Harry M. Sneed · Richard Seidl

Software-evolution

Erhaltung und Fortschreibung bestehender Softwaresysteme

dpunkt.verlag

Harry M. Sneed arbeitet seit 2003 als Tester für die Firma ANECON Software Design und Beratung GmbH. Unter anderem wirkte er dort in einigen großen Testprojekten mit und war Qualitätssicherungsbeauftragter bei einer Bundesbehörde in Deutschland. Neben seiner Projektarbeit entwickelt er Testwerkzeuge für die Unterstützung einzelner Testaktivitäten wie die Analyse der Anforderungen, die Gewinnung von Testfällen, die Generierung und Validierung von Testdaten, den Test von Webservices und die Dokumentation und Messung der Testdurchläufe.

Er hat zusätzlich Lehraufträge für Software Engineering an der Universität Regensburg, für Softwaretest an der Universität Koblenz und für Softwareevolution an der Fachhochschule Hagenberg in Oberösterreich. Außerdem lehrt er als Gastdozent für Softwaremetrik an der Universität Szeged in Ungarn.

Harry M. Sneed gehört zu den Pionieren der Softwaretesttechnologie. Er wurde 1996 von der IEEE ausgezeichnet und ist seit 2005 GI-Fellow der Deutschen Gesellschaft für Informatik. 2009 erhielt er von der IEEE den Stevens Award für seine Errungenschaften auf dem Gebiet des Software-, Reverse- und Reengineering.

Als Autor hat Sneed über 400 Fachartikel in deutscher und englischer Sprache veröffentlicht und 19 Bücher verfasst.

Richard Seidl leitet den Bereich Verifizierung, Validierung & Test bei GETEMED Medizin- und Informationstechnik AG. Er organisiert die Firm-, Hard- und Softwaretests und ist zudem für die Konzeption und Einführung der agilen Entwicklungs- und Testprozesse im Unternehmen verantwortlich. Als Autor und Mitautor hat er verschiedene Fachbücher und Artikel veröffentlicht, unter anderem »Der Systemtest« (2008), »Der Integrationstest« (2012) und »Basiswissen Testautomatisierung« (2012).

Harry M. Sneed · Richard Seidl

Softwareevolution

Erhaltung und Fortschreibung bestehender Softwaresysteme

dpunkt.verlag

Harry M. Sneed · Harry.Sneed@t-online.de
Richard Seidl · office@richard-seidl.com

Lektorat: Christa Preisendanz
Copy-Editing: Ursula Zimpfer, Herrenberg
Herstellung: Birgit Bäuerlein
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

Fachliche Beratung und Herausgabe von dpunkt.büchern im Bereich Wirtschaftsinformatik:
Prof. Dr. Heidi Heilmann · heidi.heilmann@augustinum.net

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

ISBN:
Buch 978-3-86490-041-9
PDF 978-3-86491-385-3
ePub 978-3-86491-386-0

1. Auflage 2013
Copyright © 2013 dpunkt.verlag GmbH
Wieblinger Weg 17
69123 Heidelberg

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.
Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.
Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.
5 4 3 2 1 0

Vorwort

Der Begriff »*Softwareevolution*« impliziert alle Handlungen, die auf ein Softwaresystem ab dem Zeitpunkt wirken, ab dem es produktiv eingesetzt wird. Das System wird u. a. korrigiert, geändert, erweitert, optimiert, saniert, dokumentiert und integriert. Es bleibt jedoch im Kern immer das gleiche System, auch wenn sich das Verhalten ändert. Wenn es sich jedoch so weit wandelt, dass es sich ganz anders verhält, handelt es sich nicht mehr um Evolution, sondern um eine Weiterentwicklung und dafür gelten andere Gesetze. Die Migration des Systems in eine andere technische Umgebung bzw. die Transformation des Codes in eine andere Sprache dürfte kein anderes Verhalten verursachen, ist aber trotzdem nicht als Evolution zu betrachten, denn was dabei herauskommt, ist ein anderes Produkt.

Neuentwicklung und Migration sind einmalige große Anstrengungen, um ein System in seiner Gesamtheit neu zu schaffen oder zu verlagern. Evolution bedeutet hingegen eine Reihe begrenzter Aktionen, um das bestehende System in seiner ursprünglichen Umgebung ständig zu verbessern. Softwareevolution ist mit Softwarewartung eng verwandt. Beide Begriffe beziehen sich auf die gleichen Grundaktivitäten. Der Unterschied besteht darin, dass Evolution der umfassendere Begriff ist. Viele haben unter »Wartung« (Maintenance) nur die Korrektur und Anpassung der Software verstanden. Mit dem Begriff »Evolution« ist eindeutig auch die Weiterentwicklung des Systems gemeint, also die fortdauernde Erweiterung durch neue Funktionen. Dies zu unterstreichen war der Hauptgrund für die Verwendung dieses Begriffs.

