Vorgehensmodelle

Udo Kelter

02.10.2003

Zusammenfassung dieses Lehrmoduls

Dieses Lehrmodul führt in den Problemkomplex ein, den Prozeß der Entwicklung von Softwaresystemen zu strukturieren und planbar zu machen. Derartige Strukturierungen nennt man Vorgehensmodelle. Das bekannteste Vorgehensmodell ist das Phasenmodell, das den gesamten Entwicklungsprozeß in mehrere Phasen aufteilt, in denen das System schrittweise konkretisiert wird. Wir skizzieren ferner einige alternative Vorgehensmodelle, die aufgrund bestimmter Schwächen des Phasenmodells entwickelt worden sind.

Vorausgesetzte Lehrmodule: keine

Stoffumfang in Vorlesungsdoppelstunden: 1.0

Inhaltsverzeichnis

1 Einführung **3**

2 Das Phasenmodell **5**
2.1 Anforderungsanalyse und Produktdefinition 7
2.2 Entwurf 9
2.3 Kodierung und Modultest 11
2.4 Integration und Systemtest 12
2.5 Einsatz und Wartung 12
2.6 Bewertung des Phasenmodells 13
2.7 Aufwandsverteilungen 14

3 Alternative Vorgehensmodelle **16**
3.1 Rapid Prototyping 16
3.2 Evolutionäre Softwareentwicklung 17
3.3 Iteriertes Phasenmodell und parallele Phasen 17
3.4 Das Spiralmodell 18

Literatur 20
Glossar 20
Index 20

1 Einführung

Wenn jemand, der ein Haus bauen will, sich einen Lastwagen Steine und einige Säcke Zement kommen läßt und dann mit der Maurerkelle den Grundriß in den Boden ritzt, fände man das reichlich unprofessionell. Genauso unprofessionell sind Leute[1], die, wenn sie ein Softwaresystem realisieren sollen, als erstes zum Rechner schreiten, den Compiler anwerfen und einige wichtige Systemkomponenten programmieren.

Ganz offensichtlich sollte man, bevor man ein System realisiert, zunächst einmal den tatsächlichen Bedarf und die Finanzierbarkeit des Vorhabens analysieren. Weiterhin können nichttriviale Systeme normalerweise nicht von einer einzigen Person realisiert werden, sondern die Arbeit muß entweder wegen unterschiedlicher erforderlicher Qualifikationen oder allein wegen des quantitativen Umfangs auf viele Personen verteilt werden. Dies erfordert eine sorgfältige Planung.

Bei der systematischen Entwicklung eines Softwaresystems fallen also sehr unterschiedliche Tätigkeiten an. Ein **Vorgehensmodell** legt fest, welche Einzeltätigkeiten auftreten und wie sie koordiniert werden. Es gibt sehr viele verschiedene Vorgehensmodelle[2]. Um die vielen Vorschläge einordnen zu können, ist folgende grobe Klassifikation hilfreich:

Grobgranulare Vorgehensmodelle strukturieren den Entwicklungsprozeß in wenige (ca. 5 - 10) Abschnitte, jeder Abschnitt dauert Tage oder Wochen. Ein Beispiel ist das Phasenmodell. Die einzusetzenden Sprachen und Methoden bleiben weitgehend offen bzw. werden nur sehr abstrakt benannt. In einem konkreten Projekt müssen also noch viele Details festgelegt werden.

Feingranulare Vorgehensmodelle schreiben detailliert die zu erstellenden Dokumente und die einzusetzenden Sprachen und Entwicklungsmethoden vor. Ein Beispiel ist der auf der UML

[1] Ich rede hier bewußt nicht von Informatikern.

[2] In [NoS99] werden z.B. 7 Modelle für die objektorientierte Softwareentwicklung im Detail verglichen.

basierende *Unified Process*. Es werden Arbeitsschritte beschrieben, die nur Stunden oder sogar nur Minuten dauern können.

Feingranulare Vorgehensmodelle beziehen sich immer auf bestimmte (Programmier-, Modellierungs-, Abfrage- u.a.) Sprachen und/oder Dokumentstrukturen und oft indirekt auf Standardarchitekturen, Betriebssysteme oder Frameworks. Während die Sprachen sozusagen Syntax und Semantik der zu entwickelnden Dokumente definieren, beschreiben feingranulare Vorgehensmodelle u.a. die Pragmatik der Sprachen. Die zugrundeliegenden Sprachen müssen also gut bekannt sein, die Vorgehensmodelle adressieren vielfach den "meisterhaften" Umgang mit den Sprachen (*best practices*).

Während man grobgranulare Vorgehensmodelle auf wenigen Seiten beschreiben kann, füllt die Beschreibung eines feingranularen Vorgehensmodells meist ein ganzes Buch.

Vorgehensmodelle drücken die Erfahrung aus, wie ein System am besten zu entwickeln ist. Je feingranularer ein Vorgehensmodell ist, desto detailliertere Annahmen müssen über die Art des zu entwickelnden Systems, den Anwendungsbereich und diverse Randbedingungen unterstellt werden. Alleine hierdurch kommt es zu vielen Alternativen. Darüberhinaus kann man oft geteilter Meinung sein, welches Vorgehen wirklich das beste ist; es kommt also zu konkurrierenden Modellen, oder innerhalb eines Modells wird eine Auswahl zwischen verschiedenen, je nach den Umständen auszuwählenden Vorgehensweisen angeboten, d.h. bevor man das Vorgehensmodell wirklich benutzen kann, muß man es zuerst auf ein konkretes Projekt zuschneiden.

