigkeit. Einerseits ist er auf den Provider für seine Funktionalität und anderseits für die Pflege seiner Daten angewiesen [CGKM11]. Um sich abzusichern, wird der Anwender ein Service Level Agreement mit dem Provider abschließen müssen. Diese Vereinbarung wird ihm die Kontinuität der Dienstleistung einschließlich Wartung und Evolution der von ihm benutzten Services absichern, d.h., alles, was in diesem Buch behandelt wird, wird dem IT-Anwender in Zukunft als Dienstleistung angeboten [Scho11].

Diese Abhängigkeiten mögen für viele als unüberwindbare Hürde erscheinen. Dennoch weist Nicolas Carr mit Recht darauf hin, dass die Geschichte der Menschen im technologischen Zeitalter eine Geschichte der zunehmenden Abhängigkeit ist. Wenn das nicht so wäre, würden wir alle noch in Einsiedlerhöfen mit

eigener Strom- und Wasserversorgung leben. Wir sind in der Mehrzahl alle abhängig von einer zentralen Wasser-, Gas- und Kommunikationsversorgung. Wir haben allenfalls eine Wahl zwischen Lieferanten. Dennoch dreht sich die Welt weiter und wir sind von der Sorge um diese Services des täglichen Bedarfs befreit – zum Preis der Abhängigkeit [Carr09].

So wird es auch in der IT-Welt kommen. IT-Anwender können es sich nicht länger leisten, eigene Software zu entwickeln, zu warten und weiterzuentwickeln. Alle klagen über den Mangel an Fachkräften. Allein in Deutschland sollen über 40.000 IT-Fachkräfte fehlen. Die Regierung überlegt, wie sie noch mehr Spezialisten aus dem Ausland herlocken kann. Dies ist aber keine dauerhafte Lösung. Die Menge an Software nimmt exponentiell zu und verlangt immer mehr Pflegepersonal. Es bleibt dem IT-Anwender am Ende nichts anderes übrig, als sich in die Abhängigkeit der Softwareserviceprovider zu begeben. Das Gros der Software wird in Zukunft zentral bei den großen Providern gewartet und weiterentwickelt. Die Endanwender werden nur noch für die Entwicklung und Pflege ihrer Geschäftsprozessmodelle verantwortlich sein, von denen aus die Ablauffolge der Services gesteuert wird. Die Eigenentwicklung von Software auf der Ebene der bisherigen Programmiersprachen wie COBOL, C++, C# und Java wird aufhören. Diese Programmiersprachen sind immer noch zu nahe an der Maschine. Ihre Bausteine – die Klassen, Methoden und Attribute – sind zu winzig und davon gibt es zu viele. Schon mittelgroße Anwendungssysteme haben mehrere Hundert Klassen und mehrere Tausend Methoden. Das bekommen die durchschnittlichen IT-Anwender nicht mehr in den Griff. Sie sind schlicht überfordert. Sie brauchen Sprachen auf einer höheren semantischen Ebene mit größeren Bausteinen und weniger Beziehungen. Programmiert bzw. modelliert soll nur noch in abstrakten Modellierungssprachen wie BPMN, S-BPM und BPEL werden. Auch UML wird an Bedeutung verlieren weil sie viel zu nahe an der Programmiersprache ist [ErMG12].

Wenn das so ist, warum befassen wir uns mit der Evolution solcher Legacy-Systeme in diesem Buch? Die Antwort ist, dass die jetzigen Anwendungssysteme noch lange leben werden. Auch wenn weniger neue Systeme erstellt werden, bleiben die bestehenden Systeme noch lange in Betrieb und müssen gewartet und weiterentwickelt werden. Umso größer ist der Bedarf an Wartungstechnikern, deren Stärke darin liegt, bestehende Systeme zu erhalten. Sie sind in der Lage, sich in bestehenden Code einzuarbeiten, und können diesen Code korrigieren, ändern und notfalls sanieren. Diese Fähigkeiten werden so lange verlangt, bis das letzte prozedurale oder objektorientierte Anwendungssystem durch eine serviceorientierte Lösung abgelöst ist. Bis dahin bleibt die Evolution der alten Systeme ein wichtiges Thema.

1.5 Struktur und Inhalt der folgenden Kapitel

Die folgenden Kapitel sind nach den Tätigkeiten in einem Softwareevolutionsprozess gegliedert. Die allererste Aktivität ist die Wirtschaftlichkeitsbetrachtung. Sämtliche Wartungs- und Weiterentwicklungsvorhaben müssen auf ihre Wirtschaftlichkeit geprüft werden. Erst wenn sie diese Prüfung bestanden haben, dürfen sie zur Implementierung freigegeben werden. Als Nächstes folgt eine Erläuterung der Gesetze der Softwareevolution, die man verstehen muss, um den Evolutionsprozess überhaupt planen und steuern zu können. Danach folgt ein Überblick über den Gesamtprozess mit sämtlichen Aktivitäten, die im Laufe einer Evolution vorkommen. Der erste Schritt bei der Implementierung eines neuen Release bzw. in einer neuen Iteration ist die Analyse des bestehenden Produktzustands, und zwar sowohl auf der Codeebene als auch auf der Dokumentationsebene. Erst nachdem die Analyse des Istzustands abgeschlossen ist und die Daten über Größe, Komplexität und Qualität der Software vorliegen, kann die Planung der weiteren Evolutionsschritte stattfinden, die zu einem neuen Sollzustand führen sollte.

Nach der Planung des neuen Release teilt sich das Evolutionsprojekt in vier Aktivitäten, die nebeneinander betrieben werden. Die erste Aktivität ist die Fehlerbehebung, die zweite die Änderung, die dritte die Sanierung und die vierte die Weiterentwicklung. Jeder dieser Aktivitäten ist ein eigenes Kapitel gewidmet. Nach Abschluss dieser Aktivitäten müssen zwei weitere Aktivitäten folgen – der Regressionstest, um die Qualität des veränderten Systems zu sichern, und die Nachdokumentation, um den neuen Zustand des Systems für die künftigen Releases zu beschreiben. Diese beiden Themen werden getrennt behandelt.

Daraus ergibt sich folgende Gliederung:

1. Diese Einführung
2. Die Wirtschaftlichkeit der Softwareevolution
3. Die Gesetze der Softwareevolution
4. Die Beschreibung des Evolutionsprozesses
5. Die Analyse des bestehenden Produkts
6. Die Planung des nächsten Release
7. Die Behebung gemeldeter Fehler
8. Die Änderung der gegenwärtigen Funktionalität
9. Die Sanierung der Systemschwachstellen
10. Die Erweiterung des Systemfunktionalität
11. Der Regressionstest des veränderten Systems
12. Die Nachdokumentation der Änderungen

Damit sind die operativen Aufgaben eines Evolutionsprojekts weitestgehend abgedeckt. Nicht behandelt werden Managementaufgaben wie Problemmanagement, Change Management, Releasemanagement und Configuration Management. Diese Aktivitäten gehören zum Produktmanagement und werden in einem früheren Buch vom selben Verlag zu diesem Thema behandelt [SnHT04].