Seitdem die Softwareentwicklung agil geworden ist, besteht die Notwendigkeit, Evolution auch von der Entwicklung abzugrenzen. Wenn die Entwicklung sich inkrementell in kleinen Schritten vollzieht, worin liegt dann der Unterschied zur Evolution, bei der das Produkt auch in kleinen Schritten ständig verbessert wird? Auf diese Frage kann es nur eine Antwort geben, und zwar, dass bei der Evolution das Produkt produktiv im Einsatz ist. Bei einer agilen Entwicklung dürfte dies noch nicht der Fall sein, und wenn es doch der Fall ist, dann ist es keine Entwicklung mehr, sondern Evolution.

Dies dürfte für den Leser wie ein Streit um des Kaisers Bart erscheinen, aber diese Begriffsunterscheidung ist für das Verständnis des vorliegenden Buches wichtig. Bei der Softwareevolution herrschen ganz andere Gesetze als bei der Softwareentwicklung. Im Vordergrund steht die Aufrechterhaltung der laufenden Dienstleistung. Das System, das im Einsatz ist, darf in keiner Weise in seiner Leistung beeinträchtigt werden. Wenn sein Verhalten sich ändert, dann nur geringfügig und zum Besseren. Das Gleiche trifft auf seine Konstruktionsweise zu. Kurzum, die Erhaltung der bestehenden Funktionalität hat Vorrang vor der Entstehung neuer Funktionalität. Das Ziel eines agilen Entwicklungsprojekts ist es, neue Funktionalität zu erschaffen, auch wenn es erforderlich ist, die bisher geschaffene Funktionalität zu bewahren. Die letztere ist nicht so wichtig, weil die Software noch nicht im Einsatz ist. Im Falle der Evolution ist es umgekehrt. Das Hauptziel ist die Erhaltung der bestehenden Software, auch wenn es erforderlich ist, neue Funktionalität zu schaffen.

Im Lehrfach Software Engineering an den Hochschulen und Bildungsanstalten steht die Entwicklung neuer Systeme im Mittelpunkt. Wenn überhaupt, wird der Umgang mit bestehenden Systemen nur am Rande behandelt. Diese Vernachlässigung steht im krassen Widerspruch zu dem Bedarf an Wartungspersonal in der Industrie. Laut verschiedenen Schätzungen sind über 70 % des Entwicklungspersonals mit der Erhaltung bestehender Systeme beschäftigt – Systeme, die im Durchschnitt 7 Jahre im Einsatz sind. Es versteht sich von selbst, dass die Nachbesserung und Fortschreibung solcher Systeme andere Kenntnisse verlangen als die Entwicklung neuer Systeme. Nicht nur die Vorgehensweise, sondern auch die Programmier- und Testtechniken sind verschieden.

Der Bedarf an Wartungspersonal wäre gar nicht so akut, wenn die Systeme so bleiben würden, wie sie sind, aber Softwaresysteme sind ein Spiegelbild der betrieblichen Wirklichkeit und diese ändert sich immer schneller, laut einer Untersuchung von Les Hatton [Hatt07] um 10 bis 16 % jährlich. Das heißt, bei einem System mit 200.000 Anweisungen werden jährlich mindestens 20.000 Anweisungen geändert oder hinzugefügt. Die mittlere Änderungsproduktivität liegt unter 400 Anweisungen pro Personenmonat. Dennoch kann ein Systemerhalter maximal 5.000 Anweisungen pro Jahr verändern bzw. 50.000 Anweisungen bzw. 75.000 Codezeilen betreuen. Sämtliche Untersuchungen zum Thema Softwarewartung kommen zum Schluss, dass eine Person jährlich nicht mehr als 50.000 Anweisungen bei einer jährlichen Änderungsrate von 10 % betreuen kann. Leider wird in vielen Anwenderbetrieben das Wartungspersonal gezwungen, bis zu 100.000 Anweisungen zu betreuen. Verantwortliche IT-Manager müssen erkennen, wo die Grenzen der Technologie liegen.

Tatsache ist, dass immer mehr neue IT-Systeme in die Welt gesetzt werden, ohne dass die alten Systeme aus dem Verkehr gezogen werden. Einzelne Systeme werden abgelöst, aber die Mehrzahl alter Systeme bleibt in Betrieb, weil irgendwelche Benutzer sie noch benötigen. Also müssen sie gepflegt und angepasst wer-

den. Das bindet Personal. Wenn allein in Deutschland jährlich 6.000 IT-Anwendungen mit einer mittleren Größe von 100.000 Anweisungen freigegeben werden, schafft das einen Bedarf an 12.000 zusätzlichen Wartungstechnikern. Diese Wartungstechniker müssten wissen, wie man Fehler findet oder korrigiert, wie man Code ändert ohne unerwünschte Begleiterscheinungen, wie man neue Funktionalität einbaut und wie man Systeme saniert, ohne ihr Verhalten zu beeinträchtigen.

