3. Der Software-Prozess

Software-Prozesse beschreiben den Ablauf der Entwicklung und der Pflege von Software. In diesem Kapitel werden zunächst einige grundlegenden Phänomene im Zusammenhang mit Software-Prozessen beschrieben. Dann werden ausgewählte Prozessmodelle für die Entwicklung von Software vorgestellt. Anschließend wird die Rolle von Prototypen und von Wiederverwendung und Beschaffung im Entwicklungsprozess diskutiert. Ausführungen zum Prozess der Software-Pflege schließen das Kapitel ab.

3.1 Grundlagen

3.1.1 Software-Prozesse

Den Ablauf eines Vorhabens mit der Beschreibung der Schritte, der beteiligten Personen, der für diesen Ablauf benötigten Informationen und der dabei entstehenden Informationen nennen wir einen *Prozess*.

DEFINITION 3.1. *Prozess*. Eine Folge von Schritten, die zur Erreichung eines gegebenen Zwecks ausgeführt wird. (IEEE 610.12).

Wir unterscheiden dabei zwischen ad-hoc-Prozessen, die spontan, individuell und ungeregelt ablaufen und systematischen Prozessen, deren Ablauf geplant und gelenkt wird. Auch die *Herstellung und Pflege von Software* sind Prozesse.

Im nicht professionellen Bereich wird Software typischerweise nach einem *ad-hoc-Prozess* entwickelt und gepflegt. Ein solcher Prozess hat folgende charakteristischen Merkmale: Es gibt in der Regel nur vage Sachziele und keine Termin- und Kostenvorgaben. Spezifikationen werden keine, Entwürfe nicht oder nur bruchstückhaft gemacht. Getestet wird nicht oder nur ansatzweise; Qualität ist kein Thema. Die Entwicklung hat oft den Charakter des Probierens. Die Produkte sind eher klein, kurzlebig und nicht dokumentiert. In der Regel handelt es sich um Einpersonenarbeit für den Eigengebrauch. Pflegemaßnahmen erfolgen spontan immer dann, wenn mit der Software Probleme auftreten, die sich nicht anders lösen lassen.

Im professionellen Bereich hat diese Art von Arbeit ausschließlich bei experimentellen Entwicklungen im Forschungsbereich sowie bei der Erstellung von Wegwerf-Prototypen (vgl. Kapitel 3.3) ihre Berechtigung. Man findet sie jedoch leider häufig auch in der sogenannten Endbenutzer-Entwicklung, wo Leute für ihren PC Software in dieser Art zusammenbasteln.

Wird Software als Produkt oder als Teil eines Produkts entwickelt, so braucht es zwingend *systematische* (d.h. geplante und gelenkte) *Prozesse*, sofern man keine Software-Katastrophen will. Die Abwicklung wird geplant; es bestehen klare Termin-, Kosten- und Sachziele. Der Ablauf des Prozesses wird überprüft. Mit Lenkungsmaßnahmen werden Störungen im Ablauf sowohl präventiv als auch reaktiv bekämpft. Qualität ist wichtig, da das fertige Produkt anschließend gepflegt und in der Regel weiterentwickelt werden muss.

Systematische Pflege (Wartung) ist ein kontinuierlicher Prozess, welcher die Erhaltung der Gebrauchstauglichkeit eines Software-Produkts zum Ziel hat. Der Prozess regelt nicht nur die Reaktion auf aufgetretene Probleme, sondern plant und lenkt auch die Weiterentwicklung der Software und ihre laufende Anpassung an ein sich wandelndes Umfeld.

3.1.2 Software-Projekte

Jede systematische Entwicklung von Software sowie alle größeren, abgrenzbaren Pflegevorhaben benötigen einen Abwicklungsrahmen. Dies ist in der Regel ein Software-Projekt.

DEFINITION 3.2. *Projekt*. Eine zeitlich befristete Organisationsform zur Bearbeitung einer gegebenen Aufgabe. Dient zur Regelung der Verantwortlichkeiten und zur Zuteilung der für die Arbeit notwendigen Ressourcen.

Software-Projekte unterscheiden sich von klassischen technischen Entwicklungsprojekten. Der wesentlichste Unterschied ist, dass in den klassischen Disziplinen das entstehende materielle Produkt eine Reihe von natürlichen Ansatzpunkten für Fortschrittskontrolle und Problemerkennung bietet, die es bei Software als einem immateriellen Produkt nicht gibt.

Software-Projekte müssen daher sorgfältiger geplant, überprüft und gelenkt werden als andere technische Projekte. Mögliche Mittel hierfür sind die Verwendung (und Einhaltung) geeigneter Prozesse sowie ein adäquater Umgang mit den Projektrisiken.

FESTSTELLUNG 3.1. Jede systematische Entwicklung von Software und jedes größere Software-Pflegevorhaben wird als Projekt organisiert.

3.1.3 Der Software-Lebenslauf

Der Software-Lebenslauf ist ein grundlegendes Phänomen für das Verständnis von Software-Prozessen. Betrachtet man eine beliebige Software-Komponente isoliert für sich, so stellt man fest, dass jede solche Komponente einen *Lebenslauf* hat, welcher die Stadien Initiierung – Entwicklung – Nutzung durchläuft. In der Entwicklung laufen für die Erstellung einer solchen Einzelkomponente folgende Tätigkeiten ab:

- Analysieren der Aufgabe und *Spezifizieren der Anforderungen*; Dokumentieren und Prüfen der Anforderungen
- *Konzipieren der Lösung* (Grobkonzept, Architekturentwurf); Dokumentieren und Prüfen des Konzepts
- *Entwerfen der Lösung im Detail*; Dokumentieren und Prüfen des Entwurfs
- *Codieren* und *Testen* des Programmstücks für die betreffende Einzelkomponente
- *Integrieren* mit den Programmstücken anderer Einzelkomponenten; Dokumentieren und *Testen* der Integrationsschritte
- *Installieren und Testen* der Komponente als Teil des Gesamtsystems.

DEFINITION 3.3. *Software-Lebenslauf (software life cycle)*. Zeitraum, in dem eine Software-Komponente bearbeitet oder benutzt wird. Beginnt mit der Initiierung der Komponente und endet mit ihrer endgültigen Außerbetriebsetzung. Umfasst typisch die Stadien Initiierung, Entwicklung und Nutzung, wobei die Entwicklung wiederum in die Stadien Spezifikation der Anforderungen, Konzept der Lösung, Entwurf und Programmierung, Zusammensetzung mit anderen Komponenten und Inbetriebnahme (einschließlich der Überprüfung des Entwickelten nach jedem Schritt) zerfällt.

FESTSTELLUNG 3.2. Je kleiner ein Stück Software ist, desto mehr verläuft ihr Lebenslauf linear. Bei größeren Komponenten und bei ganzen Systemen gibt es dagegen Iterationen in allen Stadien des Lebenslaufs.

Solche Iterationen werden beispielsweise durch sich ändernde Anforderungen oder durch die Entdeckung und Korrektur von Fehlern verursacht. Außerdem gibt es auch Prozessmodelle, die von vornherein Iterationen vorsehen (vgl. die Ausführungen zu Wachstumsmodellen in Abschnitt 3.2). Während der Nutzung eines Systems löst jede Anpassung oder Erweiterung die gleiche Folge von Entwicklungsaktivitäten für die anzupassenden oder neuen Komponenten aus.

3.1.4 Software-Evolution

Lehman teilt in seinem fundamentalen Artikel von 1980 die Software in drei Klassen ein.

- Software vom *S-Typ* ist solche, die vollständig durch eine formale Spezifikation beschrieben werden kann. Die Entwicklung ist erfolgreich, wenn die resultierenden Programme nachweislich die Spezifikation erfüllen. Typische Beispiele für S-Software sind Programme zum Sortieren von Daten oder zur Berechnung mathematischer Funktionen.
- Software vom *P-Typ* ist solche, die ein spezifisches, abgegrenztes Problem löst. Die Entwicklung ist erfolgreich, wenn das Problem zufriedenstellend gelöst ist. Typische Beispiele für P-Software sind Programme, welche gegebene Modelle, etwa Festigkeitsmodelle in der Statik oder Wettervorhersagemodelle in der Meteorologie berechnen.
- Software vom *E-Typ* ist solche, welche eine in der realen Welt eingebettete Anwendung realisiert. Die Entwicklung ist erfolgreich, wenn die Anwender mit der Software zufrieden sind.

FESTSTELLUNG 3.3 (LEHMAN). Nur Software vom S-Typ ist stabil. Software vom P- und E-Typ ist einer *Evolution* unterworfen. Software vom E-Typ trägt selbst zur Evolution bei.