Anwendbarkeit der Modelle. Feingranulare Vorgehensmodelle sind nicht einsetzbar, wenn die unterstellten Programmier-, Modellierungs- und sonstigen Sprachen nicht eingesetzt werden können. Dieser triviale Zusammenhang führt speziell bei dem Fall, daß ein existierendes (altes) System weiterentwickelt werden muß, dazu, daß auf moderneren Sprachen (z.B. der UML) basierende Vorgehensmodelle nicht eingesetzt werden können.

Die meisten Vorgehensmodelle unterstellen, daß ein komplett neues System entwickelt werden soll. In der Praxis müssen dagegen häufig vorhandene Systeme weiterentwickelt werden. Das entscheidende Problem stellt hier oft das Re-engineering und ggf. die Sanierung des existierenden Systems dar. Dies sind Tätigkeiten, die bei der Erstentwicklung überhaupt nicht auftreten.

Viele Systeme werden kostengünstiger auf Basis einer Standardsoftware (z.B. für eine bestimmte Branche) realisiert und nicht durch eine komplette Neuentwicklung. Zentrales Problem ist hier die Auswahl der Standardsoftware. Die anschließende Anpassung ist oft eher eine Konfigurierung als eine klassische Programmierung.

Daß man ein System komplett neu erstellt, ist daher im beruflichen Alltag eher die Ausnahme. Allerdings treten die bei der Erstentwicklung eingesetzten Sprachen, Methoden und Vorgehensweisen auch in den anderen Fällen auf, wenn auch in reduziertem Umfang.

2 Das Phasenmodell

Das bekannteste und älteste Vorgehensmodell ist das **Phasenmodell**, aufgrund der üblichen in Bild 1 gezeigten graphischen Darstellung auch **Wasserfallmodell** genannt. Der Entwicklungsprozeß wird in mehrere Phasen aufgeteilt, das Ergebnis jeder Phase wird als Eingabe in die Folgephase eingespeist. Es gibt mehrere Varianten des Phasenmodells, die sich i.w. in der Anzahl der Einzelphasen unterscheiden.

Am Ende der vierten Phase liegt das konstruierte System vor und kann dann (hoffentlich) eingesetzt werden. Die noch gezeigte anschließende fünfte Phase ist also nicht mehr Teil der eigentlichen (Erst-) Entwicklung des Systems und wird oft als **Wartungsphase** bezeichnet. Ganz grob können wir die ersten vier Phasen aufteilen in die erste, die meist **Analysephase** genannt wird und in der es um die Frage geht, "was" das System leisten soll, und die nächsten drei Phasen, in denen es um das "Wie" geht.

In jeder Phase werden bestimmte Dokumente produziert, für deren Produktion jeweils eigene Fähigkeiten und damit ggf. auch eigene Spezialisten erforderlich sind. Der Abschluß einer Phase ist erreicht, wenn

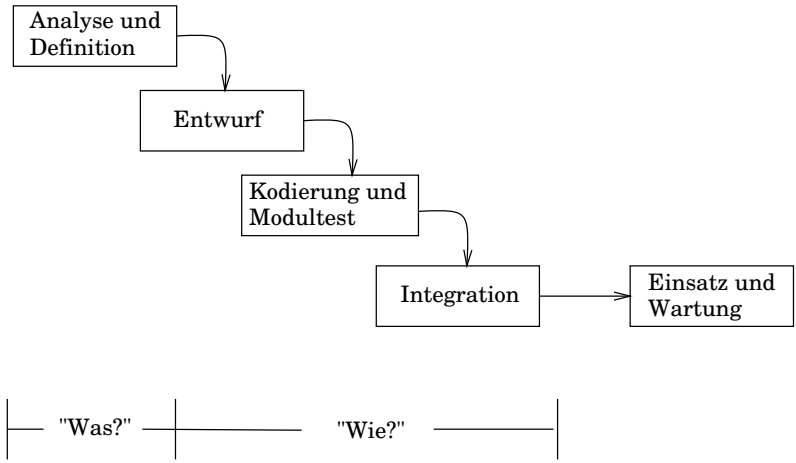

Abbildung 1: Das Phasenmodell

alle geplanten Dokumente vorliegen und vom Kunden abgenommen worden sind.

Das Phasenmodell suggeriert eine strikt lineare Abfolge der Phasen; wie wir später noch sehen werden, ist dies i.a. nicht praktikabel. Dennoch ist das Phasenmodell keineswegs unbrauchbar:

- Trotz diverser Ausnahmen und Abweichungen kann man den Ablauf vieler Projekte im großen und ganzen anhand des Phasenmodells strukturieren.
- Man kann das Phasenmodell nicht nur als zeitlichen Ablauf interpretieren, sondern als Folge von *Entwicklungsstufen des Systems*, die aufeinander aufbauen und die bzgl. der entstehenden Dokumente einander ergänzen. Das System wird anfangs eher vage und abstrakt beschrieben, später immer detaillierter und präziser. Dieser Verfeinerungsprozeß kann in manchen Systemteilen schneller und in anderen langsamer verlaufen.