Daraus ergibt sich ein beträchtlicher Ausbildungsbedarf für Softwareerhaltungsspezialisten. Ob unter dem Stichwort »Softwarewartung« oder »Softwareevolution«, junge Menschen müssen mehr im Umgang mit alter Software ausgebildet werden. Neben den Kursen in Modellierung und Entwicklung von Software sollten auch Kurse in der Erhaltung von Software angeboten werden. Gemessen an dem industriellen Bedarf müssen die Erhaltungskurse sogar den Vorrang haben. Dieses Buch ist einerseits Begleitmaterial zu einem Kurs in Softwareevolution und anderseits als ein Nachschlagwerk für praktizierende Softwareerhalter gedacht. Es soll darüber hinaus auch dazu beitragen, dass dieses wichtige Thema die Aufmerksamkeit erhält, die ihm zusteht. Die Entwicklung neuer Systeme kann man jederzeit einstellen. Die Erhaltung der bestehenden Systeme muss weitergehen. Dafür werden gezielt ausgebildete Spezialisten gebraucht.

In diesem Buch wird auf die wesentlichen Aktivitäten der Softwareevolution eingegangen – Fehlerbehebung, Änderung, Sanierung und Erweiterung sowie Regressionstesten und Nachdokumentation. Diese Aktivitäten sind in den meisten Lehrplänen der Informatik nicht ausreichend beachtet. Möglicherweise liegt es daran, dass es so schwierig ist, sie abzugrenzen und zu lehren. Das müsste sich ändern. Statt Spezifikationen für Entwicklungsaufgaben sollten Studenten fertige Programme in einem schlechten Zustand bekommen und aufgefordert werden, diese zu bearbeiten. Sie sollten den Code korrigieren, sanieren, dokumentieren, ändern und erweitern. Wie sie dabei vorzugehen haben, wird hier geschildert. Außerdem wird auf die Gesetze und die Wirtschaftlichkeit der Softwareevolution eingegangen.

Das Buch ist aber nicht nur für Studierende gedacht, sondern auch als Unterstützung für die praktische Arbeit. Die Rolle des Wartungsprogrammierers wird hier aufgewertet. Es sind schließlich diese Programmierer, die den IT-Betrieb in Gang halten. Sie haben jeden Grund, darauf stolz zu sein. Diese Lektüre sollte dazu beitragen, mehr Professionalität in ihre alltäglichen Tätigkeiten einzubringen. Die Voraussetzung dafür ist, dass sie die Bedeutung ihrer Tätigkeit verstehen und vertreten können. Auch das wird von diesem Buch befördert.

Zum Schluss ist zu sagen, dass der Stoff, der hier zusammengefasst ist, auf mehr als 30 Jahre Erfahrung in der Bearbeitung bestehender Software basiert. Der Umgang mit Legacy-Systemen war schon immer eine große Herausforderung und wird es in der Zukunft auch bleiben. Schon vor 20 Jahren habe ich in einem Artikel für die Computerwoche proklamiert: »Die größte Herausforderung der Zukunft ist die Bewältigung der Vergangenheit.« Diese Aussage gilt nach wie vor.

Dieses Buch ist dem Vater der Softwareevolution, Manny Lehman, gewidmet, der vor Kurzem in Jerusalem verstorben ist. Er hat zusammen mit Les Belady erkannt, dass komplexe Softwaresysteme das Produkt eines langen Evolutionsprozesses sind. Ich hatte das Privileg, beide als meine Gäste auf der Internationalen Software Maintenance Conference 2005 in Budapest begrüßen zu dürfen.

Das war kurz nachdem ich meinen Koautor Richard Seidl kennengelernt habe. Wir arbeiteten zusammen in einem großen Testprojekt in Dresden. Seit damals haben wir bei verschiedenen Buchprojekten zusammengearbeitet: Bücher über das Testen, das Vermessen und jetzt über Softwareevolution. Ich bin sehr froh, Herrn Seidl als Koautor für dieses Buchprojekt gewonnen zu haben. Ohne seinen wertvollen Beitrag wäre dieses Buch nie in dieser Form erschienen.