Evolution bedeutet in diesem Zusammenhang, dass ein Produkt aufgrund des sich wandelnden Umfelds selbst einem permanenten Wandel unterliegt. Software vom E-Typ ist selbst Teil dieses Umfelds und trägt mit zum Wandel bei: Ein Produkt löst durch seine Benutzung Veränderungen in seiner Umwelt aus, die ihrerseits Veränderungen des betreffenden Produkts erforderlich machen. Die gleichen Mechanismen der Evolution finden wir auch bei allen anderen technischen Produkten Bei Software geht die Evolution jedoch besonders schnell. Dies hat zwei wichtige Konsequenzen:

FESTSTELLUNG 3.4. Es ist unmöglich, Software vom P- oder E-Typ so zu entwickeln, dass sie während einer mehrjährigen Lebensdauer ohne jegliche Änderungen ihre Aufgaben zur vollen Zufriedenheit der Benutzer erfüllt.

Pflege (Wartung) ist bei dieser Art von Software kein Unfall, der durch sorgfältigere Entwicklung hätte vermieden werden können, sondern ein unvermeidlicher Bestandteil der zu leistenden Arbeit. Bis zu einem Drittel des Gesamtaufwands für ein Software-Produkt entfällt auf die Weiterentwicklung des Produkts während der Nutzung (vgl. Bild 1.3). Alle Überlegungen zu Software, seien sie technischer oder wirtschaftlicher Art, dürfen daher nicht isoliert nur für die Software-Entwicklung gemacht werden, sondern müssen sich auf die gesamte Lebensdauer der Software beziehen.

FESTSTELLUNG 3.5. Dauert die Entwicklung von Software vom P- oder E-Typ länger als ca. ein halbes Jahr, so ist es kaum möglich, nach einem Satz einmal erhobener und dann festgeschriebener Anforderungen so zu entwickeln, dass das fertige Produkt bei seiner Inbetriebnahme die Bedürfnisse der Leute, die es benutzen sollen, optimal befriedigt.

Sich verändernde Anforderungen sind bei P- und E-Software demnach ebenfalls kein Unfall, sondern problembedingt. Da solche Veränderungen Instabilitäten und Mehrkosten in den Projekten zur Folge haben können, muss dieses Phänomen im gewählten Software-Prozessmodell möglichst berücksichtigt werden.

3.1.5 Meilensteine

Meilensteine sind das wichtigste Mittel, um den Ablauf eines Prozesses zu strukturieren und den Fortschritt wirksam zu kontrollieren.

DEFINITION 3.4. Ein *Meilenstein* ist eine Stelle in einem Prozess, an dem ein geplantes Ergebnis vorliegt.

Ein Meilenstein wird *geplant*, indem das zu erreichende Ergebnis und die dafür zur Verfügung stehenden Ressourcen (Zeit, Geld, Personal, etc.) festgelegt werden. Ein Meilenstein ist *erreicht*, wenn das verlangte Ergebnis nachweislich vorliegt. Durch Vergleichen des geplanten mit dem effektiven Ressourcenverbrauch ist eine wirksame, quantitative *Fortschrittskontrolle* möglich. Meilensteine sind nur wirksam, wenn das mit einem Meilenstein verknüpfte Ergebnis so beschrieben ist, dass die Erreichung des Meilensteins *gemessen* oder mit hinreichender Sicherheit *überprüft* werden kann. Bild 3.1 zeigt drei Beispiele.

Ergebnis in einem Software-Prozess	Eignung für Meilenstein
Codierung zu 90% beendet	unbrauchbar, da nicht messbar, sondern nur schätzbar
Die Komponente xyz hat internen Abnahmetest bestanden	geeignet, da messbar (an Hand des Abnahmetest-Protokolls)
Die Anforderungsspezifikation liegt vor	geeignet, wenn Inhalt und Form des Dokuments durch ein Review überprüft werden

BILD 3.1. Eignung von Zwischenresultaten für Meilensteine

3.2 Prozessmodelle für die Entwicklung von Software

Da man Software nicht sehen kann, ist auch der Ablauf, in welchem sie entsteht, ohne besondere Maßnahmen nicht erkennbar. Es ist daher sehr wichtig, eine *Modellvorstellung* über den Ablauf der Entstehung von Software zu haben und sowohl die Planung als auch die Kontrolle der Entwicklung an diesem Modell zu orientieren. Nur so ist ein Software-Projekt führbar.

Ein Software-Prozessmodell ist genau eine solche Modellvorstellung. Die Wahl eines geeigneten Prozessmodells ist daher eine der Grundlagen für eine erfolgreiche Software-Projektführung. In den folgenden Abschnitten werden ausgewählte Prozessmodelle vorgestellt und charakterisiert.

3.2.1 Das Wasserfall-Modell

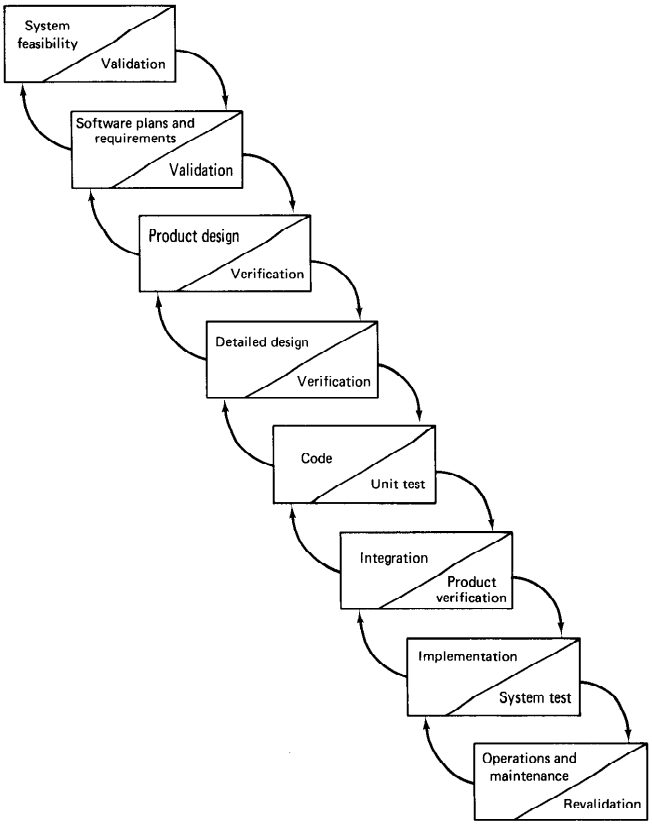
Das sogenannte Wasserfall-Modell wurde von Royce (1970) entwickelt und später vor allem von Boehm (1981) propagiert. Es ist das älteste systematische Prozessmodell für die Entwicklung von Software.

Das Wasserfall-Modell überträgt den Software-Lebenslauf, d.h. die bei der Entwicklung von Einzelkomponenten beobachtete Reihenfolge von Tätigkeiten, auf die Entwicklung eines ganzen Systems (Bild 3.2). Jede Tätigkeit bildet eine *Entwicklungsphase*. Am Ende jeder Phase steht ein Prüfschritt, welcher sicherstellen soll, dass eine Phase erst verlassen wird, wenn alle benötigten Phasenergebnisse in geprüfter und akzeptierter Form vorliegen. Trotzdem geht das Wasserfall-Modell davon aus, dass in einer Phase Probleme in den Vorgaben, d.h. den Ergebnissen der vorangegangenen Phase, auftreten können. Es sind daher lokale Iterationen zwischen aufeinanderfolgenden Phasen vorgesehen, um solche Probleme zu beheben.

DEFINITION 3.5. *Wasserfall-Modell*. Ein Modell für den Software-Entwicklungsprozess, welches die Entwicklung in eine Sequenz von Entwicklungs- und Prüfaktivitäten unterteilt. Die Reihenfolge der Entwicklungsaktivitäten folgt dem Software-Lebenslauf.

Es gibt eine große Zahl von Varianten des ursprünglichen Wasserfall-Modells, welche sich meist geringfügig in Benennung und Reihenfolge der Phasen unterscheiden. Allen diesen Modellen ist jedoch gemeinsam, dass die Unterteilung der Entwicklung in Phasen nach einer linearen Reihenfolge von Tätigkeiten erfolgt.

Aufgrund der Software-Evolution und aufgrund der Tatsache, dass die erkannten Fehler nicht immer nur in der gegenwärtigen und der vorangegangenen Phase gemacht wurden, sind bei der Anwendung des Wasserfall-Modells häufig Iterationen über mehrere Phasen hinweg erforderlich (Bild 3.3). Dies erschwert die Projektführung und hat dem Wasserfall-Modell sehr viel (berechtigte) Kritik eingetragen. Trotz seiner unbestreitbaren Vorteile (Einfachheit, Modellierung des natürlichen Lebenslaufs) und seiner immer noch beachtlichen Verbreitung sollte das Wasserfall-Modell in der Interpretation einer Folge von Entwicklungstätigkeiten heute nicht mehr verwendet werden. Gleiches gilt für alle ähnlichen Modelle, welche die Phasen als Tätigkeiten auffassen.



Quelle: Boehm, 1981

BILD 3.2. Das Wasserfall-Modell

3.2.2 Ergebnisorientierte Phasenmodelle

Ergebnisorientierte Phasenmodelle machen wie das Wasserfall-Modell den Software-Lebenslauf zur Grundlage des Prozesses. Sie gehen also ebenfalls davon aus, dass zunächst alle Anforderungen spezifiziert werden, dann das System vollständig entworfen wird, dann codiert und getestet wird, usw. In einem ergebnisorientierten Phasenmodell ist eine Phase jedoch keine Tätigkeit wie beim Wasserfall-Modell, sondern ein *Zeitintervall*.