Im folgenden gehen wir die einzelnen Phasen, die darin erstellten Dokumente und die involvierten Personen bzw. Rollen der Reihe nach durch.

2.1 Anforderungsanalyse und Produktdefinition

Hauptziel der Analysephase ist es, die Anforderungen, die der Kunde bzw. Auftraggeber an das System stellt, zu erfassen, zu analysieren und so zu notieren, daß das System in den folgenden Phasen ohne ständige Rückfragen beim Kunden realisiert werden kann.

Daß hier so stark betont wird, daß die Anforderungen des Kunden erfaßt werden müssen, mag zunächst etwas übertrieben erscheinen. Tatsächlich ist aber die gründliche Analyse der Anforderungen außerordentlich wichtig, und die Gefährlichkeit dieses Problems liegt darin, daß es notorisch unterschätzt wird. Es hat viele Systeme gegeben, die zwar funktionsfähig realisiert, dann aber vom Kunden nie eingesetzt worden sind, weil sie nicht das taten, was der Kunde eigentlich brauchte und haben wollte, sondern etwas anderes. Hauptursache für solche Vorkommnisse sind Kommunikationsprobleme zwischen dem Kunden und Entwicklern. Der Kunde ist Fachmann in irgendeinem Anwendungsgebiet und denkt und spricht in den Begrifflichkeiten dieses Anwendungsgebiets; er ist i.d.R. kein Informatiker und kann die Realisierbarkeit und Auswirkungen seiner Anforderungen nicht einschätzen. Die Entwickler auf der anderen Seite sind Fachleute in den Technologien zur Realisierung der Systeme, kennen hingegen die Anwendungsdenkwelt und vielen Erfahrungen und stillschweigenden (!) Annahmen, die dieser Denkwelt zugrunde liegen, nicht[3]. Mißverständnisse sind daher vorprogrammiert.

Selbst wenn man die Anforderungen des Kunden richtig verstanden hat, ist keineswegs gesagt, daß auch alle Anforderungen befriedigt werden können, insb. weil die Kosten zu hoch sind oder die verfügbare Zeitdauer zur Realisierung des Systems zu kurz ist. Ein zweites zentrales Thema der Analysephase ist daher eine grobe Einschätzung der Kosten und Zeiträume und die Bildung von Prioritäten, anhand derer die weniger wichtigen Anforderungen fallengelassen werden. Die englische Bezeichnung *requirements engineering* für die Analysephase betont gerade diesen Sachverhalt, daß Anforderungen "entwickelt" wer-

[3]Exakt dieses Problem adressieren interdisziplinär angelegte Studiengänge wie Technische Informatik, Medieninformatik und Wirtschaftsinformatik.

den. Generell liegt eine Leistung eines guten Systemanalytikers darin, die Anforderungen, die der Kunde zunächst einmal vorgibt, in jeder Hinsicht auf unerwünschte und dem Kunden nicht bewußte Seiteneffekte zu untersuchen, den Kunden über solche Effekte in verständlicher Weise aufzuklären und ggf. zusammen mit dem Kunden die ursprünglichen Anforderungen so zu modifizieren, daß sie praxisgerecht und finanzierbar sind.

Meist muß für die Kostenschätzung zumindest provisorisch eine grobe **Systemarchitektur** entworfen werden[4]. Hiermit zusammen hängt oft eine **Technologieauswahl** (bzgl. Datenbanksystem, Transaktionsmonitor, Komponenten-Technologie, Applikations-Framework, WWW-Technologien, Sicherheitstechnologie, Programmiersprache(n) usw.). Eine Technologie definiert nicht nur einzelne Module im entstehenden System, sondern hat i.a. auch erheblichen Einfluß auf die Gestaltungsmöglichkeiten und das Vorgehen bei der Entwicklung.

Neben den Kosten muß oft auch der Nutzen des geplanten Systems analysiert und beziffert werden, um die Wirtschaftlichkeit des Projekts sicherzustellen. Weiterhin müssen oft beim Kunden im Rahmen der Einführung des Systems betriebliche Abläufe oder andere Strukturen verändert werden, so daß sich hier ein Beratungsbedarf ergibt. Auf diese Tätigkeiten in der Analysephase gehen wir hier nicht näher ein.

Die wichtigsten in der Analysephase erstellten Dokumente sind:

- das **Lastenheft**: stellt eine erste Grobkonzeption des Systems rein textuell und umgangssprachlich dar, je nach Systemgröße ca. 5 - 20 Seiten lang
- das **Pflichtenheft**: dieses beschreibt in einer für den Kunden nachvollziehbaren Form möglichst exakt und vollständig die Funktionen und Schnittstellen des Systems und ist oft Basis für einen Vertrag und die spätere Endabnahme des Systems.
- eine erste Version eines **Projektplans** incl. Terminplan und **Aufwandsschätzung** (sofern nicht schon Teil des Pflichtenhefts)

[4]Dies ist eines der Beispiele, wo von der strikten Reihenfolge der Phasen abgewichen wird.