Harry Sneed
Wien, im Juli 2013

Inhaltsübersicht

1 Einführung in die Softwareevolution 1

2 Wirtschaftlichkeit der Softwareevolution 17

3 Die Gesetze der Softwareevolution 41

4 Der Evolutionsprozess 57

5 Softwaresystemanalyse 79

6 Softwareevolutionsplanung 97

7 Fehlerbehebungen 119

8 Änderungen 133

9 Sanierung 153

10 Softwareweiterentwicklung 179

11 Systemregressionstest 199

12 Fortlaufende Dokumentation 223

Anhang

A Glossar 253

B Abkürzungen 259

C Literaturverzeichnis 261

Index 281

Inhaltsverzeichnis

1 Einführung in die Softwareevolution **1**

1.1 Wartung und Evolution – eine Begriffsbestimmung 1

1.1.1 Zum Ursprung des Begriffes »Maintenance« 2

1.1.2 Zum Unterschied zwischen Erhaltung und Entwicklung . . . 3

1.1.3 Zum Unterschied zwischen Erhaltung und Evolution 4

1.1.4 Zum Unterschied zwischen Änderung und Erweiterung . . . 8

1.1.5 Zum Unterschied zwischen Korrektur und Sanierung 8

1.2 Iterative und evolutionäre Softwareentwicklung 9

1.3 Softwareevolution und agile Softwareentwicklung 10

1.4 Wartung und Evolution in einer serviceorientierten IT-Welt 13

1.5 Struktur und Inhalt der folgenden Kapitel 16

2 Wirtschaftlichkeit der Softwareevolution **17**

2.1 Zur Werterhaltung von Softwarekapitalgütern 17

2.2 Software als verpacktes Wissen 20

2.3 Wertgetriebene Softwareevolution 21

2.4 Einflüsse auf die Evolutionskosten 26

2.4.1 Der Einfluss der Größe auf die Softwareerhaltung 26

2.4.2 Der Einfluss der Komplexität auf die Softwareerhaltung . . 28

2.4.3 Der Einfluss der Qualität auf die Softwareerhaltung 29

2.5 Schätzung der Evolutionskosten 30

2.6 Ermittlung vom Evolutionsnutzen 33

2.6.1 Zum Nutzen der korrektiven Aufträge 34

2.6.2 Zum Nutzen der adaptiven Aufträge 35

2.6.3 Zum Nutzen der perfektiven Aufträge 35

2.6.4 Zum Nutzen der enhansiven Aufträge 36

2.7 Beispiel einer Kosten-Nutzen-Rechnung 37
2.7.1 Kalkulation des ROI für ein betriebliches Informationssystem 37
2.7.2 Kalkulation des ROI für ein Testwerkzeug 38
2.8 Schlussfolgerungen aus der Wirtschaftlichkeitsbetrachtung 39

3 Die Gesetze der Softwareevolution 41

3.1 Die Pionierleistung von Lehman und Belady 41
3.2 Lehmans Kategorisierung der Softwaresystemtypen 42
3.2.1 S-Systeme 43
3.2.2 P-Systeme 44
3.2.3 E-Systeme 46
3.3 Die fünf Gesetze der Evolution 48
3.3.1 Gesetz der fortdauernden Änderung 48
3.3.2 Gesetz der zunehmenden Komplexität 48
3.3.3 Gesetz der abnehmenden Qualität 49
3.3.4 Gesetz der sinkenden Produktivität 49
3.3.5 Gesetz des begrenzten Wachstums 49
3.4 Zur Gültigkeit der Evolutionsgesetze 49
3.5 Konsequenzen aus den Gesetzen der Softwareevolution 50
3.5.1 Releasegrößen müssen begrenzt werden 51
3.5.2 Die Dokumentation muss mit jedem Release aktualisiert werden 52
3.5.3 Der Code muss in regelmäßigen Abständen saniert werden 52
3.5.4 Die Qualität der Software soll laufend überwacht werden 53
3.5.5 Die Evolution des Systems muss geplant werden 53
3.6 Warum Software doch noch stirbt 54
3.6.1 Veralterung der Implementierungstechnologie 54
3.6.2 Ungeeignete fachliche Lösung 55
3.6.3 Abhängigkeit von Schlüsselpersonen 55

4 Der Evolutionsprozess 57

4.1 Der Softwareevolutionsprozess aus historischer Perspektive 58
4.2 Das Releasekonzept 63
4.3 Systemanalyse 67

4.4 Releaseplanung 69
4.5 Fortschreibung des Systems 71
4.5.1 Korrekturaufgaben 71
4.5.2 Änderungsaufgaben 72
4.5.3 Weiterentwicklungsaufgaben 72
4.5.4 Integrationsaufgaben 72
4.5.5 Sanierungsaufgaben 72
4.5.6 Optimierungsaufgaben 73
4.6 Systemregressionstest 73
4.6.1 Regressionstestplanung 73
4.6.2 Regressionstestspezifizierung 74
4.6.3 Regressionstestaufbau 74
4.6.4 Regressionstestausführung 74
4.6.5 Regressionstestvalidierung 75
4.6.6 Regressionstestevaluierung 75
4.7 Systemdokumentation 76
4.8 Systemauslieferung 76

5 Softwaresystemanalyse 79

5.1 Gegenstände der Systemanalyse 80
5.2 Methoden der Systemanalyse 81
5.2.1 Softwareregelprüfung 81
5.2.2 Vermessung der Software 83
5.2.3 Nachdokumentation der Software 85
5.2.4 Impact-Analyse 87
5.3 Ergebnisse der Ist-Systemanalyse 91
5.3.1 Prüfberichte 91
5.3.2 Metrikberichte 92
5.3.3 Liste der zu ändernden Bausteine 94
5.3.4 Größenmaße des Auswirkungsbereiches 95
5.4 Systemanalyse ist nicht gleich Systemanalyse 95

6 Softwareevolutionsplanung 97

6.1 Die Notwendigkeit einer Evolutionsstatistik 99
6.1.1 Statistik aus der Produktanalyse 99
6.1.2 Statistik aus der Prozessanalyse 101

6.2 Die Schätzung eines neuen Release 102
6.2.1 Messung der gegenwärtigen Systemgröße 103
6.2.2 Hochrechnung der Systempflegekosten 105
6.2.3 Schätzung der Weiterentwicklungskosten 109
6.2.4 Kalkulation der Gesamtkosten des nächsten Release 110
6.3 Die Identifikation und Zuweisung der Evolutionsaufgaben 112
6.3.1 Evolutionsaufgaben 113
6.3.2 Zuweisung der Evolutionsaufgaben 114
6.3.3 Terminierung der Evolutionsaufgaben 115
6.3.4 Management by Contract 116
6.4 Werkzeuge für die Releaseplanung 117