DEFINITION 3.6. *Ergebnisorientiertes Phasenmodell.* Ein Modell für den Software-Entwicklungsprozess, welches die Entwicklung in eine Sequenz aufeinanderfolgender Zeitabschnitte (Phasen) unterteilt und in jeder Phase einen Teil der insgesamt zu liefernden Ergebnisse erarbeitet. Die Reihenfolge der Phasen orientiert sich am Software-Lebenslauf.

Die Phasen in ergebnisorientierten Phasenmodellen sind nach diesem Ergebnis oder nach der vorherrschenden Tätigkeit, die zum Ergebnis führt, benannt. Es gibt keine Phasen-Iteration. In jeder Phase können (bzw. müssen) notwendige Nacharbeiten an in früheren Phasen erzielten Ergebnissen durchgeführt werden.

Jeder Phasenabschluss bildet einen *Meilenstein* in der Entwicklung. Weitere Meilensteine können bei Bedarf hinzugefügt werden. Die geforderten Ergebnisse der frühen Phasen sind typisch Dokumente; später bestehen die Ergebnisse aus Code und Dokumenten.

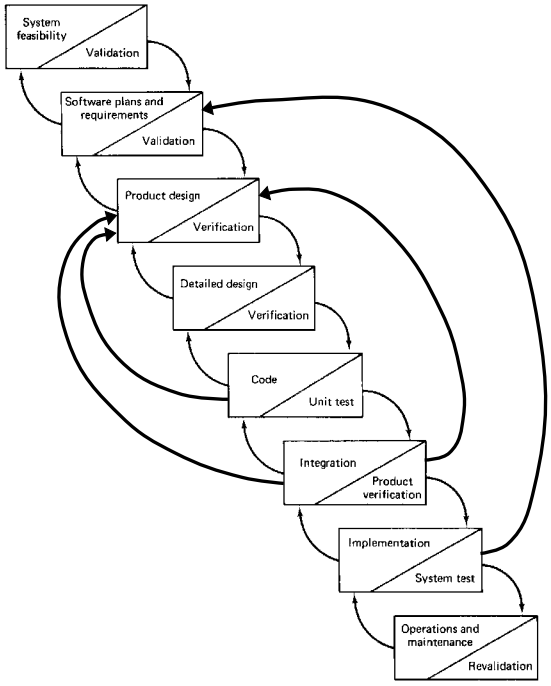


BILD 3.3. Mehrfache Iterationen im Projektablauf mit dem Wasserfall-Modell

Wie beim Wasserfall-Modell gibt es auch hier eine Vielzahl von Varianten, die sich in Reihenfolge und Benennung der Phasen unterscheiden. Bild 3.4 zeigt einen typischen Vertreter.

Ergebnisorientierte Phasenmodelle haben folgende Vorteile:

- Sie sind leicht verständlich und folgen dem natürlichen Software-Lebenslauf.
- Sie sind ein geeignetes Instrument für die Projektführung.
- Sie fördern planvolles Vorgehen mit sorgfältigem Spezifizieren, Konzipieren, Prüfen und Dokumentieren. Sie verhindern wildes Drauflos-Programmieren nach dem Motto: Einhacken–Probieren–Absturz–Ändern–Probieren–Absturz... .
- Ausführliche Spezifikationen und Entwürfe tragen erheblich dazu bei, Fehler früh zu erkennen und damit Zeit und Kosten zu sparen.

Diesen Vorteilen stehen folgende Nachteile gegenüber:

- Die Grundannahme, dass ganze Systeme genauso wie Einzelkomponenten planvoll in einer Abfolge von Schritten *konstruiert* werden können, ist vor allem für große Systeme mit langer Entwicklungs- und Lebensdauer meistens *falsch*. Solche Systeme sind in der Regel nicht konstruierbar, sondern sie *wachsen* evolutionär. Dies gilt insbesondere dann, wenn die Anforderungen an ein System nicht a priori genau bekannt sind.
- Lauffähige Systemteile entstehen erst relativ spät:
  - Die lange Durststrecke, in der nur Dokumentation produziert wird, ist nachteilig für die Motivation der Projektbeteiligten.
  - Unter Umständen wird erst sehr spät erkannt, dass vorgesehene Lösungen technisch nicht realisierbar sind, das System nicht die geforderten Leistungen erbringt oder dass es nicht das leistet, was der Auftraggeber eigentlich wollte.
- Das System muss in einem Schritt komplett in Betrieb genommen werden.

Ergebnisorientierte Phasenmodelle eignen sich daher vor allem dann, wenn

- das Projekt nicht allzu groß ist
- die Aufgaben, welche das zu erstellende System erfüllen soll, genau bekannt und präzise beschreibbar sind
- die Beteiligten über Know How im betreffenden Problembereich verfügen (z.B. ähnliche Projekte bereits abgewickelt haben)
- das Entwicklungsrisiko eher gering ist.

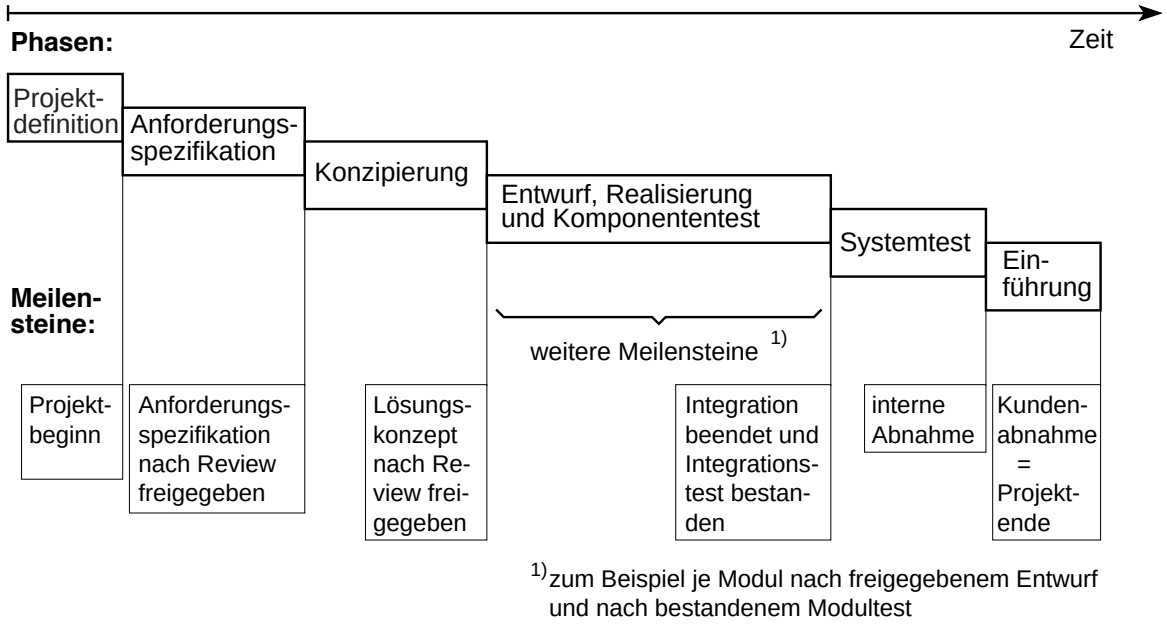


BILD 3.4. Beispiel eines ergebnisorientierten Phasenmodells

3.2.3 Wachstumsmodelle

3.2.3.1 Grundidee

Wachstumsmodelle stellen den Gedanken der *Software-Evolution* in den Mittelpunkt. Ein System wird nicht konstruiert, sondern es *wächst* in einer Reihe aufeinanderfolgender Schritte.

Es gibt verschiedene Varianten von Wachstumsmodellen unter verschiedenen Namen, z.B. Versionen-Modell, evolutionäres Modell, inkrementelles Modell. Die Ideen zu dieser Art von Modellen gehen auf Basili und Turner (1975) zurück. Sie wurden von IBM für die Entwicklung sehr großer technischer Software aufgenommen (Mills et al., 1980) und später vor allem von Gilb (1988) verbreitet.

Der Grundgedanke ist immer der folgende: Ausgehend von einer groben Zusammenstellung der Gesamtfunktionalität des gewünschten Systems wird eine Folge von *Lieferungen* mit Lieferumfang und Liefertermin definiert. Jede Lieferung ist *betriebsfähig* und kann nach Auslieferung sofort in Betrieb genommen werden. Erfahrungen mit frühen Lieferungen können bei späteren Lieferungen berücksichtigt werden.

3.2.3.2 Aufbau und Struktur

DEFINITION 3.7. *Wachstumsmodell*. Ein Modell für den Software-Entwicklungsprozess, welches die Entwicklung in eine Folge von Iterationen unterteilt. In jeder Iteration wird ein vollständiges Teilergebnis mit betriebsfähiger Software erarbeitet und ausgeliefert.