- Benutzerhandbücher, Bildschirmmasken, Kommandomenüs u.ä.: diese stellen ebenfalls Beschreibungen des Systems aus einem bestimmten Blickwinkel dar
- sofern möglich Testdaten

2.2 Entwurf

Gegenstand dieser Phase ist die Entwicklung einer **Architektur**, also eine Strukturierung des Systems in Subsysteme, Komponenten oder Module und die Festlegung der Schnittstellen zwischen diesen Komponenten. Man spricht hier auch vom "*Programmieren im Großen*". Die kleinsten Einheiten in dieser Struktur sollen so klein sein, daß sie in der anschließenden Kodierungsphase (also beim "*Programmieren im Kleinen*") von einer Person in maximal 4 Wochen realisiert werden können.

Da in der Kodierungsphase meist mehrere Entwickler parallel arbeiten, sollten die Abhängigkeiten zwischen den Modulen möglichst gering sein.

Die Komplexität der Architektur hängt naheliegenderweise von der Größenordnung des Systems und den verwendeten Programmiersprachen, Bibliotheken usw. ab; bei kleinen Systemen, wie sie z.B. im Programmierpraktikum erstellt werden, reicht es meist aus, das System in Klassen zu zerlegen und die Schnittstellen der Klassen zu spezifizieren. Bei großen Systemen untergliedert man die Entwurfsphase weiter in den

1. Grobentwurf
2. Detailentwurf
3. Implementierungsentwurf

Sofern eine Standardarchitektur für das System verwendet werden kann, deckt diese typischerweise den kompletten Grobentwurf ab.

Bei umfangreichen Systemen wird im Rahmen des Detailentwurfs oft parallel die Produktdefinition vervollständigt, indem bisher noch vage formulierte Anforderungen durch detaillierte und präzisere ersetzt werden. Sachlogisch gehören diese Tätigkeiten eigentlich zur System-

analyse, können aber aus zwei Gründen nicht in der Analysephase durchgeführt werden:

- die Analysephase würde zu lange dauern und zu aufwendig werden. In diesem Zusammenhang ist zu beachten, daß große Systeme oft in mehreren Stufen realisiert und eingeführt werden und die Anforderungen der späteren Stufen zunächst nur oberflächlich analysiert werden.
- Die Anforderungen können erst auf Basis der grundlegenden Entwurfsentscheidungen präzisiert werden.

Dies ist ein weiteres Beispiel dafür, daß eine strikt lineare Abfolge der Phasen stellenweise nicht praktikabel ist.

Der Implementierungsentwurf ist durch folgende Problematik motiviert: In großen Systemen werden oft verschiedene Programmiersprachen zur Programmierung einzelner Komponenten eingesetzt. Notiert man nun die Schnittstellen der Module in einer ganz bestimmten Programmiersprache (z.B. Definition Modules bei Modula-2 oder Prototypen bei C/C++), dann wird dieser Entwurf in vielen Details durch die Eigenschaften der Programmiersprache beeinflußt werden. In vielen Sprachen gibt es z.B. keinen Typkonstruktor `set of X`, der zu einem Basistyp `X` Mengen über diesem Typ bildet. Hier und bei anderen Beispielen muß man zu einer Ersatzkonstruktion greifen, z.B. einer Liste oder einem Array. Derartige Ersatzkonstruktionen hängen stark von der Programmiersprache ab, d.h. die entwickelte Architektur hängt von der gewählten Programmiersprache ab! Dies kann vermieden werden, indem man für den Entwurf eine "abstraktere" Sprache verwendet, die einen reichen Typvorrat hat.

Nachdem nun Grob- und Detailentwurf in dieser abstrakteren Sprache vorliegen, muß nun noch die Brücke geschlagen werden hin zur konkreten gewählten Programmiersprache. Im einfachsten Fall kann man die Konstrukte der abstrakteren Sprache direkt auf Konstrukte der Programmiersprache abbilden, i.w. braucht man hier nur Texte zu ersetzen, die Modulstrukturen bleiben unverändert. In komplizierteren Fällen müssen einzelne Basismodule (z.B. ein Modul, das einen `set`

`of integer` realisiert) hinzugefügt werden, die Architektur wird hier also strukturell erweitert oder sogar modifiziert. Diese Umsetzung ist ohne Werkzeugunterstützung relativ aufwendig, weswegen sprachunabhängige Architekturspezifikationen nur bei großen Systemen praktiziert werden.

Wenn ein System in einer einzigen Programmiersprache realisiert wird und diese Programmiersprache gute Spezifikationskonzepte bietet, dann ist eine sprachunabhängige Architekturspezifikationen wegen des Umsetzungsaufwands i.a. nicht sinnvoll, d.h. die Unterphase Implementierungsentwurf entfällt.

Wesentliche in der Entwurfsphase erzeugte Dokumente sind:
- die Architekturspezifikation (in einer formalen Notation)
- ggf. Testdaten oder Handbücher
- bei Informationssystemen: die Datenbankschemata

Die Datenbankschemata sind in gewisser Weise "Quellcode", genauso wie Typdeklarationen in Programmen; somit fallen sie scheinbar in die Kodierungsphase. Tatsächlich werden indes in einer guten Modularisierung alle wichtigen Datenstrukturen durch Module repräsentiert, die diese Strukturen einkapseln. Man kann normalerweise diese Module nur zusammen mit den Datenbankschemata entwerfen, eine Trennung dieser beiden Tätigkeiten ist nicht sinnvoll. Das Entwerfen der Datenbankschemata kann auch wegen globaler Performanceoptimierungen nur in geringem Maß parallelisiert werden.