7 Fehlerbehebungen 119

7.1 Die Fehlermeldung 119
7.2 Die Fehleranalyse 120
7.3 Fehlerursachenforschung 123
7.4 Fehlerbeseitigung 125
7.5 Fehlerkorrekturtest 126
7.6 Fehlerkorrekturdokumentation 128
7.7 Fehlerstatistik 129

8 Änderungen 133

8.1 Der Änderungsprozess 134
8.2 Der Änderungsantrag 135
8.3 Änderungsanalyse 137
8.4 Lokalisierung der zu ändernden Stellen 140
8.4.1 Analyse der Textdokumente 141
8.4.2 Analyse der Entwurfsdokumente 141
8.4.3 Analyse des Codes 142
8.4.4 Analyse der Datenstrukturen 143
8.4.5 Analyse der Testfälle 143
8.5 Änderungsfolge 143
8.6 Änderungsabschottung 144
8.7 Änderungsdurchführung 146

8.8 Änderungsvalidation 147
8.8.1 Aufbau der Integrationstestumgebung 148
8.8.2 Generierung der Integrationstestdaten 149
8.8.3 Instrumentierung des geänderten Codes 149
8.8.4 Ausführung des Integrationstests 149
8.8.5 Auswertung der Testergebnisse 149
8.9 Änderungsdokumentation 150

9 Sanierung 153

9.1 Messung als Voraussetzung der Sanierung 154
9.2 Sanierungsziele 156
9.3 Sanierungsverfahren 157
9.3.1 Einfrierungsstrategie 159
9.3.2 Gelegenheitsstrategie 159
9.4 Sanierungsmaßnahmen 160
9.4.1 Assembler-Sprachen 160
9.4.2 Prozedurale Sprachen 161
9.4.2.1 Reformatierung des Codes 163
9.4.2.2 Bereinigung des Codes 163
9.4.2.3 Umbenennung nicht sprechender Datennamen 164
9.4.2.4 Beseitigung inkompatibler Datentypen 164
9.4.2.5 Entfernung festverdrahteter Daten 164
9.4.2.6 Auslagerung der IO-Schnittstellen 165
9.4.2.7 Restrukturierung der Ablauflogik 165
9.4.2.8 Verflachung der Ablaufstruktur 166
9.4.2.9 Zerlegung des Codes in kleinere Bausteine 167
9.4.3 Objektorientierte Sprachen 167
9.4.3.1 Verflachung der Methodenlogik 169
9.4.3.2 Verflachung der Klassenhierarchie 170
9.4.3.3 Eliminierung redundanter Methoden 170
9.4.3.4 Ablösung komplexer Auswahlstrukturen 170
9.4.3.5 Verlagerung gemeinsamer Methoden und Attribute 170
9.4.3.6 Spaltung zu groß gewordener Klassen 171
9.4.3.7 Vereinfachung der Schnittstellen 173
9.4.3.8 Entfernung festverdrahteter Daten 173
9.4.3.9 Entfernung des toten Codes 174
9.4.3.10 Umbenennung der Bezeichner 174

9.4.3.11 Ergänzung der Kommentierung 175
9.4.3.12 Vereinheitlichung des Codeformats 175
9.4.4 Skriptsprachen 175
9.5 Sanierungsrevision 176

10 Softwareweiterentwicklung 179

10.1 Beauftragung einer Systemerweiterung 183
10.1.1 Anforderungsgetriebene Evolution 183
10.1.2 Modellgetriebene Evolution 184
10.1.3 Change-Request-getriebene Evolution 185
10.2 Analyse eines Erweiterungsantrages 186
10.2.1 Bestätigung des angegebenen Nutzens 187
10.2.2 Analyse der Funktionsspezifikation 188
10.2.3 Auswirkungsanalyse der beantragten Erweiterung 188
10.3 Aufwandsschätzung einer Erweiterung 189
10.3.1 Aufwandsschätzung in der anforderungsgetriebenen Evolution 190
10.3.2 Aufwandsschätzung in der modellgetriebenen Evolution 191
10.3.3 Aufwandsschätzung in der CR-getriebenen Evolution . . . 191
10.4 Genehmigung einer Erweiterung 192
10.5 Aufstellung des Erweiterungsprojekts 192
10.6 Spezifikation der Erweiterung 193
10.7 Durchführung der Erweiterung 195
10.8 Test der Erweiterung 197
10.9 Dokumentation der Erweiterung 197

11 Systemregressionstest 199

11.1 Die Problematik der Testfallselektion 199
11.2 Notwendigkeit eines unabhängigen Testteams 201
11.3 Regressionstestwerkzeuge 203
11.3.1 Werkzeuge für die statische Analyse 204
11.3.1.1 Werkzeuge zur Prüfung der Anforderungsspezifikation 204
11.3.1.2 Werkzeuge zur Prüfung des Codes und des Entwurfsmodells 205
11.3.1.3 Werkzeuge zur Prüfung der Testdokumentation 206