Jede Lieferung wird als weitgehend autonomes Teilprojekt organisiert. Da diese Teilprojekte in der Regel klein und kurz sind, kann ein ergebnisorientiertes Phasenmodell als Prozessmodell

verwendet werden. Bild 3.5 zeigt den zeitlichen Ablauf eines nach einem Wachstumsmodell organisierten Projekts.

Bei der Definition des Umfangs und der Reihenfolge der Lieferungen wird immer ein schrittweiser Ausbau des Systems bis zum gewünschten Endzustand geplant. Dabei können verschiedene Merkmale schrittweise ausgebaut werden, zum Beispiel die Funktionalität, der Bedienungskomfort, die Leistung oder die Verwendbarkeit. Mit letzterem ist beispielsweise der Ausbau von einer Einzweck- zu einer Vielzweckanwendung oder von einem dedizierten zu einem portablen System gemeint.

In Fällen, wo ein Auftraggeber keine Teillieferungen will, bzw. man nicht mit Teilprodukten auf den Markt gehen will, kann ein Wachstumsmodell trotzdem sinnvoll eingesetzt werden. Die Lieferungen erfolgen dann intern. Vorteile sind eine bessere Projektkontrolle und lieferfähige Teile, wenn der Liefertermin für das Gesamtsystem nicht eingehalten werden kann.

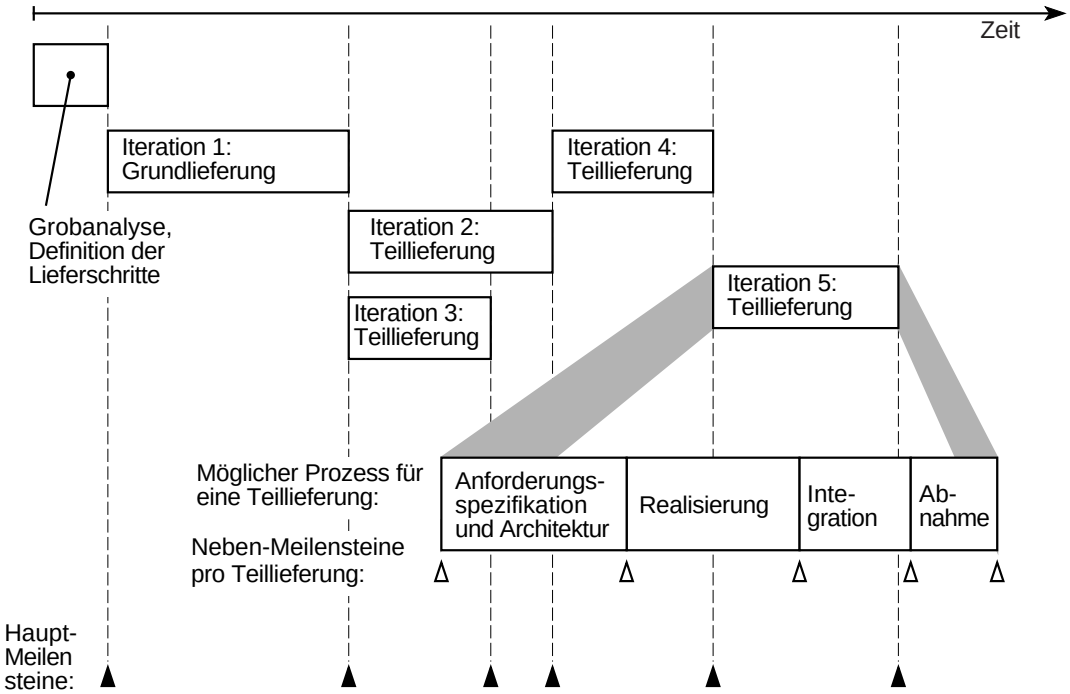


BILD 3.5. Beispiel eines Projektablaufs mit einem Wachstumsmodell

3.2.3.3 Eignung

Wachstumsmodelle haben die folgenden Vorteile:

- Sie modellieren das natürliche Verhalten der meisten großen Systeme.
- Sie sind ein sehr geeignetes Instrument zur Projektführung: Die Projektführung im Großen erfolgt mit den Meilensteinen der Lieferungen. Da jede Lieferung lauffähig sein muss, ist eine wirksame Kontrolle der Meilensteine möglich. Im Kleinen wird jede Lieferung über die Meilensteine eines ergebnisorientierten Phasenmodells kontrolliert.
- Es entstehen sehr schnell lauffähige Teile des Systems. Dies fördert die Motivation der Projektbeteiligten und hilft dem Auftraggeber, seine größten Nöte frühzeitig zu lindern.
- Sie verkürzen die Rückkopplungszyklen[1] in der Entwicklung. Dadurch werden Fehler schneller erkannt, was die Kosten für die Fehlerbehebung senkt (vgl. Kapitel 7.1, Bild 7.1).

[1] Der Rückkopplungszyklus ist die Zeitspanne zwischen der Stellung einer Aufgabe und der Bereitstellung einer Lösung, an Hand derer der Aufgabensteller den Aufgabenlösern eine Rückmeldung über Erfolg bzw. Misserfolg geben kann, d.h. ob bzw. inwieweit die Lösung die Anforderungen erfüllt.

- Das System kann sanft eingeführt werden. Es können Erfahrungen in realen Pilotanwendungen gesammelt werden. Umstellung, Schulung der Mitarbeiter, etc. können schrittweise erfolgen.

Dem stehen folgende Nachteile gegenüber:

- Es besteht die Gefahr, dass anstelle eines geschlossenen Systems viele nicht ganz zusammenpassende Teilsysteme entstehen.
- Die Konzepte und Strukturen der ersten Lieferungen können durch Ergänzungen und Änderungen in späteren Lieferungen bis zur Unkenntlichkeit entstellt werden, so dass sich die Pflegbarkeit des Gesamtsystems drastisch verschlechtert.

Systematisches, sorgfältig geplantes Arbeiten ist daher beim Einsatz von Wachstumsmodellen besonders wichtig. Die Verwendung eines Wachstumsmodells darf kein Vorwand für ein Abgleiten in die ad-hoc-Entwicklung sein!

Wachstumsmodelle eignen sich vor allem, wenn

- das System groß ist und eine lange Entwicklungszeit hat
- nicht die volle Funktionalität sofort erforderlich ist
- man ein Software-Vorhaben nicht vorab in allen Einzelheiten definieren kann, sondern zunächst Erfahrungen sammeln will oder muss
- ein Basisprodukt schnell auf dem Markt bzw. beim Kunden sein soll.

3.2.3.4 Kontinuierliche Integration

Jedes Wachstumsmodell ist dadurch charakterisiert, dass am Ende jeder Teillieferung eine betriebsfähige und auslieferbare Version des entwickelten Systems zur Verfügung steht. Wachstumsmodelle mit kontinuierlicher Integration gehen hier noch weiter und sorgen dafür, dass spätestens ab Fertigstellung der Grundlieferung *ständig* eine betriebsfähige Referenzversion des Systems zur Verfügung steht, und dass diese Referenzversion in sehr kurzen, regelmäßigen Intervallen immer wieder neu erzeugt wird. Bei jeder Erzeugung werden die bis dahin neu entwickelten Systemteile in die Referenzversion integriert. Kontinuierliche und weitgehend automatisierte Tests sorgen dafür, dass die Referenzversion auch tatsächlich betriebsfähig ist und dass die neu integrierten Teile das korrekte Funktionieren der bisher vorhandenen Software nicht beeinträchtigen. Dieses Vorgehen setzt eine ausgefeilte und gut funktionierende Konfigurationsverwaltung der Software (vgl. Kapitel 11) voraus. Bei guter Organisation ist eine tägliche Neuerzeugung der Referenzversion möglich und wird in der Praxis mit Erfolg praktiziert (vgl. Cusumano und Selby 1995). Auf Englisch wird dieses Vorgehen als “daily build” bezeichnet. Tägliche Integration funktioniert vor allem dann gut, wenn (a) ein Entwicklungsvorhaben in viele kleine, weitgehend autonome Teile zergliedert werden kann und (b) die Erzeugung einer neuen Referenzversion nur geringen Aufwand verursacht (indem sie weitestgehend automatisiert wird). Andernfalls sind längere Integrationszyklen (zum Beispiel wöchentlich oder zweiwöchentlich) oder das Grundmodell mit Integration nur am Ende jeder Teillieferung wirtschaftlicher.