2.3 Kodierung und Modultest

In dieser Phase werden die einzelnen Module ausprogrammiert und zunächst isoliert getestet, soweit dies möglich ist. Man spricht hier auch vom "Programmieren im Kleinen (PiK)". Diese Phase wird oft Implementierungsphase genannt, was aber mißverständlich ist, denn die Entwurfs- und die Integrationsphase kann man auch als Teil der Implementierung eines Systems ansehen.

Ein einzelnes Modul M ist i.d.R. alleine kein lauffähiges und testbares Programm. Um es testen zu können, muß erstens ein Hauptprogramm, ein sogenannter Testtreiber, geschrieben werden, das unser

Modul M benutzt und alle Operationen von M mit entsprechenden Parametern aufruft. Zweitens muß man für alle Module, die M seinerseits benutzt und die nicht schon vorliegen, Ersatzimplementierungen ("Dummies") schreiben. Der Aufwand für die Realisierung des Testtreibers und der Dummies kann ganz erheblich sein; deshalb muß man sich hier meist auf rudimentäre Tests beschränken und kann komplexe Testfälle erst in der Integrationsphase durchführen.

Ergebnisse der Kodierphase sind die Quelltexte und Testrahmen der einzelnen Module.

2.4 Integration und Systemtest

In der Integrationsphase werden die vorliegenden Module zusammengefügt und können nun als Gesamtsystem getestet werden. Hier können erstmals komplexere funktionale Tests durchgeführt werden; die hier gefundenen Fehler verursachen oft einen hohen Korrekturaufwand.

Ferner können hier nichtfunktionale Eigenschaften (z.B. Antwortzeiten) erstmals überprüft werden; wo nötig, müssen Systemteile optimiert, also meist signifikant verändert werden. Bei der Optimierung und oft auch bei der Fehlerbeseitigung muß wieder in die Kodierungs- oder sogar Entwurfsphase zurückgesprungen werden.

Sofern das System bisher auf einem separaten Entwicklungsrechner und Testdatenbeständen entwickelt wurde, muß es anschließend noch auf dem Produktionsrechner und "Originaldaten" getestet werden. In diesem Kontext kann ein weiterer Arbeitsschritt darin bestehen, das System beim Kunden zu installieren.

Abgeschlossen wird die Integrationsphase und damit die gesamte Entwicklung durch die Abnahme durch den Kunden.

2.5 Einsatz und Wartung

Wie schon erwähnt ist der anschließende Einsatz und die dabei stattfindenden Modifikationen an dem ausgelieferten System nicht mehr Teil der Erstentwicklung. Diese Modifikationen werden oft als War-

tung bezeichnet und diese Phase daher als Wartungsphase. Der Begriff Wartung ist indessen eher irreführend, denn unter Wartung versteht man klassischerweise Reparaturen, die infolge von Materialverschleiß notwendig werden und deren Ziel darin besteht, den Ursprungszustand wiederherzustellen. Software verschleißt aber nicht. Die Wartung von Software ist in Wirklichkeit immer eine inkrementelle Weiterentwicklung. Üblicherweise treten drei Arten von Weiterentwicklungen auf:

Fehlersuche und Korrektur: Ursache sind hier Fehler, die bei der Erstentwicklung nicht gefunden worden sind. Bei einer "perfekten" Erstentwicklung würden solche Wartungsarbeiten also nicht auftreten.

Adaption und Portierung: Ursache sind hier Änderungen an der "Umgebung" des Systems, beispielsweise der Übergang zu einer anderen Version eines Betriebssystems oder Datenbanksystems. Die Funktionalität des Systems bleibt dabei unverändert. Bei einer guten Modularisierung erstrecken sich die notwendigen Änderungen auf wenige Basismodule.

Verbesserung und Perfektionierung: Ursache sind hier modifizierte Anforderungen. Beispielsweise kann es sich nach einiger Praxis mit dem System herausstellen, daß Schnittstellen ungünstig gestaltet sind oder Funktionen fehlen, oder es können sich gesetzliche Bestimmungen oder geschäftliche Ziele ändern. Bei all diesen Beispielen müssen im Prinzip angefangen bei der Analyse alle Entwicklungsphasen für die betroffenen Systemteile erneut durchlaufen werden.

2.6 Bewertung des Phasenmodells

Historisch gesehen war das Phasenmodell das erste weit verbreitete Vorgehensmodell; gegenüber dem davorliegenden (und selbst heute noch oft anzutreffenden) Zustand, überhaupt kein Vorgehensmodell zu haben (System Chaos), war und ist es ein erheblicher Fortschritt. Stärken des Phasenmodells sind:

- Der Entwicklungsprozeß wird strukturiert.
- Es werden Zwischen- bzw. Endprodukte der Phasen, die auch **Meilensteine** genannt werden, identifiziert. Diese erlauben eine vergleichsweise gute Kontrolle des Projektfortschritts. Dies macht das Phasenmodell aus Sicht des Projektmanagements sehr attraktiv.
- Die Rolle der Qualitätssicherung wird betont. Dadurch, daß jede Phase mit bestimmten definierten Dokumenten abschließt, können direkte Maßnahmen zur Sicherung der Qualität dieser Dokumente geplant werden. Die Integrationsphase hat sogar weitgehend die Qualitätssicherung zum Gegenstand.