11.3.2 Werkzeuge für die Verwaltung und Fortschreibung der Testfälle 207
11.3.3 Werkzeuge für die Testfallauswahl 208
11.3.4 Werkzeuge für die Testdatengenerierung 209
11.3.5 Werkzeuge für die Regressionstestausführung 210
11.3.6 Werkzeuge für die Verfolgung der Testabläufe 211
11.3.7 Werkzeuge für die Vermessung der Testüberdeckung 212
11.3.8 Werkzeuge für den Abgleich der Testergebnisse 213
11.3.9 Werkzeuge für die Fehlermeldung und Fehlerverfolgung 214
11.4 Regressionstestverfahren 216
11.4.1 Regressionstestplanung 216
11.4.2 Regressionstestvorbereitung 218
11.4.3 Regressionstestausführung 218
11.4.4 Regressionstestauswertung 219
11.5 Regressionstestergebnisse 220

12 Fortlaufende Dokumentation 223

12.1 Die Rechtfertigung der Systemdokumentation 223
12.2 Automatisierte Nachdokumentation 225
12.2.1 Analyse der Anforderungsdokumente 227
12.2.2 Analyse des Entwurfsmodells 227
12.2.3 Analyse des Codes 228
12.2.3.1 Prozedurale Sprachen 229
12.2.3.2 Objektorientierte Sprachen 230
12.2.3.3 Gemeinsame Strukturen aller Sprachen 230
12.3 Multiple Sichten auf ein Softwaresystem 232
12.3.1 Statische Sichten auf die Programmlogik 232
12.3.2 Visualisierungstechniken 234
12.3.2.1 Baumdiagramme 235
12.3.2.2 Netzdiagramme 235
12.3.2.3 Sequenzdiagramme 236
12.3.2.4 Ablaufdiagramme 237
12.3.2.5 Entity/Relationship Diagramme 238
12.3.3 Grenzen der grafischen Darstellung 239
12.3.4 Texte vs. Diagramme 240

12.4 Aufbau und Nutzung eines System-Repository 241
12.4.1 Aufbau eines Repository 242
12.4.2 Besichtigung der Repository-Inhalte 243
12.4.3 Abfragen der Repository-Inhalte 245
12.4.4 Generierung von Dokumenten 246
12.5 Fortschreibung der Dokumentation 246
12.5.1 Abgestimmte Evolution aller Softwareschichten 247
12.5.2 Software = Code + Dokumentation + Test 248

Anhang

A Glossar **253**

B Abkürzungen **259**

C Literaturverzeichnis **261**

Index **281**

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).

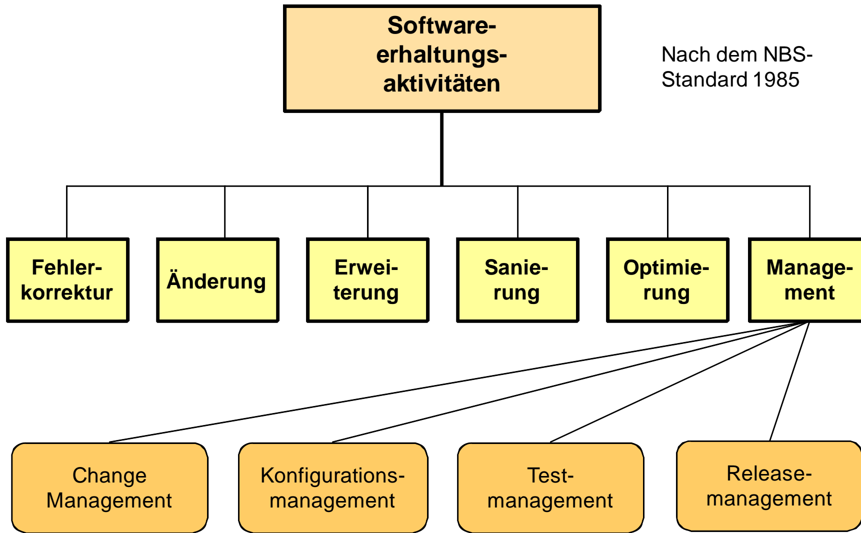

Abb. 1–1 *Softwareerhaltungsaktivitäten*

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).

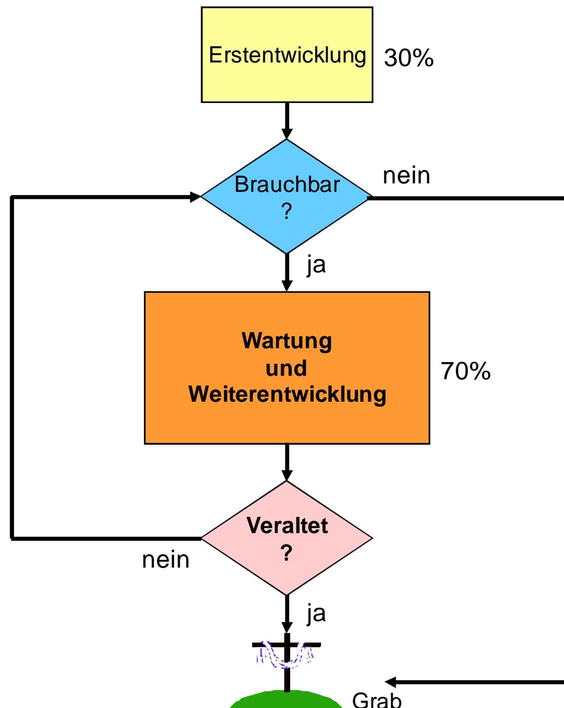

Abb. 1–2 *Das Leben eines Softwareprodukts*

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).