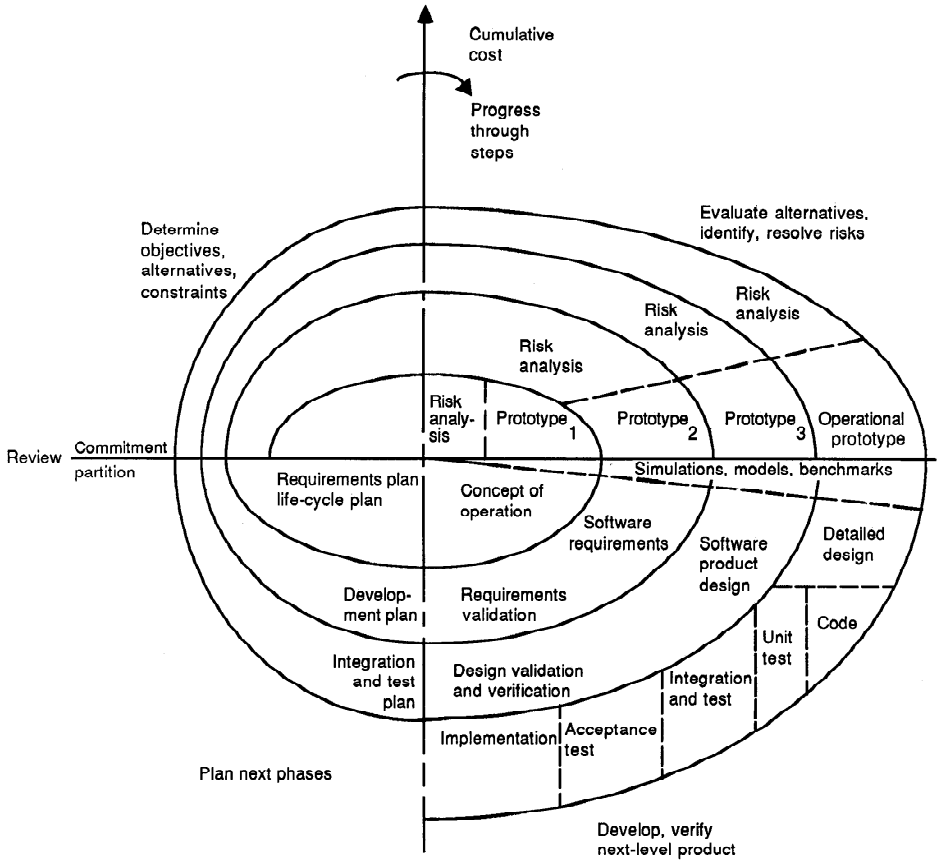
Der große Vorteil der kontinuierlichen Integration sind die gegenüber einem gewöhnlichen Wachstumsmodell nochmals stark verkürzten Rückkopplungszyklen und die damit verbundene Senkung der Fehlerkosten. Nachteilig ist, dass notwendige Restrukturierungen, welche einen größeren (und zeitaufwendigen) Umbau der bestehenden Software bedingen, häufig unterbleiben, weil sie sich nicht als Folge kleiner, inkrementeller Änderungen realisieren lassen.

3.2.4 Das Spiralmodell

Das Spiralmodell (Bild 3.6) ist eine von Boehm (1988) stammende Weiterentwicklung des Wasserfall-Modells. Es ist vor allem für die Entwicklung *großer, risikoreicher* Systeme gedacht.

Anstelle der zyklischen Abfolge „Entwickeln, prüfen, entwickeln, prüfen...“ tritt beim Spiralmodell ein vierteiliger Zyklus, der in der graphischen Modell-Darstellung den vier Quadranten des Koordinatensystems entspricht:

1. Planung des nächsten Spiralumlaufs
2. Zielsetzung bzw. Zielkorrektur
3. Untersuchung von Varianten und ihren Risiken, Variantenentscheid
4. Entwicklung und Prüfung (dieser Schritt korrespondiert mit dem Wasserfall-Modell).



Quelle: Boehm, 1988

BILD 3.6. Das Spiralmodell

Der Zyklus wird viermal durchlaufen, nämlich für *Systemdefinition*, *Anforderungen* an die Software, *Konzipierung* (Architektur-Entwurf) und *Realisierung* (Detailentwurf, Codierung, Test, Integration und Installation). Wichtig ist vor allem der dritte Schritt im Zyklus, wo die *Risiken* der Entwicklung erkannt und kontrolliert werden. Boehm selbst bezeichnet das Spiralmodell als *risikogesteuert*. Das Hauptmittel hierfür sind *Prototypen* .

Gelegentlich findet man unter dem Namen „Spiralmodell“ auch Modelle, die eigentlich Wachstumsmodelle sind, bei denen aber jeder Wachstumsschritt als ein Spiralumlauf dargestellt ist.

3.2.5 Agile Software-Entwicklung

In jüngster Zeit hat die Idee der sogenannten agilen Software-Entwicklung (agile software development) bzw. eines agilen Prozessmodells sehr viel Aufmerksamkeit gefunden. Die Grundüberlegung dabei ist es, ausgehend von den einfachen und erfolgreichen Vorgehensweisen für die Entwicklung von Kleinstsoftware einen einfachen und möglichst wenig reglementierten Prozess zu schaffen, welcher die Anwendung dieser Verfahren auch für große Software mit vielen Beteiligten ermöglicht (vgl. Kapitel 1, Tabelle 1.1).

Ein agiles Prozessmodell geht daher von folgenden Prinzipien aus:

- *Je kleiner die Teilaufgaben und je kürzer die Rückkopplungszyklen, desto besser*. Sehr kleine Teilaufgaben lassen sich in der Regel in einem Schritt realisieren, ohne dass viel Aufwand für Planung, Anforderungsklärung und Entwurf investiert werden muss. Die Validierung (d.h. die Überprüfung, ob die Lösung den Bedürfnissen der Auftraggeber entspricht) kann aufgrund sehr kurzer Rückkopplungszyklen mit vertretbaren Kosten am fertigen Produkt erfolgen. Da die Gesamtaufgabe in vielen kleinen Schritten gelöst wird, ist kontinuierliche Integration (vgl. Abschnitt 3.2.3.4) unerlässlich.
- *Der Kunde gestaltet das Produkt während seiner Entstehung*. Da Kunden zu Beginn eines Projekts häufig nicht genau wissen, was sie wollen und sich die Kundenanforderungen fortlaufend ändern (vgl. Kapitel 7), wird die Aufgabe in vielen kleinen Teilschritten angegangen, ähnlich einem Wachstumsmodell. Allerdings sind die Teillieferungen viel zahlreicher und erfolgen in viel kürzeren Intervallen als bei einem typischen Wachstumsmodell. Ebenfalls anders als bei einem typischen Wachstumsmodell erfolgt zu Beginn keine genaue Lieferungsplanung. Stattdessen entscheidet der Kunde fortlaufend über die Priorität und die Reihenfolge der zur Lösung anstehenden Teilaufgaben. Vom Entwicklungsteam erhält er ebenso fortlaufend Schätzungen über den Realisierungsaufwand für die anstehenden Teilaufgaben, so dass er seine Entscheidungen an Kosten, Nutzen und erforderlicher Zeit orientieren kann. Dieses Vorgehen setzt voraus, dass der Kunde aktiv im Software-Entwicklungsprozess mitarbeitet. Da „der Kunde“ als Person in der Regel nicht existiert, muss die Auftraggeberseite einen kompetenten und entscheidungsbefugten(!) Vertreter in das Software-Entwicklungsteam delegieren.
- *Je freier und konzentrierter die Entwickler arbeiten können, desto besser*. Menschen sind dann am produktivsten, wenn die Arbeit ihnen Spaß macht. Die meisten Software-Entwickler programmieren sehr gerne und sind an anderen Tätigkeiten, insbesondere dem sogenannten „Papierkram“, weniger bis gar nicht interessiert. Agile Software-Entwicklung versucht daher, die Entwickler so weit als möglich von Planungsaufgaben, Sitzungen und dem Schreiben von Dokumentation zu entlasten, so dass sie sich auf das Programmieren konzentrieren können.
- *Die Qualität wird an der Quelle sichergestellt*. Fehler werden vermieden, indem Programme zu zweit entwickelt werden (sogenanntes Pair Programming): Eine Person denkt und schreibt, die andere Person liest und denkt fortlaufend mit. Von der mitlesenden Person erkannte Fehler und Schwächen werden sofort behoben. Lokale Fehler in einer Teillösung werden rasch gefunden, indem für jede Teillösung gleichzeitig die Software und eine Menge von Testfällen für diese Software entwickelt werden und die Teillösung noch vor der Integration mit diesen Testfällen getestet wird.
- *Keine Arbeit auf Vorrat*. Die Struktur der Lösung wird nicht vorausgeplant, sondern entwickelt sich. Vorgegeben wird nur eine Leitmetapher für die Architektur der Software (vgl. Kapitel 6). Erst wenn sich bei der Integration von Teillösungen strukturelle Probleme ergeben, wird die bis dahin entwickelte Software restrukturiert (sogenanntes Refactoring).

Erste Erfahrungen zeigen, dass agile Software-Entwicklung unter folgenden Voraussetzungen funktioniert:

- Der Auftraggeber delegiert einen kompetenten und entscheidungsbefugten Vertreter in das Software-Entwicklungsteam des Auftragnehmers.
- Das Problem ist in hinreichend viele kleine Teilaufgaben unterteilbar.
- Das Entwicklungsteam ist klein. Bei größeren Aufgaben müssen mehrere, parallel arbeitende Teams eingesetzt werden.
- Im Entwicklungsteam gibt es einen Software-Architekten. Dies ist eine Person, welche tiefgehende Kenntnisse über Software-Architektur (vgl. Kapitel 6) hat und über einschlägige Erfahrungen verfügt. Sie gestaltet die Architektur der inkrementell entstehenden Software. Ihre Entscheidungen werden von den übrigen Teammitgliedern respektiert und umgesetzt. Erfordert

die Größe der Aufgabe ein Arbeiten mit mehreren, parallel arbeitenden Teams, so braucht es zusätzlich einen übergeordneten Software-Architekten, welcher die Gesamtarchitektur der Software festlegt und überwacht.
- Die kontinuierliche Integration ist gut organisiert und benötigt wenig Aufwand.
- Die Sicherstellung der Qualität an der Quelle wird tatsächlich konsequent durchgehalten.