Andererseits hat das Phasenmodell teilweise gravierende Schwächen. Wie schon früher erwähnt ist die lineare Abfolge der Phasen unrealistisch:

- Zur Fehlerbehebung muß ggf. wieder in frühere Phasen zurückgesprungen werden.
- Bei einer Projektdauer von ca. einem Jahr oder mehr muß damit gerechnet werden, daß sich die Anforderungen während der Entwicklungszeit ändern. In solchen Situationen wird ein inkrementelles, paralleles Durchlaufen der Phasen notwendig.
- Beim Phasenmodell kann der Kunde erst am Ende der Integrationsphase zum ersten Mal das lauffähige System ausprobieren. Eine zentrale Annahme des Phasenmodells ist daher, daß der Auftraggeber die Anforderungen an sein System festlegen kann, ohne das System jemals gesehen zu haben. Viele Kunden können dies nicht, weil ihnen die Anschauung fehlt. Dieses Problem ist besonders gravierend und hat zu Varianten des Phasenmodells und alternativen Vorgehensmodellen geführt, die wir weiter unten vorstellen werden.

2.7 Aufwandsverteilungen

Durch die weite Verbreitung des Phasenmodells liegen viele empirische Daten darüber vor, wie sich die Aufwände auf die einzelnen Phasen verteilen. Die gemessenen Zahlen hängen allerdings immer von

den Umständen ab, der genauen Zuordnung von Tätigkeiten zu den Phasen, der konkreten Ausgestaltung der Entwicklungsprozesse, der Qualität der eingesetzten Werkzeuge usw. Nichtsdestotrotz lassen sich bestimmte Trends generell beobachten, die folgenden Zahlen sollten daher als derartige Trendaussagen interpretiert werden.

Verhältnis von Erstentwicklung und "Wartung". Für dieses Verhältnis werden Relationen von 40:60 bis 10:90 genannt. Mit anderen Worten sind die Kosten der Erstentwicklung im Rahmen der Gesamtkosten während der Lebensdauer des Systems zweitrangig bis marginal, dominierend sind die Weiterentwicklungskosten.

Wenn man also die Gesamtkosten minimieren will, muß man sich vor allem auf die Weiterentwicklungskosten konzentrieren. Diese hängen vor allem von der "Wartbarkeit" eines Programms ab, also der Leichtigkeit, mit der das Programm modifiziert werden kann. Entscheidend ist hier eine gute Architektur des Systems, also die Qualität des Entwurfs. Ideal wären hier hellseherische Fähigkeiten der Entwickler, die spätere Änderungen vorausahnen und diese schon vorwegnehmen oder die Strukturen an den entsprechenden Stellen flexibel gestalten. Diese Fähigkeit wird oft als "Erfahrung" bezeichnet. Es ist kein Zufall, daß immer wieder gefordert wird, in der Entwurfsphase "besonders erfahrene" Entwickler einzusetzen.

Verhältnis der Entwicklungsphasen. Die Kostenverteilung auf die Entwicklungsphasen hängt sehr stark vom Reifegrad einer Entwicklergruppe ab.

Sofern Analyse und Entwurf mit wenig Sorgfalt und entsprechend wenig Aufwand durchgeführt werden, werden viele Fehler erst zu einem sehr späten Zeitpunkt entdeckt, schlimmstenfalls erst bei der Abnahme durch den Kunden. Grob geschätzt verzehnfacht sich der Aufwand zur Behebung eines Fehlers pro Phase, die er verspätet entdeckt wird. Die mangelnde Sorgfalt bei Analyse und Entwurf und der zu frühe Beginn des Programmierens führen dazu, daß die "eingesparte" Zeit in den frühen Phasen durch sehr hohe Aufwände in den späten Phasen bezahlt wird. Die Aufwandsverteilung könnte dann in etwa so

aussehen: 10% Analyse und Entwurf, 30% Codierung, 60% Testen.

Bei technologisch fortgeschrittenen Entwicklergruppen wird viel mehr Sorgfalt und dementsprechend Aufwand auf die frühen Phasen verwendet; die resultierenden Aufwandsverteilung zweier Firmen sind in der folgenden Tabelle angegeben:

Phase	A	B
Analyse und Definition	30 %	18 %
Entwurf	30 %	19 %
Kodierung	15-20 %	34 %
Integration und Test	20-25 %	29 %

3 Alternative Vorgehensmodelle

Die Schwachpunkte, die das Phasenmodell aufweist, haben zu einer Reihe von Alternativen geführt, die wir i.f. nur kurz skizzieren.

3.1 Rapid Prototyping

Das Rapid Prototyping ist eine Variante des Phasenmodells, bei der innerhalb der Analysephase ein "schneller" Prototyp hergestellt wird. Auch diesen kann man in gewisser Weise als Modell des Systems ansehen, allerdings mit der Besonderheit, daß er ausführbar ist. Verbreitet sind GUI-Prototypen, bei denen die Formulare und anderen Bedienschnittstellen prototypisch realisiert werden. Hinter den Bedienelementen steckt aber noch keinerlei Verarbeitungslogik.