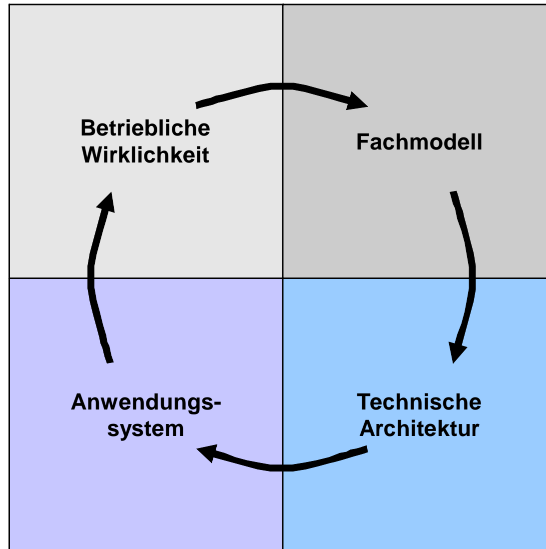

Abb. 1–3 *Evolution als permanente Nachbesserung*

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).

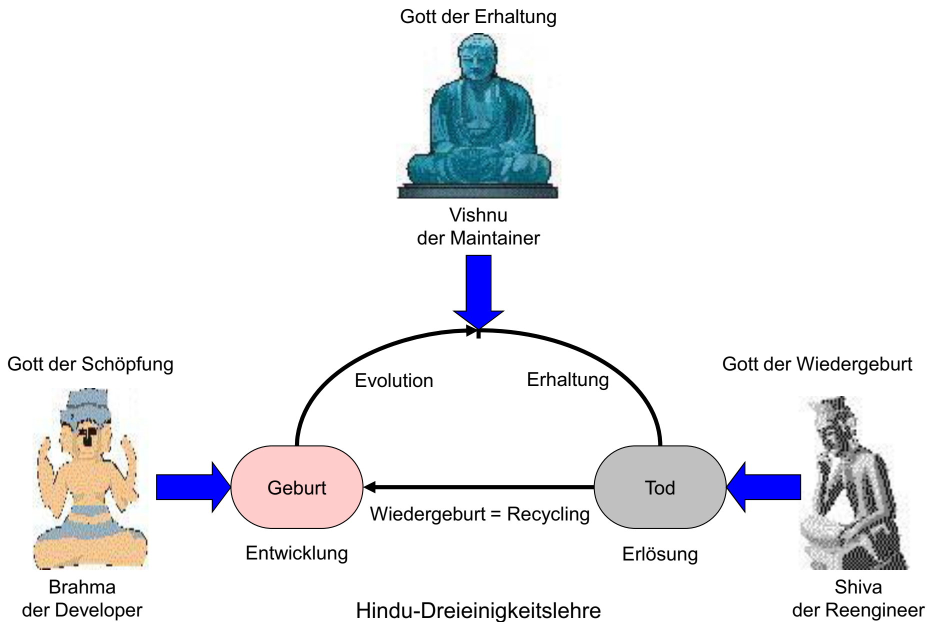

Abb. 1–4 *Das Urmodell des Softwarelebenszyklus*

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].

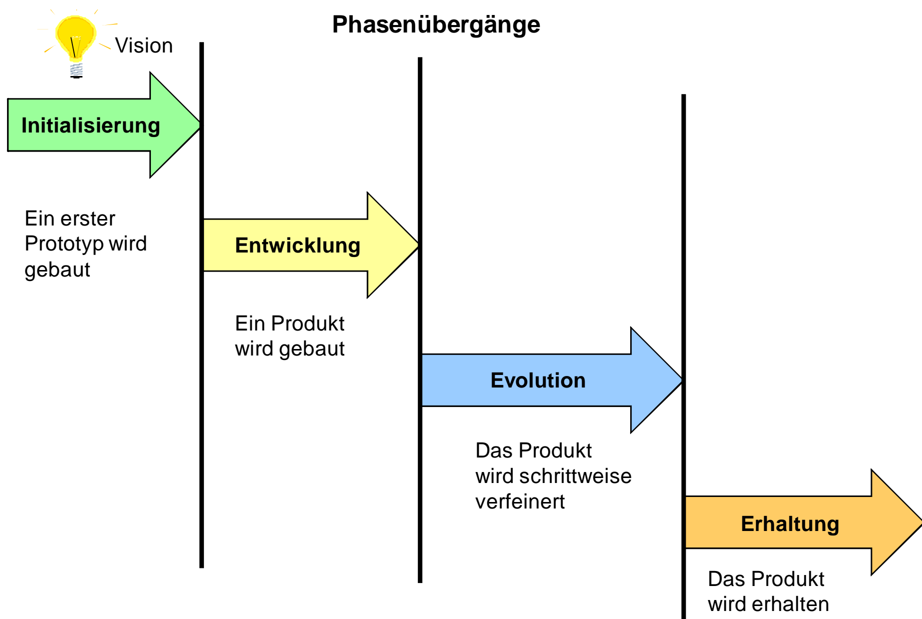

Abb. 1–5 *Phasen im Leben eines Softwareprodukts*

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.