Sind diese Bedingungen nicht erfüllt, so stürzt ein als agil geplantes Software-Projekt blitzschnell ins Chaos und in die ad-hoc Entwicklung ab: Ist der Kundenvertreter inkompetent, ist die auf Grund seiner Vorgaben entwickelte Software unbrauchbar oder das Projekt dreht sich im Kreis. Ist er nicht entscheidungsbefugt, steht das Projekt ständig still, weil auf Entscheidungen gewartet werden muss. Gibt es keinen guten Architekten, ist die Struktur der Software schlecht und sind die Weiterentwicklungs- und Pflegekosten zu hoch. Unterbleibt die Sicherstellung der Qualität an der Quelle, sind die Fehlerkosten zu hoch und das Projekt gerät rasch in den Zustand der Selbstbeschäftigung, wo jedes neue Inkrement so viele Probleme und Folgefehler in der bereits vorhandenen Software erzeugt, dass das Projekt vom Entwicklungs- zum Reparaturprojekt verkommt.

Die Vor- und Nachteile eines agilen Prozessmodells sind ähnlich denen des Wachstumsmodells, allerdings in verstärkter Form: Die Flexibilität und die fortlaufende Orientierung an den Bedürfnissen des Auftraggebers sind höher; ebenso ist aber auch die Gefahr der Erzeugung chaotischer, qualitativ schlechter Software markant erhöht.

3.3 Prototypen

Prototypen sind ein zentrales Mittel zur frühzeitigen Erkennung und Lösung von Problemen in der Software-Entwicklung.

DEFINITION 3.8. *Prototyp*. Ein lauffähiges Stück Software, welches kritische Teile eines zu entwickelnden Systems vorab realisiert.

Der Prozess der Erstellung und Nutzung von Prototypen wird als *Prototyping* bezeichnet. Es werden drei Arten von Prototyping unterschieden: *exploratives*, *experimentelles* und *evolutionäres* Prototyping (Floyd, 1984; Kieback, Lichter, Schneider-Hufschmidt und Züllighoven 1992).

Exploratives Prototyping dient zur Klärung und Bestimmung von Anforderungen an Hand von lauffähigen Programmen, welche den zu klärenden Ausschnitt aus dem Gesamtproblem realisieren. Exploratives Prototyping wird einerseits verwendet, um mit sogenannten *Demonstrationsprototypen* die prinzipielle Nützlichkeit und Machbarkeit von Systemideen zu demonstrieren (v.a. im Bereich der Akquisition und der Projekt-Initiierung). Andererseits dient es dazu, ein konkretes, von den zukünftigen Benutzern erprobbares und kritisierbares Modell einer geplanten Informatik-Lösung zu schaffen. An Hand eines solchen *Prototyps im engeren Sinn* können die Angemessenheit von Anforderungen und die Tauglichkeit von Lösungen für den geplanten Verwendungszweck untersucht werden. Beispielsweise können so Benutzerbedürfnisse geklärt oder geplante Benutzerschnittstellen erprobt werden. Bei Produktentwicklungen kann exploratives Prototyping außerdem zur Gewinnung von Wissen über einen bisher nicht beherrschten Problembereich dienen.

Experimentelles Prototyping wird verwendet, um die Realisierbarkeit kritischer Systemteile zu untersuchen und um Entwurfsalternativen zu bewerten. Beispielsweise kann so geklärt werden, ob bestimmte geforderte Leistungen (Reaktionszeiten, Datenraten, ...) erbracht werden können. Solche Prototypen werden auch als *Labormuster* bezeichnet.

Demonstrationsprototypen, Prototypen im engeren Sinn und Labormuster sind *Wegwerf-Prototypen.* Das geplante System wird auf der Grundlage des mit den Prototypen erworbenen

Wissens neu entwickelt. Es ist zulässig, solche Wegwerf-Prototypen nicht zu dokumentieren und schnelle, softwaretechnisch unsaubere Lösungen zu verwenden. Es muss allerdings sichergestellt sein, dass niemand der Versuchung erliegt, aus Software, die als Wegwerfprototyp geschrieben wurde, dann doch durch Weiterentwicklung ein Produkt zu machen. Ist ein solcher Ausbau beabsichtigt, so muss ein evolutionäres Prototyping gewählt werden (siehe unten). Die Erstellung von Wegwerfprototypen ist wirtschaftlich, wenn es gelingt, bei der eigentlichen Systementwicklung aufgrund der Erfahrung mit den Prototypen mehr Kosten zu sparen, als die Prototypen selbst gekostet haben, oder wenn mit Prototypen ein gefährliches Risiko wesentlich gemindert werden kann (vgl. Kapitel 4, Risiken).

Im Gegensatz dazu ist beim *evolutionären Prototyping* der Prototyp ein *Pilotsystem*, das den Kern des zu entwickelnden Systems bildet, operational eingesetzt wird und durch schrittweise Ergänzungen und Erweiterungen zum geplanten System weiterentwickelt wird. Dementsprechend muss ein evolutionärer Prototyp sorgfältig und nach allen Regeln der Kunst entwickelt werden. Evolutionäres Prototyping entspricht im Vorgehen einem Wachstumsmodell.

Das Arbeiten mit Prototypen ist somit kein eigenständiges Modell für die Entwicklung von Software:

- *Demonstrationsprototypen* unterstützen die Akquisition von Aufträgen und das Aufsetzen von Projekten.
- *Prototypen im engeren Sinn* sowie *Labormuster* dienen zur Risiko-Minderung und können in jedem Prozessmodell zu jedem Zeitpunkt für diesen Zweck eingesetzt werden.
- Das Arbeiten mit *Pilotsystemen* ist eine Form des Wachstumsmodells.

3.4 Wiederverwendung und Beschaffung

3.4.1 Mehrfachverwendung und Produktivität

Die Produktivität bei der Entwicklung neuer Software lässt sich nach heutigen Erkenntnissen nur in ziemlich eng begrenztem Rahmen steigern, weil wesentliche Teile des Entwicklungsprozesses kreative Arbeiten sind, die nicht automatisierbar sind und nicht durch den Einsatz von Hilfsmitteln massiv beschleunigt werden können (Brooks 1987).

Leistungssteigerungen von mehreren Größenordnungen bei gleichem Preis innerhalb weniger Jahre, wie man dies bei der Hardware beobachten kann, hat es bei der Entwicklung neuer Software nie gegeben und wird es wohl auch nie geben. Der Schlüssel für diesen Unterschied liegt in den verkauften Stückzahlen. *Die Masse macht es*.

Die Kosten für die Neuentwicklung komplexer Hardware, z.B. eines Prozessors, stehen den Kosten für die Entwicklung komplexer Software um nichts nach. Im Gegensatz zur Software aber werden Hardware-Komponenten nach ihrer Entwicklung in großen Stückzahlen verkauft, so dass die Entwicklungskosten pro verkauftem Exemplar drastisch sinken.

Die einzige Möglichkeit, die Kosten pro verkauftem Exemplar Software massiv zu senken, ist eine ebenso massive Steigerung der Stückzahl, in der diese Software verwendet, bzw. verkauft wird. Besonders wirtschaftlich sind also einerseits große Serien eines Produkts mit identischer Software und andererseits die Wiederverwendung von Software.

3.4.2 Wiederverwendung

Wiederverwendung von Software fällt nicht vom Himmel. Sie stellt sich auch nicht automatisch ein, wenn sie als Unternehmensziel postuliert wird, sondern man muss ihr auf die Sprünge helfen. Dies hat technische und organisatorische Gründe.

Technisch muss wiederverwendbare Software zunächst einmal geschaffen werden. Hierzu muss neu erstellte Software so gestaltet, in Komponenten gegliedert und dokumentiert sein, dass

entsprechende Komponenten überhaupt herausgelöst und in einem anderen Kontext sinnvoll wiederverwendet werden können.

Im Bereich des Organisatorischen gibt es im Wesentlichen drei Probleme, welche einer breiten Wiederverwendung von Software im Weg stehen:

- Wiederverwendbare Software ist in ihrer Herstellung aufwendiger als Einzweck-Software, weil sie allgemeingültiger angelegt und ausführlicher dokumentiert werden muss. Für jemanden, der nur für ein Projekt verantwortlich ist, besteht daher kein Anreiz, die für das Projekt notwendige Software wiederverwendbar zu gestalten. Wiederverwendbare Software muss deshalb explizit honoriert werden.
- Wiederverwendung scheitert häufig daran, dass Projektleiter oder Projektmitarbeiter nicht wissen, welche im Unternehmen vorhandene Software in ihrem Projekt wiederverwendet werden könnte. Es muss daher Kataloge wiederverwendbarer Software geben.
- Wiederverwendung scheitert häufig daran, dass die Urheber von potenziell Wiederverwendbarem nicht bereit sind, irgendwelche Garantien oder Pflegeverpflichtungen für ihre Software zu übernehmen. Wiederverwendung muss daher wie Beschaffung vertraglich geregelt werden.