Der Prototyp ist "schnell" in dem Sinn, daß er mit wenig Aufwand erstellt wird, typischerweise unter Benutzung von GUI-Editoren, und daher schnell verfügbar ist. Da keinerlei Zeit und Mühe auf softwaretechnische Qualität verwendet wird, ist ein schneller Prototyp ein Einweg-Produkt: nach einmaligem Gebrauch wegzuwerfen. Sinn und Zweck des Prototyps liegt allein darin, dem Benutzer eine sehr frühzeitige Anschauung über das entstehende System zu geben.

Im Prinzip kann man schnelle Prototypen für beliebige Aspekte des Systems erstellen, üblich sind nur GUI-Prototypen.

3.2 Evolutionäre Softwareentwicklung

Die evolutionäre Softwareentwicklung adressiert das gleiche Problem wie das Rapid Prototyping, allerdings wesentlich intensiver. Man geht hier von der Annahme aus, daß es schlicht unmöglich ist, die Anforderungen an das System vorab präzise zu definieren, typischerweise weil das Gebiet noch neu ist und noch wenige Erfahrungen vorliegen. Man muß diese Erfahrungen erst noch sammeln, und das kann man leider nur mit einem lauffähigen, in der Praxis eingesetzten System.

Der Gesamtprozeß einer evolutionären Softwareentwicklung besteht daher aus mehreren Abschnitten, in denen jeweils eine neue, bessere, komplett funktionsfähige Version des System erstellt wird. Diese Version wird dann eingesetzt und evaluiert, wobei aus den Einsatzerfahrungen neue bzw. modifizierte Anforderungen für die nächste Version gewonnen werden. Wenn irgendwann eine zufriedenstellende Version erreicht ist[5], endet der Entwicklungsprozeß.

Verbessert werden kann in einer Folgeversion der Funktionsumfang und / oder die Qualität vorhandener Funktionen.

Äußerst kritisch bei der evolutionäre Softwareentwicklung ist der Anpassungsaufwand. Die entstehenden Zwischenversionen sind keine Wegwerfprototypen, sondern müssen sogar besonders hochwertig entworfene Systeme sein, damit sie mit möglichst geringem Änderungsaufwand in die Folgeversion übernommen werden können.

3.3 Iteriertes Phasenmodell und parallele Phasen

Das iterierte Phasenmodell erweitert das Bild 1 um Rücksprünge in die früheren Phasen, die, wie schon dargestellt, aufgrund gefundener Mängel notwendig werden. Die Realität wird so exakter wiedergegeben. Gewonnen ist damit aber nichts, denn wann und in welchem Umfang die Rücksprünge auftreten, bleibt naturgemäß offen. Außerdem widersprechen die Rücksprünge der linearen zeitlichen Reihenfolge der Phasen.

Konsequenterweise werden in anderen Modellen Phasen nicht mehr

[5] oder das Geld alle ist ....

als Zeiträume, sondern als *Arten von Tätigkeiten* aufgefaßt, durch die zunächst vage, informelle Systembeschreibungen und darauf aufbauend immer konkretere Architekturen und Programme entwickelt werden. Alle Tätigkeitsgruppen können während der ganzen Projektlaufzeit auftreten, die Phasen liegen also parallel, es ändert sich nur der jeweilige Anteil an der Arbeitszeit: bspw. ist der Anteil der analysierenden und definierenden Tätigkeiten am Anfang sehr hoch, er sinkt zum Projektende gegen Null. Entwurfstätigkeiten haben am Anfang einen geringen Anteil, nach einer ersten Konsolidierung der Systemdefinition steigt er deutlich an, um später gegen Null zu tendieren. Parallele Phasen und evolutionäres Vorgehen sind zentrale Konzepte des *Unified Process* und seiner Varianten und beim *Extreme Programming*.

3.4 Das Spiralmodell

Das Spiralmodell [Bo86] heißt so, weil der Entwicklungsprozeß graphisch als eine Spirale in einem Koordinatenkreuz aufgezeichnet wird (s. Bild 2). Jede "Umdrehung" der Spirale steht für einen Abschnitt des gesamten Entwicklungsprozesses. Jede Umdrehung der Spirale hat den gleichen Aufbau:

1. Aufstellung von Zielen für diesen Entwicklungsabschnitt, Bestimmung von alternativen Vorgehensweisen zur Erreichung dieser Ziel (z.B. make-or-buy-Entscheidungen)
2. Evaluation der Alternativen hinsichtlich ihrer Risiken, ggf. unter Einsatz von Prototypen oder Tests, Bestimmung von Unsicherheitsfaktoren, Auswahl einer Alternative
3. Durchführung der gewählten Alternative
4. Planung des nächsten Entwicklungsziels bzw. -abschnitts
5. Review des Projektfortschritts und Entscheidung über die Projektfortsetzung

Abschnitte könnten beispielsweise die Phasen des Phasenmodells oder die einzelnen Versionen bei der evolutionären Softwareentwick-