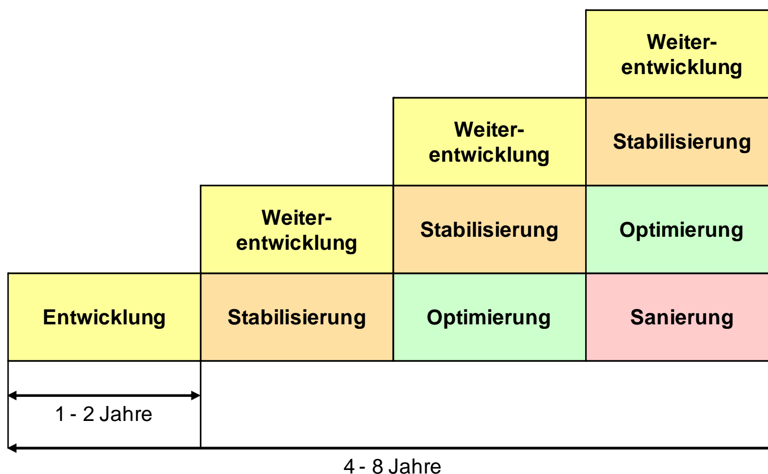

Abb. 1–6 *Evolutionäre Softwareentwicklung nach Tom Gilb [Gilb85]*

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).

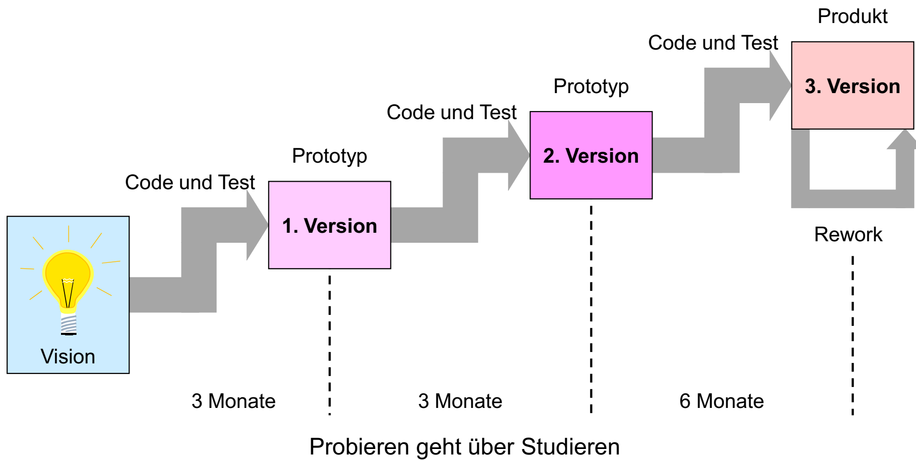

Abb. 1–7 *Inkrementelle Softwareentwicklung*

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).

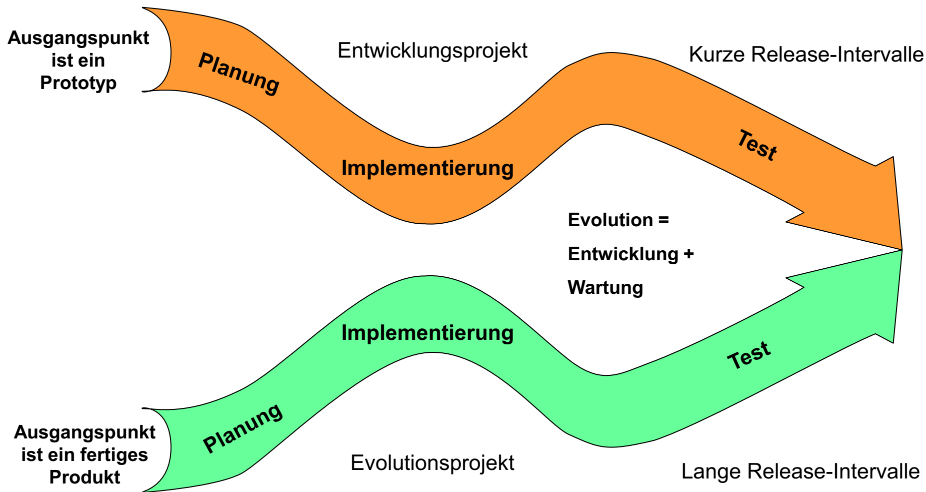

Abb. 1–8 *Konvergenz von Entwicklung und Evolution*

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