Optimal für die Förderung von Wiederverwendung wäre es, wenn es Unternehmen oder unternehmensinterne Profit Center gäbe, die vom Handel mit wiederverwendbaren Software-Komponenten leben. Ansätze in dieser Richtung sind beispielsweise kommerziell vertriebene Klassenbibliotheken und Software-Rahmen (Frameworks).

Eine andere Möglichkeit zur Förderung von Wiederverwendung innerhalb eines Unternehmens ist die Schaffung einer Informationsstelle für Wiederverwendung. Diese trägt Informationen über wiederverwendbare Software zusammen und führt einen Katalog. Interessierte Projektleiter bzw. Entwickler können sich bei der Informationsstelle nach vorhandenen Komponenten für ihr Problem erkundigen.

Wichtig ist, dass beim Sammeln das Holprinzip herrscht: die Informationsstelle sammelt selbst aktiv bei den Entwicklungsabteilungen; diese werden nicht mit zusätzlichem Aufwand belastet (sonst kooperieren sie nicht).

3.4.3 Beschaffung

Wo immer möglich, werden Systeme bzw. Software-Komponenten nicht entwickelt, sondern beschafft. Dies hat rein ökonomische Gründe: aufgrund der verkauften Stückzahlen ist beschaffte Software fast immer kostengünstiger als selbst entwickelte. Allerdings fallen neben den reinen Beschaffungskosten fast immer auch Parametrierungs- und Anpassungskosten an, so dass die Wirtschaftlichkeit einer Beschaffung in jedem Einzelfall geprüft werden muss.

Der Beschaffungsprozess ist typisch in den Software-Entwicklungsprozess eingebettet, weil Spezifikations-, Test- und Installationsarbeiten auch bei Beschaffungen notwendig sind. Die Hauptarbeit ist im Rahmen der Konzipierung der Software zu leisten (siehe Kapitel über Konzipierung).

3.4.4 Wiederverwendung und Beschaffung im Entwicklungsprojekt

Alle Anstrengungen zur Mehrfachverwendung von Software (Wiederverwendung, Verwendung beschaffter Software) sind nutzlos, wenn die Entwicklerinnen und Entwickler nicht mitmachen. Projektleitende und Entwickelnde dürfen ihren Berufsstolz nicht darin sehen, möglichst alles selbst, neu und perfekt zu machen. Sie müssen vielmehr lernen, dass nicht das Perfekte (und Teure) die beste Lösung ist, sondern das Gebrauchstaugliche, das so weit wie möglich aus vorhandenen Komponenten gefertigt ist.

Die Suche nach Beschaffbarem bzw. Wiederverwendbarem muss zur fest verankerten Tätigkeit in jedem Projekt werden. Damit zeitgerecht entschieden werden kann, sind bereits während der Anforderungsspezifikation entsprechende Vorabklärungen zu treffen.

Die Mitarbeiterinnen und Mitarbeiter dürfen nicht danach beurteilt werden, ob sie möglichst viel Neues produzieren. Sie müssen vielmehr danach beurteilt werden, ob sie Nützliches und Bedarfsgerechtes unter Verwendung von möglichst viel Vorhandenem produzieren.

3.5 Pflege (Wartung)

Pflege, oft auch Wartung (maintenance) genannt, ist der Prozess der systematischen Erhaltung der Gebrauchstauglichkeit von Programmen. Software vom S-Typ (vgl. Abschnitt 3.1.4) braucht keine Pflege, sofern sie richtig entwickelt worden ist. Die nachfolgenden Ausführungen beziehen sich daher auf Software vom P- und E-Typ. Die Pflegemaßnahmen werden in der Regel in drei Klassen unterteilt: Fehlerbehebung, Anpassung und Weiterentwicklung (Erweiterungen, Verbesserungen). Bild 1.3 zeigt die typische Verteilung des Aufwands auf die drei Klassen.

DEFINITION 3.9. *Pflege (Wartung, maintenance).* Modifikation bestehender Software und/oder Ergänzung bestehender Software durch neue Software mit dem Ziel, Fehler zu beheben, die bestehende Software an veränderte Bedürfnisse oder Umweltbedingungen anzupassen oder die bestehende Software um neue Fähigkeiten zu erweitern.

Bei großer Software, die zudem bei mehreren Kunden in Betrieb ist, ist es unmöglich, jede Änderung sofort in Betrieb zu nehmen. Der Verwaltungs- und Installationsaufwand wäre viel zu hoch. Stattdessen werden die Änderungen gebündelt und von Zeit zu Zeit en bloc als neuer *Release* freigegeben (siehe auch Kapitel über Konfigurationsmanagement).

DEFINITION 3.10. *Release*. Eine konsistente Menge von Komponenten eines Systems, die gemeinsam zur Benutzung freigegeben werden.

Bei Software bestehen die Komponenten aus einer Menge von Programm-, Daten- und Steuerkomponenten, die zusammen ein betriebsfähiges System bilden sowie aus Komponenten, welche das System und seine Benutzung dokumentieren.

Meist wird zwischen kleinen und großen Releases unterschieden und dies durch die Nummerierung des Release nach dem Schema m.n kenntlich gemacht. m.n bezeichnet den n-ten kleinen Release innerhalb des m-ten großen Release. Der Abstand zwischen zwei kleinen Releases beträgt in der Regel einige Monate; der zwischen großen Releases ein bis drei Jahre.

Pflegemaßnahmen, auch wenn sie sorgfältig durchgeführt werden, führen typisch dazu, dass die gepflegte Software größer und in ihrem strukturellen Aufbau schlechter wird. Lehman und Belady haben Zahlenmaterial über die Pflege von Software erhoben und daraus sechs Gesetze abgeleitet (Belady und Lehman 1976, Lehman 1980, Lehman 1992). Alle Gesetze gelten nur für große, während ihrer Lebenszeit ständig benutzte Programme vom P- und vor allem vom E-Typ.

REGEL 3.1. *Gesetz der kontinuierlichen Veränderung*. Ein Programm unterliegt während seiner gesamten Lebenszeit ständiger Veränderung oder es verliert nach und nach seinen Nutzen. Dieser Prozess schreitet fort, bis es kostengünstiger wird, das Programm durch ein neuerstelltes Nachfolgerprogramm zu ersetzen.

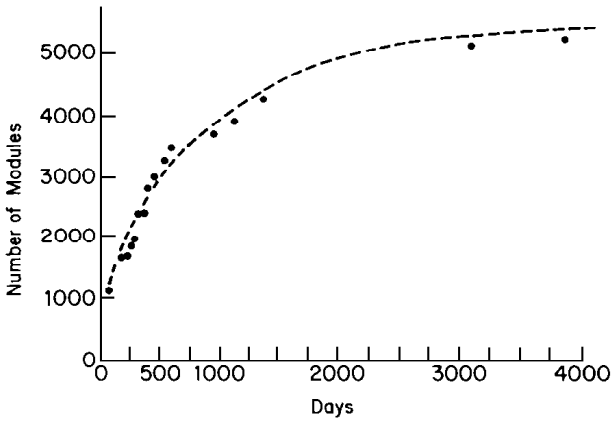
REGEL 3.2. *Gesetz der zunehmenden Komplexität.* Die Komplexität eines Programms, das ständig verändert wird, nimmt kontinuierlich zu, weil seine Struktur durch die Änderungen immer schlechter wird; es sei denn man ergreift explizite Gegenmaßnahmen.

REGEL 3.3. *Grundgesetz der Programm-Evolution*. Die Evolution eines Programms unterliegt einer sich selbst regelnden Dynamik mit statistisch bestimmbaren Tendenzen und Invarianzen.

REGEL 3.4. *Gesetz der Invarianz des Arbeitsaufwands*. Global betrachtet ist der Aufwand, der in die Pflege eines Programms gesteckt wird, während der ganzen Lebenszeit des Programms statistisch konstant.

REGEL 3.5. *Gesetz der Invarianz des Release-Inhalts*. Der Inhalt der Releases (Änderungen, Ergänzungen, Weglassungen) ist über die Folge aller Releases während der Lebenszeit eines Programms statistisch konstant.

REGEL 3.6. *Gesetz des kontinuierlichen Wachstums*. Ein Programm muss während seiner gesamten Lebenszeit kontinuierlich wachsen, wenn es gebrauchstauglich bleiben soll.