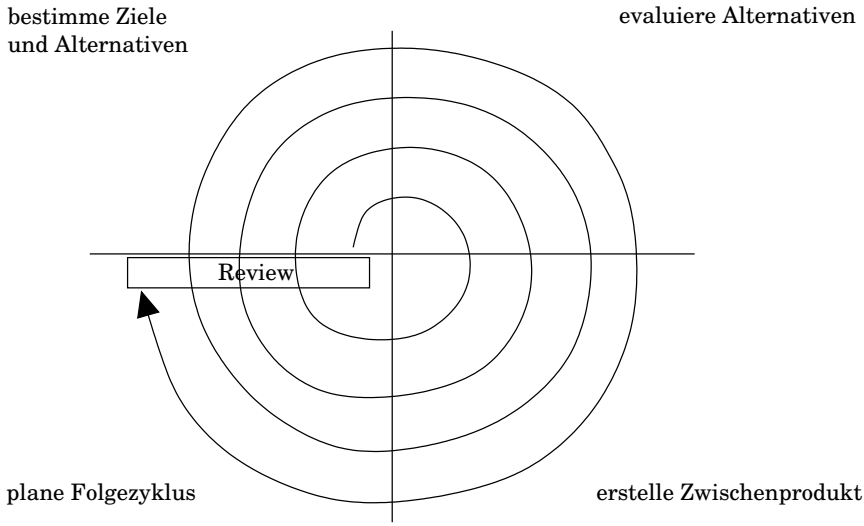

Abbildung 2: Das Spiralmodell

lung sein oder die Entwicklung irgendeines Zwischenprodukts; das Spiralmodell ist in diesem Punkt völlig offen. Im Gegensatz zu anderen Vorgehensmodellen wird hier der Entwicklungsprozeß *inhaltlich nicht vorstrukturiert.* Dies ist insofern nachteilig, als diese Planungsleistung hier im Einzelfall erbracht werden muß (sofern man diesbezüglich nicht auf die anderen Vorgehensmodelle zurückgreift). Die eigentliche Besonderheit des Spiralmodells im Vergleich zu den anderen Vorgehensmodellen liegt in der Betonung der Risikominimierung.

Das Bild der Spirale kann dahingehend mißverstanden werden, daß die vorstehenden Abschnitte ungefähr gleich lange dauern. Dies trifft nicht zu, typischerweise wird der dritte Abschnitt weitaus umfangreicher als die übrigen sein.

Literatur

[Bo86] Boehm, B.W.: A spiral model of software development and enhancement; ACM SIGSOFT 11:4, p.14-24; 1986/08

[NoS99] Noack, Jörg; Schienmann, Bruno: Objektorientierte Vorgehensmodelle im Vergleich; Informatik-Spektrum 22:3, p.166-180; 1999/06

Glossar

Evolutionäre Softwareentwicklung: grobgranulares Vorgehensmodell, bei dem iterativ praktisch einsetzbare Systemversionen erstellt werden, wobei die Anforderungen an die jeweilige Folgeversion durch den praktischen Betrieb gewonnen werden

Lastenheft: Grobkonzeption eines zu entwickelnden Systems, textuell dargestellt

Pflichtenheft: detaillierte Darstellung der Anforderungen an ein zu entwickelndes System; Ergebnis der Analyse- und Definitionsphase

Phasenmodell (*waterfall model*): grobgranulares Vorgehensmodell mitden Phasen Analyse und Definition, Entwurf, Kodierung und Modultest, Integration und Wartung.

Rapid Prototyping: Variante des Phasenmodells, bei der innerhalb der Analysephase ein "schneller" Wegwerf-Prototyp für eine interessierende Systemeigenschaft hergestellt wird

Spiralmodell: Vorgehensmodell, das die Risikoanalyse und -Minimierung betont

Unified Process: adaptierbares Vorgehensmodell für die Softwareentwicklung mit objektorientierten Methoden, insb. bei Verwendung der UML

Vorgehensmodell (*process model*): legt fest, welche einzelnen Entwicklungtätigkeiten auftreten und wie sie koordiniert werden.

Index

Adaption, 13
Analysephase, 5, 7
Anforderungen
    nichtfunktionale, 12
Architektur, 9
Aufwandsschätzung, 8

Benutzerhandbuch, 8

Datenbankschema, 11
Detailentwurf, 9
Dokument, 5, 8
    Entwurfsphase, 11
    Verfeinerungsprozeß, 6

Entwicklungskosten, 15
    Verteilung auf Phasen, 15
Entwurf, 9
evolutionäre Softwareentwicklung, 20

Fehlersuche, 13

Grobentwurf, 9

Implementierungsentwurf, 9, 10
Integrationsphase, 12

Kodierungsphase, 11
Kosten
    ∼schätzung, 8

Lastenheft, 8, 20

Meilenstein, 14
Modell
    Phasenmodell, 5

Pflichtenheft, 8, 20
Phasenmodell, 5, 20
    Analysephase, 5
    inkrementelle Entwicklung, 14
    Integrationsphase, 12
    Kodierungsphase, 11
    Linearität des ∼s, 10, 14
    Wartungsphase, 12
Portierung, 13
Programmieren im Großen, 9
Programmieren im Kleinen, 9, 11
Projektplan, 8
Prototyp, 16

Qualitätssicherung, 14

Rapid Prototyping, 16, 20
requirements engineering, 7

Spiralmodell, 20
Standardarchitektur, 9
Systemarchitektur, 8

Tätigkeit, 3
Technologieauswahl, 8
Terminplan, 8

Unified Process, 20

Vorgehensmodell, 3
Vorgehensmodelle, 20

Wartung, 15
Wartungsphase, 5, 12
Wasserfallmodell, *siehe Phasenmodell*