Quelle: Lehman und Belady (1985), p. 307

BILD 3.7. Wachstum des Betriebssystems OS 360/370 von IBM

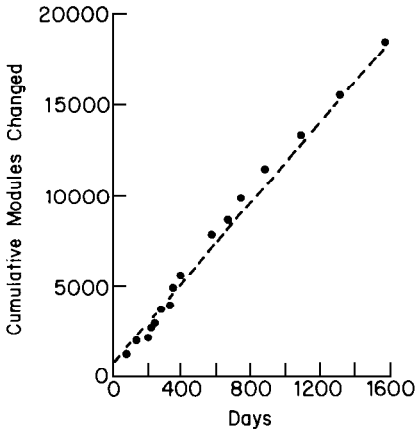
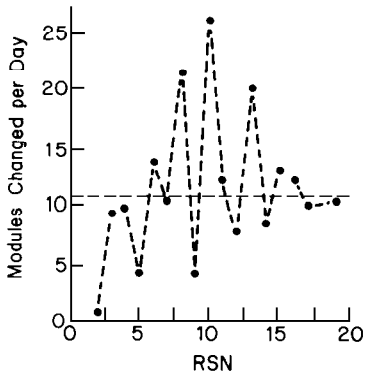


BILD 3.8. A. Kumulierte Anzahl geänderter Module



Quelle: Lehman und Belady (1985) p. 416

B. Änderungsrate über der Folge der Releases (RSN: Release sequence number)

Während die Mechanismen, die zu den Regeln 3.1, 3.2 und 3.6 führen, intuitiv klar sind, sind die Regeln 3.3 bis 3.5 eher kontra-intuitiv. Intuitiv würde man erwarten, dass der Pflegeprozess durch einschlägige Management-Entscheidungen nachhaltig beeinflussbar ist. Die von Lehman und Belady für mehrere großen Systeme erhobenen Zahlen belegen jedoch das Gegenteil. Das Wachstum folgt klar einer mathematischen Kurve (Bild 3.7), die kumulierte Anzahl geänderter Module folgt einer Geraden, d.h. die Änderungsrate ist konstant (Bild 3.8 a) und die Anzahl der pro Tag geänderten Module, aufgezeichnet über der Folge der Releases, ist statistisch konstant (Bild 3.8 b). Offensichtlich sind die Trägheitsdynamik großer Systeme und Organisationen sowie die begrenzten Fähigkeiten sowohl der Systementwickler wie der Systembenutzer zum

Be- und Verarbeiten («Verdauen») von Änderungen so stark, dass sie die Eingriffe von außen marginalisieren, den Pflegeprozess statistisch stabilisieren und ihn selbstregulierend machen.

Aufgaben

3.1 Ein zu entwickelndes System besteht aus einer Basiskomponente mit Datenbank und Benutzerschnittstelle sowie sechs voneinander weitgehend unabhängigen Anwendungsfunktionen. Für die Basiskomponente wird ein Entwicklungsaufwand von 12 Personenwochen geschätzt; der Aufwand pro Anwendungsfunktion auf je drei Personenwochen.
Sie sind Projektleiterin. Ihr Management verlangt einen Meilenstein mit dem Kriterium «System zu 80% fertig».
a) Wie reagieren Sie?
b) Formulieren Sie ein messbares Kriterium für einen solchen Meilenstein.

3.2 Gegeben sei das Problem aus Aufgabe 2.5 (Statistischen Auswertung von Betriebsdaten).
(a) Welches Prozessmodell schlagen Sie für die Entwicklung der geforderten Software vor? Begründen Sie Ihre Entscheidung.
(b) Angenommen, Ihre Abteilungsleiterin entscheidet unabhängig von Ihrem Vorschlag, dass dieses Projekt nach einem Wachstumsmodell abzuwickeln ist. Skizzieren Sie einen Lieferplan, indem Sie Anzahl und Reihenfolge der Lieferungen festlegen und für jede Lieferung entscheiden, welche Komponenten der geforderten Lösung in welchem Ausbauzustand darin enthalten sein sollen.

3.3 a) Wo und mit welchem Ziel kann Prototyping im Software-Entwicklungsprozess eingesetzt werden?
b) Schildern Sie je eine typische Projektsituation, in der die Verwendung eines Demonstrationsprototyps, eines Prototyps im engeren Sinn, eines Labormusters und eines Pilotsystems sinnvoll sind.

3.4 Ein Unternehmen betreibt ein selbst entwickeltes System zur Auftragsabwicklung. Die Anwender beklagen sich seit längerer Zeit über die mühsame und nicht mehr zeitgemäße Bedienung des Systems. Mit der Funktionalität des Systems sind die Benutzer nach wie vor zufrieden. Soeben hat die Geschäftsleitung beschlossen, die Benutzeroberfläche des Auftragsabwicklungssystem zu erneuern und hat Sie mit der Durchführung des entsprechenden Projekts beauftragt. Zur Minimierung des Projektrisikos wollen Sie Prototyping einsetzen. Welche Arten von Prototypen setzen Sie wozu ein?

3.5 Begründen Sie das Gesetz des kontinuierlichen Wachstums von Lehman (Regel 3.6). Ziehen Sie dabei auch Bild 3.7 heran.

Ergänzende und vertiefende Literatur

Basili und Turner (1975), Boehm (1981, 1988), Royce (1970) sowie Gilb (1988) sind Schlüsselquellen für die vorgestellten Prozessmodelle. Larman und Basili (2003) geben einen Überblick über die Geschichte der Wachstumsmodelle.

Barnes und Bollinger (1991) diskutieren wirtschaftliche Aspekte der Wiederverwendung und Maßnahmen zur Förderung von Wiederverwendung.

Floyd (1984) gibt eine Taxonomie der verschiedenen Arten des Prototypings; Kieback et al. (1992) geben eine kurze Übersicht über Prototyping als solches und beschreiben dann verschiedene industrielle Prototyping-Projekte.

Lehman (1980) ist der Klassiker über die Evolution von Software.

Die Welle der agilen Software-Entwicklung wurde durch *eXtreme Programming* von Kent Beck (2000) angestoßen. Weitere Bücher zu diesem Thema sind Cockburn (2002) sowie Lippert, Rook und Wolf (2002).

Das Problem der Definition und der Verbesserung der realen Software-Prozesse in Unternehmen wird in diesem Kapitel nicht behandelt. Interessierte Leserinnen und Leser seien auf Humphrey (1989) verwiesen.

Zitierte Literatur

Barnes, B.H., T.B. Bollinger (1991). Making Reuse Cost-Effective. *IEEE Software* **8**, 1 (Jan. 1991), 13-24.

Basili, V.R., A.J. Turner (1975). Iterative Enhancement: A Practical Technique for Software Development. *IEEE Transactions on Software Engineering* **SE-1**, 6. 390-396.

Beck, K. (2000). *Extreme Programming Explained – Embracing Change*. Reading, Mass.: Addison Wesley.

Boehm, B. (1981). *Software Engineering Economics*. Englewood Cliffs, N.J.: Prentice-Hall.

Boehm, B.W. (1988). A Spiral Model of Software Development and Enhancement. *IEEE Computer* **21**, 5 (May 1988), 61-72.

Brooks, F.P. (1987). No Silver Bullet. Essence and Accidents of Software Engineering. *IEEE Computer* **20**, 4. 10-19.

Cockburn, A. (2002). *Agile Software Development*. Reading, Mass.: Addison Wesley.

Cusumano, M.A., R.W. Selby (1995). *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. New York: Simon & Schuster.

Floyd, C. (1984). A Systematic Look at Prototyping. In Budde et al. (Hrsg.): *Approaches to Prototyping*. Berlin, etc.: Springer. 1-19.

Gilb, T. (1988). *Principles of Software Engineering Management*. Reading, Mass.: Addison Wesley.

Humphrey, W. S. (1989). *Managing the Software Process*. Reading, Mass.: Addison-Wesley.

IEEE (1990). *Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12-1990. IEEE Computer Society Press.

Kieback A. , H. Lichter, M. Schneider-Hufschmidt, H. Züllighoven (1992). Prototyping in industriellen Software-Projekten. *Informatik-Spektrum* **15**, 2 (April 1992). 65-77.

Larman, C., V.R. Basili (2003). Iterative and Incremental Development: A Brief History. *IEEE Computer* **36**, 6 (Jun 2003). 47-56.

Lehman, M.M. (1980). Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE* **68**, 9, 1060-1076. (Nachgedruckt als Kapitel 19 in Lehman und Belady 1985).

Lehman, M.M., L.A. Belady (Hrsg.) (1985). *Program Evolution: Processes of Software Change*. London, etc.: Academic Press.

Lehman, M.M. (1992). A Model of Software Evolution and its Implications. ABB Computer Science Conference, Kopenhagen.

Lippert, M., S. Rook, H. Wolf (2002). *Software entwickeln mit eXtreme Programming – Erfahrungen aus der Praxis*. dpunkt.Verlag.

Mills, H.D., D.O'Neill, R.C. Linger, M. Dyer, R.E. Quinnan (1980). The Management of Software Engineering, Part I-V. *IBM Systems Journal* **19**, 4. 414-477.

Royce, W. (1970). Managing the Development of Large Software Systems. *Proceedings IEEE WESCON*. 1-9. (Nachgedruckt in Proceedings 9th International Conference on Software Engineering, Monterey, 1987, 328-338.)